EÖTVÖS LORÁND TUDOMÁNYEGYETEM

TERMÉSZETTUDOMÁNYI KAR

SZÁMITÓGÉPES NAPLÓÁLLOMÁNYOK ELEMZÉSE GÉPI TANULÁSI MODELLEKKEL

SZAKDOLGOZAT

Rétfalvi Bence

Matematika BSc

Matematikai elemző szakirány

Témavezető: Lukács András

Számítógéptudományi Tanszék

Budapest

2021

Köszönetnyilvánítás

Ezúton szeretném megköszönni témavezetőmnek, Lukács Andrásnak, akinek mindig támaszkodhattam szakértelmére és segítségére. Szeretném még megköszönni családomnak és barátaimnak, a töretlen támogatásukat.

Tartalomjegyzék

1. fejezet

Bevezetés

Mai világunkban az emberek egyre inkább támaszkodnak az informatika fejlődése által létrejött eszközökre. Társadalmunk működésének elengedhetetlen részét képezik a számítógépes szoftverek. Ezek a szoftverek mindennapjaink részévé váltak, lehetővé teszik, hogy biztonságosan tudjunk pénzt utalni, ha hívást vagy üzenetet indítunk, akkor a kapcsolat a megfelelő emberrel jöjjön létre. Szórakozás, kapcsolattartás, munka, sport, és egyéb területeken is támaszkodhatunk a számítógépes rendszerekre. Ha életünk ennyire meghatározó részét alkotják szükségünk van arra, hogy minden rendben működjön. Felmerülhet a kérdés, honnan tudhatjuk, hogy minden megfelelően működik, nem történt valamilyen rosszindulatú támadás, csalás, vagy egyéb rendszer hiba? A számítógépes rendszerek annak érdekében, hogy tudjuk mi történik a háttérben monitorozzák az eseményeket, és a megfigyeléseket napló fájlokba, úgynevezett log fájlokba gyűjtik. A log fájlok soronként értelmezhető, félig strukturált szövegek, tartalmuk függ az őket generáló szoftvertől. Tartalmazhatnak figyelmeztetéseket, hibákat és általános üzenetek is a szoftver állapotáról, az éppen futó folyamatról, vagy esetleg a hardverről, mint például videókártya hőmérséklet, vagy memória telítettség. Ha a rendszerünk valamilyen, az eddigi működésétől eltérő eseményt észlel, akkor az a log üzenetekből kiolvasható. A log üzenetek nagy szoftvereknél, mint például telefonközpontok elosztó rendszerében, operációs rendszereknél, hatalmas adathalmazok. Ezeket az adathalmazokat nehéz, általában lassú és költséges kizárólag emberi erővel vizsgálni, viszont a problémák észlelése kulcsfontosságú. Amennyiben egy rosszindulatú szoftver támadja a rendszerünket, fontos hogy azt azelőtt detektáljuk, mielőtt nagyobb kárt okozna. A log adathalmazok vizsgálatára használhatunk gépi tanulásos módszereket, amelyek használatával automatizálhatjuk az

anomáliák detektálását. A rendszerünk folyamatos ellenőrzése mellett kideríthetjük, a log üzenetek és a rendszer viselkedése közötti rejtett összefüggéseket.

A log üzenetekben történő anomália keresésnek két kritikus pontja van. Egyrészt egy általános módszert találni, amellyel automatizálható az a folyamat, hogy bármilyen rendszer logból képesek legyünk olyan adatot transzformálni, hogy az értelmezhető legyen gépi tanulásos módszereknek is. A másik kritikus pont pedig, a log adatok változó hosszúságú idősor jellege. Ahhoz hogy a rendszerünkben felismerjük a normálistól eltérő viselkedést meg kell tanulnia a gépi tanulásos modellnek, a sorrendiségben tárolt összefüggéseket. Szakdolgozatomban a log feldolgozásra bemutatok egy csoportosító módszert, amellyel a kód ismerete nélkül alakíthatunk át log adathalmazt vektorizálható formába, majd vektorizálom az adatot, hogy neurális hálókkal is értelmezhető legyen. A már feldolgozott adathalmazon, pedig a tudomány jelenlegi álláspontjának megfelelő mély neurális hálókkal végzek anomália keresést. Munkámban a több cikkben is kizárólagos adathalmaznak szolgáló HDFS log adatbázist [1] használom a módszerek kiértékelésére. A szakdolgozat jelentős része a 2019-ben Microsoft fejlesztők által publikált LogRobust cikk [2] mentén történik, kiegészítve a teljes folyamat megértéséhez elengedhetetlenül szükséges részletekkel, a módszer kisebb módosításaival, illetve egy másik jellegű módszer bemutatásával. Az általános egyetemi anyagra ismertként tekintek, így az ebben szereplő fogalmakat nem fejtem ki a szakdolgozat során.

