

# **TDK-dolgozat**

Kovács Gergely  
Zsolt

Teleki Sándor

Szalay Gergő

# Gépi tanulási architektúra permutálás feladatának megoldására

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSELMÉLET ÉS SZOFTVERTECHNOLÓGIAI TANSZÉK



*Szerzők:*

Kovács Gergely  
Zsolt  
programtervező  
informatikus MSc.  
2. évfolyam

Teleki Sándor  
programtervező  
informatikus MSc.  
1. évfolyam

Szalay Gergő  
programtervező  
informatikus MSc.  
2. évfolyam

*Témavezetők:*

Pintér Balázs  
egyetemi adjunktus

Gregorics Tibor  
tanszékvezető egyetemi docens

Budapest, 2024. május 12.

# Tartalomjegyzék

|                                                                                                                  |           |
|------------------------------------------------------------------------------------------------------------------|-----------|
| <b>1. Bevezetés</b>                                                                                              | <b>1</b>  |
| 1.1. Kontribúcióink . . . . .                                                                                    | 2         |
| 1.2. Gépi tanulás, mélytanulás . . . . .                                                                         | 2         |
| 1.3. Kapcsolódó munkák . . . . .                                                                                 | 8         |
| <b>2. Módszerek</b>                                                                                              | <b>10</b> |
| 2.1. Architektúrák . . . . .                                                                                     | 10        |
| 2.2. Architektúra közvetlen permutálási feladatra . . . . .                                                      | 10        |
| 2.3. Attention mechanizmus a közvetlen permutálási architektúrához . . .                                         | 12        |
| 2.4. A közvetett permutálási feladat emberi megközelítése . . . . .                                              | 12        |
| 2.5. Architektúra a közvetett permutálási feladatra . . . . .                                                    | 13        |
| 2.6. Attention mechanizmus a közvetett permutációs architektúrához . . .                                         | 15        |
| 2.7. A létrehozott <i>attention</i> összeillesztése a másoló mechanizmus <i>attention</i> -<br>ionével . . . . . | 17        |
| <b>3. Kiértékelés</b>                                                                                            | <b>19</b> |
| 3.1. Mérőszámok . . . . .                                                                                        | 19        |
| 3.2. Eredmények . . . . .                                                                                        | 19        |
| 3.2.1. Közvetlen permutációs architektúra eredményei . . . . .                                                   | 19        |
| 3.2.2. Közvetett permutációs architektúra eredményei szintetikus<br>példákon . . . . .                           | 20        |
| 3.2.3. Közvetett permutálási architektúra eredménye valós példán . .                                             | 21        |
| <b>4. Diskusszió</b>                                                                                             | <b>23</b> |
| <b>5. Összefoglalás</b>                                                                                          | <b>24</b> |
| <b>Irodalomjegyzék</b>                                                                                           | <b>26</b> |

## Kivonat

Permutálásról akkor beszélünk, amikor egy sorozat elemeit egy adott (bijektív) leképezéssel újrendezzük. A leképezés lehet például a sorozat sorba rendezése, véletlenszerű megkeverése vagy egy, az értékektől független, mégis a bemenetben szereplő logikából adódó átrendezés.

Munkánk során a bemenetben szereplő második sorozat sorrendje alapján való átrendezésre kerestünk és adtunk megoldást a gépi tanulás eszközeivel. Kutatásunk eredménye két olyan architektúra, mely képes a sorrendiséget felismerni és alkalmazni a szintén a bemenetben megadott sorozatra csupán pozicionális információkra támaszkodva. A megalkotott két architektúra a hagyományos rekurrens szekvencia-szekvencia architektúrák kibővítése. Architektúránkhoz különböző benchmarkokat hoztunk létre, melyeken megmutattuk, hogy az általunk definiált összes feladatot képes kezelni a megfelelő háló. Ezzel szemben az általános, elterjedt architektúrák (*RNN*, *RNN+Attention*, *Copying Mechanism*, *Transformer*) nem, vagy csak korlátozottan képesek ezen feladatok megoldására. Módszerünk általános megoldást ad több különböző feladatra is, amelyeket az általunk létrehozott benchmarkokon mérünk ki.

# 1. fejezet

## Bevezetés

A gépi tanulás módszerei széleskörben elfogadottak az informatikatudományban és a hozzá kapcsolódó multidiszciplináris területeken. A gépi tanulás, és azon belül is a mélytanulás több ismeretlen vagy esetlegesen lassú algoritmust helyettesít a mindennapi életben. Ennek ellenére a mélytanulásnak is vannak hiányosságai, melyek sokszor az alkalmazott architektúrák jellegéből fakadnak. Ennek kiküszöbölésére különböző javításokkal egészítik ki az adott architektúrát, ezáltal újabb feladatokra történő alkalmazásokat tesznek lehetővé. Ilyen kiegészítésnek tekinthetjük például az Attention-t [1] vagy a Másoló mechanizmust [2].

Egy szintén hasznos képesség, ami gyakorlati kutatásaink [3] során nélkülözhetetlennek bizonyult, az a permutálás képessége. Neurális kódvisszafejtés során a hosszú switch utasítások esetén létrejövő ugrótáblák (*jump table*) esetén egy az ugrótábla elején lévő elágazási fa eseteit kell a hozzájuk tartozó címkék címeinek sorrendjében visszaadni. Ennek a képességnek a megtanulása viszont továbbra is nyitott feladat.

Magát a permutálás feladatát többféleképpen is értelmezhetjük.

Az első értelmezés szerint adott egy adatsorozat és egy ezzel megegyező hosszú indexsorozat. A modell feladata megtanulni egy olyan kimenet előállítását, melyben az adatsorozat elemei az indexek által meghatározott sorrend szerint helyezkednek el. Ebből kifolyólag a modellnek az indexekhez többletjelentést kell társítania, mely jelzi, hogy az adott index a bemeneti adatsorozat hányadik elemére hivatkozik. Ezen többletjelentés szükségessége miatt viszont a használt indexek a tanítás után nem módosíthatóak és nem bővíthetőek, hiszen a modellnek az új indexek esetén nem lesz többletinformációja azok jelentéséről. Ez a megkötés egy limitációt von a feladat általánosíthatóságára vonatkozóan. A továbbiakban ezt a feladatot **közvetlen**

**permutálási feladat**nak fogjuk nevezni. Egy példa ilyen jellegű feladatra az alábbiakban látható:

a c b , 3 2 1  $\rightarrow$  b c a

A második értelmezés szerint adott három megegyező hosszú sorozat: az első az adatokat tartalmazza, míg a második az adatokhoz tartozó kulcsokat, a harmadik pedig ezen kulcsok egy permutációját. A modell feladata megtanulni az adatokat a kulcsok alapján permutálni. Ebben az esetben a kulcsok tetszőlegesen bővíthetők amennyiben a modell képes felismerni a három sorozatot, és a kulcsok különböző előfordulásait összekötni. A továbbiakban ezt a feladatot **közvetett permutálási feladat**nak fogjuk nevezni. Az alábbiakban egy ilyen példa látható:

a b c , 1 2 3 , 3 1 2  $\rightarrow$  c a b

Mindkét feladat esetén elképzelhető, hogy a sorozatok nincsenek egyértelműen elválasztva egymástól, esetlegesen az első két sorozathoz tartozó elemek párokban helyezkednek el, vagy más szabály alapján vannak elrendezve az adatok.

Munkánk során mindkét feladat megoldását tűztük ki célul, utóbbi esetén az új kulcsokra való általánosítás képességét fenntartva.

## 1.1. Kontribúcióink

Az alábbiakban ismertetjük a kontribúcióinkat:

- Egy olyan architektúra létrehozása, mely képes a közvetlen permutálási feladatot megtanulni és megoldani.
- A közvetett permutálási feladatot megoldó architektúra létrehozása, ennek általánosítási képességeinek kimérése és az eredmények összevetése az elterjedt architektúrákkal.
- Benchmarkok létrehozása a közvetett permutálási feladathoz.

## 1.2. Gépi tanulás, mélytanulás

Napjainkban a **mesterséges intelligencia** legközismertebb és leginkább széles körben használt területe a **gépi tanulás**, azon belül is a **mélytanulás**, ami mesterséges neuronhálókkal foglalkozik. A legatomibb építőeleme egy ilyen hálónak a

neuron, mely egy adott  $x$  bemenet esetén azt egy  $w$  súllyal megszorozza, majd megnöveli egy  $b$  értékkel:  $w \cdot x + b$ . Ezen  $w$  és  $b$  értékek a neuron paraméterei (a  $w$  a súly vagy *weight*, a  $b$  az eltolás vagy *bias*), melyeket a tanítás során változtat, tanul a rendszer.

A legtöbb esetben nem egy skalár bemenetet kell feldolgozni, hanem egy adott dimenziós vektor bemenetet kap a neuron. Ekkor a  $w$  értéke sem skalár lesz, hanem egy, a bemenettel megegyező dimenziójú vektor, mely minden értéke külön súlyként kezelhető. Annak érdekében, hogy a kimenet egy skalár legyen, a két vektor között egy skaláris szorzatot végzünk, melyhez ezután hozzáadjuk a továbbra is skalár  $b$  eltolást:  $w \cdot x^T + b$ .

