Eötvös Loránd Tudományegyetem

Természettudományi kar

Tanulás matematikai adatokon

Ujvári Máté

Szakdolgozat

Matematika BSc

Témavezető:

Zombori Zsolt

Budapest, 2025

Köszönetnyilvánítás

Szeretném kifejezni hálámat Zombori Zsoltnak és Darabos Dánielnek a számos értékes tanácsért és a támogatásért, amelyet a dolgozat elkészítése során kaptam. A szakmai iránymutatáson túl különösen nagyra értékelem segítőkész és biztató hozzáállásukat, amely jelentős szerepet játszott abban, hogy elmélyedjek a kutatási témámban.

Tartalomjegyzék

1. fejezet

Bevezetés

A természetes nyelvi feldolgozás (NLP) területén az utóbbi években jelentős áttörést hoztak a transzformer-alapú nyelvi modellek. A nagy nyelvi modellek (Large Language Models, LLM-ek), mint például a GPT, kiemelkedő teljesítményt nyújtanak számos nyelvi és logikai feladatban. Ezek a modellek azonban rendkívül nagy paraméterszámuknak és a sok tanító adatnak köszönhetik sikerességüket, nem pedig a hatékony, feladatspecifikus tanulásnak.

A formális nyelvek abban különböznek a természetes nyelvektől, hogy egyértelmű és gépek számára értelmezhető kommunikációra jöttek létre, így sokkal szigorúbb szabályrendszer jellemzi őket, és nyelvi struktúrájuk is jóval kötöttebb. A programozási nyelvek és matematikai kifejezések esetében rendelkezésre áll az úgynevezett absztrakt szintaxisfa (Abstract Syntax Tree – AST), amely épp az imént említett tulajdonságokról ad információt. Az AST egy fában jeleníti meg a szöveg logikai és szintaktikai felépítését.
Formális szövegeken finomhangolunk egy LLM-et (Large language model) - azaz a saját adatainkon tanítjuk tovább a már előtanított modellt -, és azt vizsgáljuk, hogy a bemenetek logikai felépítését használva sikerül-e javítani a teljesítményét. A programozási nyelvek például a formális nyelvek családjába tartoznak, melyek szövegeihez - a kódokhoz - létezik absztrakt szintaxisfa. Ezen kívül, a kódolás terén elért eredmények rendkívül jól hasznosíthatóak, így a kódkiegészítés, kódgenerálás izgalmas kutatási irányok lehetnek. Egy olyan kisebb feladatot választottunk, amelyhez könnyű tanító adatot szerezni, és amelyen ki tudjuk próbálni a logikai felépítés felhasználását.
Egyszerű aritmetikai műveletek megoldását szeretnénk hatékonyabban megtanítani a modellnek. Egy nagy modellen nehéz lenne kísérletezni, mert az óriási paraméterszám hosszú tanítási időt és rengeteg adatot igényel, egyébként pedig a legsikeresebb LLM-ek elég jól teljesítenek ilyen feldatokon, így talán kevésbé lenne kimutatható, ha sikeres a módszer. Emiatt a 135 millió paraméterrel rendelkező SmolLM2-t [3] fogjuk finomhangolni a saját adatainkon. (Viszonyításképp, a GPT-3-ban a paraméterek száma 175 milliárdra tehető [5].) A kérdés tehát, hogy egy kontrolláltabb tanítás során javul-e a modell általánosító

képessége.

A 2. fejezetben a transzformer architektúra alkotóelemeit és működését mutatom be. A 3. fejezetben a formális szövegek speciális tulajdonságairól és logikai felépítésük felhasználásáról esik szó. A kísérletek menetét, eredményeit a 4. fejezetben fogom ismertetni. Az 5. fejezetben pedig a kísérletek értékelésére kerül sor, emellett további lehetőségekről és lehetséges hibákról lesz szó.

2. fejezet

Háttérismeretek

A fejezet elkészítésekor az alábbi cikkekre és blogbejegyzésekre támaszkodtam: [20], [4], [11], [9], [14], [17], [22], [19]. Emellett felhasználtam Szemenyei Márton *Deep learning a vizuális informatikában* című jegyzetének 1., 2. és 5.3. fejezetét is [15].

2.1. Tanulási alapfogalmak

Tanuló eljárások

A tanuló eljárások valamilyen feladat megoldásához készített, paraméterekkel rendelkező modellek. A cél, hogy a paramétereket a tanítás során úgy változtassuk, hogy a lehető leghatékonyabban oldja meg a modell az adott problémát. Megfelelően nagy és jó minőségű adathalmazzal képesek vagyunk egyes feladatokon az emberi teljesítményt is felülmúlni . Ilyenek például a Go játék (AlphaGo), a sakk (AlphaZero), vagy a képosztályozás (ResNet, EfficientNet). Mindez azt jelzi, hogy egy adott feladatnál leggyakrabban nem az emberi tudás felhasználása, utánzása a cél, hanem a hibázás minimalizálása valamilyen optimalizáló eljárás segítségével.

Veszteségfüggvény (*Loss function*)

A veszteségfüggvény a modell teljesítményét méri, és a paraméterek a változói. Más és más módon vizsgáljuk a helyes válaszoktól való eltérést az egyes feladatokon. Regressziós feladatoknál például négyzetes eltérést szokás számolni, klasszifikációnál pedig az osztályokra adott valószínűségeket hasonlítjuk össze a várt válaszban szereplő osztállyal. A veszteségfüggvény minimalizálásával egy olyan modellhez jutunk, amely a várt válaszokhoz leginkább illeszkedő predikciókat adja. A minimalizálás gradiens módszerrel történik. A függvény gradiensével ellentétes irányba mozdítjuk a paramétereket, hiszen ebben az irányban csökken leginkább a függvény értéke. A lépés hossza egy hiperparaméter nagyságán múlik. A hiperparamétereket nem tudjuk optimalizálási módszerekkel

finomítani, mert ezek legtöbbször épp a módszer tulajdonságait, vagy a modell működését írják le.

Túltanulás (Overfitting)

Ezt a fogalmat a gépi tanulásban olyan esetekre használjuk, melyeknél a modell túl jól megtanulja a tanító adatokat, de általánosítani nem tanul meg, ezért rosszul teljesít új adatokon. Akkor szokott előfordulni, ha nincs elég adatunk ahhoz képest, hogy mennyire sok paramétere van a modellnek. Ilyenkor a komplex modell „megjegyzi" a bemenet-kimenet párokat, de egy új feladaton tanácstalan. Ez olyan, mintha a szorzótáblát megtanulnánk tökéletesen, de a szorzás technikáját nem értenénk, ekkor persze kétjegyű számokkal már nem számolnánk olyan meggyőzően. Jellemző, hogy amíg a tanulóadaton egyre jobb eredményt ér el a modell, addig a validációs adaton romlik a teljesítménye.

Epoch

Az epoch kifehezés a gépi tanulásban 1 iterációt jelent, amely során a modell minden adatpontot látott egyszer. A tanulási folyamat általában több epochból áll, hiszen az adatpontok többszöri megtekintésével precízebben tudja tanulni a modell az adathalmaz mintázatait. Az epochok száma is egy hiperparaméter. Ha túl sokszor látja a modell az egyes adatpontokat, könnyen túltanulhat a modell, ugyanis az általános tulajdonságok helyett megtanulhatja a tanítóadatokat.

2.2. A transzformer architektúra

A *transzformer* egy neurális hálózat architektúra, mely a modern tanuló rendszerek nélkülözhetetlen komponense. Egy ilyen modell bemenetként egy szövegszekvenciát kap, és kimenetként egy másik szekvenciát állít elő. Ezen szekvenciák elemeire *token*ként hivatkozunk. A modell belsejében levő paraméterek határozzák meg a bemenet és kimenet közötti leképezést, és ezen paramétereket mintaadatokhoz való illesztés segítségével hangoljuk be, ezt hívjuk *tanításnak*. Transzformereket rengeteg nyelvi feladathoz használtak sikeresen, mint például szöveggenerálás, fordítás, összefoglalás stb.

Vaswani és munkatársai [20]vezették be a transzformer architektúrát. Korábban rekurrens neurális hálózatokat (Recurrent Neural Network - RNN) alkalmaztak ilyen feladatokra. Egy RNN a bemeneti szekvencia tokenjei alapján próbálja megjósolni a kimenet következő tokenjét. A bemenet és kimenet korábbi részei egy állapotvektorban vannak elkódolva, így a modell képes a tokenek múltbeli kontextusát is figyelembe venni. Az állapotvektor karbantartása miatt az RNN a kimeneti tokeneket egyenként, egymás után tudja csak előállítani, így nem használja ki a GPU-k térnyerésével megjelenő párhuzamosítási lehetőségeket. Így a tanítás meglehetősen lassú lehet.