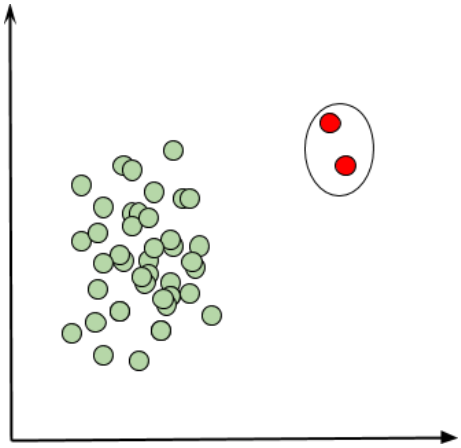
A dolgozat bevezető utáni, második fejezetében, bemutatom az anomália keresés problémakörét, kihívásait, illetve ismertetem a mély tanuláson alapuló anomália keresők alapjait. A harmadik fejezetben a log üzenetek ismertetése után, a log üzenetek feldolgozására és vektorizálására mutatok egy módszert. A negyedik fejezetben, bemutatom a kiválasztott anomália kereső modelleket, részletesen leírom működésüket. A modellek segítségével készített mérések az ötödik fejezetben olvashatók, itt a modellek paramétereinek hangolására, és az összehasonlítására is kitérek. Végül a hatodik fejezetben levonom a konklúziókat, összefoglalom a szakdolgozatban leírtakat.

2. fejezet

Anomália keresés, számitógépes naplófájlokban

Az anomália keresés egy olyan folyamat, ahol azokat az adat rekordokat, amelyek valamilyen módon az adathalmaz többségi részétől eltérnek meghatározzuk. Ezeket nevezzük anomáliának. Az anomáliáknak több tipusa van, vannak egyszerű otlier-ek, ezek egyszeri esetek, mérési hibák. Vannak tényleges anomáliák, ezek különböző nemvárt események hatására jönnek létre, és az egyes esemény típusok általában hasonló típusú anomáliákat képeznek. Egy anomália létrejövetel több okból is fakadhat, egyrészt valamilyen külső káros tevékenység hatására, hibából, de akár az ok lehet egy egyelőre még nem ismert folyamat felbukkanása is. Például, egy közösségi média profil feltörése anomáliát képezhet a bejelentkezési szokások adathalmazán, akár arra nézve, hogy hol jelentkeznek be, vagy esetleg, hogy a jelszót mennyiszer kísérlik meg beírni. A való életben kulcsfontossággú, hogy ezen anomáliákat detektáljuk, és az okokat kiderítsük. A bejelentkezéses példánál maradva, ha képesek vagyunk felfedezni, egy vélhetően illetéktelen bejelentkezést, akkor további autentikációs feltételeket kérhetünk a felhasználótól, hogy azonosítsa magát. Ezzel elkerűlhetjük az illetéktelen felhasználását a felhasználó adatainak.

Az anomália keresés egy fontos és nagy energiákkal kutatott aspektusa a adatbányászati feladatoknak. Éppen releváns módszerekre add betekintést jó néhány összefoglaló cikk, amelyek mély tanulásos és egyszerű gépi tanulásos eszközöket is használ [3, 4, 5]. Adathalmazok növekedésével, az anomáliák száma és a modellek számítási komplexitása is növekszik, így szükség van minél gyorsabb és pontosabb megoldásokra.



2.1. ábra. Az ábrán egy két dimenziós adathalmaz látható, a zöld pontok egy osztályból származnak, míg a pirosak outlier-ek.

2.1. Anomália keresés kihívásai

Az anomáliák megtalálásában több, általában jellemző kihívással kell szembenéznünk. Egy ilyen kihívás, az eltérő adatoknak a változó alakja. Egy anomália nem feltétlenül hasonlít egy másik anomáliára. Előfordulhatnak olyan anomáliák, amelyeket nem is figyeltünk még meg, így a kereső módszernek képesnek kell lennie, olyan adatokat anomáliának címkézni, amelyek nem hasonlítanak a többi anomáliára, viszont a normális adatra sem. Az adathalmaz elkészítésében problémát jelent, hogy az összes adatra nézve általában viszonylag kevés anomáliát találni, és az egyensúlytalan halmazokon jellemzően nehezebb tanítani. Az adathalmaz elkészítésben megoldás lehet, hogy elég nagy címkézett adatot hozunk létre, hogy annak csak egy részét használva egyensúlyt teremtsünk, de ez igen erőforrás igényes és nem mindig kivitelezhető.

Ha egy számítógépes rendszerben hiba történik, akkor a fejlesztőknek a log fájlok, amelyek információt tartalmaznak a programm futásáról, általában segítséget nyújtanak egy probléma megoldásánál. Ezeken a log üzeneteket végzett anomália kereséssel tudjuk, mi is megállapítani a rendszerünk nemvárt viselkedését, vagy megtalálni annak az okát. A számítógépes log fájlok elemzésénél tovább nehezíti a dolgunk, hogy a bejövő adat alapvetően egy magas dimenziós idősor, ahol számításba kell vennünk a log üzenetek érkezésének idejét is. Ezeknek a log üzeneteknek a feldolgozása szintén nehezíti a helyzetet, erre a 3. fejezetben térek ki részletesebben.

2.2. Mély anomália kereső módszerek

Ahhoz, hogy hatékonyan tudjuk kezelni az említett problémákat, mély anomália kereső módszereket fogunk használni. Jelölje X az adathalmazt, $x \in \mathbb{R}^d$ az adathalmaz egy elemét. Az anomália keresés első lépésében, megadunk egy $\phi(s) : s \to \mathbb{R}^z$ függvényt, ahol $z << d$. A ϕ függvény kimenete egy kisebb dimenziós reprezentánsa a bemenetnek. Ahhoz, hogy valamit anomáliának bélyegezzünk szükségünk van egy függvényre, amely egy anomália értéket készít egy reprezentánshoz. Az anomália érték készítő függvény $\psi(s) : s \to \mathbb{R}$. kimenete az anomália érték, ezt általában úgy határozzuk meg, hogy minél magasabb az érték, annál inkább tekinthető az adat rekord anomáliának.

