NYILATKOZAT

Név: Becsó Gergely

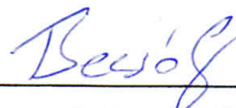
ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: JRU5UJ

Szakdolgozat címe:
Az utazóügynök probléma közelítő megoldásai neurális hálókkal

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2022.05.31



a hallgató aláírása

Eötvös Lóránd Tudományegyetem
Természettudományi Kar

Az utazóügynök probléma közelítő megoldásai neurális hálókkal

Becsó Gergely

Matematika BSc
Alkalmazott matematika szakirány

Témavezető:
Lukács András
Számítógéptudományi Tanszék

Budapest, 2022

Tartalomjegyzék

1. Köszönetnyilvánítás **3**

2. Bevezetés **4**

3. Az utazóügynök feladat (TSP) **5**

3.1. Javító algoritmusok 6

3.2. Konstruktív algoritmusok 6

3.3. A gyakorlatban használt programok 8

4. Mély tanulás **9**

4.1. Egy neurális háló felépítése, működése 9

4.2. Graph Neural Networks 10

4.3. Attention 11

5. Reinforcement Learning **14**

5.1. Egy dinamikus környezet matematikai formalizmusa 14

5.2. Mély Q-tanulás 15

5.3. REINFORCE 17

5.4. Actor-Critic 19

6. Eddigi TSP háló-modellek **20**

6.1. Elvárások egy TSP-hálótól 20

6.2. Tang IterGNN-je 20

6.3. Mele konstruktív algoritmusa 22

6.4. Wu transformer modellje 25

7. Mérések **30**

Hivatkozások **32**

1. Köszönetnyilvánítás

Szeretném megköszönni témavezetőmnek, Lukács Andrásnak a nagyszerű témajavaslatot, a rendszeres konzultációkat, az iránymutatást az internetes szakirodalom útvesztőjében és azt, hogy bevezetett a gépi tanulás izgalmas világába.

Szeretném megköszönni mindazoknak a nagyszerű embereknek, akiket kereshettem kisebb-nagyobb elakadásaimmal. A teljesség igénye nélkül - Ács Juditnak, Csiszárik Adriánnak, Vas Bernadettnek, Csanády Bálintnak.

És végül, de nem utolsósorban szeretném megköszönni családomnak és barátaimnak a végtelen türelmüket, megértésüket és támogatásukat, amit folyamatosan kapok tőlük.

2. Bevezetés

Középiskola óta tudtam, hogy egyszer meg szeretném érteni a gépi tanulást. Egyszer azt hallottam egy programozó ismerősömtől, hogy "a világon kb 40 ember érti, a többi pedig fekete dobozként használja, pedig nem is érti rendesen". Az egyetemi tanulmányaim alatt nagyon megszerettem a gráfalgoritmusokat, az optimalizálási feladatokat és a modellezés folyamatát, amikor egy való életbeli jelenséget leírunk matematikával. Ezen érdeklődési területeim mentém elindultam szakdolgozati témát és témavezetőt keresni, és rendkívül gyorsan irányítottak Lukács Andráshoz. Kiderült, hogy manapság intenzíven kutatott terület a mélytanulás alkalmazása a kombinatorikus optimalizálási feladatokon [Cap+21], főleg NP-nehéz feladatok közelítő megoldására. A területen a mélytanulás összes technikáját igyekeznek alkalmazni mind a klasszikus algoritmusok helyettesítésére, mind a teljesítményük javítására.

Egy év alatt életszerűtlennek tűnt, hogy a gépi tanulás alapfogalmaitól az egész terület áttekintéséig tudnék jutni, így kiválasztottam az utazóügynök problémát (TSP), mely az egyik legnépszerűbb kombinatorikus optimalizálási feladat mély tanulásos körökben. Egyszerűen leírható, szinte minden fajtája NP-teljes, általános eredményeket mutatnak fel mind a mai napig klasszikus eszközökkel is és az életben rengeteg alkalmazása van [MSM10]. Ezek között megtalálható járművek útvonal-optimalizálásától áramkör tervezésig mindenféle érdekes terület.

Ilyen szűkítések mellett is rengeteg irányba el lehetett indulni a számtalan különféle kutatási irány miatt. Megismerkedtem a megerősítéses tanulással, mely segítségével a Google Deepmind csapata komoly és látványos eredményeket ért el több játékban, felülmúlván a legjobb embereket és a legtöbb korábban alkotott botot (Sakk, Go, StarCraft, Dota). Tanultam az attention mechanizmusról, mely segített új szintre emelni a természetes nyelvfeldolgozást, és jelenleg újabb és újabb áttörésekben játszik fontos szerepet a gépi tanulás több területén. Olvashattam konvolúciós hálókról, melyek elvét hasznosították a gráfalgoritmusok világában is.

Az induláshoz a cikkek mellett hasznos támpontot adtak korábbi évek szakdolgozatai, főleg Harsányi Benedek [Ben] és Kulcsár Zoltán [Zol] munkái.

Az alábbi dolgozatban a 3. fejezetben átnézem az utazóügynök feladat alapfogalmait, fajtáit és releváns algoritmusait. A 4. fejezet a mély tanulásról szól, az alapok gyors áttekintése mellett szó esik a GNN-ekről, ami a régebbi gráfokhoz alkotott hálók alapja és az attention-ről, amely egy újabb módszer. Az 5. fejezetben folytatjuk a mélytanulás világának megismerését, a megerősítéses tanulás alapjait tekintjük át. Majd a 6. fejezetben a TSP három különböző mélyhálós megközelítését vizsgáljuk, végül a 7. fejezetben a 6.4. alfejezetben látott architektúrával néhány saját kísérlet eredményét mutatom be.

3. Az utazóügynök feladat (TSP)

Az utazóügynök feladat egy gráfelméleti probléma, ahol egy súlyozott gráfban kell egy minimális összsúlyú Hamilton-kört keresni, azaz egy olyan kört, ami minden csúcsot pontosan egyszer érint. Nevét, Travelling Salesman Problem (TSP) onnan kapta, hogy egy városokat és az őket összekotő utakat tekintve gráfnak, a feladat megoldása megmondja egy utazó ügynöknek, hogy tudja a lehető legkevesebb utazással eljuttatni a portékáját minden városba.

Adott $G(V,E)$ gráf, egy $c: E \to R$ súlyfüggvény, a feladat a V csúcsok egy olyan permutációjának meghatározása, hogy minden $i \in (0,1\dots n)$-re (v_i, v_{i+1}) él, $v_0 = v_n$, és $\sum_{n=0}^{n} c_{v_i v_{i+1}}$ minimális. Külön megkérdezhető, hogy létezik-e megoldás, és ha létezik megoldás, mi az optimális megoldás.

A feladatot Karl Menger írta le elsőként 1930 körül. Többször megfoglamazták postások, iskolabuszok segítségével, míg 1949-ben Julia Robinson adta neki a jelenlegi nevét. A 20. század második felében rengetegen kutatták a témát. Mivel a feladat NP-teljes, így olyan megoldást, ami polinomiális futásidőben talál optimális megoldást, nem ismerünk. Már egy egyszerű gráf Hamilton-körének létezése is NP-teljes. Viszont mivel az alkalmazásoknál a jó közelítő megoldások is bőven elegendőek, vannak heurisztikus módszereink, közelítő algorimusaink a megoldás keresésére.

Az első polinomiális közelítő algoritmusa TSP egy alesetére Cristofides nevéhez fűződik 1976-ból, ez $3/2$ közelítő volt. Ezen 2011-ben és 2020-ban sikerült javítani $3/2-\varepsilon$ közelítőre [KKG20], ahol $\varepsilon > 10^{-36}$. Jelenleg a metrikus esetre ez a legjobb, amit tudunk. Ennél erősebb eredmény is van egy kisebb feladatra, az euklideszi TSP-re Sanjeev Arora 1998-ban alkotott egy a PTAS[1] algoritmust [Aro98], amiért 2010-ben Gödel-díjat kapott.

A '90-es években a létező legjobb módszerek összegyúrásából megszületett a Concorde program, ami a gyakorlatban a legjobban használható program napjainkig. 1991-ben Geralt Leinelt létrehozta a TSPlib nevű könyvtárat, ami rengeteg, kölünböző nehézségű feladatot tartalmaz. A kutatók ezt használják napjainkban a különböző heurisztikák összehasonlítására. A feladatnak több fajtáját különböztetjük meg, bizonyos plusz megszorítások szerint.

3.1. Definíció. Metrikus TSP

Ebben a feladatban vannak térbeli pontjaink és egy metrikánk a téren. Úgy kapjuk meg a gráfot, hogy a pontok lesznek a csúcsok, és ezen egy teljes gráfot veszünk. A súlyfüggvény egy adott élre a kezdő és végpontjának a távolsága a tér metrikája szerint.

3.2. Definíció. Euklideszi TSP

A gyakorlatban a leginkább használt verzió az ún. Euklideszi TSP feladat, ami a metrikus TSP azon alesete, ahol az Euklideszi tér L_2 távolságát használjuk élsúlyozásra. Ez az eset adta a legtöbb motivációt a téma kutatására, hiszen a való életben rendszeresen előforduló feladatról van szó. A feladat ilyen megkötésekkel is NP-teljes [Pap77].

3.3. Definíció. Szimmetrikus/Asszimetrikus TSP

Létezik a szimmetrikus és az asszimetrikus felosztás (G irányítatlan, vagy irányított). Ez előkerül például forgalomirányításnál.

[1] Polinomial-time approximation scheme

3.1. Javító algoritmusok

A javító algoritmusok inputja egy TSP feladat és egy (nem optimális) megoldása. A javító algoritmus ezen egy valamilyen javító lépést tesz meg. A teljesség igénye nélkül ilyenek a k-opt algoritmusok, a Lin-Kerninghan, a szimulált hűlés, a hangyaboly-optimalizálás vagy a genetikus algoritmusok.

Az ilyen javító algoritmusok működését jobban megérthetjük, ha elképzeljük a Hamilton-körök gráfját. Ebben a gráfban a csúcsok a lehetséges Hamilton-körök, és két csúcsot összeköt él, ha egy javító lépéssel át lehet jutni egyik körből a másikba. Ekkor minden javító algoritmus egy sétát generál ezen a gráfon. Mivel a gráfnak akár $n-1!$ csúcsa is lehet, így egy javító algoritmus futásidejére nem tudunk általános esetben hasznos felső becslést adni.

2-opt

A 2-opt egy lokális javító algoritmus, ami 2-opt lépésekből áll. Az algoritmus kiindul egy megengedett megoldásból, majd végrehajt egy 2-opt lépést, ami a következőképpen néz ki: a jelenlegi permutációnk $(v_1, v_2, \dots v_n)$. Keressünk két élet (v_i, v_{i+1} és v_j, v_{j+1}, $i < j$), melyek kezdő és végpontjait felcserélve egy jobb (vagy legalább másik) megoldást kapunk. A lépés végén a permutációnk: $(v_1, v_2, \dots v_i, v_j, v_{j-1}, \dots, v_{i+1}, v_j, v_{j+1}, \dots v_n)$. Ha a gráfunk teljes, minden állatpotban n^2 különböző 2-opt lépés lehetséges.

Meggondolható, hogy ilyen 2-opt lépések sorozatával tetszőleges megoldásból tetszőleges megoldásba átmozoghatunk, de ha csak olyan lépéseket engedünk meg, amikor éppen javítunk, akkor könnyen beragadunk egy lokális optimumba. A 2-opt lépés erejét mutatja az, hogy a fejezet elején felsorolt összes javító algoritmus 2-opt lépéseket használ, valamilyen kiegészítéssel, hogy lokálisan rontó lépéseket is megengedjünk.

k-opt és v-opt

A k-opt lépés a 2-opt általánosítása. Az éppen aktuális körtúrát szétbontjuk k útra, majd ezeket valamilyen módon újra összekötjük egy körré. A 2-opt esetében 2 utat kétféleképp lehet összeilleszteni, így ez egy egyértelmű lépés, azonban $k = 3$ esetén 8 féle összeillesztés van (azaz 7 lépést hívhatunk 3-optnak), általános esetben pedig $(k-1)!2^{2(k-1)}$ módon illeszthetjük össze a k szakaszunkat.

A v-opt módszerek valamilyen sorrendben tesznek k-opt lépéseket, ahol k lépésről-lépésre változhat. A leghíresebb ilyen módszer Shen Lin és Brian Kerninghan algoritmusa, amit 1973-ban publikáltak [LK73].