Ennek eredményeként le tudunk képezni egy adott dimenziójú bemenetet egy skalárra, viszont ez a lépés önmagában információvesztéssel jár. Annak érdekében, hogy a bemenetből minél több információt legyen képes a háló továbbítani valamilyen feldolgozott formában, több ilyen neuronnak is oda kell adni az adott bemenetet, majd a kapott kimeneteket összefűzve továbbítani későbbi feldolgozásra. Ezen neuronoknak a súlyai a gyakorlatban eltérnek, így mindegyik egy kissé eltérő módon dolgozza fel az adott bemenetet. Ezen súlyok legyenek rendre  $w_1, b_1, \dots, w_k, b_k$ , melyekkel az alábbi számításokat hajtjuk végre:  $w_1 \cdot x^T + b_1, \dots, w_k \cdot x^T + b_k$ . A gyakorlatban a gyorsabb számítás érdekében ezeket úgynevezett vektorizálás segítségével számítjuk ki, vagyis az egyes  $w_i$  súlyokat egy mátrixba rendezzük ( $W$ ), hasonlóan a  $b_i$  értékeket egy vektorba ( $b$ ), és az így kapott két műveletre módosítjuk a számítási képletet:  $W \cdot x^T + b^T$ . Az imént leírt  $k$  darab neuront együttesen egy (lineáris) rétegnek nevezzük.

Viszonylag egyszerűen adódik a gondolat, hogy a háló számítási komplexitását két ilyen réteg egymás után helyezésével próbáljuk növelni. Sajnálatos módon viszont igen hamar kiderül, hogy két egymás után helyezett réteg önmagában nem növeli meg a rendszer komplexitását. Tekintsük az egymás utáni két lineáris réteg képletét adott  $x$  bemenetre:  $y = W_1 \cdot x^T + b_1^T$  és  $z = W_2 \cdot y^T + b_2^T$ , vagyis  $W_2 \cdot (W_1 \cdot x^T + b_1^T) + b_2^T$ . Az asszociatív és disztributív tulajdonságokat felhasználva átalakíthatjuk a kifejezést az alábbira:  $W_2 \cdot W_1 \cdot x^T + W_2 \cdot b_1^T + b_2^T$ . Észrevehetjük, hogy lényegében egyetlen réteg számítási képletével megegyező alakot kaptunk, ahol  $W_1 \cdot W_2$  az új  $W$  és  $W_2 \cdot b_1^T + b_2^T$  az új  $b$ , tehát nem sikerült növelnünk a rendszer bonyolultságán.

Annak érdekében, hogy ezt elkerüljük, nemlineáris, vagy más szóval aktivációs rétegekkel kell fenti lineáris rétegeket elválasztanunk. Ezek olyan függvények, melyek

nem írhatóak le véges sok lineáris függvény segítségével (így egyetlen eggyel sem). Leggyakrabban használt példák ilyen függvényekre a *sigmoid*  $\left(\frac{1}{1+e^{-x}}\right)$ , a *tangens hiperbolikus*  $\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)$ , a *ReLU*  $(\max\{0, x\})$  és a *softmax*  $\left(\frac{e^{x_i}}{\sum e^{x_j}}\right)$ . Ezen függvények segítségével már összetettebb hálókat is megadhatunk, ezzel megalapozva a **mélytanulást**.

Tanulás során általánosságban a háló kap egy bemenetet, melyből az aktuális súlyok alapján visszaad egy kimenetet. Ezen kimenetből számolunk egy hiba értéket, mely általában egy elvárt kimenettől való eltérés. A kapott hibát ezután visszaterjesztjük a hálón keresztül az egyes súlyokra deriválás alkalmazásával, úgynevezett gradiens módszerrel. Miután az így visszavezetett hiba minden egyes súly esetén meg lett határozva, egy egységes eljárás keretében frissítjük a súlyokat oly módon, hogy a későbbiekben a rendszer kisebb hibával tudjon egy hasonló bemenetet kezelni.

## Rekurrens hálók

Nem minden esetben lehet a bemenetet fix méretű vektorként kezelni. Legismertebb példa ilyen esetre a szekvenciális adatok feldolgozása, mely során egy változó méretű bemeneti példát dolgozunk fel. Ide soroljuk a szöveg-, illetve hangfeldolgozást, esetlegesen a mozgókép feldolgozást, mindazon példákat, amelyek esetén a sorrendiségnek fontos többletjelentése van a bemenet szempontjából.

Könnyen meggyőződhetünk arról, hogy előrecsatolt lineáris rétegek segítségével ilyen jellegű adatok feldolgozása nehezen megoldható. Amennyiben esetlegesen létezik felső korlát a szekvencia hosszára úgy még képesek lennénk beadni az adatokat egy lineáris rétegnek kiterjesztve extrémális értékekkel, viszont ekkor a lineáris rétegek pozícióhoz kötve tanulnák meg az összefüggéseket, mely — például ha szövegek feldolgozására gondolunk — nem túl célszerű. Ekkor ugyanis más súlyokkal vennénk figyelembe az *alma* szót az *itt egy alma* és az *alma finom* szövegekben, mivel eltérő pozícióban szerepel a szövegekben, ami redundáns tanulást eredményezne és korlátozná az általánosítási képességét a rendszernek. Másrészt a háló súlyainak száma rendkívül magas lenne.

Összességében tehát nem tudjuk előrecsatolt lineáris rétegekkel közvetlenül feldolgozni a bemenetet. Annak érdekében, hogy ezt a problémát kezelni tudjuk, ki kell lépünk az előrecsatolt hálók világából, hiszen a bemenetben előjövő sorrendiség (időbeliség) illetve maga a bemeneti értékek is külön információval bírnak. Ezen gondolatmenet eredményeként jöttek létre az első rekurrens hálók, melyek egységen-

ként, például tokenenként dolgozzák fel a változó hosszú bemenetet, miközben az eddig feldolgozott értékekből kinyert információt egy speciális, úgynevezett rejtett állapotvektorba tárolják el. A feldolgozást egyetlen réteg végzi, mely megkapja az előző lépés kimenetét képező rejtett állapotot és a soron következő tokent, ezekből pedig kiszámolja az új rejtett állapotot (és ebből az esetleges kimenetet). Az adat feldolgozásának végén az utolsó rejtett állapot tárolja az összegyűjtött információt.

## LSTM és GRU

Az egyszerű rekurrens hálóknak van egy komoly hátránya. A fent leírt számítási egység, mely minden adatot egyesével feldolgoz, a számításai során minden lépésben veszi az előző rejtett állapotot és lineáris rétegek kombinációjával (mely továbbra is minden lépésben megegyezik az adott bemenet feldolgozása során) új értéket számol. A gradiens minden lépés esetén egy nem egység normájú súlymátrixszal megszorozva. Ennek eredménye egynél nagyobb norma esetén exponenciális robbanás a gradiens értékében, míg egynél kisebb norma esetén exponenciálisan lecseng, majd eltűnik a gradiens értéke. Egyik esetben se lesz képes a háló a szekvencia első részében található információk alapján tanulni.

Ennek nyomán jött létre az *LSTM* [4], mely a konstans hiba visszavezetés segítségével orvosolta ezt a problémát. Az *LSTM* emellett megoldást nyújtott a hosszú és a rövid távú kapcsolatok kezelésére is úgynevezett memóriacellák, illetve kapuk segítségével. A háló három kaput tartalmaz, mely megtanulja, hogy adott bemenet és rejtett állapot esetén milyen új információkat tároljon el, hagyjon el, illetve tartson meg a bejövő, illetve az eddig eltárolt információból.

Ezen architektúrának egy egyszerűsített változataként jelent meg a *GRU* [5], mely a rejtett állapoton keresztül biztosítja a konstans hiba visszatérjedést, elhagyva ezzel a memóriacellákat.

## Szekvencia-szekvencia transzformáció

A fentiekben immár megoldást találtunk szekvenciális adatok feldolgozására, viszont egyes esetekben szükséges lehet szekvenciális adatokat visszaadni is. Ebbe a feladatkategóriába tartozik például a fordítás, a kép összefoglalás és a hanggenerálás. A továbbiakban azon példát tekintjük, amikor a bemenet és a kimenet is egy

szekvencia. Ezt a feladatot nevezzük *seq2seq* feladatnak, az ezen feladatot megoldó hálókat pedig *seq2seq architektúrának* [1], [6].

A rekurrens architektúráknál persze *seq2seq* nélkül is vissza tudnak adni szekvenciális adatot: az aktuálisan visszaadott rejtett állapotot fel lehet használni egy lineáris réteg segítségével, hogy kimenetet generáljunk az adott időpillanatban. Ezt az ötletet felhasználva minden egyes lépésben visszaadhatunk egy kimenetet a bemeneti szekvencia feldolgozása során. Az ötletnek több hiányossága is van. Az egyik hiányosság, hogy csak a bemeneti szekvenciánál nem hosszabb kimenet megadása lehetséges ily módon. A másik hiányosság, hogy a kimenet elejének előállításakor a modell nem éri el a bemeneti szekvencia végén lévő információkat.

Térjünk vissza egy pillanatra a szekvenciális adatok feldolgozásához. Az ott elmondottak értelmében egy megfelelő rekurrens háló képes egyetlen rejtett állapotvektorba leképezni a bemenetben lévő információt. Emellett tudjuk, hogy a rekurrens hálók képesek kimenetet generálni minden egyes lépésben. Az egyetlen kérdés, hogy ezek alapján hogyan tudjuk megoldani a problémát?