2.3. Komponensek

A fent említett cikkben az enkóder-dekóder struktúra felépítését mutatják be. Nagyvonalakban a következőképp működik. Az enkóder megkapja a tokenizált szöveget, ezt feldolgozza, értelmezi, és valós vektor reprezentációt állít elő, amely valamilyen módon tartalmazza a szöveg „lényegét". Tehát az enkóder által előállított vektorok megragadják az egyes tokenek jelentését és a szekvenciában betöltött szerepüket. Első ránézésre meglepő lehet, hogy vektorokkal képesek vagyunk megragadni olyan összetett és emberi jellegű fogalmakat, mint a jelentés vagy a kontextus. A transzformer viszont valójában nem érti a nyelvet, hanem mintázatokat tanul. Azt tanulja például, hogy egyes szavak gyakrabban fordulnak elő egymás mellett. A nagy számítási kapacitás, a hatékony működés és a sok tanuló adat teszi lehetővé, hogy képesek legyünk egy többszáz dimenziós vektortérben jelentést kódolni.

Ezután a dekóder megkapja az enkóder kimenetét, majd egy szekvenciát generál autoregresszíven: azaz a tokeneket egyenként állítja elő, mindig figyelembe véve a korábban generált tokeneket, amelyeket bemenetként használ a következő lépéshez. A tokenek generálása azt jelenti, hogy a dekóder egy valószínűségi eloszlást ad meg a tokenekre, amely alapján kiválaszthatjuk a legvalószínűbbet, vagy mintavételezhetünk is, ha diverzebbé szeretnénk tenni a generált szöveget. Ez a folyamat egy speciális - a szekvencia végét jelző token - az EOS (*End of Sequence*) token generálásáig tart, vagy amíg el nem érjük a maximális szekvenciahosszt.

Az önfigyelem(Self-Attention)

Ez a réteg nagy újítás az RNN-ekhez képest, itt szeretnénk a tokenek kapcsolatát, a kontextust megtanítani a modellnek. Minden input vektorhoz különböző lineáris rétegek kimeneteiként kapunk egy query, egy key és egy value vektort. Minden query vektort minden key vektorral összeszorozzuk, majd a kapott értékeket egy mátrixba rendezzük, tehát az így kapott A figyelemmátrixban $A_{i,j} = q_j * k_i$, ami az i. token j. tokenre fordított figyelmét méri.

$$A = QK^T$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{A}{\sqrt{d_k}}\right)V$$

Ahol az Q az a mátrix, melyet a query vektorok alkotnak, amíg K és V, a key és a value vektorokat tartalmazó mátrixok, Attention(Q, K, V) a figyelmi réteg kimenete, d_k pedig a key vektorok dimenziója. Nagyobb dimenziószám mellett nagyobb értékek kerülnek A-ba, így leosztjuk $\sqrt{d_k}$-val a softmax elvégzése előtt. A softmax normalizáló függvény egy $x \in \mathbb{R}^n$ vektort, valószínűségi eloszlássá alakít, tehát olyan [0,1]-beli számok alkotják az új vektort, melyek összege 1.

$$\text{softmax}(x) := \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}}$$

A mátrix sorait tekintjük vektoroknak, és ezeken végezzük a softmax függvényt, így minden sorban 1 lesz az értékek összege.

A gyakorlatban nem csak egy ilyen figyelmi mechanizmust alkalmazunk, hanem többet is egyszerre. Ezt nevezzük többfejű önfigyelemnek (*multihead self-attention*). Ilyenkor többféle nézőpontból is elemzi a rendszer a tokenek kapcsolatát, és az egyes fejekkel különböző tulajdonságait tanulja a szövegeknek.

Bemenet és beágyazás

A bemenetet tokenizáljuk, majd minden token kap egy $x_i \in \mathbb{R}^{|V|}$ vektort. Ezt a vektort *one-hot encoding* módszerrel állítjuk elő, azaz a szótárban k.-ként szereplő token vektora a következő lesz:

$$(x_k)_j = \begin{cases} 1, & \text{ha } j = k \\ 0, & \text{különben} \end{cases}$$

Az E $\in \mathbb{R}^{|V| \times d}$ mátrix segítségével transzformáljuk a vektorokat egy d dimenziós vektortérbe. $|V|$ a szókincs mérete (vocab size), d pedig a beágyazási dimenzió (embedding dimension).

$$\mathbf{e_i} = E[x_i] \tag{2.1}$$

E mátrix minden eleme egy-egy paraméter, melyeket a tanulás során változtatunk, hogy a folyamat végére, "okosabb" kezdeti beágyazáshoz jussunk. Vylomova és munkatársai képesek voltak a mátrixot úgy optimalizálni, hogy a szavak jelentésbeli hasonlósága megjelenjen a beágyazásban [21]. Például az a vektor, amely a *király* és *királynő* szavak beágyazásának különbségét reprezentálta, közel volt a *férfi* és *nő* szavak beágyazásának különbségéhez . Ez a statikus vektor, a one-hot vektorral ellentétben már tartalmaz valamilyen információt az adott szóról, viszont ebben a lépésben minden tokent külön-külön vizsgál a modell, így a kontextusról nem ad információt. Például, ha csak a *nő* szót látjuk kontextus nélkül, képtelenek vagyunk eldönteni, hogy éppen az igére, vagy a főnévre utal. Ezt az információt az önfigyelmi rétegek injektálják a vektorba.

Az önfigyelem egyáltalán nem használja fel a tokenek szekvenciabeli helyét. Például az aritmetikai műveleteknél ez probléma lehet, ugyanis a kétjegyű számokat számjegyenként tokenizáljuk. Nyilván nem mindegy, hogy 12-vel vagy 21-gyel számolunk. Az osztás nem kommutatív művelet, így itt is érdemes lenne tudni a tokenek sorrendjét. Tehát szeretnénk valamilyen módon megjeleníteni a tokenek reprezentációjában a sorrendet. A beágyazott vektorokhoz hozzáadunk egy-egy *PE (positional encoding*) vektort, amelyek

meghatározására a következő módszert használjuk:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \tag{2.2}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right), \tag{2.3}$$

ahol $0 \leq i \leq \frac{d}{2} - 1$, pos pedig a szekvenciában elfoglalt hely.

2.1. ábra. A hozzárendelt értékek a pozíció és a dimenziószám függvényében

Azt szeretnénk, hogy minden pozícióhoz különböző vektor tartozzon, korlátos értékekből álljanak a vektorok hosszú szekvenciák esetén is, és adott távolságban lévő pozíciókhoz tartozó vektorok különbsége legyen hasonló. A dimenziószám növekedésével csökkentjük a szinusz és koszinusz függvények frekvenciáját. Ha 1-től növekvő sorban vizsgáljuk a pozitív, egész számokat, azt látjuk, hogy az utolsó helyiérték minden lépésnél változik, az előtte lévő csak minden második lépésnél, és így tovább. Itt a [-1,1] intervallumból veszik fel az értékeket a függvények, és ahogy a 2.1-es képen is látszik, balról jobbra haladva, egyre lassabban változnak az egyes pozíciókhoz tartozó értékek.

Reziduális normalizáció

A következő két technikát mindig párban alkalmazzuk a transzformerekben, ezért együtt fogom kezelni ezeket. Réteg normalizáció alkalmazásával és átugró kapcsolatok létrehozásával is a tanulási folyamatot szeretnénk stabilabbá tenni. Mélyebb hálóknál megjelenő probléma, hogy az első rétegekben szinte egyáltalán nem változnak a súlyok. Ez azért van, mert gradiens számolásnál láncszabályt alkalmazunk, és ezzel sok 0 közeli értéket szorzunk össze. Az átugró kapcsolatokat egy x bemenethez f(x)-et rendelő

réteggel együtt alkalmazzuk.

$$y = x + f(x)$$

A bemenetet hozzáadjuk a réteg kimenetéhez, és azt szeretnénk elérni, hogy a modell a kívánt kimenet és a bemenet közti különbség előállítását tanulja meg. Abban az esetben amikor például $y \approx x$, könnyebb megtanulni, hogy $f(x) = 0$, mint azt, hogy $f(x) = x$. Ezt a módszert a ResNet [9] nevű neurális hálóban használták először, amely maga mögé utasította az addigi képfelismerő modelleket.

Adott az $x = [x_1, x_2, \ldots, x_n]$ vektor, ekkor a következő módon normalizáljuk:
Kiszámoljuk a szórásnégyzetet és az átlagot:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$$

A bemenetet normalizáljuk, hogy 0 átlagú, 1 szórású eloszlást kapjunk:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Ezután egy lineáris transzformációt hajtunk végre γ és β tanulható paraméterekkel:

$$y_i = \gamma \cdot \hat{x}_i + \beta$$

(ϵ egy kis konstans a 0-val való osztás elkerülése érdekében.)

Maszkolt önfigyelem

A transzformer tokenenként állítja elő a választ egy adott szekvenciához. A tanulási folyamatot jelentősen meglassítaná, ha minden egyes token után be kéne adni a helyes válasz soron következő szavát, így egyszerre megadjuk a teljes generálandó szekvenciát, és párhuzamosan állítjuk elő a tokeneket. Ekkor viszont minden pozícióból figyelhetnénk a későbbi tokenekre, amivel arra sarkallnánk a modellt, hogy a helyes válasz aktuális pozíciójában lévő tokenjét generálja. Élesben nem áll rendelkezésre a válasz, így ez egyértelműen zsákutca lenne a tanulás szempontjából. Vaswani és társai egy úgynevezett figyelemmaszk bevezetésével orvosolták a problémát:
Az M figyelemmaszk egy $n \times n$ méretű mátrix, ahol n a tokenek száma, és

$$M_{ij} = \begin{cases} 0, & \text{ha } j \leq i \\ -\infty, & \text{ha } j > i \end{cases}$$

Az M mátrixot hozzáadjuk A-hoz, így a softmax elvégzése után minden token 0 figyelmet fog fordítani az utána következő tokenekre, a korábbiakat pedig nem befolyásolja a maszk.