3.2. Konstruktív algoritmusok

Az euklideszi TSP feladatot a postások napi rendszerességgel oldják meg, de ők jellemzően nem valamilyen algoritmust kérnek meg, hogy adják meg az aznapi útvonalat, hanem a józan észben bízva veszik sorra a címeket. A gyakorlatban az ilyen józan észen alapuló megfontolásokból születnek a heurisztikák, ezekből néhányat járunk körben ebben a fejezetben.

Mohó (legközelebbi szomszéd)

A mohó megközelítése a feladatnak, hogy elindulunk egy tetszőleges v_0 pontból és minden lépésben a legközelebbi szomszédot vesszük hozzá a T túrához. Ha minden pontot hozzávettünk,

térjünk vissza v_0-ba.

Input: $G(V,E,c)$ élsúlyozott gráf, $v_0 \in V(G)$ kezdőpont
Output: T körtúra

$T \leftarrow \emptyset$
$M \leftarrow V \setminus v_0$
$v \leftarrow v_0$
while $M \neq \emptyset$ **do**
  v_i legyen v legközelebbi olyan szomszédja, hogy $v_i \in M$
  $T \leftarrow T \cup vv_i$
  $M \leftarrow M \setminus v_i$
  $v \leftarrow v_i$
end while
$T \leftarrow vv_0$
  return T

Azaz, az i-ik lépésben adott egy $T = (v_0, v_1, \ldots, v_{i-1})$ út, amit már bejártunk és mi a v_{i-1} pontban tartózkodunk. Vegyük a legközelebbi szomszédot, amit még nem láttunk, vegyük be a körtúrába a hozzá vezető utat. Az út végén vissza is kell menni v_0-ba. Ez az algoritmus a TSPLIBbeli példákon áltagosan $1,25$-ször hosszabb megoldásokat ad, mint az optimum.

Legtávolabbi pontot beszúró

Ennél a módszernél szinte minden lépésben van egy K zárt körutunk, ezt fogjuk bővíteni.

Az első 3 lépésben csinálunk egy háromszöget, majd mindig a legtávolabb eső csúcsot fogjuk beszúrni a körbe.

Input: $G(V,E,c)$ élsúlyozott gráf, $v_0 \in V(G)$ kezdőpont
Output: T körtúra

$T \leftarrow v_0$
$M \leftarrow V \setminus v_0$
v_1 legyen v_0 legtávolabbi olyan szomszédja, hogy $v_1 \in M$
$M \leftarrow M \setminus v_1$, $T \leftarrow T \cup v_0 v_1$
v_2 legyen olyan, hogy $c(v_1 v_2) + c(v_0 v_2)$ maximális
$M \leftarrow M \setminus v_2$, $T \leftarrow T \cup v_0 v_2 \cup v_1, v_2$
while $M \neq \emptyset$ **do**
  for all $v_i \in M$ **do**
    $edge_i \leftarrow uw \in T$, úgy, hogy $c(uv_i) + c(wv_i)$ minimális
    $cost_i \leftarrow c(uv_i) + c(wv_i)$
  end for
  $v \leftarrow v_i : cost_i$ maximális
  $M \leftarrow M \setminus v$
  $T \leftarrow T \setminus edge_i$
  $T \leftarrow T \cup uv \cup wv$
end while
  return T

Az algoritmus megoldása a TSPlib példáin átlagosan $1,16$-szorosa az optimumnak.

Multi fragment (MF)

A multi-fragment szintén egy mohó megközelítése a feladatnak, ha egy él kicsi, valószínűleg nem lesz tőle hosszú a túra. Tehát minden lépésben vegyük azt az élet, aminek bevétele nem okoz kört, ezen belül pedig a lehető legrövidebb és vegyük hozzá az élhalmazunkhoz. Az algoritmus végén egy kört kapunk.

Input: $G(V,E,c)$ élsúlyozott gráf
Output: T körtúra

$T \leftarrow \emptyset$
$L \leftarrow V$ növekvő sorrendben
for $i \in L$ **do**
  if $T \cup i$ fa **then**
    $T \leftarrow T \cup i$
  end if
end for
$u, v : deg_T(u) = deg_T(v) = 1$
$T \leftarrow T \cup uv$ **return** T

2015-ben belátták , hogy a metrikus TSP-re ez az algoritmus $O(log(n))$ közelítő [BH15].

Clarke-Wright (CW)

Clarke és Wright 1964-ben publikálták a tanulmányukat [CW64], amiben egy általánosabb jármű-ütemezési feladatra adtak egy heurisztikus megoldást. Ennek egy alesete az utazó-ügynök feladatot oldja meg. A algoritmus kis módosítással $O(log(n))$ közelítő [BH15].

Ennél a technikánál kiválasztunk egy megkülönböztetett v_0 kezdőpontot. Az éleket aszerint fogjuk értékelni, hogy mennyit takarítunk meg, ha az útvonalon u-ból v-be direkt megyünk, az uv_0v úthoz képest.

Input: $G(V,E,c)$ élsúlyozott gráf, v_0 kezdőpont
Output: T körtúra

$saving_{v_iv_j} = c(v_iv_0) + c(v_jv_0) - c(v_iv_j)$, $i, j \neq 0$
$L \leftarrow E(G \setminus v_0)$ csökeknő sorrendbe rendezve *saving* szerint
for $i \in L$ **do**
  if $T \cup i$ fa **then**
    $T \leftarrow T \cup i$
  end if
end for
$u, W : deg_T(u) = deg_T(w) = 1$
$T \leftarrow T \cup uv_0 \cup wv_0$
  return T

3.3. A gyakorlatban használt programok

David Applegate, Robert E. Bixby, Vašek Chvátal, és William J. Cook 1990-ben megalkották a Concorde programot. A jelenlegi tudásunk szerinti legjobb módszereket ötvözték. 2006-ban kiszámolták a segítségével egy 85,900 városból álló feladat optimális megoldását. Mivel a Concorde szabadon elérhető akadémiai használatra, az különböző fejlesztéseket hozzá szokás hasonlítani. A Concorde a pontos megoldást branch-and-cut módszerrel keresi, emellett össze-

hasonlításul képes kiszámolni különféle heurisztikus megoldásokat is: véletlenszerű túra, Lin-Kernighan, legközelebbi szomszéd, mohó és a Boruvka módszereket. A program sajátossága az egyedi fájl-formátuma, a .tsp.

Egy másik, hasonló szerepet betöltő, bár jóval kevésbé híres szoftver az LKH-3 (Lin-Kernighan-Helsgaun). Keld Helsgaun fejlesztette ki, 2017-ben publikálta ezt a Lin-Kernighan algoritmusán alapuló megoldóprogramot. A Concordehoz hasonlóan, akadémiai használatra ez is szabadon felhasználható. Ugyan az őse, az LKH-2 mindössze négyféle problémát (szimmetrikus és asszimetrikus TSP, Hamilton-kör és -út) tudott megoldani, az LKH-3 már 29 különböző feladat megoldására alkalmas. Ezek mindegyike valamilyen útvonaltervezési vagy csomagszállítási feladat, különböző megszorításokkal.

Az ELTE-n fejlesztett Lemon szoftverben is implementáltak több különböző heurisztikus megoldást (a két mohó algoritmust, különféle beszúrásos módszereket, Cristofides algoritmusát és a 2-optot).

A Google OR Tools csomagjában is található egy pontos TSP-megoldó. Az OR Tools egy Google által fejlesztett nyílt forráskódú csomag, amiben olyan problémákra adnak megoldásokat, mint például a lineáris programozási feladatok vagy különféle gráf feladatok.

4. Mély tanulás

A való életbeli feladatmegoldáshoz a matematika úgy áll, hogy minden feladatothoz építünk egy matematikai modellt, melyben a feladatot és megoldását is matematikai objektumokkal reprezentálunk. Jelen esetben feladat megoldása egy paraméterezett leképezés a bemeneti adatok teréből a megoldások terébe. Ezt az f_θ paraméteres leképezést keressük - mikor pontosan, mikor közelítő módon. A gépi tanulás alapötlete, hogy keressünk egy olyan f_θ paraméteres függvényt, ami megfelelő paraméterekkel (θ^*) "jól" megoldja a feladatot. Tanulás alatt a θ paraméterek megfelelő beállítását értjük és ezt a folyamatot szeretnék automatizálni.

Minnél általánosabb problémát szeretnénk megoldani, annál nagyobb paraméterhalmazra van szükségünk a jó modellezéshez. Jellemzően a gép tanulásnál θ egy olyan vektor, aminek a dimenziója akár a milliós nagyságrendet is elérheti. Ekkora paraméterhalmazt beállítani igen erőforrásigényes, ezért a számítógépek fejlődésével újabb és újabb áttörések várhatóak.

4.1. Egy neurális háló felépítése, működése

A neurális hálók világa mostanában népszerű téma, így a témakör alaptechnikáit ismertnek tekintem. A neurális hálók vektoradatokkal képesek dolgozni, így egy probléma megoldásakor az első lépés az input adatok vektorizálása. Szerencsére viszonylag kicsi (legfeljebb pár száz) dimenziójú vektorokkal a gyakorlatban rengeteg problémát le lehet írni. Képfeldolgozásnál például pixelenként 3 szám (RGB kódolás szerint) megadása tökéletesen leírja a képet. Gráfokat tudunk szomszédsági mátrix formájában, akár egyéb gráfbeágyazásokkal reprezentálni. Az életben pedig képek és gráfok segítségével már rengeteg feladatot le lehet írni.

A vektorizált adaton egy neurális háló különböző műveleteket végez. A műveletek alapegysége a mesterséges neuron, amiről a technika a nevét is kapta. Egy mesterséges neuronnak vannak

bemenetei (melyeket a vektorizált adatunkból, vagy más neuronokból kap) és vannak súlyai (amelyeket a tanulás folyamán kell beállítani). A neuron az bemeneteket skalárszorozza a súlyokkal, majd egy nem-lináris függvényt alkalmaz rá. Ez a szám a neuron kimenete. A neuronokat rétegekbe rendezzük, egy rétegnek így egy vektor (esetleg mátrix) bemenete és egy vektor (esetleg mátrix) a kimenete. A különböző feladatokra különböző típusú rétegeket fejlesztettek ki. Néhány ilyen rétegtípusról később szó fog esni ebben a dolgozatban is. Az egymás után csatolt rétegek alkotják a neurális hálót. Ennek a bemenete és a kimenete is egy vektor, amit valahogy értelmezni kell. Ez az értelmezés feladatonként változik.

Ahhoz hogy a hálókat használni tudjuk, kell egy módszer, amivel be tudjuk állítani a paramétereket, be tudjuk tanítani a hálót. Ehhez szükségünk van egy (jellemzően nagy) címkézett adathalmazra. Megnézzük adott állapotban adott bemenetre mit ad a függvényünk, megnézzük ez milyen messze van az ideális kimenettől (veszteségfüggvény), majd ez alapján frissítjük a paramétereket. Ennek módszere a gradiens emelkedés módszere. A frissítés a visszaterjesztés (backpropagation) alogritmussal történik. A gyakorlatban a gradiens módszernek fejlettebb változatait használjuk, a legelterjettebb talán az Adam Optimizer névre hallgató eljárás.

A megoldás szépsége, hogy mivel a háló kimenetét a veszteségfüggvénnyel értelmezzük valamilyen módon, nem kell külön foglalkoznia a hálónak a kimenete értelmezésével. A veszteségfüggvény minimalizálásával automatikusan olyan eredményeket fog nekünk kiadni, amit pont úgy kell értelmezni, ahogy mi mindig is értelmeztük az adatot. A megoldás nehézsége, hogy jellemzően rengeteg adatra van szükségünk, amikhez egy jó címkézés is tartozik. A gépi tanulás egyik fő kihívása, hogy ilyet nem mindig egyszerű szerezni vagy generálni.

A megoldás egy érdekessége, hogy a gradiens módszerből következik egy nagyon fontos jelenség: ha a súlyok valamilyen módon szimmetrikusak, a visszaterjesztés során az oda tartozó gradiensek (és ezáltal a súlyok változásai) is szimmetrikusak maradnak. Nekünk pedig az a fontos, hogy a különböző bemeneteket különböző súllyal vegyük figyelembe. Ezért fontos, hogy a hálókat random súlyokkal szokás inicializálni. Ennek a véletlennek az egyenetlenségeit fogja a tanítás úgy formálni, hogy ez valamilyen lényegi információt hordozzon.