Egy helyett két rekurrens hálót fogunk alkalmazni. Az első (kódoló) a bemenet feldolgozására és egyetlen rejtett állapotba való kódolására. A másodikat (dekódoló) pedig az így létrejövő rejtett állapot felhasználására, dekódolására, a kimenet megadására. A dekódoló minden lépésben az addig elkészített kimenetet folytatja, így változó hosszúságú kimenet generálására képes.

## Attention

A fentiekben megismerkedtünk a fejlett rekurrens hálókkal, melyek képesek hosszú távú összefüggéseket is felismerni, illetve a szekvenciából szekvenciába képező hálókkal. Utóbbi remekül képes megoldani a feladatát mindaddig, amíg a bemenet és a kimenet hossza nem túl nagy. Felmerülhet a kérdés, hogy mégis mi lehet az oka, hogy hosszabb be-, illetve kimenetek esetén rohamosan visszaesik a teljesítménye. A válasz: a kapcsolat a be- és kimenet között egyetlen rejtett állapot, melynek képesnek kellene lennie az összes releváns információt eltárolni a bemenetből, ami a kimenethez szükséges. Ez a rejtett állapot egy fix méretű vektor, így csak limitált információt képes eltárolni. A már jól ismert kérdéssel állunk szemben: Hogyan lehetne ezt megoldani?

Érezhető, hogy a rejtett állapot méretének, vagyis a vektor dimenziójának növelése nem hosszú távú megoldás a problémára. Egy olyan megoldás kellene, mely

lehetőséget biztosít a dekódoló számára, hogy a rejtett állapoton túl is tudjon információt szerezni a bemenetből. A bemenetben lévő információ a kódoló által visszaadott rejtett állapotokban fellelhető, mivel minden bemeneti tokenhez tartozik egy ilyen rejtett állapot. Viszont nem minden állapot releváns minden kimenet meghatározásánál.

A kérdés: melyek egy kimenet előállításához szükséges rejtett állapotok, és hogyan érjük el őket? A kérdésekre a választ bízzuk rá magára a hálóra: a dekódoló az aktuális rejtett állapotot, illetve a kódoló rejtett állapotait felhasználva készít egy súlyozást a bemenet minden tokenéhez, majd ezen súlyozás alapján vegye figyelembe a kódoló által létrehozott rejtett állapotokat. Ennek eredményeként a dekódoló mindig az aktuális rejtett állapot alapján tud beleolvasni a bemenetben elkódolt információba, ezzel kiegészítve a rejtett állapotban lévő időbeliséget.

A fent leírt működést nevezik *attention* mechanizmusnak [1], tekintve, hogy a háló megtanul figyelni a bemenet egyes szavaira, így nyerve információt.

## Másolás

Az *attention* segítségével már hosszabb szekvenciákat is képesek kezelni a hálók, viszont a szótár (vocabulary) mérete korlátozott, és az abban nem szereplő tokeneket nem kezelik.

Erre egy megoldás a másolás, melynek célja, hogy a bemenetben lévő ritkán előforduló, esetleg ismeretlen tokeneket vagy azoknak akár egy sorozatát a háló megtanulja kiemelni, és helyesen tokenről tokenre a kimenetbe generálni. Felvetődhet az ötlet, hogy ezeket a tokeneket is rakjuk be tanítási példaként, viszont az összes lehetséges szót kezelni ilyenformán nem lehetséges.

Jobb lenne, ha a háló képes lenne valahogyan rámutatni az egyes bemeneti tokenekre, mint ahogy az *attention* esetén is tette, és így kiválasztani a következő visszaadandó tokenet. Amilyen egyszerűnek hangzik ez az állítás az *attention* megismerése után, olyan egyszerűen meg is valósítható: a dekódoló aktuális rejtett állapotával vesszük a súlyozását a kódoló által visszaadott rejtett állapotoknak, majd ezen súlyozás alapján kiválasztjuk a következő kimeneti tokenet. Az egyetlen dolog, amire még figyelniünk kell, hogy a ne vesszen el a háló azon lehetősége, hogy a kimenetben nem lévő tokeneket is képes visszaadni. Ennek érdekében minden egyes lehetséges token esetén összeadjuk a hagyományos generálás által adott súlyozást az új mó-

don számolt súlyozással, ügyelve arra, hogy ha egy token a bemenetben többször is előfordul, akkor azon előfordulásokhoz tartozó összes súlyt figyelembe vesszük.

A fentiekben leírt módszerek remekül működnek egyetlen token kiemelése esetén, viszont mi történik abban az esetben, ha egy sorozatnyi tokenet szeretnénk kiemelni a bemenetből a kimenetbe? Jó lenne egy, az *attention*-höz hasonló technika, mely során az előzőleg másolt elemet figyelembe tudnánk venni, hogy a környezetében lévő potenciálisan másolni kívánt új elemek nagyobb súlyozást kapjanak. Szerencsére az előző lépés során meghatároztuk, hogy melyik bemeneti tokenet milyen súllyal kell figyelembe venni. Ezt a súlyozást most ismételtén felhasználhatjuk egy újabb *attention* elkészítésére (hiszen az eredeti motivációja is az *attention* volt). Vegyük a legutoljára visszaadott tokenet, és amennyiben az másolva volt a bemenetből, úgy a bemenetben lévő előfordulásaihoz tartozó rejtett állapotokat vesszük figyelembe a már meglévő súlyozás alapján.

Ez a módszer jól képes működni, ha a bemenetből fordított sorrendben szeretnénk másolni a kimenetre, hiszen akkor egy másolt token után a következőhöz tartozó rejtett állapotban már található információ a bemenetben őt megelőző tokenről. Viszont jó lenne, ha a bemenettel megegyező sorrendben is tudnánk könnyedén tokensorozatot másolni. Ennek érdekében a bemenetet úgynevezett kétirányú rekurrens hálóval dolgozzuk fel. A kétirányú rekurrens háló valójában két rekurrens háló együttes használatát jelenti, amelyek közül az egyik előre fele dolgozza fel a kimenetet, míg a másik a fordított sorrendben. A létrejövő két rejtett állapotot ezután összefűzik tokenenként, így egy adott bemeneti tokenhez tartozó rejtett állapot tartalmazni fog a token előtti és utáni részből is információt.

Az előbbiekben ismertetett módszert nevezik másoló mechanizmusnak, avagy *copying mechanism*-nek [2].

### 1.3. Kapcsolódó munkák

A permutáció, mint feladat kezelése már több különböző területen is megjelent a gépi tanulás kapcsán.

Chughtai és mások [7] az  $S_5$  csoportot vették alapul a gépi tanulás egyetemességének (*Universality*, [8]) mérésére. Munkájuk során az 5 hosszú permutációk kompozíciójának tanulását figyelték különböző reprezentációkat alapul véve. Tekintve, hogy

a reprezentációk mind beágyazást jelentettek, a kompozíciók pedig mátrixszorzáson alapultak, így nem a bemenet egy általános permutálását végezték el.

Santa Cruz és mások [9] összekevert képek visszaalakítására tanítottak be egy architektúrát. A feladatot (képek helyreállítása) vizuális permutációs feladatnak nevezték. Ellentétben a mi munkánkkal, architektúrájuk az elvárt sorrendet magából az adatból kellett, hogy kinyerje, így a feladat tekinthető egy rendezésnek.

Hasonló feladatot tűzött ki Mena és mások [10], akik nem csak képek, hanem számsorozatok rendezésére megoldást nyújtottak.

Bemeneti tokenek, tokenszekvenciák átemelését oldották meg Gu és mások [2]. Munkájukban a bemenetben szereplő tokenek egy-az-egyben történő másolását valószínűsítették meg csupán azok kontextusára alapozva. Ezen feladat egy általánosításaként tekinthető a bemenet egy részének adott permutációjának visszaadása.