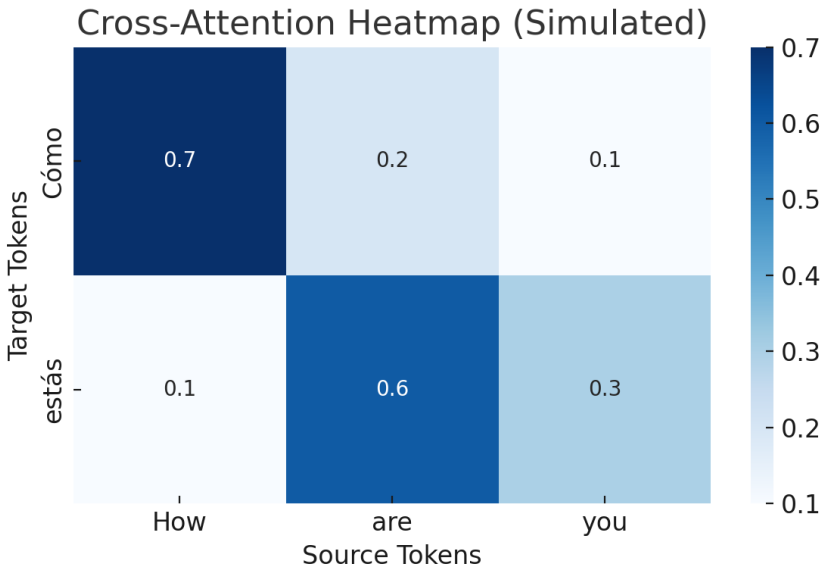
A szakirodalomban sokszor egyesekből álló alsó háromszögmátrixként hivatkoznak az alapértelmezett figyelemmaszkra. Ezzel valójában azt jelzik, hogy a 0-ás pozíciókra nem figyel a token, de számoláskor a fenti M mátrixot használjuk.

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{A}{\sqrt{d_k}} + M\right)V$$

A modelleket sokszor fixált méretű bemeneteken tanítják a számítási nehézségek elkerülése végett. A rövidebb szekvenciákat kiegészítik padding tokenekkel, melyeket tanítás során szintén kimaszkolunk a fenti módszerrel. Erre azért van szükség, mert ezek a tokenek nem hordoznak semmilyen információt, ezt viszont meg kellene tanulnia a modellnek, ami számítási kapacitás pazarlása lenne, másrészt a padding megtanulása torzítaná a veszteségfüggvényt.

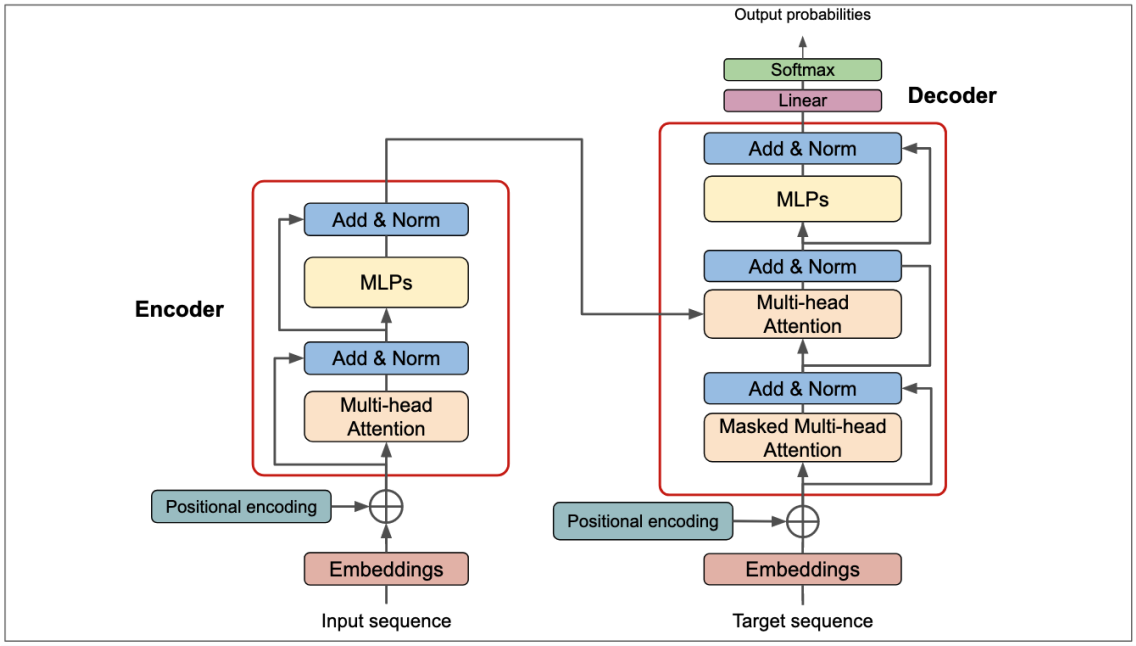
Keresztfigyelem(*Cross attention*)

A keresztfigyelem abban különbözik az önfigyelemtől, hogy a bemenet- és a kimenetszekvencia kapcsolatának megértését célozza. A V és K mátrixokat az enkóder kimenetéből állítja elő a modell, a Q vektor, pedig a maszkolt önfigyelemből érkezik. Innentől a réteg kimenetét ugyanúgy számoljuk, mint az önfigyelemnél. A 2.2. ábrán láthatjuk, hogy a generált tokenek az input tokenjeit figyelik, és megfigyelhetjük azt is, hogy a sorok összege 1.



2.2. ábra. Egy egyszerű példa szimulált figyelemmátrixszal.

2.4. Működés



2.3. ábra. Az enkóder–dekóder architektúrát bemutató ábra forrása a Deep Revision vizuális összefoglalója [16].

Az enkóder

Az enkóderben N enkóderblokkot alkalmazunk egymás után. A figyelmi rétegek mellett minden egyes blokk tartalmaz egy teljesen összekapcsolt, előrecsatolt (feed-forward) neurális hálót, melyet minden pozícióra külön-külön alkalmazunk. (A 2.3. ábrán MLP (Multilayer Perceptron) jelöléssel látható.) Mindkét réteg után alkalmazunk reziduális normalizációt. (Az ábrán "Add Norm")

Az előrecsatolt háló két lineáris transzformációból áll, közöttük egy ReLu [1] aktivációs függvénnyel:

$$\mathrm{ReLu}(x) = \max(0, x) \tag{2.4}$$

$$\mathrm{FFN}(x) = \mathrm{ReLu}(xW_1 + b_1)W_2 + b_2 \tag{2.5}$$

A W_1, W_2 súlymátrixok és a b_1, b_2 eltolások rétegenként eltérnek. Az első lineáris transzformációval általában egy nagyobb dimenziójú térbe transzformálunk, amelyből a második liníris transzformálás után kapunk egy, a bemeneti vektorral megegyező dimenziójú vektort.

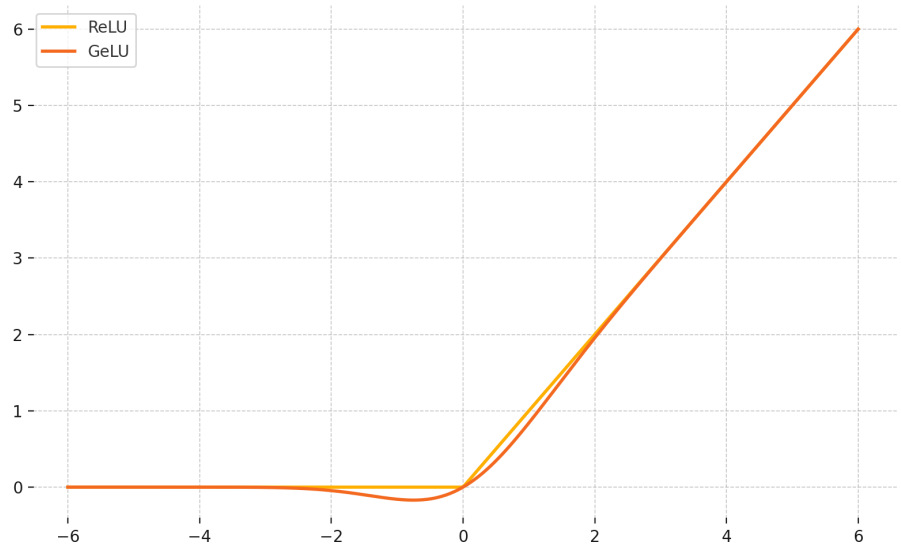
A sikeres nagy nyelvi modellekben ReLu függvény helyett GeLu-t (Gaussian Error Linear Units) használnak aktivációs függvényként [10]. Ezt a következő módon számol-

juk:

$$\text{GELU}(x) = x \cdot \Phi(x) \tag{2.6}$$

ahol $\Phi(x)$ a standard normális eloszlás eloszlásfüggvénye. Mivel az eloszlásfüggvénnyel nem tudunk gyorsan számolni, a következő közelítést használjuk a gyakorlatban:

$$\frac{1}{2}x\left(1 + \tanh\left(\frac{\sqrt{2}}{\pi}\left(x + 0.044715x^3\right)\right)\right)$$



2.4. ábra. A ReLu és GeLu függvények görbéi.