Egy 2021-es anomália keresési módszereket összefoglaló cikkben [4], a mély anomália kereső módszereket három csoportra osztják kiértékelésük alapján. Az egyik csoportba tartoznak azok a megoldások, ahol egy mély hálóval elkészített kisebb dimenziós reprezentánsokat felhasználják, és egy ettől független módszerrel számolnak ezen reprezentánsok terében egy anomália értéket a rekordokhoz. Ez a módszer lehet akár bármilyen gépi tanulásos módszer. Másik csoportba tartoznak azok a modellek, amelyek a kis dimenziós reprezentánsból közvetlenül számolnak egy anomália értéket valamilyen függvény segítségével, illetve vannak az úgynevezett „end-to-end” megoldások, amelyeknél egy teljesen összefüggő mélyháló készíti el a becsléseket. Mivel idősorokkal dolgozunk rekurrens hálókat fogok alkalmazni az adatok időbeli összefüggéseinek megtanulásához.

3. fejezet

Log feldolgozás

3.1. Log üzenetek

A log üzenetek félig strukturált szövegek, a jellegük változhat a IT rendszertől és szoftvertől függően. A rendszerben egy folyamat futásának hatására keletkeznek, a rendszer állapotát közvetítik. Az üzenet lehet figyelmeztető, hiba üzenet, vagy egyéb üzenet arról, hogy milyen folyamatok milyen eredményekkel futottak le. Tartalmazhatnak hardver állapot információkat is, mint például memória kihasználtsága, vagy a processzor hőmérséklete. Jellemzően tartalmaznak idő, identifikáló bélyeget, illetve egy azonosítót milyen folyamat során jött létre az üzenet. Mivel a log üzenetek páronként eltérőek, így nem tudunk sima nyers log üzeneteken tanítani egy modellt, ehhez előfeldolgozásra lesz szükség.

```
20264 INFO dfs.DataNode$DataXceiver: Receiving block blk_813542614119066696 src: /10.251.106.50:40068 dest: /10.251.106.50:50010
20264 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_-2159450366950106948 terminating
20264 INFO dfs.DataNode$PacketResponder: Received block blk_-2159450366950106948 of size 67108864 from /10.251.201.204
20265 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_1106692684454494876 terminating
20265 INFO dfs.DataNode$PacketResponder: Received block blk_1106692684454494876 of size 67108864 from /10.251.202.209
20278 INFO dfs.DataNode$DataXceiver: Receiving block blk_2165197841358336737 src: /10.251.202.209:54773 dest: /10.251.202.209:50010
```

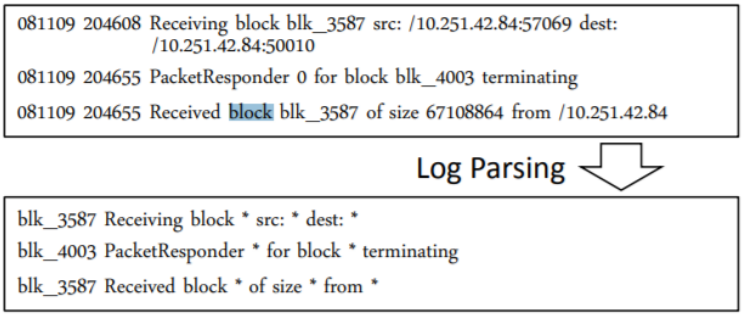
3.1. ábra. Log üzenetek az általam használt HDFS adathalmazból

Az előfeldolgozásnak esetünkben két lépése van. Első lépésben a log üzeneteket csoportosítjuk hasonlóság szerint. Két log hasonlóságát azon részei határozzák meg, amelyek állandó jellegűek, ezek általában szövegek, és követi őket egy változó rész, amely attól függően mikor, milyen folyamat részeként jött létre változik. Az állandó részt szavakra bonthatjuk, ezeket a szavakat hívjuk konstansnak. Eseménynek pedig egy log üzenet sémát hívunk, amelyhez a sémához hasonló log üzeneteket csoportosíthatjuk. A hasonlóságot később definiálom. Második lépésben pedig ezeket az eseményeket vektorizáljuk,

és összegyűjtjük egy azonosító szám mentén, így megfeleltethetünk minden azonosítónak egy vektor listát, ezt a listát a modellünk már képes lesz értelmezni.

3.2. Log üzenet csoportositás

Több log üzenet csoportosító módszert különböztetünk meg, vannak olyan módszerek amelyek az eseményeket a logot generáló kód ismeretében készítik el, a forráskódból kinyerik a megfelelő információkat egy log esemény létrehozásához [6]. Ezzel a módszerrel alapvető probléma, hogy függenek az adott programozási nyelvtől, illetve valós esetekben gyakran nem áll rendelkezésünkre az egész forráskód. A módszerek egy másik csoportja, amelyek nem igénylik a forráskód ismeretét, csak a nyers logokat. Én a Drain algoritmust [7] fogom használni, amely az utóbbi módszer családra egy példa.