## 2. fejezet

# Módszerek

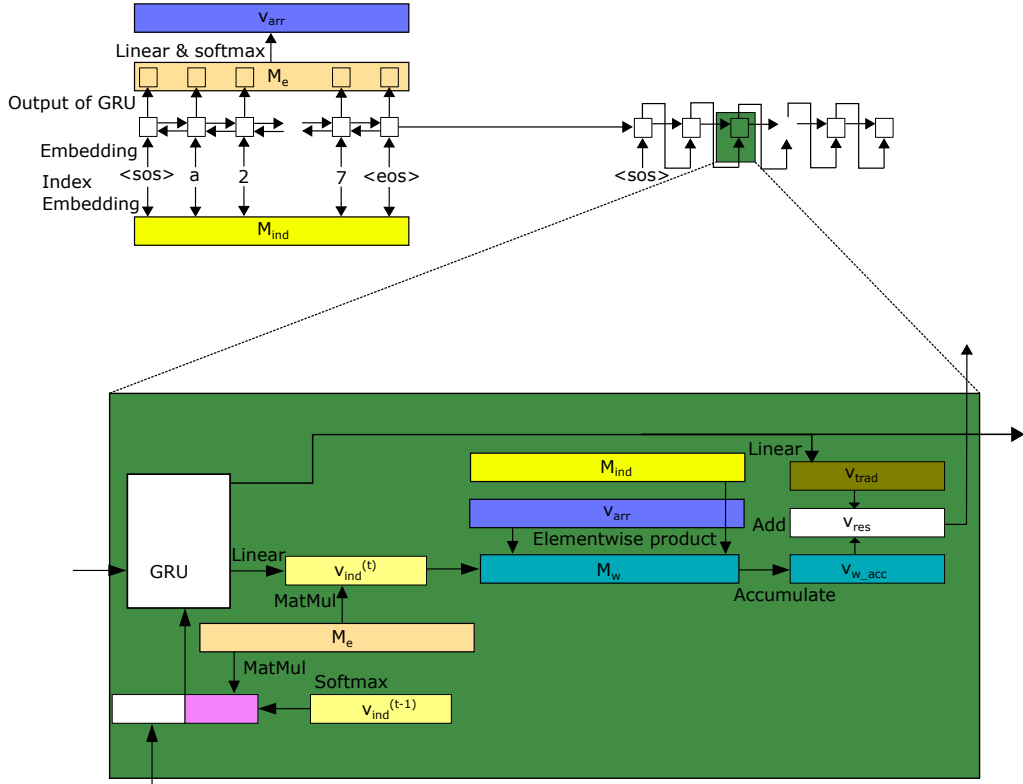
### 2.1. Architektúrák

Munkánkban a két feladat megoldására két különböző architektúrát hoztunk létre, és tanítottunk be. Mindkét architektúra alapja egy *seq2seq* architektúra kétirányú *GRU* kódolóval és egyirányú *GRU* dekódolóval. A továbbiakban az adat tokeneket ADAT, a permutációban lévő elemeket PERMUTÁCIÓ, a közvetlen feladat esetén az index tokeneket INDEX, a közvetett feladat esetén a kulcs tokeneket KULCS jelzővel fogjuk jelölni.

### 2.2. Architektúra közvetlen permutálási feladatra

A közvetlen permutálási feladat esetén a tanítás során ismert, hogy a leghosszabb permutáció mérete mekkora, ennél hosszabb eset kezelése a feladat leírásának megfelelően nem elvárható. Hasonlóan nem elvárható, hogy az ADAT értékek szétszórva helyezkedjenek el, ekkor nem lehet meghatározni, hogy mi is számít ADAT tokennek, így feltehető, hogy az ADAT tokenek egymás után helyezkednek el. Ezen többletinformáció birtokában hoztuk létre az első architektúrát.

Az architektúra a bemenet minden elemét feldolgozza a kétirányú *GRU*-val. Ennek rejtett állapota a dekódolónak továbbadódik, míg kimeneteire ( $M_e$ ) egy lineáris réteget alkalmazunk, mely minden bemeneti tokenhez egyetlen értéket rendel, mely azt mutatja meg, hogy az adott tokenről kezdődik-e az ADAT tokenek sorozata. Ezen értékekre alkalmazzuk a *softmax* függvényt, így egy valószínűségi eloszlást kapva:  $v_{arr} = softmax(\text{Lin}(M_e))$



2.1. ábra. A közvetlen permutációs feladathoz készített architektúránk ábrája. Az ábra felső részén látható a kódoló-dekódoló architektúra, kétirányú *GRU* kódolóval. A kódoló kimenetét felhasználjuk a  $v_{arr}$  vektor előállításához, mely megadja, hogy melyik bemeneti tokenről kezdődnek az ADAT értékek. Emellett a bemenet minden tokenjére alkalmazunk egy beágyazást, mely megadja, hogy hányadik indexet jelöli az adott token, mint INDEX érték. A dekódoló működése a kép alsó részén látható. A kódoló kimenetét súlyozzuk az utoljára generált INDEX tokenek eloszlásával ( $v_{ind}^{(t-1)}$ ), melynek eredményét a dekódolóban lévő *GRU* bemenetéhez fűzünk. A dekódolóban lévő *GRU* új rejtett állapotát egyrészt a hagyományos generálásra ( $v_{trad}$ ) használjuk fel. Másrészt kombináljuk a kódoló kimeneteivel, ezáltal megkapva a  $v_{ind}^{(t)}$  vektort, melyet a kezdőpozíció súlyozásával és a bemeneti tokenek indexelésével elemenkénti szorzással kombinálunk. Az így kapott  $M_w$  súlyozás az azonos pozícióra mutató elemek összevonásával felhasználható a permutációs token prediktálására ( $v_{w\_acc}$ ).

A bemeneti tokenekre alkalmazunk egy második beágyazást is, melynek mérete a maximális permutáció méretével megegyezik:  $M_{ind} = \text{Emb}(inp)$ . Ezen beágyazás értékeit, mint egy tömbbe való indexelést, fogjuk felhasználni.

A dekódoló tokenfeldolgozást követő rejtett állapotát egy lineáris réteggel való feldolgozást követően kombináljuk a kódolási kimenettel, ezáltal súlyozást megadva a következő INDEX tokenhez:  $v_{ind} = M_e^T \cdot \text{Lin}(v_d)$  Ezzel a súlyozással, illetve az ADAT sorozat kezdőpozíciójának eloszlásával elemenként megszorozzuk a második beágyazás eredményét:  $M_w = v_{arr} \odot M_{ind} \odot v_{ind}$  (ahol  $\odot$  az elemenkénti szorzást

jelenti). Ennek eredménye egy súlyozás, mely minden egyes bemeneti tokenre megadja, hogy tőle, mint az ADAT szekvencia első eleme, egy adott indexre mekkora eséllyel van a következő kimeneti token.

Tekintve, hogy ily módon az indexelésekben létrejöhet átfedés (a második token harmadik indexe és a negyedik token első indexe ugyanazon pozíciókat jelöli), így az azonos pozícióra mutató értékeket összeadjuk.

A másoló mechanizmushoz [2] hasonlóan a bemenetben szerepelhet több azonos token is, mely pozíciója a kimenet szempontjából nem releváns. Ezt kezelendő, tokenenként összeadjuk a bemenetben a token előfordulási pozícióiban található súlyokat ( $v_{w\_acc}$ ), így minden egyes tokenhez csupán egyetlen érték fog tartozni végül. Az így kapott értékeket hozzáadjuk a hagyományosan generált token eloszláshoz ( $v_{trad}$ ), így megkapjuk a dekódoló egy lépésének a kimenetét ( $v_{res}$ ).

### 2.3. Attention mechanizmus a közvetlen permutálási architektúrához

Ahogy a másoló mechanizmus [2] esetén is, úgy ennél az architektúránál is lehet egyfajta *attention* működéssel kiegészíteni a hálót a hatékonyság növelésének érdekében. Tudjuk, hogy a legutoljára mutatott INDEX token környezetében van egy másik INDEX token, vagy potenciálisan más érdekleges információ. A legutolsó INDEX token súlyozásából tudunk készíteni egy eloszlást, mellyel figyelembe vesszük a bemeneti *GRU* kimeneteit. Ezen információt így a bemenet részeként átadhatjuk a dekódoló *GRU*-jának. A közvetlen permutáció feladatához készült architektúra *attention* mechanizmussal a 2.1 ábrán látható.

### 2.4. A közvetett permutálási feladat emberi megközelítése

Ezen feladat kapcsán próbáltuk feltárni, hogy az emberek hogyan közelítik meg ezt a problémát. Ennek keretében egy kisebb informális felmérést végeztünk. A felmérésben az érintett személyek az alábbi bemenetet kapták: **a 2 b 3 c 1 <sep> 1 3 2**, és meg kellett határozniuk a kimeneti tokensorozatot ügyelve arra, hogy lépésként milyen részfeladatokat oldanak meg. Az eredmény általánosságban nem

meglepő: szinte mindenki először végignézte a kapott token sorozatot, majd a `<sep>` token utáni elemeket kezdték el feldolgozni sorban. A feldolgozás egy lépéseként megkeresték a bemenet `<sep>` előtti részében az adott tokenet, majd az első kettő (vagy a `<sep>` előtti utolsó kettő) token sorrendjéből kiindulva vették az adott pozíció előtti elemet, és kimásolták a lépés eredményeként a kimenetre.

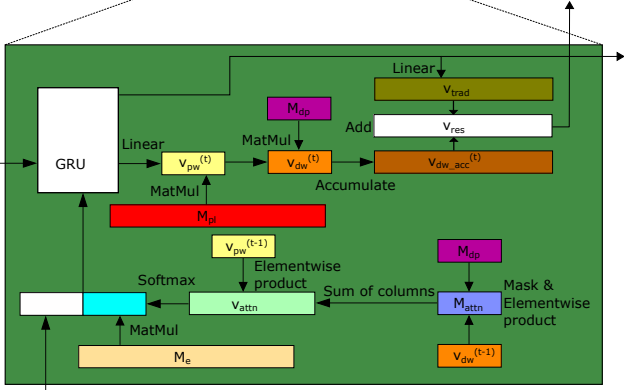
Ez a szabadosan megfogalmazott algoritmus mutatott rá arra a képességre, melyet az architektúrának majd el kell sajátítania a feladat sikeres kezeléséhez. Fel kell ismernie, hogy a permutációt megadó sorozat hol helyezkedik el, ezt meg kell különböztetni a többi elemtől. Ezt követően meg kell keresni, hogy az egyéb részekben mely token jelenti a kapcsolatot a feladat szempontjából (a példa esetében ez ugyanaz a token egy korábbi pozícióban). Miután megvan a kapcsolatot biztosító token, meg kell határozni az általa reprezentált értéket (mely a példa esetében az előtte lévő token).

Az alábbiakban ezt a gondolatmenetet megvalósító architektúrát részletezzük.