A ReLu függvény mélyebb hálókon való alkalmazásakor előfordul, hogy rengeteg neuron inaktívvá válik. Mivel a negatív x-ekre 0-t vesz fel a derivált és a függvény is, ezek a súlyok már nem változnak a tanulás során [2]. Ez a folyamat részben természetes, viszont ha egy jelentős része a paramétereknek nem tanul, az csökkenti a model kapacitását, és lassítja a tanulást. A 2.4. ábrán látszik, hogy a ReLu-val ellentétben, a GeLu $x < 0$ értékekre nem azonosan 0, és a deriváltja sem 0 minden negatív x-re. ezzel kisebb valószínűséggel hal el egy-egy neuron a tanulás során.

A beágyazás után már "érti" a modell a szavak jelentését, az önfigyelemmel a kontextust, a mondatban betöltött szerepet tesszük hozzá. Az *esős az idő* és a *szép az idő* szekvenciáknál másképp fogja reprezentálni az *idő* tokent az enkóder. A különbség még jelentősebb a többértelmű szavak eltérő reprezentációi között, mivel ezek esetében nem csupán a kontextus vagy a stílus változik, hanem a szó jelentése is. A modell számára ilyenkor a környező tokenek, vagy akár a teljes szekvencia alapján válik egyértelművé, hogy az adott szónak éppen melyik jelentése releváns.

A dekóder

A dekóderben is többször ismételjük a fő blokkot. Ez tartalmaz egy maszkolt többfejű önfigyelmi réteget, egy többfejű keresztfigyelmet és egy előrecsatolt hálót. Itt is alkalmazunk minden réteg után reziduális normalizációt. A blokk ismétlése után egy lineáris réteggel előállítjuk a logitokat, azaz egy szótár méretű vektort, így minden lehetséges tokenhez tartozni fog egy érték. Ebből egy softmax függvénnyel eloszlást készítünk, majd ennek segítségével kiválasztjuk a szekvencia következő tokenjét. Bár ismételjük a blokkokat, az alrétegekben lévő paraméterek eltérnek blokkonként, így más-más szempont szerint tanul a modell, ezzel pedig gazdagabb reprezentációt kapunk.

Decoder-only modellek

Az enkóder-dekóder modelleket leggyakrabban szövegátalakítási feladatokra, például fordításra, összefoglalásra használják. Szövegek, kódok kiegészítésében a decoder -only modellek bizonyultak hatékonyabbnak, többek között a GPT is egy ilyen architektúrán alapszik. Egy ilyen modellben csak a dekóder jelenik meg, de az is a keresztfigyelmi rétegek nélkül. A bemenete enkóder híján a tokenek beágyazása. Nagy előnyük, hogy nincs szükség tanításkor bemenet-kimenet párokra, mint az enkóder-dekóder modelleknél, így sokkal könnyebb a tanításukhoz adatot szerezni.

3. fejezet

Koncepció

A fejezet a [12]. blogbejegyzés és a [6]. könyv 1.6. fejezetének felhasználásával készült. ,

3.1. Formális és természetes szövegek

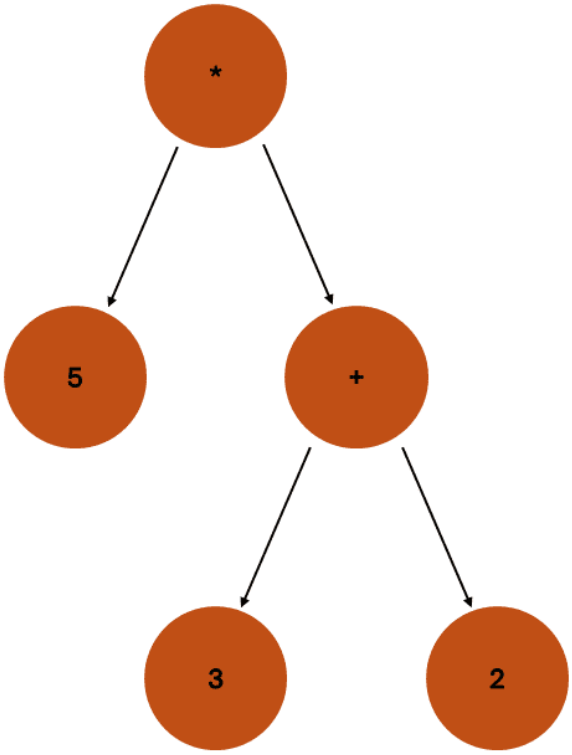
A természetes nyelvek spontán fejlődtek az emberi kommunikáció során. Évezredek alatt kulturális és társadalmi folyamatok hatására alakultak ki. A formális nyelveket mesterségesen tervezték meghatározott célokra, például számítási folyamatok leírására, logikai következtetések modellezésére, vagy tudományos definíciók precíz megfogalmazására. Ilyen nyelvek a programozási nyelvek, vagy a matematikai logika nyelve.

A természetes nyelvekben előfordulnak többértelmű kifejezések, mondatok, és sokszor csak a kontextusból, hangsúlyból, vagy valamilyen háttértudás alapján tudjuk megfejteni a valódi jelentést. Ezzel szemben, a formális nyelvek egyik legfontosabb jellemzője az egzaktság. Szigorúan definiált szintaktikai és szemantikai szabályok mentén működnek. Egy formális nyelv minden érvényes kifejezése egyértelműen értelmezhető, és ezek jelentése nem függ a kontextustól. Például egy programkódot csak akkor tud értelmezni a gép, ha minden utasítást ismer, és ezeknek pontosan egy jelentése van. Pythonban, a "print"Hello"", szintaktikailag helytelen parancs, ezért nem fut le a kód, amíg nem javítjuk "print("Hello")"-ra a hibás sort. Ha a "print("Hello")" parancsnak két jelentése is lenne, szintén falba ütközne a gép, hiszen nem hoz önállóan döntést, egyértelmű kell legyen számára, hogy mi a teendő az adott sor elolvasása után.

3.2. Az absztrakt szintaxisfa (AST)

Az AST, egy fa alapú adatstruktúra, melyben a formális szövegek logikai felépítését tároljuk. A csúcsok szintaktikai szerkezeteket, például utasításokat, kifejezéseket, függvényhívásokat vagy operátorokat reprezentálnak. Az „absztrakt” jelző arra utal, hogy a fa nem jeleníti meg a kód azon részleteit, melyek nem szükségesek a kód értelmezéséhez —

például a 5 * (3 + 2) kifejezés szintaxis fájában nincs zárójelezés, hiszen az AST a szöveg hierarchiáját is tárolja, tehát zárójelek nélkül is egyértelmű marad, hogy az 5-öt a 3 és a 2 összegével kell összeszorozni. A 3.1-es lépen az előbbi példa szintaxisfája látható:



3.1. ábra. 5 * (3 + 2) szintaxisfája. Az irányítás a hierarchikus felépítés miatt fontos.

3.3. Ötlet

A nagy nyelvi modellek sikeresnek bizonyultak természetes szövegekkel, és a formális nyelvek szabályait, felépítését is képesek megtanulni, de ehhez rendkívül sok erőforrásra van szükségük. Itt lenne érdemes felhasználni a bemenetek szintaxis fáját, hiszen ez éppen a szöveg struktúráját tárolja. Az AST beadásával kijelölhetnénk egy irányt a modellnek tanulás szempontjából, hiszen így direktebben kapna információt a szövegek felépítéséről, szabályairól.

Mivel az AST egy fa, kézenfekvő gondolat, hogy használjunk gráf neurális hálózatot(GNN) [23]. Ez a mélytanulási modell gráfokat kap bemenetként, és a szerkezeti összefüggések megtanulásával prediktál.

Ha a GNN-nél maradunk, szükséges az architektúra testreszabása. Az AST nem jeleníti meg a tokenek szekvenciabeli helyét, ami osztásnál, többjegyű számoknál probléma lenne. Ezt orvosolhatnánk a transzformereknél alkalmazott pozíció kód(Positional enco-

ding) használatával. Gráf neurális hálózatot osztályozás és regressziós feladatok mellett még rengeteg problémára lehet alkalmazni. Ezen a feladaton a hálónk kimenete egy eloszlás lenne a tokenekre, csakúgy mint a transzformereknél. Ezekkel a módosításokkal egy olyan gráf neurális hálózathoz jutnánk, ami kifejezetten hasonlít egy transzformerre. A transzformerek sikeressége pedig jól dokumentált, így érdemes lenne egyből erre indulni.