A nagyobb, hasznos modellek teljes betanításához jelentős gépigényre van szükség. Sokaknak ez nem áll rendelkezésére, de ők is dolgozhatnak ilyen modellekkel, ugyanis létezik a "transfer learning" módszere. Ez azt jelenti, hogy ha van egy nehéz feladat, amihez egy (nagyon) hasonlót már megoldottak, akkor annak súlyait felhasználva kezdőállapotnak a jó megoldáshoz viszonylag közel tudunk kezdeni, így rengeteg munkát spórolhatunk.

4.2. Graph Neural Networks

A mélytanulás alapintuíciója, miszerint mesterséges neuronok üzeneteket küldözgetnek egymásnak. Ezt az alképzelést ha szeretnénk gráfokra alkalmazni, több lehetséges megközelítést kapunk. Ezek körül egy csoportot hívunk Gráf Neurális Hálónak (GNN). Egy népszerű és látványos technika az ún. Graph Network Block-ok (GB block) használata. Ezek segítségével kidolgozott gráfalgoritmusokat lehet szépen átírni a mélytanulás világába, általános gráfelméleti feladatokat lehet megfogalmazni. Többek között így a TSP-t is.

GN blokk [Tan+20]

A bemeneti gráfot reprezentáljuk $G = (u, V, E)$, ahol u egy globális attribútumvektor, $V =$

$\{v_1, \ldots v_n\}$ csúcsattribútumvektorok és $E = \{e_1, \ldots e_n\}$ élattribútumvektorok. Minden esetben igaz, hogy s_i, r_i, az él két végpontja bele van kódolva az e_i élattribútum vektorba. Egy klasszikus GN blokknak hat metódusa van. Három üzenetküldő függvény (ϕ^e, ϕ^v, ϕ^u) és három összesítő ($\rho^{e \to v}, \rho^{e \to u}, \rho^{v \to u}$) függvény. Az üzenetküldő függvények valamilyen vektorból vektorba képező függvények, például egy MLP (multi layer perceptron). Az összesítő függvényeknek valamilyen permutáció invariáns összesítésnek kell lenniük, például maximum vagy átlag. Először minden s_k csúcs küld egy e'_k üzenetet minden belőle kimenő e_k élre. Ezt majd r_k csúcs fogadja. Az üzenetet ϕ^e a fentieken kívül a globális attribútumokból számolja

$$e'_k = \phi^e(e_k, v_{s_k}, v_{r_k}, u).$$

Ezután minden fogadó csúcs a $\rho^{e \to v}$ függvény segítségével összesíti.

$$\mathbf{e'_i} = \rho^{e \to v}(e'_k | r_k = i)$$

Az összesített üzenetek segítségével frissítjük a csúcsok attribútumait

$$v'_i = \phi^v(\mathbf{e'_k}, v_i, u)$$

Az összes elküldött üzenet összesítjük, ezzel majd a globális attribútumot szeretnénk frissíteni

$$\mathbf{e'} = \rho^{e \to u}(e'_k)$$

Összesítsük a csúcsattribútumokat is, ezzel is a globális attribútumokat frissítjük majd

$$\mathbf{v'} = \rho^{v \to u}(v'_i)$$

Mindezekből frissítjük a globális attribútumokat

$$v'_i = \phi^u(\mathbf{e'}, \mathbf{v'}, u)$$

Ezek a GN blokkok egy nagyon általános eszközként funkcionálnak, amiket jól lehet egy-egy feladat megoldására használni.

4.3. Attention

A természetes nyelvfeldolgozás a mélytanulás egy fontos alkalmazási területe. Vaswani és társai 2017-ben publikálták az "Attention is All you Need [Vas+17]" c. cikküket, ameyben bemutatták az attention mechanizmust és az erre épülő transzformer architektúrát. Az attention lényege, hogy az input egyes elemeinek a beágyazását a többi input kontextusában módosítja. Előnye a korábbi verziókkal szemben, hogy rendkívül jól párhuzamosítható, mátrix-szorzások és vektoronkét végzett egyszerű függvények kompozíciójáról van szó. A módszer erejét többek között az is mutatja, hogy a természetes nyelvfeldolgozáshoz készített módszert éppen TSP problémához fogjuk használni.

A modell első alkalmazása az volt, hogy egy mondatot egyik nyelvről a másikra fordítson. Ezt úgy tette meg, hogy a mondatot mátrix alakban reprezentálta (egy szó egy vektor) majd egy sorozat elkódoló blokk segítségével ezt beágyazta egy látens térbe. Innen egy másik nyelvre kikódolta egy sorozat dekódoló blokk segítségével. Ebben a dolgozatban mindössze az elkódolás rész értése szükséges.

Egy elkódoló blokk egy jellemzően egy attention rétegből és egy sűrű rétegből áll. A szokásos trükköket (batch normalization, reziduális kapcsolások) természetesen lehet használni, de az egyszerűség kedvéért most ezek nélkül tárgyaljuk. Egy általános mély-tanulásos réteggel ellentétben az attention nem egy vektort vár bemenetnek, hanem egy mátrixot, melynek csak az egyik dimenziója fix. Ez teszi lehetővé, hogy mondatokat fogadjon inputnak, hiszen azok hossza nem fix.

Attention kiszámolása szavanként

Az attention rétegben minden $x_i \in R^d$, $i \in [1, \ldots, I]$ szó-vektornak külön-külön módosítások jutnak. A módosítás függ a többi szótól, de egy adott szó beágyazása annak a szónak a beágyazása marad. Az attention mechanizmus hat egyszerű lépésre felbontható. Első lépésként szavanként három segédvektort definiálunk:

- query, $q \in R^{d_q}$
- key, $k \in R^{d_k}$
- value, $v \in R^{d_v}$

Ezen segédvektorok előállításához három mátrixot ($W^q \in R^{d \times d_q}$, $W^k \in R^{d \times d_k}$, $W^v \in R^{d \times d_v}$) használunk, amik a tanítható paraméterek a rétegben. $d_q = d_k$ minden esetben, de ezen belül választható paraméterek. d_v ezektől függetlenül változtatható paraméter, de a kimenetben az egyes szavak beágyazásának a dimenziója d_v lesz, így gyakran a $d_v = d$

- $q_i = x_i W^q$ $\forall i$-re
- $k_i = x_i W^k$ $\forall i$-re
- $v_i = x_i W^v$ $\forall i$-re

Az attention kiszámolásának második lépése, az ún. *score* érték. Ez azt fogja megmondani, hogy az adott szó értelmezésénél a többi szó mennyire releváns. Az i-edik szóhoz tartozó *score* vektor j-edik koordinátája megadja, hogy az i szó értelmezésekor a j szó mennyire fontos. A koordinátákat a megfelelő *query* és *key* vektorok skalárszorataként kapjuk. Magas érték azt jelenti, hogy nagy a két szó között az összefüggés.

- $score_i = (q_i \cdot k_1, q_i \cdot k_2, \ldots q_i \cdot k_I)$

A harmadik lépés egy technikai lépés, a kapott pontszámokat osszuk el a *key* vektorok dimenzójának gyökével. Ez empirikus tapasztalatok alapján kiegyensúlyozottabb gradiensekhez vezet.

A negyedik, hogy az egyes *score* vektorokra egy $softmax$ függvényt alkalmazzunk, ami egy valószínűségi eloszlást csinál belőle. Ennek az eloszlásnak a koordinátáit jelöljük $softmax_{ij}$-nek, ahol i azt a szót jelöli, amihez tartozik, j pedig az egyes koordinátákra vonatkozik. Jellemzően $softmax_{ii}$ egy nagy szám lesz.

Az ötödik lépés, hogy az összes v_j vektort megsúlyozzuk a megfelelő $softmax_{ij}$ értékkel. Így az irreleváns szavak beágyazása apró, míg a fontos szavak beágyazása nagy súllyal kerül majd számításba.

A hatodik lépés, hogy az így megsúlyozott v_j vektorokat összeadjuk. Ez lesz az x_i', az x_i szó beágyazása az attention réteg kimenetében.

A réteg teljes kimenete így az x_i' vektorok konkatenációjaként kapott mátrix lesz.

Az attention implementációja mátrixokkal

A fenti lépésekben is látszik, hogy egy adott szó számolásának i-ik lépésénél a többi szó számolásának legfeljebb $i-1$-ik lépésre van szükség.

Az első lépés a következőképp fog kinézni. $X \in R^{l \times d}$ a bemeneti mátrix, Q, K, V rendre a *query*, a *key* és a *value* vektorok mátrixai.

- $Q = XW^q$
- $K = XW^k$
- $V = XW^v$

Ezután, ha a $softmax$ függvényt mátrixokon megfelelően értelmezzük, a 2-6. lépések így írhatóak le:

$$X' = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

Az alapkoncepció ezzel kész is van.

Multi-head attention

Egy gyakran használt technika még az ún. $multi-headattention$. Az elképzelés itt az, hogy a réteg rugalmasságán javíthatunk, ha a fenti attention mechanizmust egymástól függetlenül, párhuzamosan többször (m-szer) elvégezzük saját W^q, W^k és W^v mátrixokkal. Így kapunk m különböző X' mátrixot kimenetnek. Az $m > 1$ darab mátrix nyilvánvalóan több információt képes hordozni, mint egy. Ugyanakkor ezt az információmennyiséget bele kellene sűríteni egy kisebb mátrixba, különben túl nagyok lesznek a dimenziók. Erre a standard megoldás az, hogy konkatenáljuk az m mátrixot $Z = [X_1'|\ldots|X_m']$ és veszünk egy W mátrixot, tanítható paraméterként. ZW így olyan alakú lesz, mint egy X'.

Attention gráf-algoritmusokban

Eredetileg ugyan szövegfeldolgozásra alkották meg az attention mechanizmust, de nagyon hasznosnak bizonyult a gráf-algoritmusok esetében is. Jellemzően egy gráfalgoritmusnál egy gráfhoz szeretnénk valamilyen értéket rendelni. Ehhez a gráfnak kell valamilyen beágyazását nézni egy vektortérbe.

Az euklideszi TSP problémánál teljes gráfokról van szó, így nem szükséges azzal foglalkozni, melyik él létezik és melyik nem. Továbbá az élsúlyokat is megkaphatjuk, ha a koordinátákkal reprezentáljuk a csúcsokat. Természetes módon adódik hát, hogy a gráf csúcsaihoz rendeljünk vektorokat beágyazásként, majd az attention-nel a csúcsok beágyazását finomíthatjuk a többi csúcs kontextusában.

5. Reinforcement Learning

Előfordul, hogy egy problémáról nem állnak rendelkezésünkre címkézett adatok, de valamilyen környezetben döntéseket kell hoznunk, amire érkeznek visszajelzések. Reinforcement Learning (RL), azaz a megerősítéses tanulás egy tanulási paradigma, ami az ilyen típusú információ alapján képes nagy mennyiségű címkézett adatot generálni és ezen tanulni.

A módszer leghíresebb alkalmazásai a számítógépes játékok világához kapcsolódik: a Google Deepmind csapata kifejlesztette az AlphaGo-t, amely 2015-ben egyenlő feltételek mellett legyőzte a legjobb profi játékost. 2019-ben az AlphaStar nevű program győzedelmeskedett profi játékos ellen a StarCraft nevű valós idejű stratégiai játékban, majd a legmagasabb fokozatot érte el az élő ranglistán. 2018-ban az OpenAI Five egy öt botból álló csapat győzedelmeskedett az emberi ellenfeleken a Dota 2 nevű játékban. A korábbi példákhoz képest itt meg kellett tanulniuk koordináltan dolgozni. Sakkban a legjobb játékosokat már felülmúlta az AlphaZero nevű program és versenyképes a régebbi módszerekkel készített sakkbotokkal szemben is. De hasznos a módszer tőzsdebotok esetében, matematikai bizonyítások gépi előállításában és mint jelen dolgozatból kiderül NP-nehéz problémák megoldásánál is. Specifikusan minimális összsúlyú Hamilton körök keresésénél.