3.2. ábra. Log parsolás példa

Habár a Drain algoritmus kizárólag nyers logon dolgozik, ennek ellenére a paraméterek behangolásával, egészen pontos leképezést tud képezni a logokból. Több frissebb log elemzéssel foglalkozó cikkben is a Drain-t használják [8, 2] a log üzenetek csoportosításához.

Drain

Első körben, szükséges megvizsgálnunk a log üzeneteket, hogy vannak-e olyan részei, amelyek nem kizárólagosan szám jellegűek, viszont nem tekinthetők konstansnak, ilyenek például a betűkből és számokból álló identifikáló bélyegek, blokk számok. Amennyiben vannak ilyen kifejezések, azokat reguláris kifejezések segítségével töröljük. Általában egy-két ilyen reguláris kifejezéssel elérhetjük hogy a csoportosítás használható eseményeket képezzen az adathalmazból, természetesen ez függ a log adathalmazunktól. Az üzenet szám

típusú részeit * -ra cseréljük, hogy egy log sor kizárólag konstans elemeket, illetve * karaktereket tartalmazzon, ez teszi összehasonlíthatóvá a log üzeneteket. Következő lépésben csoportokba soroljuk a logokat, az előző lépésben kapott alakjuk alapján. Mivel minden log üzenet összehasonlítása a többi üzenettel túlságosan időigényes lenne, egy fa struktúrát használunk a csoportok létrehozásához. A fának a gyökér illetve belső csúcsai nem tartalmaznak log csoportokat, ezeknek a csúcsokban szabályok vannak, amelyek szerint küldhetjük a log adatokat tovább a megfelelő gyerek csúcsba. A fa leveleibe a log csoportok listája kerül. Egy log csoport két részből áll, egy log eseményből és az eseményhez tartozó log sorok kulcsaiból. Az eseményeket úgy hozzuk létre, hogy minden egyes üzenet besorolható legyen egy esemény alá. A fa szerkezetét két paraméter határozza meg, a fa mélysége, amely azt határozza meg, milyen mélyre építhetjük a fát, illetve a maximum gyerekszám, amely azt hogy egy csúcsnak maximum mennyi gyerek csúcsa lehet. Ezen paraméterekkel szabályozhatjuk, hogy ne szaporodjanak el a log események túlságosan.

A fa felépítése

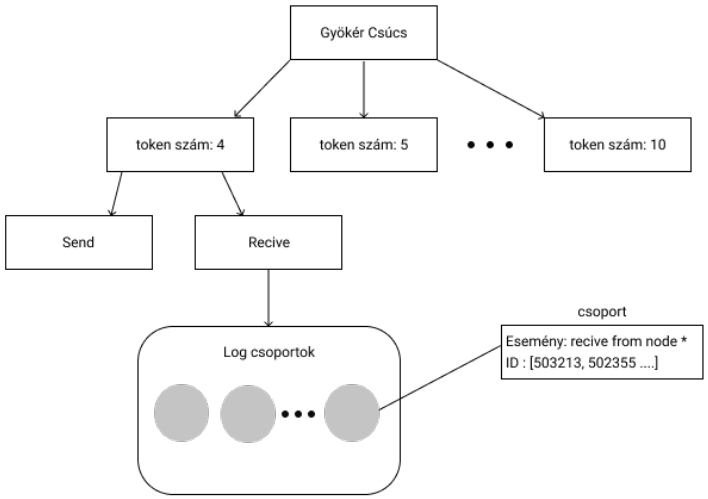
Első szinten a beérkező log üzeneteket aszerint küldjük tovább egy ágba, hogy milyen hosszúak, azaz hogy hány tokent tartalmaznak. A token egy konstans szó a log üzenetben. Amennyiben egy a többitől eltérő hosszúságú sor fut be, létrehozunk egy új gyerek csúcsot. Második szinten, a log üzenet első tokenje alapján küldjük tovább az adatot, ha megegyeznek a tokenek akkor ugyanoda kerülnek, ha még nincs ilyen token létrehozunk egy új gyerek csúcsot. A következő szinteken sorban a tokenek szerint csoportosítunk, pl.: a harmadik szinten a második token alapján, ahogyan a második szinten. A fa leveleiben alakulnak ki a log csoportok. Ezen szinten ahhoz, hogy a logokat valamilyen csoport eseményébe besoroljuk, szükségünk van egy hasonlósági metrikára a log üzenet és a log esemény között. Ennek a metrikának a kifejezéshez szükségünk van az equ függvényre amelyet a következőképpen definiálunk:

$$equ(a;b) = \begin{cases} 1, & \text{ha } a = b; \\ 0, & \text{egyébként:} \end{cases}$$

Ennek segítségével a hasonlóság a következőképpen írható le:

$$\text{simSeq} = \frac{\sum_{i=1}^{n} equ(l(i); e(i))}{n}$$

Itt $l(i)$ a beérkező log sor i-edik tokenje, $e(i)$ pedig egy esemény i-eidk tokenje, n pedig $min(jlj; jej)$, ahol jlj, jsj a log sor illetve az esemény token számát jelentik. A Drain módszer egyik hiperparamétere egy hasonlósági döntési határ, amennyiben a simSeq metrika egy eseményre, és egy log üzenetre ezt a döntési határ értéket meghaladja, akkor a log üzenet hasonló az eseményhez. A log üzenet leérkezik, ha nincs még csoport, vagy ha nem hasonlít egyetlen log csoport eseményére sem, akkor létrehoz egy csoportot. Amennyiben talál hasonló log eseményt, akkor bővíti az eseményhez tartozó csoport kulcsait a saját kulcsával, illetve az esemény azon változóit, amelyek nem egyeznek meg a beérkező loggal egy * ra cseréli. Fontos megemlíteni, hogy az algoritmus kizárólag azokat a token egyezéseket veszi figyelemben, ahol a log sorban elfoglalt hely is megegyezik.