Használjunk egy olyan modellt, amely a formális kifejezést kapja meg bemenetként, az AST-ben tárolt információkat pedig használjuk fel a figyelem mechanizmusban. Az alapértelmezett, vagy kauzális figyelemmaszk, a padding és a későbbi tokenek "letakarására" szolgál, ehelyett az AST-beli szomszédság alapján készítjük el a maszkot. Egy token önmagára és a szintaxisfabeli gyerekeire figyel. Ilyen módon próbáljuk modellezni az AST-ben való felfelé haladást. A (3 + 4) * (1 + 2) kifejezés kiszámolásakor nem kell látnia a "3" tokennek a "2" tokent, hiszen nincsenek közvetlen kapcsolatban, a szorzásjel pedig elég, ha a (3 + 4)-ből és az (1 + 2)-ből származó információt kapja meg. Az lenne az ideális, ha a modell valóban megtanulná a megoldás módszerét, ahelyett, hogy szövegként kezelné a kifejezést, és a következő tokent próbálná kitalálni. Azzal, hogy rekurzívan számolunk, tehát csak akkor végezzük el a szorzást az előbbi példa megoldásánál, amikor az összeadásokat már kiszámoltuk, azt reméljük, hogy általánosabban tud tanulni a modell, hiszen így mindig egy műveleti jeles részfeladatokat kell kiszámolnia. Azt szeretnénk megtudni, hogy meg tudja-e tanulni a modell a megoldás lépéseit.

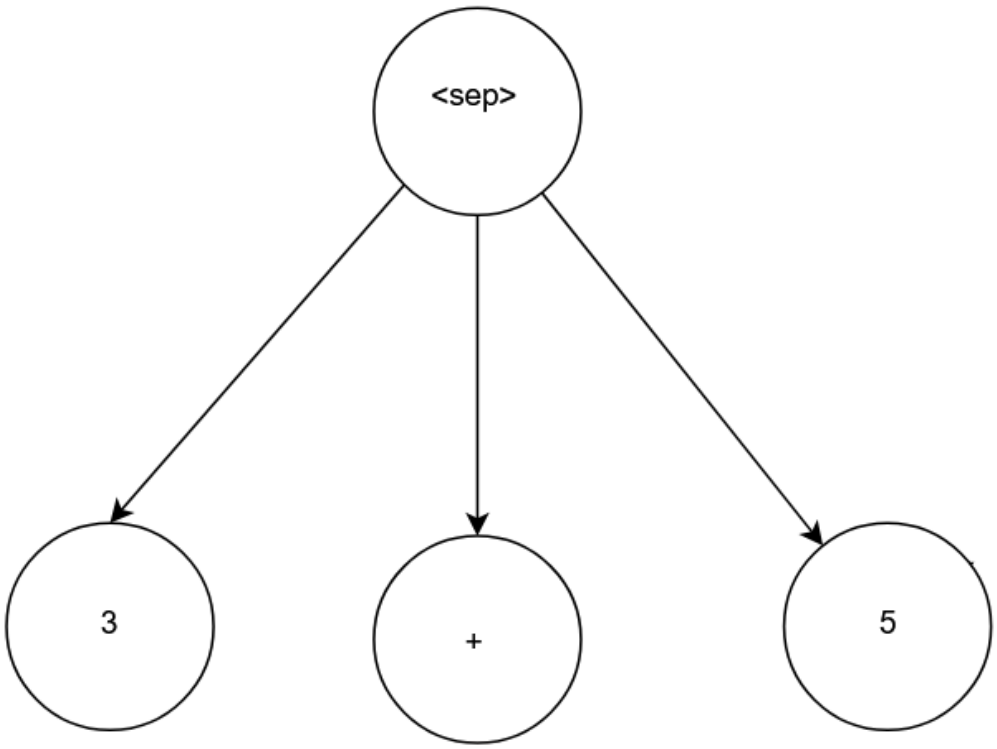
3.4. Tokenizálás

A szöveg feldarabolásába is érdemes lenne belenyúlni. Bár a szavak feldolgozása párhuzamosan történik, bizonyos szempontból a dekóderben az információ a szekvencia elejétől a vége felé áramlik, hiszen a tokenek reprezentációjának létrehozásakor csak az előttük lévő tokeneket használjuk. Az AST-ben a gyökér felé áramlik az információ, az alsó szintek megoldása után tudunk csak feljebb lépni, és kiszámolni a következő műveletet. A fenti példa tokenizálásánál az AST-ben nem szereplő zárójeleket is tokenizálnánk, és nem is lennének topologikus sorrendben a fa csúcsai. Az AST mélységi bejárása logikus választás lehet mindkét probléma megoldására, mert ha a befejezési sorrend szerint rendezzük a csúcsokat, akkor nem lesz olyan, hogy a v csúcsot u előtt tokenizáljuk úgy, hogy van uv irányított él a fában. Másrészt így a logikai szempontból felesleges tokenek - például zárójelek - nem kerülnek a tokenizált szekvenciába.

3.5. A figyelemmaszk beadása

Az AST-ben minden részfa gyökerét egy új tokennel helyettesítettük. Erre használjuk a <sep> jelölést. Egyrészt a többjegyű számokat számjegyekre darabolva a tokenizáljuk, ezeket szeretnénk összefogni a <sep> tokennel, másrészt, az egy operátoros részfeladato-

kat is egy-egy <sep> tokenhez kötjük, Például a 3 + 5 kifejezés szintaxisfájára eddig úgy tekintettünk, hogy a "+" gyerekei a 3 és az 5, innentől pedig a <sep> token gyerekei a 3, a + és az 5 lesznek.



3.2. ábra. 3 + 5 kifejezés AST-je kibővítve <sep> tokennel

Az AST átalakítása után tokenizálunk, majd előállítjuk a figyelemmaszkot a szekvenciához. Példaként a 31 + 52 ∗ 29 kifejezés tokenekre bontása és figyelemmaszkja (a 0-ás helyeket takarjuk ki):

['3', '1', 'sep', '+', '5', '2', 'sep', '*', '2', '9', 'sep', 'sep', 'sep']

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{pmatrix}$$

A figyelemmaszkot legtöbbször a padding és a későbbi tokenek letakarására használják, és a modellek nagy része egy szekvenciahossz méretű vektort vár maszkként. Így a felhasználóknak nem kell foglalkozniuk azzal, hogy előállítsanak egy mátrixot a szekvenciához, elég jelezniük, hogy mely tokenek részei a bemenetnek, és melyek szolgálnak paddingre. A beadott vektorból pedig automatikusan elkészül a kauzális maszk. A szakirodalomban 4D maszkként hivatkoznak arra a formátumra, amelyet mi is alkalmazni szeretnénk a figyelemmechanizmusban [18]. Gépi tanulásban az adatokat jellemzően nem egyesével, hanem *batch*-ekben dolgozzuk fel, vagyis az adathalmazt kisebb, azonos méretű részhalmazokra (batch-ekre) bontjuk. Egy batch minden egyes példájához külön készíteni kell annyi figyelemmaszkot, ahány feje van a többfejű önfigyelemnek. A fejek párhuzamosan dolgoznak, és nem szükséges, hogy ugyanazt a maszkot használják. A figyelemmaszk egy kétdimenziós mátrix, tehát a maszkok egy [batch méret$\times$fejek száma$\times$ szekvenciahossz $\times$ szekvenciahossz] méretű, 4 dimenziós tenzorba rendezhetőek.

Akadnak sikeres kísérletek a 4D maszk alkalmazásával. Például Miao és munkatársai memóriahatékonyabb módon tudtak szöveget generálni [13] a következő módosítással: az azonos kezdetű szekvenciákat egy szekvenciába fűzték össze, és a figyelemmaszkot úgy alakították ki, hogy a különböző szövegből érkező tokenek ne lássák egymást.
Például az *A kutya alszik*, *A kutya ugat* és az *A kutya eszik* szekvenciákból az *A kutya alszik ugat eszik* szekvenciát készítették el, és a figyelemmaszk segítségével kitakarták az alszik, ugat és eszik tokeneket egymás elől. Így az *A kutya* közös mondatrész csak egyszer kerül a memóriába, ugyanakkor ez a változtatás nem rontja a modell teljesítményét.

4. fejezet

Kísérletek

4.1. Környezet

A tanításhoz a Hugging Face transformers [8] könyvtárát használtuk. A Hugging Face nyílt forráskódú modelleket, adathalmazokat bocsát a felhasználók rendelkezésére, A kutatók és a fejlesztők saját modelljeiket, adathalmazaikat is feltölthetik a platformra. A transformers könyvtár előre tanított modelleket tartalmaz, például természetes nyelvfeldolgozáshoz, vagy számítógépes látáshoz, A könyvtár segítségével használhatjuk az előtanított modelleket, vagy finomhangolhatjuk őket saját adatainkon. A modellek előtanításkor rengeteg adatot, szöveget kapnak különböző területekről, de ahhoz, hogy egy specifikus feladaton jól teljesítsenek, sokszor szükség van a paramétereik finomhangolására az adott feladatnak megfelelő adatokon. Ez egy gazdaságos módja a teljesítmény növelésének egy-egy feladaton, mert a nulláról való tanítás nagyon költséges, és az előtanított modell valamennyire rátanul a nyelvi struktúrára, tehát jóval előrébb jár az előtanítás után, mint véletlen paraméterekkel. Például a SmolLM2-t webeoldalak mellett matematikai adatokon és kódokon is tanították, mégsem teljesít jól a minket érdeklő aritmetikai műveleteken. Az előtanított modell értelmes angol mondatokat generál, vagy a feladatban felfedezett mintát folytatja. Finomhangolás után azt megtanulja, hogy számjegyekből és tizedesvesszőből kellene állnia a válasznak, a kísérletek bemutatásakor pedig látni fogjuk, hogy nagyrészben el is találja a megoldást.