5.1. Egy dinamikus környezet matematikai formalizmusa

Az ilyen modellekben a programunkat ügynöknek (agent) hívjuk, amely egy környezetben létezik. Az ügynök akciókat hajt végre, amik befolyásolják a környezet állapotát. Minden állapot valamilyen jutalmat ad az ügyöknek. A környezet leírásához szükségunk van:

- S: az állapotok halmaza
- A: az akciók halmaza
- R: a jutalom-eloszlásfüggvény ($S \times A \to \mathbb{R}$)
- P: az állapot-átmenet eloszlásfüggvény ($S \times A \to S$)

Egy ilyen környezetben az ügynökünknek tudnia kell egy π: $S \to A$ leképezést (vezérelv/policy), ami alapján adott $s \in S$ állapotban $a \in A$ lépést meglép. T trajektóriának a (s_i, a_i, r_i) hármasok sorozatát hívjuk, ezt járja be az ügynök. Ennek hossza (N) lehet előre meghatározott vagy pedig valamilyen leállási feltételből futás közben meghatározott. Egy trajektória értékének a $\sum_{i=1}^{n} \gamma^i r_i$ értéket hívjuk. Az itt szereplő $\gamma \in [0,1]$ érték az ún. kvantálási hányados. Ha $\gamma = 1$, egy trajektória összértéke a különálló jutalmak összege. Azért van rá szükség, mert ha $\gamma < 1$, akkor a

távoli jövőben kapott jutalmakat kisebb súllyal veszi figyelembe a modell, ami jellemzően az egyszerűbb megoldásra vezet.

Egy trajektóriát a következőképpen lehet generálni:

Input: S, $s_0 \in$ S kezdőállapot, A, R, P, γ, N időkeret
Output: T trajektória
  for $i \in [0,\ldots,N]$ **do**
    $a_i \leftarrow \pi(s_i)$
    $r_i \leftarrow R(s_i, a_i)$
    $T \leftarrow$(s_i, a_i, r_i)
    $s_{i+1} \leftarrow P(s_i, a_i)$
  end for
    return T

Felfedezés és kihasználás A fenti módszer mohó egy trajektória generálására, mert mindig a policy által mondott legjobb lépést tesszük meg. A jelenlegi információink alapján ez a lehető legjobb döntés, ha eredményt szeretnénk elérni, de tanulásra kevésbé alkalmas. Könnyen lehet, hogy egy eddig felfedezetlen ágon sokkal jobb eredményeket lehet elérni. Ezért fontos, hogy egy megfelelő egyensúlyt tartsunk az ismeretlen lehetőségek felfedezése és az ismereteink kihasználása, további vizsgálata között. Egy egyszerű mód erre az ún. ε-mohó algoritmus, ahol nem mindig $\pi(s_i)$ lépést lépjük, hanem ε eséllyel egy véletlenszerű másik lépést. Egy további módszer, hogy $\pi(s)$ egy eloszlást ad az s állapotban érvényes lépéseken, és mi eszerint az eloszlás szerint sorsolunk egy lépést.

A tanulás folyamán mi egy optimális π^* leképezést közelítünk:

$$\pi^* = \arg\max_{\pi} E[\sum_{i=0}^{N} \gamma^i r_i | \pi]. \tag{1}$$

5.2. Mély Q-tanulás

A tanításhoz és a környezetben való tájékozódáshoz hasznos számunkra az ún. Value és Q-Value függvények.

Value:

$$V^{\pi}(s) = E[\sum_{i=0}^{N} \gamma^i r_i | s_0 = s, \pi]$$

Q-Value:

$$Q^{\pi}(s,a) = E[\sum_{i=0}^{N} \gamma^i r_i | s_0 = s, a_0 = a, \pi]$$

Ezek leírják, hogy mennyire hasznos az ügynöknek egy adott állapotban lenni, illetve egy adott állapot adott lépését mennyire hasznos megtenni. Legyen továbbá Q^* függvény az alábbi:

$$Q^*(s,a) = \max_\pi E[\sum_{i=0}^{N} \gamma^i r_i | s_0 = s, a_0 = a, \pi]$$

Ez a függvény segít nekünk címkézett adatokat generálni, ugyanis teljesül rá a Bellmann-egyenlőség:

$$Q^*(s,a) = E_{s'\ P(s,a), r\ R(s,a)}[r + \gamma \max_{a'} Q^*(s',a') | s,a]$$

A Q^* függvény egy adott policy alapján rendeli az állapotokhoz és kezdőlépésekhez a várható értékét a policy által generált trajektóriának. A Bellmann-egyenlőség ehhez azt teszi hozzá, hogy ez pont ugyanaz, mintha vennénk a jelenlegi lépés jutalmát és a lépés eredményeképpen kapott állapot legjobb lépésének a Q^* értékét. Az az érték, hogy meglépjük az adott lépést, majd mindig a policy szerinti legjobb lépést lépjük a továbbiakban. Innen egy egyszerű módszer a tanulásra a következő iteratív algoritmus:

Input: S, A, R, P
Output: V(s)
  while V(s) nem konvergál **do**
    for all $s \in S$ **do**
      for all $a \in A$ **do**
        $Q(s,a) \leftarrow E[r|s,a] + \gamma \sum_{s'|inS} P(s'|s,a) V'(s')$
      end for
      $V(s) \leftarrow \max_a Q(s,a)$
    end for
  end while

Ez az algoritmus nagyon lassan konvergál. Be kell járnia a teljes állapotteret, ami elképesztően nagy lehet (esetenként végtelen nagy). Speciális esetekben dinamikus programozást lehet alkalmazni a Bellman egyenlet alapján, de általános esetben nem használható. Ennél hatékonyabb megoldást kell keresni és itt tudjuk felhasználni a mélytanulás módszereit. Innen kapjuk a legegyszerűbb megerősítéses tanulási módszert, a mély Q-tanulást (deep Q-learning).

A tanulás folyamán generálunk állapotokat, megnézzük mit mond rá az épp aktuális Q függvényünk, címkézéshez pedig a Bellman-egyenletet hívjuk segítségül, és így rögtön adódik a veszteségfüggvény is.

- $Q(s,a,\theta) \approx Q^*(s,a)$
- $y_i = E_{s'\ P(s,a), r\ R(s,a)}[r + \gamma \max_{a'} Q^*(s',a',\theta_i) | s,a]$
- $L_i(\theta_i) = E_{s,a}[(y_i - Q(s,a,\theta_i))^2]$

Fontos trükk, hogy tanulás közben létrehozunk egy nagy halmazt, amiben tároljuk a már kigenerált példákat. Ebből fogunk véletlenszerűen vételezni eseteket, amiken lehet felügyelt módon tanulni. Ez abban segít, hogy ne csak a közvetlen közelmúlt alapján javítsunk, hanem a régebbi emlékeket is felhasználjuk. Ez segít a gradienseket kiegyensúlyozottabbá tenni. Ez az ötlet nem feltétlen a mély Q-tanuláshoz kötődik, máshol is elsüthető.

  $D \leftarrow 0,$ $\triangleright$ N kapacitású memória

$Q(s,a) \leftarrow$ véletlenszerű súlyok
for epizód = $1 \dots M$ **do**
s_0 véletlenszerű kezdőállapot
for t= 0 ... T **do**
ε eséllyel válasszunk véletlenszerű a_t akciót, egyébként $a_t = \max_a Q^*(s_t, a, \theta)$
tegyük meg a_t lépést a környezetben, innen megkapjuk r_t jutalmat és s_{t+1} következő állapotot
$D \leftarrow D \cup (s_t, a_t, r_t, s_{t+1})$
vételezzünk egy batch (s_j, a_j, r_j, s_{j+1}) átmenetmintát D-ből
$$y_j \leftarrow \begin{cases} r_j & \text{ha } s_{t+1}\text{-ből nem indul továbblépés} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \theta) & \text{egyébként} \end{cases}$$
egy gradiens lépést teszünk $(y_j - Q(s_{j+1}, a', \theta))^2$ veszteségfüggvény szerint
end for
end for

Ilyen módon már tudjuk kezelhető módon közelíteni a Q^* függvényt, és sokszor ez is elég egy jó vezérelv megalkotásához. Sokszor viszont $Q^*(s,a)$ rendkívül összetett lehet, akár már csak abból is, hogy magas dimenziós az állapottér. Ha emellett van egy vezérelv, ami jól működik és sokkal egyszerűbb, akkor felesleges megtanulni. Ebből a gondolatból következik egy másik megközelítése a feladatnak, amikor is rögtön a vezérelvet tanuljuk.

5.3. REINFORCE

5.1. Definíció. Policy Gradiens módszerek: Policy gradiens módszernek hívjuk azokat a módszereket, amelyek rögtön a policyt tanulják meg. Vegyünk egy paraméterezett π_θ vezérelv osztályt ($\theta \in R^m$), ezen belül keressük az optimális θ^* paramétert. Felmerül a kérdés, hogyan értékelünk egy vezérelvet? Kézenfekvő módon megadhatjuk a jóságot a

$$J(\theta) = E[\sum_{i>0} \gamma^i r_i | \pi_\theta] \tag{2}$$

függvénnyel. Eszerint keressük az optimális θ^* paramétert.

A REINFORCE egy népszerű policy gradiens módszer. Azon alapul, hogy ugyan a fenti függvény nehezen számolható, de kis trükközéssel át tudjuk alakítani kezelhetőbb formába. Ezenkívül azt is kihasználjuk, hogy egy mintából számolt gradiens várható értéke ugyanaz, mint az igazi gradiens, így jól közelíthető mintavételezéssel. τ jelöljön egy trajektóriát, ekkor a fenti hasznossági-függvény a következő módon is felírható:

$$J(\Theta) = E_{\tau \ p(\tau,\Theta)}[r(\tau)] = \int_\tau r(\tau) p(\tau, \Theta) d\tau \tag{3}$$

Ha a trajektróia valamilyen módon folytonos, akkor egy integrál lesz a (2) függvényből, ha nem, akkor is értelmezhető a trajektória egy diszkrét mértékként, így a τ szerinti integrálás pont egy ilyen nagy összeg. $r(\tau)$ jelenti egy trajektória összértékét. Ennnek a gradiense a következő lenne:

$$\bigtriangledown_\Theta J(\Theta) = \int_\tau r(\tau) \bigtriangledown p(\tau, \Theta) d\tau \tag{4}$$

Ennek a pontos kiszámolásához venni kellene az összes lehetséges trajektóriát, ami életszerűtlen.

A belső függvényt a logaritmus deriváltjával tehetjük szebbé

$$\bigtriangledown_\Theta p(\tau, \Theta) = p(\tau, \Theta) \frac{\bigtriangledown_\Theta p(\tau, \Theta)}{p(\tau, \Theta)} = p(\tau, \Theta) \bigtriangledown_\Theta \log p(\tau, \Theta). \tag{5}$$

Ezután a hasznossági függvény

$$\bigtriangledown_\Theta J(\Theta) = \int_\tau r(\tau) p(\tau, \Theta) \bigtriangledown_\Theta \log p(\tau, \Theta) d\tau.$$

Vegyük észre, hogy ez a függvény pont a várható értéke az $r(t) \bigtriangledown_\Theta \log p(\tau, \Theta)$ függvénynek. Azaz

$$\bigtriangledown_\Theta J(\Theta) = E_{\tau\ p(\tau,\Theta)}[r(\tau) \bigtriangledown_\Theta \log p(\tau, \Theta)].$$

Mivel ez egy várható érték, Monte Carlo módszerrel közelíthető, azaz veszünk trajektóriákat, azokra kiszámolom a fenti függvényt és átlagolom őket.

Egy trajektória valószínűségét egyszerűen kapjuk:

$$p(\tau, \Theta) = \Pi_{i>0} p(s_{i+1}|a_i, s_i) \pi_\Theta(a_i|s_i)$$

$$\log p(\tau, \Theta) = \sum_{i>0} \log p(s_{i+1}|a_i, s_i) + \log \pi_\Theta(a_i|s_i)$$

Ebben az a szép, hogy mivel az átmeneti valószínűségek nem függenek Θ-tól és összegben vannak, ha gradienst számolunk Θ szerint, eltűnnek.

$$\bigtriangledown_\theta \log p(\tau, \Theta) = \sum_{i>0} \bigtriangledown_\Theta \log \pi_\Theta(a_i|s_i).$$

Innen adódik a hasznossági függvény közelítése egy trajektória alapján:

$$\bigtriangledown_\theta J(\Theta) \approx \sum_{i>0} r(\tau) \bigtriangledown_\Theta \log \pi_\Theta(a_i|s_i). \tag{6}$$

Ez alapján már lehetséges tanulni, de jellemzően a gradienseknek nagy lesz a varianciája, ezért lassan tanul. Apró trükkök segítségével ezen lehet javítani, de a lehetséges javítási módszereket a ma is kutatják. Ilyen trükkök között van az, hogy egy adott akció értékeléséhez csak az utána jövő jutalmakat vesszük figyelembe

$$\bigtriangledown_\theta J(\Theta) \approx \sum_{i>0} \Big(\sum_{i'>i} r_{i'} \Big) \bigtriangledown_\Theta \log \pi_\Theta(a_i|s_i) \tag{7}$$

Újra bevezethetjük a korábban már látott kvantálási hányadost:

$$\bigtriangledown_\theta J(\theta) \approx \sum_{i>0} \Big(\sum_{i'>i} \gamma^{i'-i} r_{i'} \Big) \bigtriangledown_\theta \log \pi_\theta(a_i|s_i) \tag{8}$$

Sokszor a javítás irányáról több információt kaphatunk, ha egy $b(s)$ viszonyítási alaphoz képest értékeljük az eredményeket.

$$\bigtriangledown_\theta J(\theta) \approx \sum_{i>0} \Big(\sum_{i'>i} \gamma^{i'-i} r_{i'} - b(s) \Big) \bigtriangledown_\theta \log \pi_\theta(a_i|s_i). \tag{9}$$

Ezek után összeáll egy egyszerű policy gradiens algoritmus:

for iteráció = 1,2 ... T **do**
  generáljunk trajektóriákat a jelenlegi policy alapján
  minden lépésnél, minden trajektóriánál számoljuk ki $R(t) = \sum_{i'>i} \gamma^{i'-i} r_{i'} - b(s_t)$ értéket
  frissítsük a policy-t g gradiens-becslés alapján, ami $\bigtriangledown_\theta \log \pi_\theta(a_i|s_i) R(t)$ kifejezések összege
end for

A REINFORCE algoritmus előnye, hogy segítségével képesek vagyunk direkt a policy-t közelíteni. Hátránya a lehetséges trajektóriák nagy száma, így a variancia nagy, a különböző kisebb javítások mellett is. Rengeteg adatot igényel a konvergenciához, amit ugyan elő tudunk állítani, de a folyamat erőforrásigényes.