## 2.5. Architektúra a közvetett permutálási feladatra

A közvetlen permutálási feladattal ellentétben a közvetett permutálási feladat sokkal általánosabb, tekintve, hogy a három sorozatból adódóan nincsen a méretre megszorítás. Az architektúra alapjául itt is egy *seq2seq* architektúrát használunk, az előbbiekben bemutatotthoz képest jelentősebb kiegészítésekkel.

A kódoló kétirányú *GRU*-jának kimeneteire ( $M_e$ ) alkalmazunk egy lineáris és egy *softmax* réteget, melyek az aktuális token osztályozását adják NEM\_RELEVÁNS, ADAT, KULCS, PERMUTÁCIÓ címkékkel egy valószínűségi eloszlásként. Az így létrejövő eloszlást ezután szétválasztjuk külön ADAT, KULCS és PERMUTÁCIÓ vektorokra, a NEM\_RELEVÁNS kategóriát elhagyva. Ezen vektorokat elemenként megszorozzuk a kódoló által adott kimenettel, így megkapva az ADAT, KULCS és PERMUTÁCIÓ címkékhez tartozó  $M_d$ ,  $M_k$ ,  $M_p$  mátrixokat. Ezt követően elvégzünk két mátrixszorzást és egy-egy módosított logaritmikus *softmax*-ot:  $M_{dk} = \text{LogSoftmax}(M_d \cdot M_k^T)$ ,  $M_{kp} = \text{LogSoftmax}(M_k \cdot M_p^T)$ . A létrejövő két négyzetes mátrix oldalhossza megegyezik a bemenet hosszával. Jelentésük az ADAT és a KULCS, illetve a KULCS és a PERMUTÁCIÓ közötti összefüggések (a kódoló kime-



2.2. ábra. A közvetett permutációs feladathoz készített architektúránk ábrája. Az ábra középső részén látható a kódoló-dekódoló architektúra. A kétirányú *GRU*-val rendelkező kódoló kimenetére ( $M_e$ ) egy osztályozást alkalmazunk, mellyel meghatározzuk az egyes bemeneti tokenek relevanciáját mint PERMUTÁCIÓ, KULCS, ADAT és NEM\_RELEVÁNS. Ezen értékek figyelembevételével (a NEM\_RELEVÁNS-t elhagyva) súlyozzuk a kódoló kimenetét ezzel megkapva az  $M_p$ ,  $M_k$  és  $M_d$  mátrixokat. Az első kettőt, és a második kettőt felhasználva előállítjuk a PERMUTÁCIÓ és a KULCS tokenek közti összefüggést leíró  $M_{kp}$  mátrixot, illetve a KULCS és az ADAT közti  $M_{dk}$  mátrixot. Ezen két mátrixszorzataként megkapjuk a PERMUTÁCIÓ és az ADAT közti átsúlyozó mátrixot az  $M_{dp}$  mátrixban. A dekódoló részletes működése az ábra alján látható. A dekódoló ki van egészítve az architektúra saját *attention* mechanizmusával, mely során a permutációs kimenet generálási eredményét ( $v_{dw}^{(t-1)}$ ) a hasonlósági mátrixszal ( $M_{dp}$ ) elemenként összeszorozzuk ( $M_{attn}$ ), majd ennek oszloponként vett összegét a permutációs tokenek súlyozásával ( $v_{pw}^{(t-1)}$ ) elemenként összeszorozzuk ( $v_{attn}$ ). Az időbeliség miatt az előző dekódoló lépésben számolt vektorok ( $t-1$ ) indexszel, a jelenlegi lépésben számolt elemek pedig ( $t$ ) indexszel vannak jelölve. Végül ebből a *softmax* függvény segítségével valószínűségi eloszlást készítünk, és ezen súlyozással vesszük figyelembe a kódoló által adott kimeneteket ( $M_e$ ). A dekódolóban lévő *GRU* utáni rejtett állapotot egyrészt felhasználjuk a hagyományos generálásra ( $v_{trad}$ ). Másrészt vesszük vele a permutációs tokenek egy súlyozását ( $v_{pw}^{(t)}$ ), melyet átsúlyozunk az  $M_{dp}$  mátrixszal a megfelelő ADAT tokenekre, ezzel megkapva a permutáció által adott eredményt ( $v_{dw}^{(t)}$ , illetve összevonás után  $v_{dw\_acc}^{(t)}$ ).

netek önmagukkal vett skalárszorzatai a címkék szerint súlyozva), mely megmutatja, hogy egy adott kulcshoz milyen valószínűséggel tartoznak az adott adatok, illetve hasonlóan a permutációs tokenekhez a kulcsok.

A módosított logaritmikus *softmax* először monoton növő módon leképzi a *softmax* valószínűségeket az  $[1, e]$  intervallumba, majd ezekre alkalmazza a logaritmus függvényt:  $\text{LogSoftmax}(x) = \log(1 + (e - 1) \cdot \text{softmax}(x))$

A két mátrixot ezután összeszorozzuk, hogy megkapjuk a másodrendű összefüggéseket megmutató mátrixot az ADAT és PERMUTÁCIÓ között, ezek KULCS-hoz való hasonlóságain keresztül:  $M_{dp} = M_{dk} \cdot M_{kp}$ . Annak érdekében, hogy elkerüljük, hogy egy adott token legjobban önmagával függjön össze (ami egy várható viselkedés lenne), 0 értékkel kimaszkoljuk mindkét mátrix főátlóját, ezzel a többi tokenre helyezve a hangsúlyt (az esetleges *padding* PERMUTÁCIÓ tokenekhez tartozó értékeket hasonlóan kimaszkoljuk).

A dekódolóban használt egyirányú *GRU* esetén a kapott új rejtett állapotot egyrészt egy lineáris rétegen keresztül hagyományos módon felhasználjuk a következő kimeneti token eloszlásának kiszámításához ( $v_{trad}$ ). Másrészt, a rejtett állapotot ( $v_h$ ) mátrixszorozzuk az  $M_p$  mátrixszal egy-egy lineáris réteg alkalmazása után:  $v_{pw} = \text{Lin}(M_p) \cdot \text{Lin}(v_h)$ . Az így kapott vektor egy súlyozása lesz a bemeneti tokeneknek, és ez alapján a súlyozás alapján vesszük figyelembe az adott lépésnél PERMUTÁCIÓ szerint az egyes tokeneket. Ezen vektort az  $M_{dp}$  mátrixszal történő szorzással átsúlyozzuk a PERMUTÁCIÓ súlyokról az ADAT súlyokra:  $v_{dw} = M_{dp} \cdot v_{pw}$ . Ezt követően már tudjuk, hogy melyik bemeneti tokenet milyen súllyal kell figyelembe venni, mint a soron következő kimeneti token. A kimenet szempontjából viszont teljesen irreleváns, hogy két azonos token esetén melyik kapja a nagyobb súlyt. Ebből kifolyólag minden egyes lehetséges kimeneti token esetén a hozzá tartozó súlyokat összeadjuk ( $v_{dw\_acc}$ ), és a következő kimeneti token ezen súlyozását hozzáadjuk a hagyományos úton számolt súlyozáshoz ( $v_{res}$ ).

## 2.6. Attention mechanizmus a közvetett permutációs architektúrához

A másoló mechanizmus [2] saját *attention* mechanizmusa után már adott a motiváció az előbbieken bemutatott architektúra kiegészítésére egy saját *attention* mechanizmussal. A másoló mechanizmus esetén viszont a figyelmet jelentő súlyok megegyeznek a végső kimenetet jelentő súlyokkal (a bemeneti tokenenként vett összevonás előtti állapotban). Amennyiben az architektúra szempontjából az ADAT to-

kenekre való figyelmet szeretnénk elérni, úgy ez a viselkedés minden további nélkül itt is megvalósítható.

Ezzel szemben, ha a PERMUTÁCIÓkra szeretnénk az architektúra figyelmét irányítani, úgy a végső kimenetet vissza kell alakítanunk a PERMUTÁCIÓkra az ADAT tokenekről. Azt, hogy mely ADAT értékeket kell figyelembe venni, a másoló mechanizmushoz hasonlóan a következő dekódoló lépés bemenetével meghatározhatjuk. Ekkor azon kimeneti súlyokat vesszük figyelembe, mely helyeken a bemenetben a dekódoló bemenetével (tehát a legutoljára visszaadott kimeneti tokennel) megegyező token áll. Minden más sorhoz tartozó súlyt kinullázhatunk. A kérdés, hogy ezeket az értékeket hogyan tudjuk visszaalakítani a megfelelő PERMUTÁCIÓ tokenek pozíciójára.

Tekintve, hogy a PERMUTÁCIÓ tokenekről az ADAT tokenekre való átsúlyozást az  $M_{dp}$  mátrix valósítja meg, így adódik, hogy ezt a mátrixot próbáljuk meg a visszaalakításra is felhasználni. A legegyszerűbb lenne a mátrix inverzét venni, ha ez létezne. Viszont mivel ezt a mátrixot egy az egyben a háló állítja elő, így ilyen jellegű elvárásunk nem lehet a mátrix felé. Esetleg vehetnénk a mátrix pszeudo-inverzét, viszont ezen számítás igen költséges, és könnyen válhat numerikusan instabillá.