A 4.1. kódrészlet sorainak futtatásával generálhatunk szöveget a modellel. Telepítjük a Hugging Face könyvtárakat, betöltjük a modellt és a tokenizert. Az utóbbi jelenti azt a mechanizmust, amely a szekvenciát tokenekre darabolja, majd megkeresi azokat a modell szótárában, és hozzájuk rendeli a szótárbeli helyüket. A szekvencia feldarabolására és a szótár előállítására különböző technikák léteznek. Ha betöltöttük a modellt és a tokenizert, GPU-ra váltunk, mert rendkívül meggyorsítja a tanulást. Tokenizáljuk a szekvenciát, majd végigküldjük a rétegeken. Az utolsó lépést a kódrészletben az *encode*, *generate*, *decode* parancsokra hajtja végre a modell. (Itt az *encode* és a *decode* szavak

nem az enkóder-dekóder architektúra komponenseire utalnak, hanem egy token szöveges és tokenizált alakjai közötti átalakítás mechanizmusát jelölik.)

```
pip install transformers
from transformers import AutoModelForCausalLM, AutoTokenizer
checkpoint = "HuggingFaceTB/SmolLM2-135M"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
device = "cuda"
model = AutoModelForCausalLM.from_pretrained(checkpoint).to(device)
inputs = tokenizer.encode("Gravity is", return_tensors="pt").to(device)
outputs = model.generate(inputs)
print(tokenizer.decode(outputs[0]))
```

A kód a Hugging Face platformról érhető el (Hugging Face, 2024). [7]

A tanításhoz a transformers könyvtárból hívunk meg egy előre megírt tanító ciklust, ennek futtatása után pedig elmentjük a finomhangolt modellt az új súlyaival. Az, hogy a modell és a tanítás már meg van írva, rengeteg kódolást spórol meg a felhasználóknak, viszont a sok-sok állítható paraméter és módosítási lehetőség mellet is néhol rugalmatlannak bizonyult a felület. Például nem könnyű a modell rétegeibe belenyúlni, vagy a bennük végrehajtott számolások eredményét kinyerni.

4.2. Adathalmaz

A kísérletek elvégzéséhez először is szükségünk van egy adathalmazra, amin finomhangolhatjuk a modellt. Olyan kifejezéseket generálunk, amelyek 2 műveleti jelet, 3 számot és esetleg zárójelet tartalmaznak. Például (9/2) + 1.
Kezdetben egyjegyű számokat használunk, azt feltételezve, hogy az ilyen feladatokat hamarabb megtanulja a modell, később pedig legfeljebb kétjegyű számokra váltunk. A Python eval() függvényével ki tudjuk számolni a szintaktikailag helyes kifejezéseket, így minden feladathoz megvan a generálandó megoldás is.

Első lépésben, az volt a célunk, hogy megtanulja a modell az EOS tokent, vagyis hosszú mondatok generálása helyett álljon le néhány számjegy után. A legtöbb modell szótárában van egy speciális token kifejezetten a szekvencia végének jelzésére, ehelyett mi a „@@” tokent használtuk erre a célra. A modell betanulásakor valószínűleg nem volt nagy hatása ennek a tokennek, így ezt minden válasz végére odabiggyesztettük, és nem ütköztünk semmilyen nem várt viselkedésbe. Az adathalmazunkat ilyen párok alkotják:
"prompt": "(9 / 2) + 1 =", "response": "14.5@@"

Mivel a decoder-only modellek nem bemenet-kimenet párokon tanulnak, hanem egybefüggő szövegeken, a trl.SFTTrainer() tanító ciklust használjuk, amely egybefűzve kapja meg a promptot és a választ, de csak az utóbbin méri a hibázás mértékét.

4.3. A finomhangolt modell kipróbálása

Betöltjük a tanítás után elmentett paramétereket, és elkezdhetünk válaszokat generálni. Erre a műveletre a 4.1. kódrészletben bemutatott model.generate() függvény helyett a model() függvényt használjuk, ugyanis az előbbi nem támogatja a 4D maszk használatát. A model() kimenete tartalmazza a logitokat, amelyből már meg tudjuk határozni a generálandó tokent. A model.generate() függvénnyel ellentétben a model()-t tokenenként meg kell hívnunk addig, amíg el nem érjük a maximális szekvenciahosszt, vagy nem kapunk EOS tokent. Minden token generálásával bővül a szekvencia, így ilyenkor mindig új figyelemmaszkot kell készítenünk.
Egy $k \times k$ méretű mátrixot szeretnénk egy $(k + 1) \times (k + 1)$ méretűre bővíteni különböző módokon. Kauzális maszk esetén nem kell kitakarnunk semmit az új token reprezentációjának előállításakor, viszont az AST használatával megjelenik egy probléma. A teljes bemenethez nem tudunk AST-t rendelni. Például a 3 + 5 = szekvencia esetében, csak a 3 + 5 kerül be az AST-be, mert ez a kifejezés egyenlőségjellel együtt szintaktikailag helytelen. Mire figyelhet az egyenlőségjel és az utána generált tokenek? Az egyik megoldás, amit vizsgáltunk, hogy nem takarunk ki semmit, ekkor ugyanúgy bővítjuk a mátrixot, mint kauzális maszknál. A másik az, hogy az utolsó <sep> tokenre és az addig generált tokenekre figyel minden token az egyenlőségjeltől kezdődően, hiszen azt reméljük, hogy a fa gyökerébe gyűrűzik fel az információ. Ekkor a <sep> token reprezentációjából meg kellene tudnia tippelni a modellnek a feladat megoldását.

4.4. Eredmények

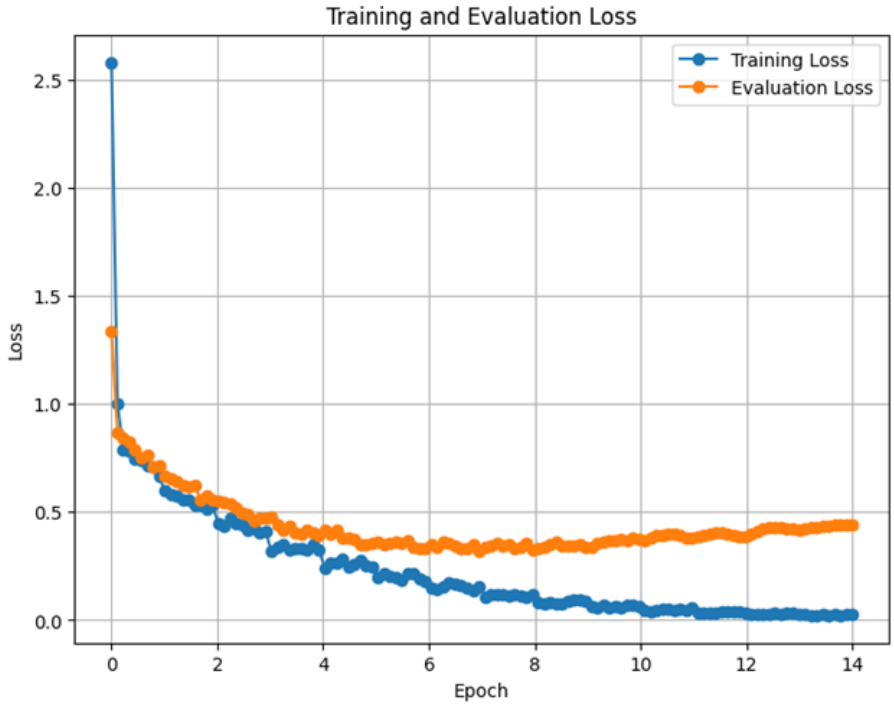
Először egyjegyű számokból álló, két műveleti jeles kifejezéseken finomhangoltuk a modellt. Az AST maszk előállításához a 3.5. szekcióban bemutatott módon kibővítjük a szintaxisfát, és ezzel a tokenizált szekvenciánk is hosszabb lesz. Ennek kapcsán megvizsgáltuk, hogy a szekvenciába nem illő tokenek megjelenése nem rontja-e az alapértelmezett kauzális maszkot használó modell teljesítményét. A 4.3. szekció végén említett két maszkolási módszer közül az elsőre hivatkozom majd AST maszkként. 10000 adatponton tanítva a következő eredmények születtek:

Maszk típusa	**Pontosság (%)**
Kauzális maszk	80,8
Kauzális maszk, <sep> tokenekkel bővített bemenethez	76,2
AST maszk	52,2

4.1. táblázat. Különböző figyelemmaszkokhoz tartozó modellpontosság. (A pontosság azt mutatja meg, hogy a modell az 1000 feladatot tartalmazó teszthalmazon a válaszok hány százalékában adott helyes megoldást.)