5.4. Actor-Critic

Van tehát két módszerünk, a mély Q tanulás és a REINFORCE, különböző előnyökkel és hátrányokkal. Kézenfekvő módon gyúrjuk őket össze egybe, hátha kiegészítik egymás hiányosságait. Az Actor-Critic algoritmus onnan kapta a nevét, hogy két részből áll, az Actorból és a Criticből. Az Actor egy policy-t jelent és valamilyen direkt policy gradiens módszerrel tanul. A critic dolga pedig, hogy az megtippelje, mennyire értékes állapotokba megy az actor, azaz Q tanulást végez.

- Critic: a $V_\phi(s)$ vagy a $Q_\phi(a|s)$ függvényt tanulja algoritmustól függően
- Actor: $\pi_\theta(s)$ policy-t tanulja

Ez azért hasznos nekünk, mert a Criticnek elegendő pontosnak lennie azokon a mintákon, amiket az Actor gyakran generál. Az actornak pedig segítség a Critic, mert csökkenthetjük vele a varianciát. Nézzünk egy Actor-Critic pszeudokódot:

Inicializáljuk s_0, θ, ϕ paramétereket véletlenszerűen, $a \leftarrow \pi_\theta(s_0)$
for t= 0 ... T **do**

$r_t \leftarrow R(s_t, a_t)$, $s_{t+1} \leftarrow P(s_t, a_t)$
$a_{t+1} \leftarrow \pi_\theta(s_{t+1})$
$\theta \leftarrow \theta + \alpha_\theta Q_\phi(s,a) \bigtriangledown_\theta \log \pi_\theta(s_t|a_t)$
$\delta_t \leftarrow r_t + \gamma Q_\phi(s_{t+1}, a_{t+1}) - Q_\phi(s_t, a_t)$ ▷ Mennyit tévedett a critic?
$\phi \leftarrow \phi + \alpha_\phi \delta_t \bigtriangledown_\phi Q_\phi(s_t, a_t)$
end for

α_θ és α_ϕ itt tanulási ráták külön az Actornak és a Criticnek.

Mint minden alapvető mély tanulásos módszernek, ennek is rengeteg válfaja létezik. Például, ha a critic egy $V(s)$ függvényt tanul, ezt használhatjuk *baseline* függvénynek. Egy másik lényeges kérdés, hogy mikor és mi alapján frissítjük a súlyainkat. A fenti algoritmusban minden lépéshez a hibát a rögtön rákövetkező lépésből és az ott számott Q értékből számoltuk. Monte Carlo módszereknél végigpörgetjük a trajektóriát egy leállási feltételig és csak utána frissítjük a súlyokat. A kettő közötti átmenetek az ún. n-step return (n-lépéses kiértékelés) módszerek. Itt a trajektória első n lépését szimuláljuk és a maradékra használjuk a Q függvény becslését.

6. Eddigi TSP háló-modellek

Ebben a részben körbejárjuk, hogy milyen általános tulajdonságokat várhatunk el egy gráfalgoritmust reprezentáló neurális hálótól, majd három ígéretes és alapjaiban különböző elképzelést nézünk meg és hasonlítunk össze [Wu+19] [MGM21] [Tan+20].

6.1. Elvárások egy TSP-hálótól

Mielőtt elmerülünk a különféle algoritmusok részleteiben, vegyük végig, milyen természetes követelményeknek kell egy TSP feladatot megoldó algoritmusnak megfelelnie. Az alapkövetelmények között van, hogy egy homogén függvényt kapjunk a végén. Adott λ értékkel megszorozva a súlyokat az összes út és túra összsúlya is λ-szorosára változik, így $f_\theta(\lambda G) = \lambda f_\theta(G)$ elvárható. Hasonlóan szükséges, hogy csúcspermutációkra ne legyen érzékeny a modell, hiszen a feladatot lényegében ez nem változtatja meg. Tehát ha G és G' izomorfak, akkor $f_\theta(G) = f_\theta(G')$ alapelvárás.

Egy további fontos szempont az általánosítás. A TSP feladatnál a szűk keresztmetszet a gráfok mérete. Elvárható, hogy a modell tudjon általánosítani még nem látott, de a tanultaknál nem nagyobb gráfokra. Nehezebb cél, hogy a modell kis csúcsszámú gráfokon való tanulás után jó eredményeket produkáljon jóval nagyobb gráfokon. Ez utóbbi gyakorlati szempontból nagyon hasznos lenne, hiszen a legtöbb modell ott akad el, hogy nagyon erőforrásigényes nagy gráfokon tanulni.

6.2. Tang IterGNN-je

Hao Tang és társai 2020-ban publikált munkája [Tan+20] két nagy problémát vett a fókuszba a gráfalgoritmusok világában: az adaptív mélységű hálók lehetőségeit vizsgálták és alkottak egy háló-modell típust, amely felépítéséből adódóan homogén függvényeket tud modellezni.

Az eddigi gráfokra épített mély hálós megoldások működésében egy réteg alatt egy csúcs a szomszédjainak üzent. Ezért ilyen megközelítéssel egy fix mélységű háló egy olyan gráfokhoz kötődő problémát, amihez az egész gráfról kell valamilyen információ, nem tudhat megoldani tetszőleges méretű gráfon. Korábban is foglalkoztak már adaptív futásidejű neurális hálókkal (többek között [Gra16]), most pedig gráfalgoritmusokon szeretnénk ezt a ötletet hasznosítani. Több hagyományos gráfalgoritmus ugyanis valamilyen iteratív formulát követ. A fenti TSP heurisztikák mindegyike egy iteratív lépést ismétel, amíg valamilyen leállási feltétel nem teljesül. Ezt az elképzelést építették bele az iteratív modulba.

Iteratív Modul

Egy iteratív algoritmus fontos része a lépés, amit iterálunk. Ezt a lépést jelöljük f függvénnyel, ez az iteratív algoritmus törzse. Jelüljük g-vel azt a függvényt, ami a leállási feltételt segít majd nekünk kiszámolni. Minden lépéshez legyen s_k az aktuális állapot és $c_k = g(s_k)$ a magabiztossági pontszám. A $c_k \in [0,1]$ érték megadja, mekkora valószínűséggel szeretnénk ebben az állapotban leállni. Így annak a valószínűsége, hogy adott s_k állapototTal tér vissza az algoritmus:

$$P(s_k) = \Pi_{i=1}^{k-1}(1-c_i)c_k.$$

Ezzel a sztochasztikus leállási feltétellel az az egy probléma, hogy nem deriválható, - ami az SGD alkalmazásához szükséges lenne. Várhatóérték számolás helyett azt a megoldást találhatjuk, hogy a "folytassuk" $P(s_k) = \Pi_{i=1}^{k}(1-c_i)$ valószínűség legyen kisebb valamilyen ε küszöbértéknél és visszatérési értéknek vegyünk egy várható értéket $h = \sum_{i=1}^{k} P(s_i)c_i$.

Az algoritmus tehát a következőképpen néz ki:

Input: f lépésfüggvény, g leállási függvény, x bemenet, ε leállási érték
  $k \leftarrow 1$
  $s_0 \leftarrow x$
  while $\Pi_{i=1}^{k-1}(1-c_i) > \varepsilon$ **do**
    $s_k \leftarrow f(s_{k-1})$
    $c_k \leftarrow g(s_k)$
    $k \leftarrow k+1$
  end while
    return $\sum_{j=1}^{k}\left(\Pi_{i=1}^{j-1}(1-c_i)\right)c_j s_j$

Reprezentálja f-et és g-t egy-egy GNN, megkapjuk az ún. IterGNN modult.

Ebben a formában ez egy általános modell, ami könnyen lekövetheti a klasszikus gráfalgoritmusok szerkezetét. Az adaptív leállási feltétel segítségével az elméleti lehetőség adott, hogy különböző csúcsszámú gráfokra általánosítson a modell. Nehézséget még az okozhat, hogyan ágyazzuk bele különböző csúcsszámú gráfokat fix dimenziójú vektortérbe.

Belátható [Tan+20], hogy ez az iteratív modul általános közelítője iteratív algoritmusoknak.

Homogenitás

6.1. Definíció. Homogén függvény:

Egy vektorokon értelmezett f függvény pozitív homogén $\iff f(\lambda x) = \lambda f(x), \forall > 0$ és $\forall x$

6.2. Definíció. Homogén függvény gráfokon:

Legyen $G(V,E)$ gráf, $x_v|v$ csúcs- és $x_e|e \in E$ élattribútumokkal.

Egy gráfokon értelmezett f függvény pozitív homogén $\iff$ $\forall G$ gráfra

$$f(G,\{\lambda x_v|v\},\{\lambda x_e|e \in E\}) = \lambda f(G,\{x_v|v\},\{x_e|e \in E\}).$$

6.3. Definíció. HomoMLP és HomoGNN:

A HomoMLP olyan MLP (multi-layer perceptron), ahol minden neuronnak 0 az eltolássúly és az aktivációs függvénye homogén.

HomoGNN olyan GNN, amiben minden modul homogén függvény.

Könnyen látható, hogy homogén függvények kompozíciója is homogén. Legyen f, g homogén függvények, ekkor

$$f(g(\lambda x)) = f(\lambda g(x)) = \lambda f(g(x)).$$

Ebből következik, hogy ekkor a HomoGNN-ek és a HomoMLP-k is homogén függvények. Belátható, hogy a HomoMLP-k általános közelítői homogén függvényeknek [Tan+20] .

6.3. Mele konstruktív algoritmusa

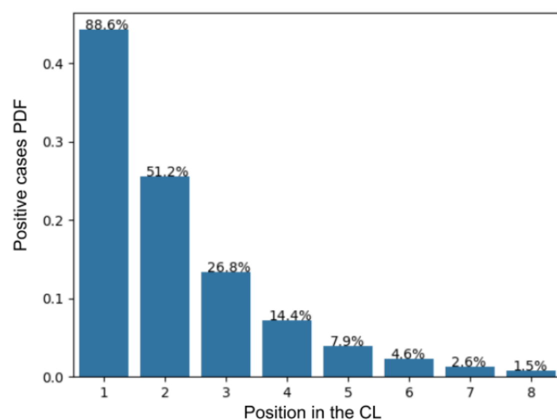
Mivel az addigi próbálkozások rendre kudarcot vallottak a kis gráfokról nagy gráfokra való általánosításnál, 2020-ban Chaitanya K. Joshi és társai előterjesztették, hogy alapjaiban kell újragondolni az általánosítás kérdéskörét [Jos+21]. Többek között innen is inspirálódva alkották Umberto Junior Mele, Luca Maria Gambardella és Roberto Montemanni 2021-ben "Egy Új Konstruktív Heurisztikát Gépi Tanulástól Vezérelve" [MGM21]. A cikkben áttekintik különböző konstruktív heurisztikák erősségeit és hátrányait, majd megkísérlik hálókkal javítani a gyengeségeket. Az alábbiakban részletezzük, ez hogyan történik. A később leírt algoritmusban a háló feladata, hogy egy adott élre megmondja, otpimális-e. Az, hogy ezt mi alapján teszi, illetve milyen sorrendben kapja az éleket, az alábbi empirikus tapasztalatok alapján dől el.