3.3. ábra. Drain fa felépítése

Drain algoritmus csak a szöveg alapú tokeneket vizsgálja, a számokat egy *-al helyettesíti, illetve a maximum gyerekszámot meghaladó csúcsoknál, a következő többiektől eltérő tokent szintén *-al helyettesíti.

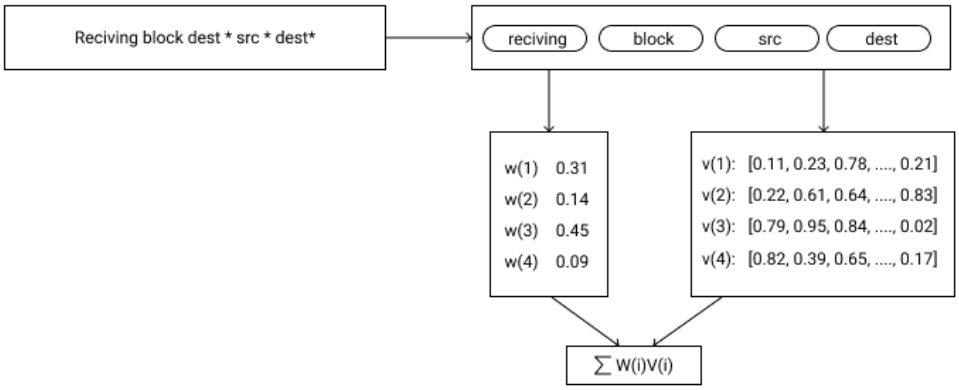
3.3. Log vektorizálás

Mivel a mély neurális hálókat kizárólag numerikus adattal tudunk tanítani, így a log vektorizálás a folyamat egy kihagyhatatlan része. A log feldolgozás ezen fázisában fognak elkészülni a már csoportosított logokból a numerikus vektor típusú adat reprezentánsok. Itt több megoldással találkozhatunk, a legkézenfekvőbb, hogy eseményenként one-hot-encoding módszerrel készítünk reprezentációt. Ennek jól látható hátránya, hogy nem képes az események közötti összefüggések eltárolására, illetve hogy ha a log üzenetek között egy az eddigiektől eltérő esemény keletkezik újra kell tanítanunk a modellt, ami valós problémáknál alkalmazva nem biztos, hogy megengedhető. Alternatív megoldást kínál egy cikkben mutatott módszer [2], amely a log üzenetek szavainak szemantikáján kereszül próbálja értelmezni őket, ezt mi is részletesebben fogjuk tárgyalni.

Szemantikus vektorizáció

Első körben a log eseményeket kell átdolgozni, a vektorizáláshoz. Célunk, hogy a log eseményekre konstanstonként létrehozzunk egy vektort, majd ezeket aggregáljuk az adott esemény mentén. Mivel szeretnénk, hogy a szemantikailag hasonló konstansok hasonló vektort kapjanak, egy előre elkészített szó beágyazást fogunk használni. Ezeket a beágyazásokat szintén gépi tanulásos módszerekkel hozzák létre, olyan modellekkel mint a word2vec [9] vagy a GloVe [10]. Ezen modellek működésére a szakdolgozatban nem térek ki. Ahhoz hogy ezeket a beágyazásokat alkalmazni tudjuk a log esemény konstansain, szükségünk van arra, hogy egy esemény minden konstans eleme egy értelmezhető angol szó legyen. Gyakran viszont a konstansok változó nevek mint például “isTrue” vagy nem szóközzel elválasztott kifejezések. Ezen korlátok áthidalására formázzuk az eseményeket, hogy minden konstans értelmezhető angol szó legyen, amelyek nem bírnak jelentéssel kivesszük, az összekötött szavakat szétszedjük.

Az események konstansainak értelmezhető angol szóvá való átalakításával olyan szövegeket kapunk, amelyet át tudunk alakítani vektor listává, az egyes szavak beágyazását vektorként használva. Beágyazásnak, egy a Wikipédiát, és a Gigaword5 [11] adatait felhasználva tanított GloVe model 100 dimenziós vektorait használtam. Így egy eseményt egy vektor listává alakítottuk, amely olyan hosszú mint az eseményben szereplő konstansok száma, illetve minden vektora 100 dimenziós.