A mátrixinvertálási lehetőségek így a háttérbe szorultak. Próbáljuk meg végiggondolni, hogy milyen más módon lehetne visszaképezni az ADATot a PERMUTÁCIÓra. Vegyük észre, hogy nem szükséges a teljes visszaalakítás, hiszen mindössze pár token súlyát kell visszaképezni, nem az egész kimenetét. Emellett ezen súlyokat a mátrix adott sorai határozzák meg. Ezen sorokat tudjuk súlyozni a hozzájuk tartozó kimeneti súlyokkal, hiszen azt szeretnénk, hogy minél nagyobb kimeneti súlyt határoz meg egy adott sor, annál inkább legyen hangsúlyos az *attention* szempontjából. Így kapunk soronként egy súlyozást a PERMUTÁCIÓ tokenekre ( $M_{attn}$ ), ezen súlyozásokat immár nyugodtan összeadhatjuk oszloponként, majd a PERMUTÁCIÓ tokenek (előző lépéshez tartozó) súlyozásával ( $v_{pw}$ ) elemenként megszorozhatjuk ( $v_{attn}$ ), így minden PERMUTÁCIÓ tokenhez egyetlen súlyt rendelve a bemenetben.

Sajnos az így létrejövő súlyozás a gyakorlatban nem csak egy, hanem az összes bemeneti tokenhez rendel egy súlyt. Ezzel szemben az lenne a cél, hogy csak a legnagyobb súlyok maradjanak meg. Ezt egy elcsúsztatott *ReLU*-val is meg lehetne valósítani, viszont ekkor a határ nagyon éles és mesterséges lenne. Emellett jó lenne, ha a súlyozás helyett egy valószínűségi eloszlást kapnánk a bemeneti tokenekre.

A *softmax* függvény részleges megoldást jelent az első felvetésre, míg triviálisan megoldja a másodikat. Annak érdekében, hogy a *softmax* jelentősen kiemelje a nagy értékeket, a számítása előtt meg lehet szorozni egy konstanssal az összes értéket. Habár a *softmax* invariáns az eltolásra, a szorzás esetén már jelentős transzformáció érhető el a kimenetén.

Ez egy bevett módszer, melyben a szorzó reciprokát gyakran „hőmérsékletnek” (*Temperature*) nevezik, és minél nagyobb a hőmérséklet, annál egyenletesebb eloszlást eredményez a *softmax*, és minél kisebb a hőmérséklet, annál jobban kiemeli a nagyobb értékeket.

Ezzel a módszerrel már sikerül hatékonyan és numerikusan stabilan visszaalakítani a kimeneti ADAT súlyokat a PERMUTÁCIÓ tokenekre egy *attention* mechanizmust megalkotva.

A közvetett permutálás feladatához készült architektúra *attention* mechanizmussal a 2.2 ábrán látható.

## 2.7. A létrehozott *attention* összeillesztése a másoló mechanizmus *attention*ével

A fentiekben leírt *attention* és maga az architektúra remekül képes teljesíteni az eredetileg megfogalmazott feladaton. Az architektúra bővítése szempontjából viszont nem mellékes, hogy a korábban már ismert bővítésekkel (*attention*, másoló mechanizmus) hogyan lehet kompatibilissé tenni az architektúrát. Természetesen az *attention* esetén nincs probléma az összeillesztéssel, hiszen csak a kódoló kimenetét használja fel, esetlegesen lineáris rétegekkel lehet mindkét esetben a kódoló kimenetét feldolgozni a komplexitás növelése érdekében.

A másoló mechanizmus [2] esetében viszont már nem ilyen egyszerű az összeillesztés. A kimenet meghatározását össze lehet vonni a két esetben, hiszen mindkét módszer a bemenet tokenjeihez rendel súlyokat, így a tokenenkénti összegzést el lehet végezni egyben. Ezzel szemben a módszerek *attention*ei már kevésbé tudnak szinkronban működni, tekintve, hogy mindkét módszer a kimenet alapján fektet hangsúlyt a bemenetben szereplő egyes tokenekre. Míg a másoló mechanizmus esetén közvetlenül a kiemelt tokenekre figyel a háló, addig az általunk létrehozott *attention* indirekció segítségével a PERMUTÁCIÓ tokenekre irányítja a figyelmet, így a kettő

együttes használata inkonzisztens viselkedést von maga után és a figyelem elveszti a lényegi szerepét.

Ezen problémát kezelendő, kimaszkoljuk a két kimenet azon súlyait, amely pozícióban a másik kimenet súlya nagyobb (egyenlőség esetén egy előre meghatározottat választunk). Így minden pozícióhoz tartozó súly esetén csak az egyik kimenet *attentione* aktiválódhat csak, így a háló már képes egyértelműen rámutatni a bemenet adott pontjaira minden egyes kiválasztott token esetén.

## 3. fejezet

# Kiértékelés

### 3.1. Mérőszámok

Munkánk során kétféle mérőszámot alkalmaztunk. Ezek közül az első a tokenenkénti pontosság vagy egyszerűen tokenpontosság. Ez a mérőszám megadja, hogy a kimenetben szereplő tokenek hány százalékát találta el a háló.

A második mérőszám a tökéletes predikciók aránya. Ez esetben a vizsgált példák esetén a hibátlanul (minden token megfelelő) visszaadott példák arányát tekintjük.

### 3.2. Eredmények

Architektúránkat, melyeket feladattól függetlenül PermNet névvel illetünk, különböző benchmarkokon teszteltük, ezeket és az eredményeket az alábbiakban részletezzük.

#### 3.2.1. Közvetlen permutációs architektúra eredményei

A közvetlen permutációs feladat esetén három különböző méréssel vetettük össze az egyes architektúrák hatékonyságát.

Az első mérés esetén 1 és 10 közötti hosszúságú permutációkat kellett a hálóknak helyesen megtanulniuk. A második mérés esetén csupán 10 hosszú permutációkkal dolgoztunk, hasonlóan a harmadik méréshez, ahol a skálázhatóságot vizsgálva 20 hosszú permutációk voltak az adathalmazban. Ez utolsó mérést néhány esetben, ahol érdemi eredmény elérésére nem láttunk esélyt, nem végeztük el. A tesztadat-

halmaz mindhárom esetben ugyanolyan példákat tartalmaz, mint a tanítóadatok: 1-10 hosszú, 10 hosszú, illetve 20 hosszú permutációkat.

A bemeneteket minden esetben egy ADAT és egy INDEX szekvencia jelentette. Az adatok mind szintetikusak lettek generálva majd szétválasztva tanítási- és tesztadathalmazra.

Az eredmények értelmezéséhez érdemes megjegyezni, hogy  $n$  hosszra egy véletlenszerű permutációt kimenetként adó hipotetikus modell is  $\frac{1}{n}$  tokenenkénti pontosságot érne el. (Bizonyítást ld. a függelékben.) Ehhez hozzászámolva a kimenet kezdetét és befejezését jelző egy-egy tokent, amik fix hosszú permutációknál mindig változatlan helyen szerepelnek, a teljes kimeneten a véletlenszerű modell átlagos tokenenkénti pontossága  $\frac{3}{n+2}$ . Ez 10-es hosszra 25%, 20-ra 13.6%-ot jelent, az ezt meg nem haladó eredmények így érdemi tanulást semennyire sem mutatnak.

|                    | 1-10   |        | 10     |        | 20     |        |
|--------------------|--------|--------|--------|--------|--------|--------|
|                    | acc.   | perf.  | acc.   | perf.  | acc.   | perf.  |
| <i>GRU</i>         | 21.75% | 0%     | 17.17% | 0%     | –      | –      |
| <i>Attention</i>   | 99.99% | 99.98% | 24.92% | 0%     | 11.35% | 0%     |
| <i>Copying</i>     | 99.97% | 99.86% | 91.66% | 0%     | 94.01% | 0%     |
| <i>Transformer</i> | 41.87% | 0%     | 39.24% | 0%     | –      | –      |
| <i>PermNet</i>     | 99.54% | 95.44% | 99.73% | 97.63% | 99.90% | 99.32% |

3.1. táblázat. Közvetlen permutálási feladaton elért eredmények. Tokenenkénti pontosság (acc.), ill. hibátlanok aránya (perf.)

### 3.2.2. Közvetett permutációs architektúra eredményei szintetikus példákon

Közvetett permutációs feladat esetén három különböző mérést is elvégeztünk, melyek során összehasonlítottuk az architektúránkat az általános *seq2seq* architektúrákkal (*GRU*, *attention*, *Transformer*).

Elsőként itt is az 1 és 10 közötti permutáció hosszakra vizsgáltuk az architektúrák képességeit. A példák első részében különböző ADAT-KULCS párok vannak felsorolva, majd a bemenet végén szerepelnek a PERMUTÁCIÓ címkék, mint a KULCS címkék egy permutációja.

Emellett hasonló módon megmértük a különböző architektúrák teljesítményét csak 10, illetve ahol ez alapján érdemi eredményt lehetségesnek tartottunk, csak 20 hosszú permutációkból álló adathalmazokon is.

|                    | 1-10   |        | 10     |        | 20     |        |
|--------------------|--------|--------|--------|--------|--------|--------|
|                    | acc.   | perf.  | acc.   | perf.  | acc.   | perf.  |
| <i>GRU</i>         | 19.20% | 0%     | 16.83% | 0%     | –      | –      |
| <i>attention</i>   | 56.83% | 17.98% | 16.35% | 0%     | –      | –      |
| <i>Copying</i>     | 99.90% | 99.32% | 99.99% | 99.97% | 15.93% | 0%     |
| <i>Transformer</i> | 99.97% | 19.77% | 39.15% | 0%     | –      | –      |
| <i>PermNet</i>     | 99.99% | 99.99% | 100%   | 100%   | 99.99% | 99.99% |

3.2. táblázat. Közvetett permutálási feladaton elért eredmények. Tokenenkénti pontosság (acc.), ill. hibátlanok aránya (perf.)