A kísérletből azt szűrtük le, hogy nem okoz nem várt viselkedést a szekvencia <sep> tokenekkel való bővítése. A pontosság csökkenése arra utalhat, hogy a modellnek azt is meg kell tanulnia, hogy az extra tokeneket nem kell figyelembe venni. A 4.1-es táblázatban látható eredményekhez 6-6 epoch tanulást futtattunk minden esetben. Később a második maszkkal 14 epochon keresztül tanítottuk a

modellt. Minél tovább tanítottuk, annál rosszab eredményt ért el a kiértékelésnél. A veszteségfüggvények görbéi (4.1. ábra) arra utaltak, hogy túltanulás történt.



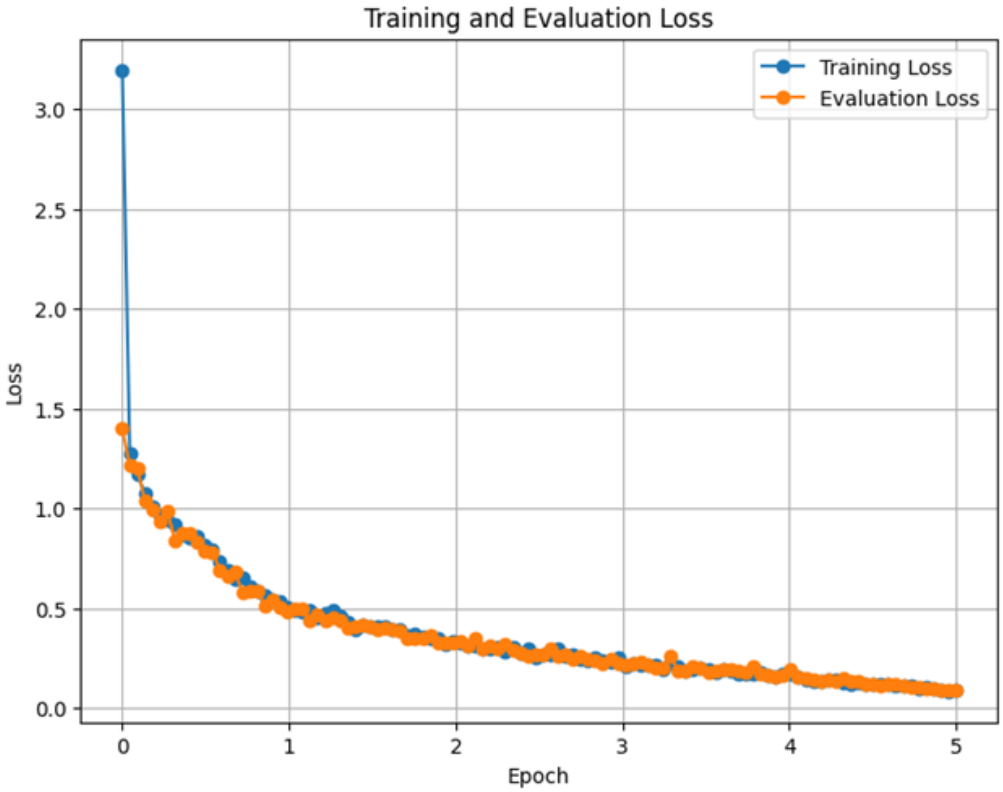
4.1. ábra. A tanuló adaton és a validációs adaton kapott veszteségek a tanulás során. Látható, hogy a validációs adaton kapott veszteség nőtt a 6. epoch után.

Azzal próbáltuk orvosolni a problémát, hogy 100 000-re növeltük az adatpontok számát. A nagyobb adathalmazzal elvégzett kísérletek mind hibásak, ugyanis ilyen aritmetikai műveletekből csak 64 000 különböző létezik. Ha a modellnek azokat a kifejezéseket kell megoldania, amelyeket a tanulás alatt akár többször is láthatott, biztosan jó teljesítményt fog nyújtani, még akkor is, ha nem tanult meg igazán jól általánosítani.

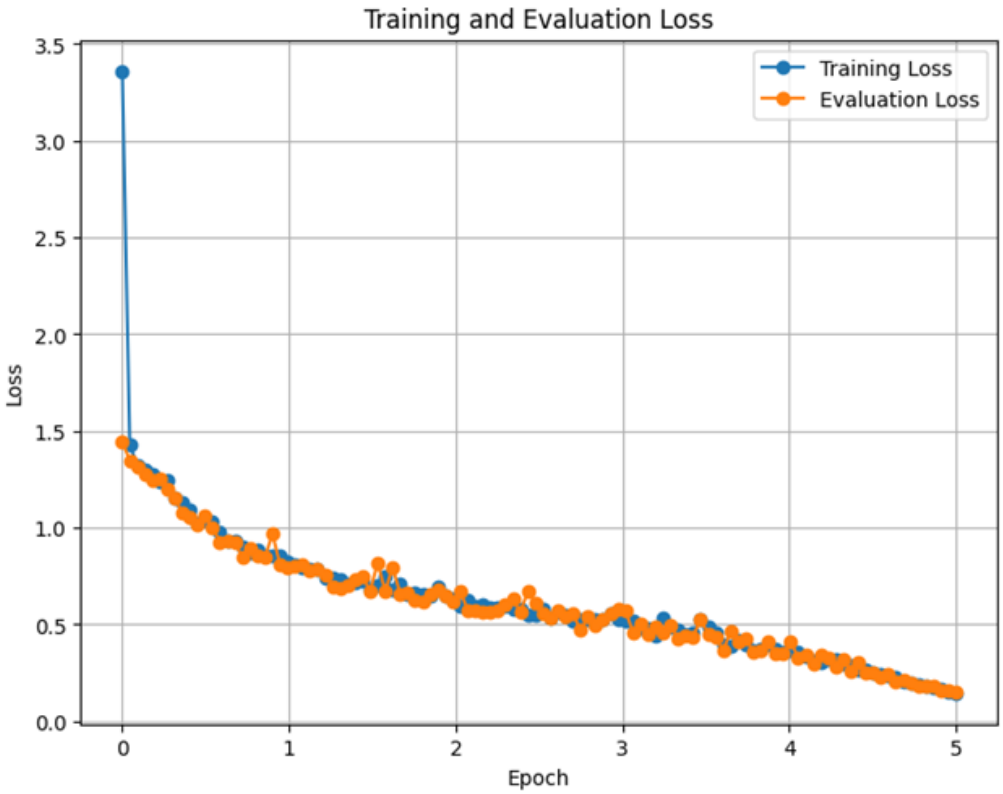
Innentől megengedtünk kétjegyű számokat is a kifejezésekben, és 50 000 adatponton tanítottuk a modellt. 3 epoch alatt a kauzális maszkkal is csak 16%-os teljesítményt ért el a modell, így egyszerűbbé tettük a feladatot azzal, hogy csak a "+" és a "-" műveleti jeleket használtuk a kifejezések generálásakor. A 4.3. szekció végén említett 2. maszkolási módszert is kipróbáltuk, ez lesz az "AST maszk 2", az "AST maszk 1" alatt az eddig használt AST maszkot értem a továbbiakban. A 4.2. táblázatban látható tanításokat 50 000 adatponton végeztük el.

Epoch	Kauzális maszk	AST maszk 1	AST maszk 2
3	64,1%	52,3%	22,7%
5	83,9%	70,4%	77,6%

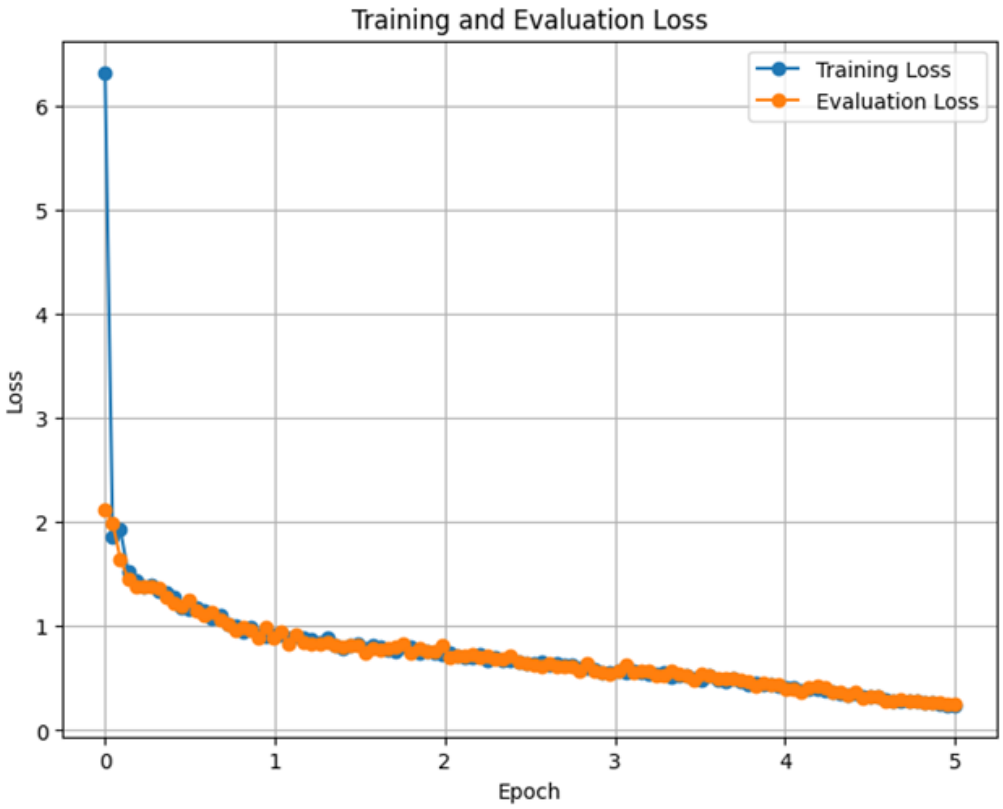
4.2. táblázat. A különböző maszktípusokhoz tartozó pontosság epochonként