3.4. ábra. Szemantikus log vektorizáció

A következő lépésben az eseményeket egyetlen vektorrá aggregáljuk, és így minden eseményre képezünk egy beágyazást. Egy esemény beágyazás kiszámítása a következő:

$$E = \frac{\sum_{i=1}^{n} (w(i) \quad v(i))}{n}$$

Ahol n egy esemény hossza, $v(i)$, az esemény i-edik szavának beágyazott vektora, $w(i)$ pedig a TF-IDF (term frequency-inverse document frequency) súlya az esemény i-edik szavának. TF-IDF definiálása:

$$TF(x) = T(x)=T$$

$$IDF(x) = log(L=L(x))$$

Itt T az események hosszának (konstans elemeik száma) az összege, $T(x)$ pedig az x konstans előfordulásának a száma, az egész esemény halmazon. L a log eseményeknek a száma, $L(x)$-el pedig azt jelöljük mennyi log eseményben fordul elő az x konstans. Mivel a vektorizálás után a log események vektorainak értékei több nagyságrendben eltérhetnek, a nagyobb abszolút értékkel rendelkező elemek elnyomhatják a többit, így a model hajlamos lesz csak ezekre hagyatkozni a tanulás során. Később szó lesz a gradiens leereszkedés módszeréről, és a módszerrel való gyorsabb tanulás érdekében is szükséges, hogy a változók hasonló eloszlásból származzanak. A felsorolt problémák megoldására standardizálni fogjuk a log eseményeket. Egy log esemény standardizálása a következő módon történik:

$$E_{standard} = \frac{E \quad u}{}$$

Ahol E az esemény aggregált vektora u az E elemeinek számtani átlaga és a szórása. A transzformáció után egy esemény vektor szórása egy lesz és az átlaga nulla, így centralizálódik a vektor, ha a bemenet normális eloszlásból származik, akkor az eredmény standard normálisból fog.

Miután az összes eseményt vektorizált és standardizált alakra hoztuk, a log üzeneteket a folyamat azonosító számuk alapján szétválogatjuk, és rendezzük az időbélyeg alapján növekvő sorrendben. Az így kapott azonosítókhoz tartozó log üzenet listának az elemeit kicseréljük arra az eseményre, amelyhez a Drain algoritmus rendelte őket. Az így kapott esemény listának, pedig az elemeit ki tudjuk cserélni, az esemény vektor típusú beágyazására. Így minden azonosítóhoz rendeltünk, egy listát melynek elemei vektorok. A lista hossza megegyezik az azonosítóhoz tartozó log üzenetek számával, mivel a log üzenetek száma nem állandó egy azonosítóhoz, ez egy változó hosszúságú lista lesz. Az elemei állandó hosszú vektorok, a szavak mérete GloVe reprezentáció méretétől függ. A létrehozott azonositóhoz tartozó szekvenciákkal pedig elkezdhetjük tanítani modellünket.

4. fejezet

Modellek

Mint ismertettem a 2-dik fejezetben, mély hálós modelleket fogok alkalmazni az anomália keresés megvalósítására. Egy a LogRobust [2] cikkben bemutatott modellt, illetve egy autoenkódert [12]. Mivel az adat szekvenciális jellegű mindkét hálóban szükség van egy rekurrens háló részre, erre az LSTM-struktúrát használtam.

4.1. Szekvenciális adat kezelése LSTM hálóval

Az LSTM (Long-Short term memory) modellt 1997-ben publikálták [13], és áttörést jelentett a rekurrens hálózatok világában, több olyan problémát át tudott hidalni, amely az addigi rekurrens hálókkal nem volt lehetséges, vagy csak sokkal lassabban.

Legyen $t; k \in \mathbb{N}$ időpontok és $t < k$. Egy egyszerű rekurrens háló képes változó hosszú időbeli összefüggéseket is megtanulni, ezt pedig egy visszacsatolt szerkezettel éri el, amely a következőképpen írható le:

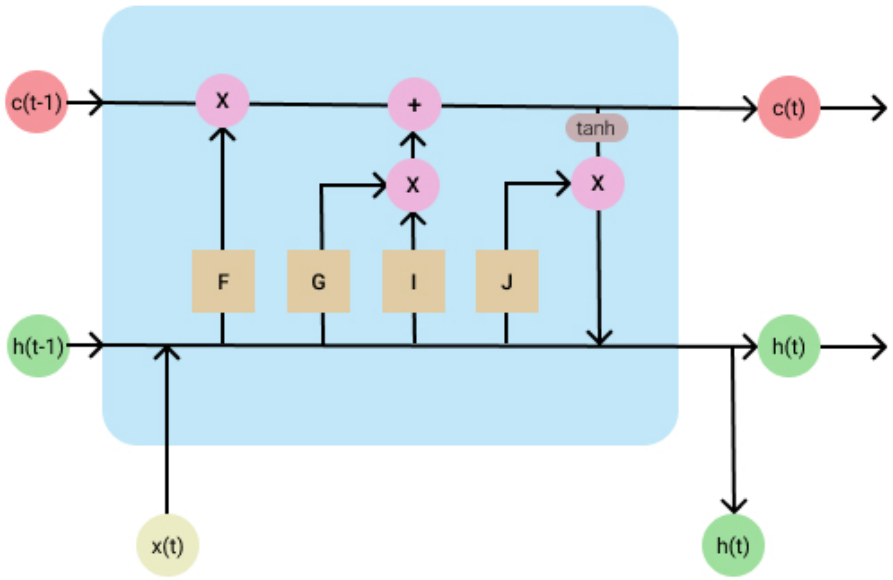
$$h_t = (W_x \quad x_t + W_h \quad h_{t\ 1} + b)$$

$$y_t = (W_{out} \quad h_t)$$

Ahol x_t az adott t időpontbeli bemenet, h_t-t a t-edik időpontban lévő belső állapotnak nevezzük. $W_x \in \mathbb{R}^{dim(h)\ \ dim(x)}$, $W_h \in \mathbb{R}^{dim(h)\ \ dim(h)}$, $W_{out} \in \mathbb{R}^{dim(y)\ \ dim(h)}$, $b \in \mathbb{R}^{dim(h)}$, és pedig nem lineáris aktivációs függvények. Ez a módszer hatékony rövid szekvenciák

értelmezéséhez, de gyakorlatban hosszú távon a bemenetek adta információ elveszik a hálóban, és így nem képes információt tárolni x_t-ből y_k-ra ha jk tj nagy.