Statisztikai tanulmány

A konstruktív heurisztikák vagy a csúcsokon (beszúrásos heurisztikák) vagy az éleken (mohó, CW, MF) iterálnak végig. Fu és társai előálltak 2020-ban [FQZ20] azzal az ötlettel, amit Meleék "Candidate List (CL)"-ként emlegetnek. Az elképzelés, hogy minden csúcshoz alkotunk egy "jelölt-listát" a belőle kiinduló csúcsokból, ezeket élhossz szerint növekvő sorrendbe rendezzük és azt vizsgáljuk egyes csúcsok mekkora eséllyek lesznek benne az optimális túrában[2].

Generáltak 1000 véletlen Euklideszi gráfot, ezek csúcsszáma uniform véletlen eloszlásból jött 100 és 1000 közül, a csúcsok pedig uniform egyenleten jönnek az egységnégyzetből. Az optimális megoldást a Concorde számolta ki. Az eredményeket az alábbi ábrán láthatjuk

[2]Véletlen gráfnál annak a valószínűsége, hogy két különböző optimális megoldás létezik, elhanyagolható.



Ebből két fontos információt szűrhetünk le. Egyrészt a CL-ek második helyezettje az esetek felében lesz optimális. Másrészt a CL-ek első 5 helye az optimális élek 95%-át tartalmazza. Elegendő lehet tehát első körben azon éleket vizsgálni, amelyek valamely CL első 5 helyén szerepelnek. A másik fontos kérdés, hogy milyen sorrendben vizsgáljuk meg az éleket. Minnél több él van már bent, új élet annál kisebb valószínűséggel választunk ki, hiszen minden él új feltételelket jelenthet.

A kérdés megválaszolásához az MF és a CW heurisztikákat tesztelték a TSPlib 54 elemént, amelyek csúcsszáma 100-tól 1748-ig terjedt. Azt vizsgálták, a két módszer hányszor választott be optimális élt (true positive rate, TPR) és hányszor választott be nem-optimális élt (false positive rate, FPR). Ezek a mérőszámok félrevezetőek lehetnek, hiszen az MF jobban koncentrál a rövid élekre, így vezessük be a pozitív valószínűségi rátát $PLR = \frac{TPR}{FPR}$ (positive likelihood rate, PLR). A pontos eredmények a cikkben [MGM21] olvashatóak. Az eredmények alapján az MF többször választ optimális rövid élt mint a CW, hosszabb éleken (CL-en belül 3+ hely) fordítva. Azonban a rövid éleken a CW felülmúlja az MF-et.

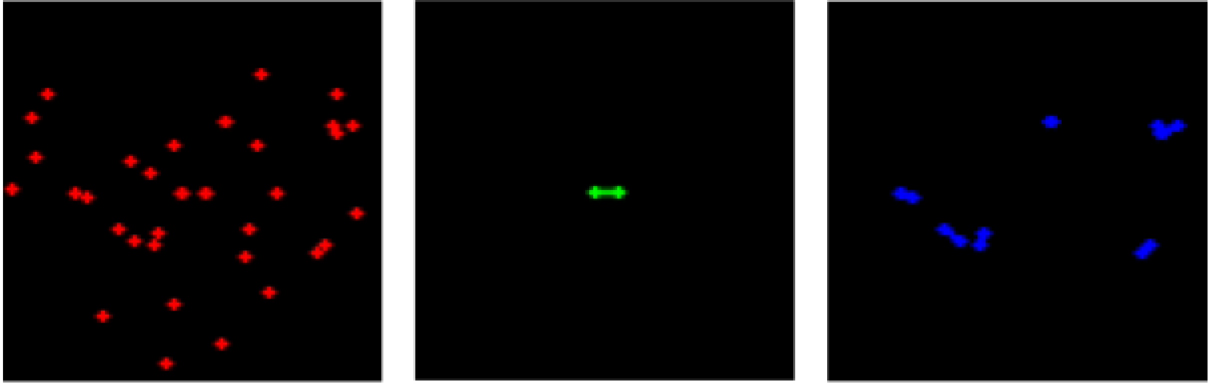
Az algoritmus

A fentiek fényében szeretnénk alkotni egy konstruktív heurisztikát. Ezt nevezzük a továbbiakban ML-Constructive-nek (ML-C). Ez két fő fázisból fog állni.

Az első fázisban minden csúcshoz létrehozzuk a CL-t, és ezek első m helyezettjét egy nagy közös L listában eltároljuk. Empririkus tapasztalatok alapján a legjobb m értéknek a 2 bizonyult. Az L listát valamilyen heurisztika szerint rendezzük, jelen esetben ez a élhossz szerint csökkenő lesz. Ezeken a listán fogunk végigmenni és ha nem hoz létre kört, alkalmazzuk rá az tanított hálónkat (továbbiakban ML-döntéshozó, később részletezzük). Ez becsli annak a valószínűségét, hogy ez az él optimális. Ha bizonyos értéknél nagyobb valószínűséget ér el, hozzáadjuk a bevett élek listájához.

A második fázisban valamilyen heurisztika szerint be kell fejezni a körutat. Több lehetséges megoldásból a CW-t választották a szerzők. Az m érték és az ML-döntéshozó elfogadási küszöbének megfelelő beállítása nem triviális feladat. Túl nagy m érték esetén sok él kerül be, és nagyobb eséllyel fogadunk el nem optimális éleket az első fázisban. Azonban ha túl kevés élet vizsgálunk az első fázisban kevés információnk lesz a második fázisban feltehetően optimális megoldásokról, és a második fázis alatt kerülhet be több nem optimális él.

Az ML-döntéshozó



Az ML-döntéshozó feladata, hogy az első fázisban vizsgálandó éleket jóváhagyja. Ehhez veszi az él két végpontjához a legközelebbi 30-30 csúcsot és ezekből alkot egy $96 \times 96 \times 3$ képet. A kép három színcsatornáján (zöld, kék, piros) a különböző információk érkeznek. A zöld mutatja a vizsgált élt és a két végpontját. A két a már beszúrt éleket az adott csúcsok között, a piros pedig a csúcsokat.

A 30-30 legközelebbi csúcs egy heurisztikus választás. Az egész gráfot nem lehet ideilleszteni, hiszen a cél, hogy tetszőlegesen nagy gráfokra általánosítsunk. Tetszőlegesen nagy gráf pedig konstans méretű képen nem fér el. A 30 csúcs úgy tűnik, megfelelően tudja a lokális kontextust ábrázolni és nem túlságosan sok.

Maga az ML-döntéshozó egy 10 réteg mély ResNet, bemenete a fent tárgyalt képformátum, kimenete két neuron. Egyike az elfogadási valószínűség, másik az elutasítási valószínűség.

Az modellben négy reziduális kapcsolás van. Minden kapcsoláson belül az első rétegben kettes "stride" értéket használunk. A kernelek a konvolúciós rétegekben 3×3-asak, cserébe minden reziduális kapcsolásnál duplázódnak a tanulható tulajdonságok. A kimenet előtt egy sűrű réteg és egy átlagolási operátor segít 2 neuronba sűríteni az információt.

A tanítás egy 38400 gráfból álló adathalmazon történt. Az egyes gráfok csúcsszáma 100 és 300 között uniform eloszlásból vételeztettek, a szerkezetüket véletlen módon generálták, egyenletes eloszlással az egységnégyzetből. A kiértékelés egy 1000 gráfból álló adathalmazon történt, hasonló módon az egységnégyzetből vételeztünk 500 és 1000 közötti pontot és 54 TSPlib példán is tesztelték az algoritmust. Az optimális megoldást minden esetben a Concorde szolgáltatta.

A tanítás érdekessége, hogy kétféle veszteségfüggvényt alkalmaztak. Az első körülbelül 1000 példán egyszerű keresztentrópia számolta a veszteséget, majda további példáknál pedig az erre a feladatra kifejlesztett "megerősítéses veszteség" (reinforcement loss) is segített frissíteni a hálózatot.

Értékelés

Mele 54 TSPlib példán vetette össze az ML-C-t az MF, és a CW heurisztikákkal és az optimális megoldással (Concorde). Vizsgálták azt is, hogy feltételezzük az ML-döntéshozó tökéletességét, ekkor milyen eredményt képes produkálni az ML-C algoritmus. Ezt jelölték ML-SC-vel. Továbbá különböző algoritmusokat is vizsgáltak, amikor is az ML-döntéshozó helyett valamilyen heurisztikát használtak:

- Y: minden L-beli elemet elfogad, ami nem hoz létre kört.

Százalékos hiba összahasonlítás a konstruktív heurisztikák esetén	
Heurisztika	Átlagos hiba
MF	17.906
CW	9.431
Y	11.345
AE	12.082
BE	8.815
ML-C	8.035
ML-SC	4.374

- E: a CL-eloszlás szerint sztochasztikusan fogadunk el egy élet. AE 20 futás alapján az átlagteljesítmény, BE 20 futásból a legjobb teljesítmény.

A méréseik megmutatták, hogy az ML-C algoritmus pontossága nem romlott lineárisan a csúcsszámok növekedésével. A módszer erőssége, hogy képes általánosítani nagy gráfokra. Tanulni 100-tól 300-ig terjedtek a tanító halmazban lévő gráfok és elfogadhatóan teljesített 1700 csúcsnál is nagyobb gráfon is. Ugyanakkor az is igaz, hogy nem sok ilyen nagy gráf van, amelyre kiszámolták az optimumot, így nem feltétlen szabad ebből a tesztelésből messzemenő következtetéseket levonni.

6.4. Wu transformer modellje

Wu és társai megközelítésében a javító algoritmusok segítségével lehet megragadni az Euklideszi TSP feladatot [Wu+19]. A 2-opt lépések segítségével mindig el lehet jutni az optimumba, a nehézséget az okozza, hogy nem egyértelmű, adott állapotban melyik 2-opt lépést kellene meglépni. A megoldás egy olyan policy, amely előre megjósolja a hosszútávú lehetőségeket és ennek megfelelően lép. A probléma ilyen megfogalmazása természetes módon vezet a megerősítéses tanuláshoz.

Az itt prezentált megoldásra később Attention2-opt (A2-opt) néven fogok hivatkozni. A problémát a következő Markov döntési folyamattal tudjuk leírni:

Állapotok: Adott $G(V,E)$ gráf és egy $(v_1,\dots,v_n)$ permutáció a csúcsokon. A permutáció jelenti a csúcsok bejárási sorrendjét. Az s állapot tehát egy ilyen permutációt jelöl.

Akciók: Két csúcs kiválasztása meghatároz egy 2-opt lépést. Tehát egy akció minden esetben $a=(v_i,v_j)$ alakban áll elő.

Átmenetfüggvény: Az átmenet adott s_t állapotból a_t akicóból determinisztikusan történik s_{t+1} állapotba a következő módon:

$$\begin{aligned} s_t &= (v_1,\dots v_n) \\ a_t &= (v_i,v_j) \\ s_{t+1} &= (v_1,v_2,\dots v_i,v_j,v_{j-1},\dots,v_{i+1},v_j,v_{j+1},\dots v_n) \end{aligned}$$

Jutalom: Mivel az optimum elérését nem tudjuk biztosítani, a cél a kezdeti állapot

összhosszának minnél jelentősebb rövidítése. Legyen $f(s)$ az s állapotban a körtúra összhossza, s^* az eddigi futás során talált legjobb megoldás. Kezdetben $s_0^* = s_0$, majd

$$r_t = r(s_t, a_t, s_{+1} = f(s_t^*) = f(s_t^*) - min(f(s_t^*), f(s_{t+1})) \tag{10}$$

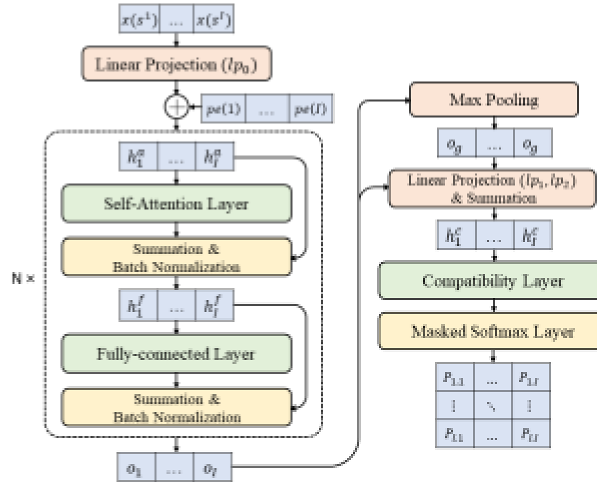
Ilyen definícióval a jutalomfüggvény akkor pozitív, ha az eddiginél jobb megoldást találtunk. Emellett ezzel a jutalomfüggvénnyel nem büntetjük a lokális rontásokat, ami kulcsfontosságú a lokális minimumokba ragadás elkerülése végett. A cél egy trajektória halmozódó jutalmának ($R_T = \sum_{i=1}^{n} \gamma^i r_i$) maximalizálása. A jutalomfüggvény $\gamma = 1$ esetén a javítás a kezdőállapothoz képest.