4.2. ábra. A veszteségfüggvény alakulása kauzális maszkkal (5 epochig)



4.3. ábra. A veszteségfüggvény alakulása AST maszk 1 esetén (5 epochig)



4.4. ábra. A veszteségfüggvény alakulása AST maszk 2 esetén (5 epochig)

5. fejezet

Diszkusszió

A kísérletek azt jelzik, hogy az AST maszkkal való tanulástól nem javult a modell teljesítménye, mégis érdemes lenne megvizsgálni, hogy hosszabb tanítási ciklus során a különböző maszktípusokkal milyen végső teljesítményt tudunk elérni. A veszteségfüggvények görbéje (4.2., 4.3., 4.4. ábrák) alapján úgy tűnik, még egyik maszkkal sem érte el a maximális teljesítőképességét a modell. Ez a feladat persze sokkal több erőforrást igényel. Talán az adathalmaz méretét is változtatni kellene a túltanulás elkerülése végett, de legfeljebb kétjegyű számokból álló kifejezésekből már jóval több van, így sok, egymástól különböző feladat generálása nem jelentene akkora nehézséget.

5.1. További kísérletek, lehetőségek

A fejeken különbözőek a súlyok, így többféle mintát képes tanulni a modell, ez hatványozottan igaz lehet, ha különböző maszkokat használunk az egyes fejeken. A SmolLM2 modell 9 fejjel rendelkezik, így érdekes lenne vizsgálni, hogy miként változik a teljesítmény, ha 1-től növeljük az AST maszkos fejek számát.
A 4D maszk beadása a modell számára kétféleképpen valósítható meg. Vagy egy [batch méret $\times$ fejek száma$\times$szekvenciahossz$\times$szekvenciahossz] méretű tenzort adunk meg, vagy egy [batch méret$\times$ 1 $\times$ szekvenciahossz $\times$ szekvenciahossz] méretűt, ekkor automatikusan úgy kezeli, hogy minden fejen ugyanaz a maszk. Bár az előbbi tenzor elkészítése nem nehéz feladat, a vele végzett kísérletek során azt tapasztaltuk, hogy a tanulás sebessége jelentősen lelassul a korábbiakhoz képest.

Az AST maszkolás sokkal kevesebb információt bocsájt a modell rendelkezésére egy-egy token reprezentációjának elkészítéséhez, mint a kauzális maszkolás. Egy másik hasznos kísérletsorozat lenne, ha egyre kevésbé korlátoznánk a modell lehetőségeit a kimaszkolt tokenek számának fokozatos csökkentésével. Megengednénk a tokeneknek, hogy ne csak a gráfbeli gyerekeiket, hanem a gyerekeik gyerekeit, vagy akár minden leszármazottjukat láthassák. Azt vizsgálnánk, hogy melyik új figyelmi kapcsolat megjelenése von maga után jelentősebb javulást a teljesítményben, akkor közelebb kerülhetnénk a figyelmi mechanizmus és a maszkolás szerepének mélyebb megértéséhez.

5.2. Lehetséges hibák

Ha az AST-ben szintenként felfelé haladva számolunk, akkor a felsőbb szinten lévő csúcsok később kapcsolódnak be a folyamatba, mint például a levelek. A transzformerben persze sok-sok blokk van egymás után fűzve, így nem fordulhat elő, hogy nem "jutunk el" a gyökérig, de például a (3+5)+2 kifejezésnél a két összeadás művelet más mechanizmussal történik. Amíg az első rétegben a 3 + 5 <sep> tokenje egyből elkezdheti az összeadás feldolgozását, addig a második összeadáshoz tartozó <sep> token reprezentációjának előállításakor figyel egy olyan <sep> tokenre, amely reprezentációja nem tartalmazza még a hozzátartozó összeadást.

Probléma lehet, hogy megváltoztattuk a figyelemmaszkot az előtanításhoz képest. Ez a folyamat talán sokkal többet nyom a latba a modell működésének szempontjából, mint a finomhangolás, hiszen nagyságrendekkel több adaton és ideig történt. Felmerülhet, hogy használjuk a modellt előtanítás nélkül. Ekkor véletlen paraméterekkel töltenénk be a modellt, ezzel tiszta lappal indulva. Így az AST-maszk lenne az egyetlen maszkolási technika, amivel találkozik a modell, viszont ez egy óriási erőforrásigényű feladat, még egy kisebb architektúránál is.

Összefoglalás

Egy kisebb LLM teljesítőképességét vizsgáltuk aritmetikai műveletek megoldásán egy speciális figyelemmaszk alkalmazásával. Célunk az volt, hogy a bemeneti szekvenciák logikai struktúrájának felhasználásával "ösztönözzük" a modellt a matematikai műveletek hatékony tanulására, .

Dolgozatomban bemutattam a transzformer architektúra felépítését és működését. Ezután ismertettem a kifejezések szintaxisfájának felhasználási módját a tanulás során. Szót ejtettem az AST figyelemmaszkba történő leképzésének menetéről, és az így kapott AST maszk felhasználásának technikai részleteiről.
Bár az AST maszk használata nem eredményezett jobb teljesítményt, az implementáció működőképesnek bizonyult, és bőven maradt még izgalmas kísérlet, kiaknázatlan kutatási irány. A hosszú tanítási idő gyakran utunkat állta, és korlátozta a lehetőségeinket, viszont a kód kis módosításokkal alkalmas lehet nagyobb léptékű kísérletek elvégzésére is.

Irodalomjegyzék

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

[2] AIML.com. What is dying relu or dead relu and why is this a problem in neural network training?, 2023.

[3] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big – data-centric training of a small language model, 2025.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[6] Allen B. Downey. *Think Python: How to Think Like a Computer Scientist (2nd Edition)*. Green Tea Press, 2015. Accessed: 2025-04-21.

[7] Hugging Face. Smollm2-135m, 2024. [Large language model].

[8] Hugging Face. Transformers documentation. `https://huggingface.co/docs/transformers/index`, 2025. Accessed: 2025-05-27.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[10] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[11] Azeem I. Understanding resnet architecture: A deep dive into residual neural network, November 2023. Medium.

[12] Jessica Lopez. Basic understanding of abstract syntax tree (ast), 2020. Accessed: 2025-04-21.

[13] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiya-

an Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949. ACM, April 2024.

[14] Gabriel Mongaras. How do self-attention masks work?, October 2022. Medium.

[15] Szemenyei Márton. Deep learning alkalmazása a vizuális informatikában, April 2019. Lecture notes, Budapest University of Technology and Economics.

[16] Jean Nyandwi. The Transformer Blueprint: A Holistic Guide to the Transformer Neural Network Architecture. *Deep Learning Revision*.

[17] Sebastian Raschka. Understanding and coding self-attention, multi-head attention, causal-attention, and cross-attention in llms, January 2024. Ahead of AI.

[18] Ruslan S. 4d masks support in transformers, January 2024. Hugging Face Blog.

[19] Deepanshu Sachdeva. Understanding transformers step by step — word embeddings, June 2023. Medium.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[21] Ekaterina Vylomova, Laura Rimell, Trevor Cohn, and Timothy Baldwin. Take and took, gaggle and goose, book and read: Evaluating the utility of vector differences for lexical relation learning, 2016.

[22] Cameron R. Wolfe. Decoder-only transformers: The workhorse of generative llms, March 2024. Deep (Learning) Focus.

[23] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

Alulírott Ujvári Máté nyilatkozom, hogy szakdolgozatom elkészítése során az alább felsorolt feladatok elvégzésére a megadott MI alapú eszközöket alkalmaztam:

Feladat	**Eszköz**	**Hely**	**Megjegyzés**
Python kód generálása	GPT-4-o	A kutatás során	
Táblázat készítése	GPT-4-o	32. oldal, 4.1, 4.2	
Nyelvhelyesség ellenőrzése	GPT-4-o	Teljes dolgozat	
LaTeX szintaktika	GPT-4-o	Teljes dolgozat	
Képek generálása	GPT-4-o	2.1, 2.2, 2.4	

A felsoroltakon túl más MI alapú eszközt nem használtam.