Az LSTM modellek ezt a struktúrát egy bonyolultabb belső logikával hidalják át. Képzeljük el a modellt úgy, mint egy rekurrens hálózatot csak a blokkban nem egy sűrű réteg van, hanem egy összetettebb szerkezet. Itt egy blokk három bemenetet kap, egy cella állapotot $c_{t\ 1}$, amely hivatott eltárolni hosszú távon az információt, egy belső állapotot $h_{t\ 1}$, amely az előző lstm blokk kimenete, illetve a t-edik időpontban bejövő új adatot x_t.



4.1. ábra. LSTM blokk belső szerkezete, a + operáció mátrixok összeadását jelenti, míg az az elemenkénti szorzást.

A 4.1 ábrán látható egy LSTM blokknak a struktúrája. Itt F, G, J sűrű rétegek, szigmoid aktivációs függvénnyel, ami a következőképpen van definiálva:

$$(u) = \frac{1}{1 + e^{\ u}}$$

Ennek a függvénynek a segítségével, képes a modell a feladat megoldása szempontjából felesleges információk elfelejtésére, mivel minél kisebb érték érkezik be, annál inkább fogja a szigmoid függvény egy nullához körüli értékbe képezni. I pedig egy sűrű réteg tangens hiperbolikusz aktivációs függvénnyel. Az aktivációs függvényeket arra használjuk, hogy a modellünk nem lineáris összefüggéseket is képes legyen megtanulni, így ezek általában

nem lineárisak, illetve a tanulásra használt gradiens módszer miatt fontos, hogy differenciálhatóak legyenek. Egy modul felépítése a következő képpen írható le:

$$c_t = f_t \quad c_{t\ 1} + g_t \quad i_t$$

$$h_t = \quad (W_J \quad [h_{t\ 1}; x_t] + b_J) \quad \tanh(c_t)$$

ahol

$$f_t = \quad (W_F \quad [h_{t\ 1}; x_t] + b_F)$$

$$g_t = \quad (W_G \quad [h_{t\ 1}; x_t] + b_G)$$

$$i_t = \tanh(W_I \quad [h_{t\ 1}; x_t] + b_I)$$

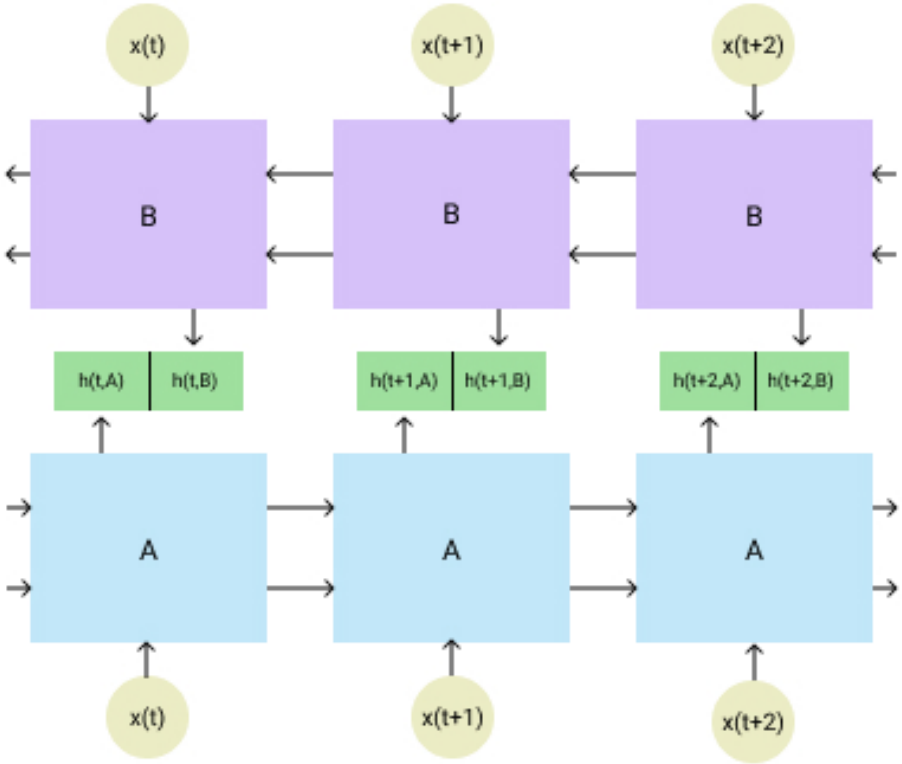
Itt $W\ 2\ \mathrm{R}^{dim(h)\ \ dim(h)+dim(x)}$ és $b\ 2\ \mathrm{R}^{dim(h)}$. A $dim(h)$-ra úgy hivatkozunk mint az LSTM háló belső mérete, ez egy hiperparamétere az LSTM hálóknak, amely meghatározza, milyen nagy dimenzió számmal számoljon a háló, $dim(x)$ pedig egy bemeneti vektor hossza. A belső méret minél nagyobb, a háló annyival komplexebb összefüggések megtanulására képes, viszont a változók növekedésével csökken a tanulás sebessége. A kimenete egy modulnak pedig h_t, a belső állapota a t-edik időpontban. Az LSTM hálók képesek hosszabb idősorok közötti összefüggések megtanulására, ezért tökéletes eszköz a hosszú log szekvenciák értelmezéséhez.