Az algoritmus a végén egy sztochasztikus vezérelv segítségével volt kiválasztani minden állapotban egy megfelelő akciót. Így a π stratégiánk egy valószínűségi eloszlást fog adni a lehetséges lépéseken. A folyamat kezdetén veszünk valamilyen módon egy s_0 kezdőállapotot. Innen pedig T alkalommal kiválasztunk egy a_t lépést a stratégiánk szerint, ami s_{t+1} állapothoz vezet. A folyamatot következő valószínűségi láncszabállyal tudjuk leírni:

$$P(s_T|s_0) = \prod_{i=0}^{T} \pi(a_t|s_t). \tag{11}$$

Vegyük észre, hogy nem definiáltunk leállási állapotot, mivel nincs garanciánk arra, hogy adott állapotban már megtaláltuk az optimális megoldást. Így T az algoritmus paramétere, amit rugalmasan állíthatunk szükség szerint. Tanulás folyamán kisebb T-k lehetnek hasznosak, míg inferálás során a jobb eredmény érdekében nagyobb értékeket érdemes neki adni.

Az architektúra



Az elméleti tökéletes stratégiát nem ismerjük, de közelíthetjük egy paraméteres π_θ neurális hálóval, ahol θ a tanítható paraméterek.

Adatvektorizálás: Euklideszi TSP problémákat szeretnénk megoldani, így elegendő pontokat venni a síkon. Ezeket a koordinátáikkal reprezentálhatjuk, az élek súlyait pedig könnyen számolhatjuk. Tehát kezdetben minden $v \in V$-hez tartozik egy $x_v = (x_1, x_2)$ beágyazás ($|V| = I$). A háló bemenete így egy $X \in R^{2 \times I}$ mátrix, ami $s = (v_1, \ldots, v_I)$.

Gráfbeágyazás: Érdemi információ megtanulásához szükségünk van magasabb dimenziószámra és fontos elkódolni a permutáció sorrendiségét. Az első feladatra egy közös $lp_0 : R^2 \to R^d$ lináris leképezést használunk ($d = 128$), amely minden x_i csúcshoz egy h_i beágyazást rendel.

A második feladatra a Transformer architektúráknál kifejlesztett "helyzeti elkódolás" (positional encoding) adja a megoldást. Jelen esetben a $pe(i,d)$ szinuszoid helyzeti elkódolást használjuk, amely szinusz és a koszinusz függvények segítségével állít elő vektorokat, amiket hozzáadva az elkódolt h_i vektorainkhoz segítenek megőrizni a sorrendiséget.

$$pe(i,d) = sin(i/10000^{\frac{\lfloor d/2 \rfloor}{d_h}}), \text{if } d \ mod \ 2 = 0 \tag{12}$$

$$pe(i,d) = cos(i/10000^{\frac{\lfloor d/2 \rfloor}{d_h}}), \text{if } d \ mod \ 2 = 1 \tag{13}$$

Így minden csúcshoz a reprezentáló vektort $h_i = h_i + pe(i,d)$ alakban kapjuk meg.

Attention elkódolás Ezután megkezdődik az Attention mechanizmus felhasználása. Három elkódoló blokkot használunk a következőből:

- Adott $(h_1^a, \dots h_I^a)$
- Self-Attention réteg
- Reziduális összegzés és batch normalizálás
- Adott $(h_1^f, \dots h_I^f)$
- Sűrű réteg
- Reziduális összegzés és batch normalizálás

A self-attention réteg az egy-fejes self-attentiont jelenti itt. Adott $H^a = [h_1^a, \dots h_I^a]$ bemeneti mátrix esetén, ahol az oszlopok a csúcsleíró-vektorok, a self-attentiont a korábban tárgyalt módon számoljuk:

$$H'^a = V_a \cdot softmax_c(\frac{K_a^T Q_a}{\sqrt{d_k}}), \tag{14}$$

ahol is $Q_a = W^q H^a$ a query, $K_a = W^k H^a$ a kulcs és az $V_a = W^v H^a$ érték mátrixok. $W^q \in R^{d_q \times d_m}$, $W^k \in R^{d_k \times d_m}$ és $W^v \in R^{d_v \times d_m}$ tanítható paraméterek. Jelen esetben $d_q = d_k = d_v = 128$.

Mind az attention, mind a sűrű réteg megtartja a beágyazások $128 \times I$ dimenzióját.

A csúcspár kiválasztása Az előző blokkok végén jelölje az aktuális csúcsbeágyazást o_i. Ebből egy max-pooling segítségével megkaphatunk egy gráf-beágyazást. Azaz $o_g = \max(\{o_1, \dots o_I\})$, azaz csúcsonként a maximális beágyazás értéke. Ezek után $lp_1 : R^{128 \times I} \to R^{128 \times I}$ és $lp_2 : R^{1 \times I} \to R^{128 \times I}$ tanítható lineáris leképezések segítségével

	TSP20			TSP50			TSP100		
Módszer	Érték	Gap	Futásidő	Érték	Gap	Futásidő	Érték	Gap	Futásidő
Concorde	3.83	0.00%	5m	5.69	0.00%	13m	7.76	0.00%	1h
LKH3	3.83	0.00%	42s	5.69	0.00%	6m	7.76	0.00%	25m
OR-Tools	3.86	0.94%	1m	5.85	2.87%	5m	8.06	3.86%	23m
AM (N=1280)	3.83	0.06%	14m	5.72	0.48%	47m	7.94	2.32%	1.5h
AM (N= 5000)	3.83	0.04%	47m	5.72	0.47%	2h	7.93	2.18%	5.5h
A2-opt (T=1000)	3.83	0.03%	12m	5.74	0.83%	16m	8.01	3.24%	25m
A2-opt (T=3000)	3.83	0.00%	39m	5.71	0.34%	45m	7.91	1.85%	1.5h
A2-opt (T=5000)	3.83	0.00%	1h	5.70	0.20%	1.5h	7.87	1.42%	2h

megkapjuk $h_c^i = lp_1(o_i) + lp_2(o_g)$ beágyazást. Ennek az a célja, hogy a gráf egészéről még információkat sűrítsen a csúcsokba. Ezután áteresztjük a adatainkat egy ún. compatibility rétegen, amit ezen a cikken kívül máshol nem használtak még - az attention mechanizmus egy módosított verziójáról van szó. Ennek a kimenete $Y \in R^{I \times I}$ és a mátrix minden eleme egy csúcspárnak feleltethető meg. Ezek után maszkolunk (a főátlóbeli elemek nem érvényes lépések és azt a lépést sem szertnénk választani, ami visszavinne az előző állapotba). A többi elem szórását egy $C \cdot tanh(Y_{ij})$ függvény segítségével kontrolláljuk (jelen esetben $C = 10$), majd pedig egy $softmax$ függvény segítségével egy P valószínűségi eloszlássá alakítjuk.

Az algoritmus tanítása és tesztelése alatt nem mohó módon a vesszük P maximális elemét, hanem a különböző (v_i, v_j) párokból P eloszlás szerint sorsolunk.

Tanítás

A modellt egy Actor-Critic típusú rendszerként tanítjuk. A fenti architektúra lesz az Actor. A Criticet hasonló módon készítjük el, két fontos különbséggel. 1) Max-pooling helyett átlagolunk a gráf-beágyazás legenerálásakor. 2) Az egész rendszer végén egy sűrű réteget alkalmazunk, aminek a kimenete egy szám. Fontos továbbá, hogy n-lépéses kiértékelést használunk.

Tanító adatokat menet közben generálunk, egyenletes eloszlással veszünk véletlen pontokat az egységnégyzetből. A pontos foyamat az Actor-Critic n-lépéses kiértékeléssel algoritmus szerint történik.

Értékelés

Wu és társai az algoritmust 10240 példán tanították, melyek az egységnégyzetben vett véletlen pontokból generált gráfok voltak. 4-lépéses visszatérést alkalmaztak, a γ kvantálási hányados 0.99 értékre lett állítva. Tanítás alatt $T = 200$ trajektóriahosszal dolgoztak, tesztelésnél jóval nagyobbal.

Az eredmények értékeléséhez összehasonlítási alapul több szoftvert is felhasználtak: 1) Concorde 2) LKH3 3) OR-Tools és egy mélytanuláson alapuló megoldóprogramot 4) AM [KHW18]. A korrekt összehasonlítás kedvéért AM mintavételezési N paraméterére az általuk használt 1280as érték mellett az az 5000-es értékkel is teszteltek, ami az A2-opt modellünk maximális lépésszáma volt.

Megállapítható, hogy viszonylag kis méretű gráfokon a hibaszázalék pár százalék és felülmúlja a másik mélytanulás alapú módszert. Két egyszerű 2-opt heurisztikával is összehasonlították. Az egyik az első megtalált javítást lépi meg, a másik a lehetséges

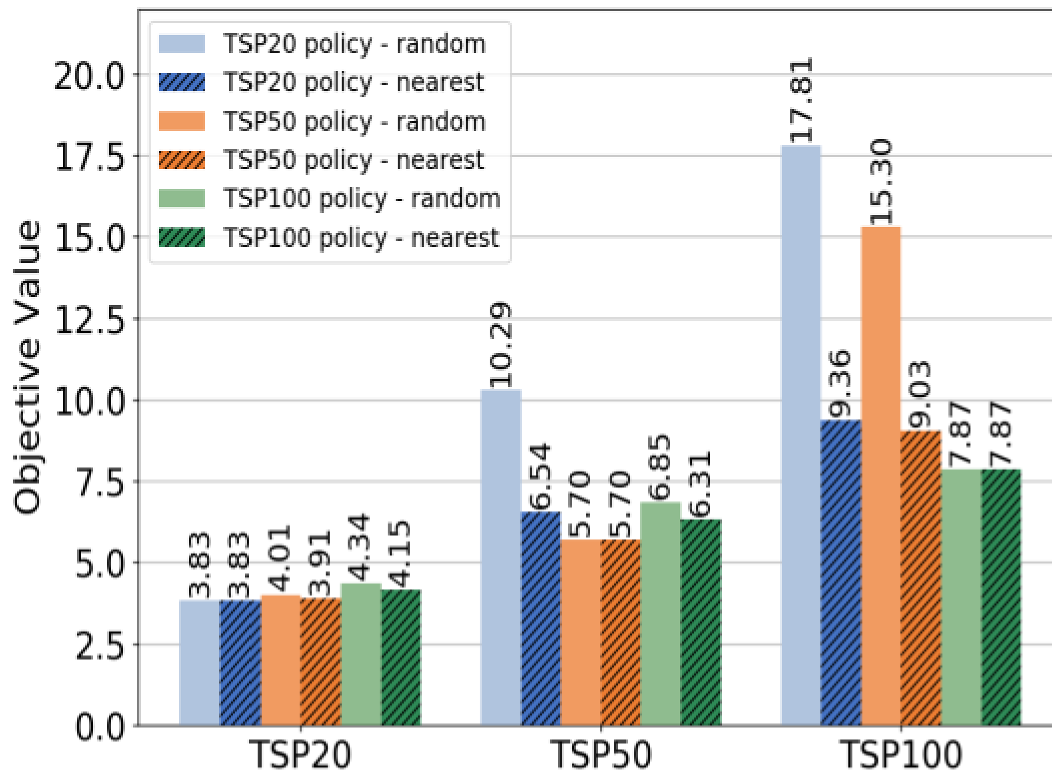
Algorithm 1 Actor-Critic n-lépéses kiértékeléssel

Input: π_θ actor háló, tanítható θ paraméterekkel, v_ϕ critic háló, tanítható ϕ paraméterekkel, E epochszám, B batchméret, T trajektóriahossz

for $e = 1, 2, \ldots, E$ **do**
  generáljunk M feladatot véletlenszerűen
  for b = 1, 2, . . . , B **do**
    alkossuk meg M_b batchet, $t \leftarrow 0$
    while t<T **do**
      $d\theta \leftarrow 0$, $d\phi \leftarrow 0$ ▷ nullázzuk a gradienseket
      $t_s \leftarrow t$; s_t az aktuális állapot
      while $t - t_s < n$ **do** ▷ n-szer léptetjük a rendszert
        $a_t \leftarrow \pi_\theta(s_t)$ ▷ és megjegyezzük a történteket
        $r_t \leftarrow R(a_t, s_t)$
        $s_{t+1} P(a_t, s_t)$
        $t \leftarrow t + 1$
      end while
      $J \leftarrow v_\phi(s_t)$
      for i $\in \{t-1, , \ldots, t_s\}$ **do**
        $J \leftarrow r_i + \gamma J$
        $\delta \leftarrow J - v_\phi(s_i)$
        $d\theta \leftarrow d\theta + \sum_{M\,b} \delta \bigtriangledown \log \pi_\theta(a_i|s_i)$
        $d\phi \leftarrow d\phi + \delta \bigtriangledown v_\phi(s_o)$
      end for
      frissítsük θ-t $\frac{d\theta}{|M_b|(t-t_s)}$-el
      frissítsük ϕ-t $\frac{d\phi}{|M_b|(t-t_s)}$-el
    end while
  end for
end for

legjobbat. Ezek az algoritmusok egy olyan kiegészítést kaptak, ha megérkeztek egy lokális maximumba egy véletlenszerű állapotból újrakezdik - a végén a legjobb megtalált túrával térnek vissza. Az A2-opt minden esetben jobb eredményt ért el ezeknél.