Tesztelve az architektúránk skálázhatóságát az azonos hosszú permutációkat tartalmazó benchmarkokból készítettünk egy 40, illetve egy 100 hosszúságú változatot is. Elsőre ezek a hosszak rövidnek tűnhetnek, viszont a lehetséges példák száma a PERMUTÁCIÓ hosszok faktoriálisával arányos, így a 40 hosszú eset esetén már megközelíti a  $10^{48}$  lehetőséget. Ennek értelmében a kitűzött feladat esetén a háló csak akkor tud értékelhető eredményt elérni, ha képes ténylegesen megtanulni a feladat megoldásához szükséges szabályt.

Architektúránk képességeit kimértük egy általánosabb feladaton is, amelyben a PERMUTÁCIÓ részben az KULCSok többször is előfordulhattak, vagy esetlegesen egyszer sem (Szótár, avagy Dict feladat). A bemenet első felében lévő ADAT-KULCS párok száma itt is megegyezett a PERMUTÁCIÓ részben található címkék számával.

|         | 40     |        | 100    |        | Dict   |        |
|---------|--------|--------|--------|--------|--------|--------|
|         | acc.   | perf.  | acc.   | perf.  | acc.   | perf.  |
| PermNet | 99.99% | 99.96% | 99.93% | 99.75% | 99.84% | 98.81% |

3.3. táblázat. PermNet eredményei nagyon nagy permutációkon és általános Dict feladaton

### 3.2.3. Közvetett permutálási architektúra eredménye valós példán

Architektúránk képességeit egy valós problémán is kimértük. Egy másik kutatásunk során egy kódvisszafejtő modellt fejlesztünk, mely C kódra alakít vissza assembly nyelven írt kódot. A visszaalakítás során a `switch` utasításból létrejövő úgynevezett ugró táblák (*jump table*) esetén felmerülő másolások a *switch* utasítás blokkjainak sorrendjében kell, hogy megtörténjenek. Ezzel szemben a másolandó

értékek egy bináris fa alakban vannak elhelyezve a kódban a gyorsabb kiértékelés érdekében. A másolandó értéket a sorrenddel összekötő kapocs az ugrás címét jelző címke, mely előfordul mind az ugrás helyén (a másolandó értékkel vett összehasonlításakor), mind az ugrás céljánál (a sorrendet meghatározó pozícióban), magánál a címkénél.

Architektúránk ezen a feladaton a Copying architektúrához képest 6 százalékponttal (85,7%-ról 91,7%-ra) jobb eredményt ért el tökéletes predikciókat tekintve. Ez jelentős előrelépést jelent, hiszen a hibás példák aránya 40%-kal csökkent.

Ezzel megmutattuk, hogy olyan esetekben, amikor egy feladat része egy permutáció megoldása, nagy előrelépés érhető el az architektúránk alkalmazása esetén.

## 4. fejezet

# Diszkusszió

Az összehasonlító táblázatokban (3.1 és 3.2 táblázatok) a 20 hosszú permutációk egyes mezői nem lettek kimérve, tekintve, hogy bizonyos architektúrák már a 10 hosszú esetben sem tudott érdemi eredményt elérni. Emellett látható, hogy mindössze a Copying architektúra volt képes kezelni a 10 hosszú permutációkat, viszont a 20 hosszú eseteken eredménytelennek bizonyult. Ezzel szemben a változó hosszú példák (1-10) több architektúra is jobb eredményeket ért el. Ezek tanításakor észrevehető, hogy elsőként a rövidebb példákat kezdi el megtanulni a háló, majd ezt követően fokozatosan kezdi elsajátítani a hosszabb példák kezelésének képességét.

Közvetett permutáció esetén megfigyelhető a rejtett állapot méretének lineáris skálázódása a példák hosszának növekedésével (lásd 5. fejezet). Ez a tendencia viszont megszakad a 100 hosszú permutációkat tartalmazó példák esetén, melyet a 40 hosszú permutációs példák tanuló modellel azonos méretű háló is képes volt kezelni. Fontos megjegyezni, hogy a 100 hosszú permutációkat tartalmazó bemenetek több mint 300 tokenből álltak, kimenetek pedig 100 tokennél hosszabbak. Ilyen hosszú általánosabb példák esetén egy egyszerű *attention* [1] önmagában is nehezen birkózik meg, nem egy olyan összetett feladat esetén, mint a permutáció.

Kiemelendő, hogy a Másoló mechanizmus [2] igen magas tokenenkénti pontosságot ért el a közvetlen permutálási feladaton, mégis a 10, illetve 20 hosszú permutációk esetén 0% volt a hibátlanul kezelt példák aránya. Ennek oka, hogy habár magát a feladat lényegi részét szinte hibátlanul volt képes kezelni az architektúra, elvesztette a nem másolandó tokenek helyes kezelésének képességét, így a kezdő és záró tokeneket nem volt képes helyesen prediktálni. Emellett megjegyzendő, hogy az összes architektúra közül ennek volt a legkevésbé stabil a tanulása.

## 5. fejezet

# Összefoglalás

Munkánkban a permutáció feladatát vizsgáltuk a mélytanulás eszközeivel. Ennek keretében két különböző feladatot állítottunk fel, mindkettőre külön-külön egy-egy architektúrát megadva, mely képes megoldást nyújtani a feladatra skálázható módon. Létrehozott architektúrákat összevetettük a state-of-the-art architektúrák képességeivel az adott feladatokon. Ezáltal igazoltuk, hogy architektúránk olyan kiegészítést adhatják a szekvencia-szekvencia hálóknak, amelyekkel azok új képességekre tehetnek szert.

A közvetett permutáció feladatának gyakorlati alkalmazásaként egy párhuzamos kutatásunkból merített alkalmazáson vizsgáltuk a Copying mechanizmus [2] kiegészítését a feladathoz készített architektúrával. Ennek eredményeként 40%-kal sikerült csökkenteni az architektúra által elrontott példák arányát, ezzel 91,7%-ra növelve az architektúra teljesítőképességét.

# Köszönetnyilvánítás

Köszönjük témavezetőink segítő útmutatását, illetve Bertalan Dániel Lászlónak, Czett Mátyásnak és Mészáros Botondnak az esettanulmányban való részvételt.

A Kulturális és Innovációs Minisztérium ÚNKP-23-2 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.



NEMZETI KUTATÁSI, FEJLESZTÉSI  
ÉS INNOVÁCIÓS HIVATAL

# Irodalomjegyzék

- [1] Dzmitry Bahdanau, Kyunghyun Cho és Yoshua Bengio. „Neural Machine Translation by Jointly Learning to Align and Translate”. (2015). Szerk. Yoshua Bengio és Yann LeCun. URL: <http://arxiv.org/abs/1409.0473>.
- [2] Jiatao Gu és tsai. „Incorporating Copying Mechanism in Sequence-to-Sequence Learning”. *CoRR* abs/1603.06393 (2016). arXiv: 1603.06393. URL: <http://arxiv.org/abs/1603.06393>.
- [3] Gergő Szalay és tsai. „Neurális kódvisszafejtés másoló mechanizmussal”. (2022).
- [4] Sepp Hochreiter és Jürgen Schmidhuber. „Long short-term memory”. *Neural computation* 9.8 (1997), 1735–1780. old.
- [5] KyungHyun Cho és tsai. „On the Properties of Neural Machine Translation: Encoder-Decoder Approaches”. *CoRR* abs/1409.1259 (2014). arXiv: 1409.1259. URL: <http://arxiv.org/abs/1409.1259>.
- [6] Ilya Sutskever, Oriol Vinyals és Quoc V Le. „Sequence to Sequence Learning with Neural Networks”. *Advances in Neural Information Processing Systems*. Szerk. Z. Ghahramani és tsai. 27. köt. Curran Associates, Inc., 2014. URL: <https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf>.
- [7] Bilal Chughtai, Lawrence Chan és Neel Nanda. „A toy model of universality: Reverse engineering how networks learn group operations”. *International Conference on Machine Learning*. PMLR. 2023, 6243–6267. old.
- [8] Yixuan Li és tsai. „Convergent learning: Do different neural networks learn the same representations?”. *arXiv preprint arXiv:1511.07543* (2015).
- [9] Rodrigo Santa Cruz és tsai. „Visual permutation learning”. *IEEE transactions on pattern analysis and machine intelligence* 41.12 (2018), 3100–3114. old.

- [10] Gonzalo Mena és tsai. „Learning latent permutations with gumbel-sinkhorn networks”. *arXiv preprint arXiv:1802.08665* (2018).
- [11] Ashish Vaswani és tsai. „Attention is All you Need”. *Advances in Neural Information Processing Systems*. Szerk. I. Guyon és tsai. 30. köt. Curran Associates, Inc., 2017. URL: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.

## Véletlen permutáció által helyesen megadott tokenek aránya

Vizsgáljuk meg, hogy egy adott  $n \in \mathbb{N}$  esetén az  $n$  hosszú elemenként különböző tokeneket tartalmazó szekvencia véletlenszerű permutálása esetén várhatóan hány elem marad a helyén. Jelöljük az  $n$  hosszú permutációban az összes jó helyen lévő token darabszámát  $a_n$ -nel. Amennyiben  $n = 1$ , úgy  $a_1 = 1$ , hiszen egyetlen módon lehet egy elem sorrendjét megadni, amikor is az egyetlen elem jó helyen van.

Tekintsünk egy  $n > 1$  számot és az általánosság megszorítása nélkül vizsgáljuk az első helyen lévő elemet. Ha az első helyen lévő elem a helyén marad, akkor a maradék  $n - 1$  elem esetén  $a_{n-1}$  jó helyen lévő elem fog szerepelni, emellett az első helyen lévő elem is jó helyen van, ez  $(n - 1)!$  különböző sorrend esetén mindig teljesülni fog.

Ha az első helyen lévő elem nincsen jó helyen, úgy ott egy másik tokennek kell lennie, ezt a tokenet  $n - 1$  féle képen választhatjuk meg. A maradék  $n - 1$  elem esetén  $a_{n-1}$  jó helyen lévő elem lesz, viszont ebbe belekerült az eredetileg első helyen lévő token is, ami itt már természetesen sose lehet jó helyen. Tekintve, hogy szimmetriai okokból az  $n - 1$  hosszú esetben minden token ugyanannyiszor szerepel jó helyen, így a maradék  $n - 1$  token esetén, mely tartalmazza az első tokenet is,  $a_{n-1} \cdot \frac{n-2}{n-1}$  jó helyen lévő elem van. A két eset lefedi az összes lehetséges esetet egy  $n$  hosszú sorozat esetén, és tekintve, hogy az első token alapján választottuk szét diszjunkt módon a két esetet, így az összegük megadja az  $a_n$  értékét:

$$a_n = a_{n-1} + (n-1)! + (n-1) \cdot a_{n-1} \frac{n-2}{n-1} = (n-1) \cdot a_{n-1} + (n-1)!$$

**Sejtés:**  $a_n = n!$

**Bizonyítás:**  $n = 1$  esetén  $1! = 1 = a_1$  teljesül. Tegyük fel, hogy  $n - 1 \in \mathbb{N}^+$ -ig minden pozitív egészre teljesül. Tudjuk, hogy  $n$  esetén  $a_n = (n - 1) \cdot a_{n-1} + (n - 1)!$ , itt használva az indukciós felvetést:

$$a_n = (n - 1) \cdot (n - 1)! + (n - 1)! = (n - 1 + 1) \cdot (n - 1)! = n \cdot (n - 1)! = n!$$

Így megkaptuk, hogy  $n$  esetén is teljesül, hogy  $a_n = n!$   $\square$

Ebből már meg tudjuk határozni, hogy egy véletlenszerű permutáció esetén várhatóan mennyi elem marad a helyén. Tekintve, hogy az összes lehetséges permutációk száma  $n!$  és a lehetséges permutációk esetén az összes helyén maradó tokenek száma

$n!$ , így a várható értéke a helyén maradó tokenek számának  $\frac{n!}{n!} = 1$  minden esetben  $n$ -től függetlenül.

## Hiperparaméterek

| Paraméter              | Értékek a különböző modellekben |        |           |        |
|------------------------|---------------------------------|--------|-----------|--------|
|                        | Közvetlen                       |        | Közvetett |        |
|                        | 1-10!                           | 10!    | 1-10!     | 10!    |
| Tanító adatok száma    | 103 515                         | 90 000 | 90 292    | 90 000 |
| Tesztadatok száma      | 11 502                          | 10 000 | 10 033    | 10 000 |
| Beágyazás mérete       | 128                             | 128    | 128       | 128    |
| Rejtett állapot mérete | 1024                            | 1024   | 1024      | 1024   |
| Tanítási ráta          | 0.0003                          | 0.0003 | 0.0003    | 0.0003 |
| Batch méret            | 4096                            | 4096   | 4096      | 4096   |
| Epochok száma          | 200                             | 200    | 200       | 200    |

1. táblázat. GRU Hiperparaméterei

| Paraméter              | Értékek a különböző modellekben |           |        |           |         |
|------------------------|---------------------------------|-----------|--------|-----------|---------|
|                        | Közvetlen                       |           |        | Közvetett |         |
|                        | 1-10!                           | 10!       | 20!    | 1-10!     | 10!     |
| Tanító adatok száma    | 102 960                         | 90 000    | 90 000 | 39 329    | 180 000 |
| Tesztadatok száma      | 11 440                          | 10 000    | 10 000 | 4370      | 20 000  |
| Beágyazás mérete       | 256                             | 128       | 256    | 256       | 256     |
| Rejtett állapot mérete | 1024                            | 1024 1024 | 1024   | 1024      | 1024    |
| Tanítási ráta          | 0.0003                          | 0.0003    | 0.0003 | 0.0003    | 0.0003  |
| Batch méret            | 1024                            | 512       | 512    | 1024      | 1024    |
| Epochok száma          | 200                             | 200       | 200    | 800       | 200     |

2. táblázat. *attention* [1] Hiperparaméterei.

| Paraméter              | Értékek a különböző modellekben |         |         |           |         |         |
|------------------------|---------------------------------|---------|---------|-----------|---------|---------|
|                        | Közvetlen                       |         |         | Közvetett |         |         |
|                        | 1-10!                           | 10!     | 20!     | 1-10!     | 10!     | 20!     |
| Tanító adatok száma    | 100 000                         | 100 000 | 100 000 | 100 000   | 100 000 | 100 000 |
| Tesztadatok száma      | 200 017                         | 200 000 | 200 000 | 209 600   | 200 000 | 200 000 |
| Beágyazás mérete       | 64                              | 64      | 64      | 64        | 128     | 128     |
| Rejtett állapot mérete | 512                             | 512     | 512     | 512       | 512     | 512     |
| Tanítási ráta          | 0.0001                          | 0.0001  | 0.0001  | 0.0001    | 0.0001  | 0.0001  |
| Batch méret            | 512                             | 512     | 512     | 512       | 2048    | 2048    |
| Epochok száma          | 200                             | 200     | 200     | 200       | 100     | 200     |

3. táblázat. Copying [2] Hiperparaméterei.

| Paraméter              | Értékek a különböző modellekben |        |           |        |
|------------------------|---------------------------------|--------|-----------|--------|
|                        | Közvetlen                       |        | Közvetett |        |
|                        | 1-10!                           | 10!    | 1-10!     | 10!    |
| Tanító adatok száma    | 152 640                         | 90 000 | 90 292    | 90 000 |
| Tesztadatok száma      | 16 960                          | 10 000 | 10 033    | 10 000 |
| Beágyazás mérete       | 256                             | 256    | 256       | 256    |
| Rejtett állapot mérete | 1024                            | 1024   | 1024      | 1024   |
| Rétegek száma          | 4                               | 4      | 4         | 4      |
| Fejek száma            | 12                              | 12     | 12        | 12     |
| Tanítási ráta          | 0.001                           | 0.001  | 0.001     | 0.001  |
| Batch méret            | 256                             | 256    | 256       | 256    |
| Epochok száma          | 500                             | 500    | 500       | 500    |

4. táblázat. Transformer [11] Hiperparaméterei.

| Paraméter              | Értékek a különböző modellekben |         |         |           |         |         |
|------------------------|---------------------------------|---------|---------|-----------|---------|---------|
|                        | Közvetlen                       |         |         | Közvetett |         |         |
|                        | 1-10!                           | 10!     | 20!     | 1-10!     | 10!     | 20!     |
| Tanító adatok száma    | 100 000                         | 100 000 | 750 000 | 400 000   | 100 000 | 100 000 |
| Tesztadatok száma      | 200 000                         | 200 000 | 250 000 | 9 464 100 | 200 000 | 200 000 |
| Beágyazás mérete       | 64                              | 128     | 128     | 128       | 128     | 128     |
| Rejtett állapot mérete | 32                              | 64      | 256     | 256       | 32      | 128     |
| Tanítási ráta          | 0.0003                          | 0.0001  | 0.0001  | 0.0003    | 0.003   | 0.0003  |
| Batch méret            | 4096                            | 4096    | 4096    | 4096      | 8192    | 4096    |
| Epochok száma          | 1000                            | 1000    | 1000    | 100       | 100     | 100     |

5. táblázat. PermNet Hiperparaméterei. A két feladatra két különböző architektúra lett kialakítva, melyek részletes leírásai a 2 fejezetben olvashatók.

| Paraméter              | Értékek a különböző modellekben |                   |         |
|------------------------|---------------------------------|-------------------|---------|
|                        | 40!                             | Közvetett<br>100! | Dict    |
| Tanító adatok száma    | 100 000                         | 100 000           | 100 000 |
| Tesztadatok száma      | 200 000                         | 200 000           | 200 000 |
| Beágyazás mérete       | 128                             | 128               | 128     |
| Rejtett állapot mérete | 512                             | 512               | 32      |
| Tanítási ráta          | 0.0001                          | 0.00003           | 0.003   |
| Batch méret            | 1024                            | 512               | 8192    |
| Epochok száma          | 100                             | 100               | 100     |

6. táblázat. PermNet Hiperparaméterei a skálázhatóságot vizsgáló tanítások esetén.