A modellt véletlenül generált adatokon tanítjuk, hasonló méretű, de addig nem látott példákra a modell jól általánosít. Az első nehézséget akkor tapasztalhatjuk, amikor kisebb méretű adatokon tanítva nagy gráfokon tesztelünk. Ekkor véletlenszerű kezdőállapotból indítva gyenge eredményt produkál, de ezen sokat lehet javítani, ha kontruktív heurisztika (például mohó) által generált megoldásból indítjuk. Kérdéses, hogy ebben az esetben mennyit javított A2-opt az eredeti megoldáson.



A **TSPlib** könyvtárban legyfeljebb 300 csúcsú szimmetrikus Euklideszi példákból 36 darab van, ezen mérték A2-opt, AM és az OR-Tools teljesítményét (a Concordehoz viszonyítva). A mélytanulás hátránya, hogy a tanulás alatt látott példáktól nagyon különböző eseteken általában rosszul teljesít, ezért is meglepő, hogy az OR-Toolt több esetben is felülmúlta az A2-opt. AM-et az esetek döntő többségében felülmúlta A2-opt. Az átlagos optimalitási rés A2-opt esetében 17.12% lett, AM esetében 133.54% (N = 5000 esetén).

7. Mérések

Természetesen jómagam is reprodukálni szerettem volna legalább részben a fenti eredményeket - főleg Wu és társai munkáját. Yi-Ning Ma 2020-ban közzétette ennek egy implementációját [Ma]. Ebben a repositoryban található egy előre tanított modell, ami egy epochban 10, darabonként 5120 példát tartalmazó batchen tanult 100 epochon át. Ez több mint 5 000 000 példa. Ezek a példák 20 csúcsú gráfok, melyek az egységnégyzetből egyenletes

eloszlással jönnek.

A kiértékelés feladattípus minden esetben fix 10000 példán történik, melyeket 10 batchbe osztunk, 1000 hosszú trajektóriákkal. Ezeket a pédákat ugyanabból az eloszlásból generáltam, mint a tanító adatokat, de az algoritmus még sosem látta őket.

Generáltam saját teszthalmazokat, egyenletes eloszlású 10, 20, 25, 30 csúcs gráfokat és két speciálisat: TSP20-kör 20 csúcs gráfokat tartalmaz melyek egy körvonalról érkeznek egyenletes eloszlással, TSP20-negyed pedig egy 0.5 oldalú négyzetből adja egyenletesen a 20 pontot. Mindegyik halmaz 10000 példát tartalmaz, ezeken kiértékeltem az előre tanított modellt.

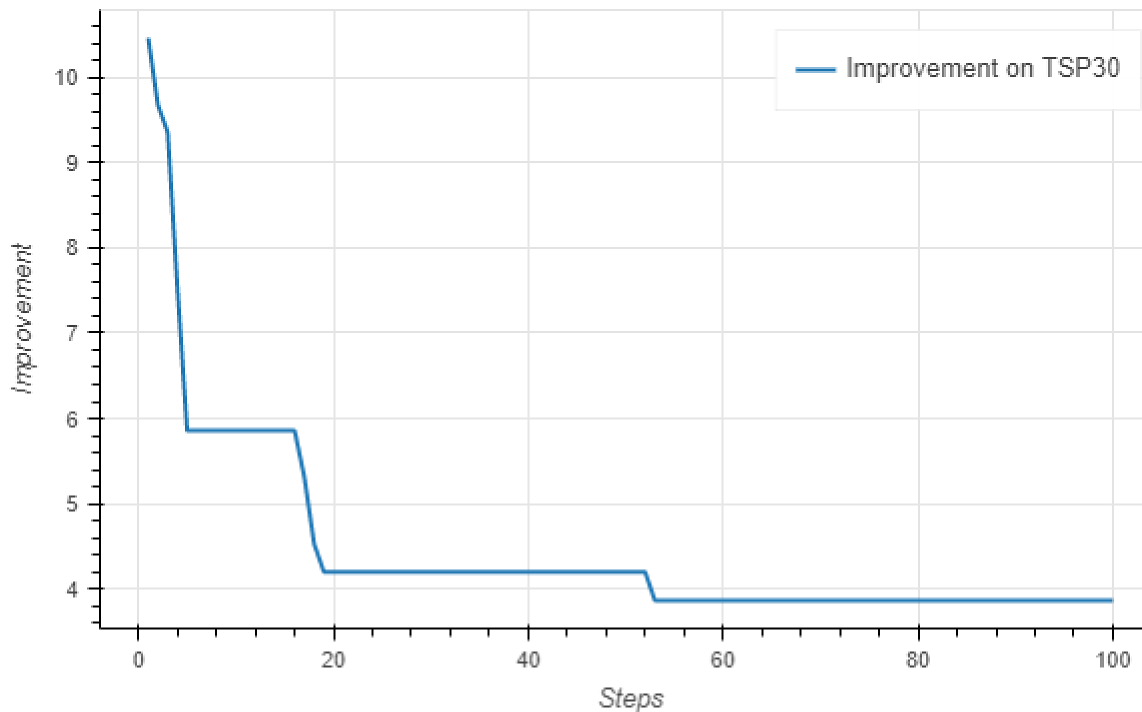
	Kezdeti átlaghossz	1000 lépés utáni átlaghossz	Javulás
TSP10	5.444400	2.922996	2.521403
TSP20	10.426547	3.846777	6.579770
TSP25	13.414846	4.670010	8.744838
TSP30	15.845465	6.492388	9.353079
TSP20_kör	12.812243	3.078409	9.733834
TSP20_negyed	5.172065	1.938554	3.233511

Ezekután a tanulási ráta manuális állítgatásával megvizsgáltam, lehet-e a modellt finomhangolni. Ennek érdekében 5 epochon át futtattam a TSP10, TSP20, TSP20-kör és a TSP20-negyed adathalmazokon. A legjobb tanulási továbbtanulásra a 10^{-6} lett, két nagyságrenddel kisebb, mint az eddig használt.

	Kezdeti adatok			Finomhangolás után		
	Kezdeti átlaghossz	Legjobb eredmény	Javulás	Kezdeti átlaghossz	Legjobb eredmény	Javulás
TSP10	5.444400	2.922996	2.521403	5.215194	2.871809	2.343385
TSP20	10.426547	3.846777	6.579770	10.426547	3.836845	6.589701
TSP20-kör	12.812243	3.078409	9.733834	12.812243	3.075942	9.733834
TSP20-negyed	5.172065	1.938554	3.233511	5.172065	1.931667	3.240397

Látható, hogy speciális adathalmazon való továbbtanítással közel egy század eredményt sikerült javítani a legtöbb eredményen.

Az alábbi ábrán látható az eredeti modell teljesítménye 100 lépés alatt egy 30 csúcs gráfon.



Hivatkozások

[CW64] Geoff Clarke és John W Wright. “Scheduling of vehicles from a central depot to a number of delivery points”. *Operations research* 12.4 (1964), 568–581. old.

[LK73] Shen Lin és Brian W Kernighan. “An effective heuristic algorithm for the traveling-salesman problem”. *Operations research* 21.2 (1973), 498–516. old.

[Pap77] Christos H. Papadimitriou. “The Euclidean Traveling Salesman Problem is NP-Complete”. *Theor. Comput. Sci.* 4 (1977), 237–244. old.

[Aro98] Sanjeev Arora. “Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems”. *Journal of the ACM (JACM)* 45.5 (1998), 753–782. old.

[MSM10] Rajesh Matai, Surya Singh és M.L. Mittal. “Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches”. 2010. nov. ISBN: 978-953-307-426-9. DOI: 10.5772/12909.

[BH15] Judith Brecklinghaus és Stefan Hougardy. “The approximation ratio of the greedy algorithm for the metric traveling salesman problem”. *Operations Research Letters* 43.3 (2015), 259–261. old.

[Gra16] Alex Graves. “Adaptive Computation Time for Recurrent Neural Networks”. *CoRR* abs/1603.08983 (2016). arXiv: 1603.08983. URL: http://arxiv.org/abs/1603.08983.

[Vas+17] Ashish Vaswani és tsai. “Attention is all you need”. *Advances in neural information processing systems* 30 (2017).

[KHW18] Wouter Kool, Herke van Hoof és Max Welling. *Attention, Learn to Solve Routing Problems!* 2018. DOI: 10.48550/ARXIV.1803.08475. URL: https://arxiv.org/abs/1803.08475.

[Wu+19] Yaoxin Wu és tsai. "Learning Improvement Heuristics for Solving the Travelling Salesman Problem". *CoRR* abs/1912.05784 (2019). arXiv: 1912.05784. URL: http://arxiv.org/abs/1912.05784.

[FQZ20] Zhang-Hua Fu, Kai-Bin Qiu és Hongyuan Zha. "Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances". *CoRR* abs/2012.10658 (2020). arXiv: 2012.10658. URL: https://arxiv.org/abs/2012.10658.

[KKG20] Anna R. Karlin, Nathan Klein és Shayan Oveis Gharan. "A (Slightly) Improved Approximation Algorithm for Metric TSP". *CoRR* abs/2007.01409 (2020). arXiv: 2007.01409. URL: https://arxiv.org/abs/2007.01409.

[Tan+20] Hao Tang és tsai. *Towards Scale-Invariant Graph-related Problem Solving by Iterative Homogeneous Graph Neural Networks*. 2020. DOI: 10.48550/ARXIV.2010.13547. URL: https://arxiv.org/abs/2010.13547.

[Cap+21] Quentin Cappart és tsai. *Combinatorial optimization and reasoning with graph neural networks*. 2021. DOI: 10.48550/ARXIV.2102.09544. URL: https://arxiv.org/abs/2102.09544.

[Jos+21] Chaitanya K. Joshi és tsai. "Learning TSP Requires Rethinking Generalization". en. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPICS.CP.2021.33. URL: https://drops.dagstuhl.de/opus/volltexte/2021/15324/.

[MGM21] Umberto Junior Mele, Luca Maria Gambardella és Roberto Montemanni. "A New Constructive Heuristic driven by Machine Learning for the Traveling Salesman Problem". *CoRR* abs/2108.10224 (2021). arXiv: 2108.10224. URL: https://arxiv.org/abs/2108.10224.

[Ben] Harsányi Benedek. *Gráf konvolúciós hálózatok és alkalmazásaik*. URL: https://web.cs.elte.hu/blobs/diplomamunkak/bsc_mat/2021/harsanyi_benedek.pdf.

[Ma] Yi-Ning Ma. *Yining043/TSP-improve: An improvement-based deep reinforcement learning algorithm presented in paper https://arxiv.org/abs/1912.05784v2 for solving the TSP problem.* URL: https://github.com/yining043/TSP-improve.

[Zol] Kulcsár Zoltán. *Az utazó ügynök probléma és alkalmazásai*. URL: https://web.cs.elte.hu/blobs/diplomamunkak/bsc_matelem/2017/kulcsar_zoltan.pdf.