4.2. LogRobust

A LogRobust modellt 2019-ben publikálták, a Microsoft fejlesztői [2]. Egy olyan supervised log anomália kereső elkészítése volt a cél, amely robosztus azokra log adatbázisokra, ahol a log üzenetek gyakran ismétlődnek, konstansaik sorrendje felcserélődhet, illetve fontos kritérium volt, hogy jól tudjon reagálni új események bejövetelére is. A fenti feltételek teljesítésével a LogRobust használható valós problémák megoldására. A modell felépítésében három részből áll. Elöször egy kétirányú LSTM hálót használunk, majd ennek a kimeneti vektorait össze aggregáljuk, ezzel létrehozunk egy fix dimenziós reprezentánst. Az utolsó háló részben pedig a reprezentánsból készítünk egy predikciót, hogy az adat anomália, vagy sem.

A modell bemenetének első rétege egy kétirányú LSTM rész. A kétirányú rekurrens hálóknál két ugyanolyan struktúrájú háló működik egymás mellett. A különbség, hogy

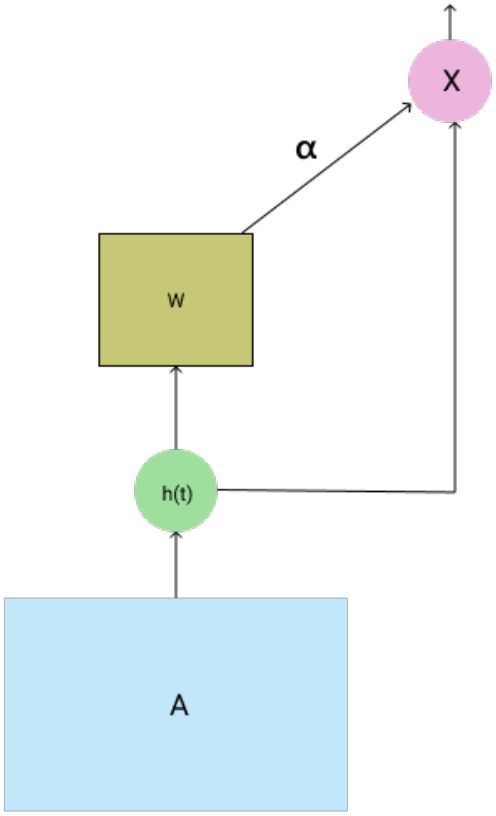
míg az egyik időben előre küldi tovább a belső és cella állapotot, addig a másik pedig pont fordítva, így minden idő állapotra két belső állapot jön létre, ezeknek általában az idő egységenként vett konkatenáltja lesz a kimenet. Ezt a megoldást természetesen LSTM modelleknél is alkalmazhatjuk [14, 15]. Ezzel a kétirányúsággal a LogRobust könyebben tudja értelmezni egy log szekvencia teljes egészét. A kétirányú összefüggések megtanulása egy mélyebb megértést eredményez. Az LSTM résznek minden időpontban létrejött kimenetét használjuk, és így a bemeneti vektorok számával egyenlő számú vektor jön létre, a kimeneti vektorok hossza $2 \quad dim(h)$. A kettes szorzó azért szükséges, mert minden időpontban két belső állapot jön létre a model kétirányúsága miatt.



4.2. ábra. Kétirányú rekurrens hálózatok struktúrája

Ezután egy egyszerűsített attention réteg következik. Az attnetion mechanizmus az emberi agynak a részletekre való fókuszálási képességének implementálása egy neurális hálóban. Az emberi agy képes részletekre fókuszálni attól függően, hogy a probléma megoldásához mi adja a legtöbb információt. Például ha a feladat az, hogy határozzuk meg a képen lévő ember korát, az arcot sokkal inkább vesszük figyelembe mintsem mást. Az

attention mechanizmust rekurrens hálózatokon először 2014-ben használták, és képesek voltak elérni áttörő eredményeket [16], azóta az adatbányászat több területén is képfeldolgozásban, és egyéb feladatokban is sikeresen alkalmazták.



4.3. ábra. A logRobust háló attention része, elemenkénti szorzást jelől, A egy LSTM blokk, W pedig egy sűrű réteg.

Ebben az esetben minden idő állapotra, azaz minden bemeneti log eseményre számolunk egy súlyt, és ideális esetben ez a súly azt tanulja meg, hogy egy esemény mennyire fontos a predikció szempontjából. Az meghatározása a következő minden t-re

$$_t = \tanh(W \ h_t)$$

Ahol $W \ 2 \ \mathrm{R}^{2\,dim(h)\ 1}$. Itt látjuk, hogy amikor egy súlyt elkészítünk kizárólag a h_t-t használjuk, nem pedig az egész környezetet, így sokkal inkább egy eseményre koncentrálunk a súlyok elkészítésénél, nem pedig a teljes kontextusra. Persze a kétirányú LSTM réteg miatt h_t tartalmaz információt a többi kimeneti állapotról is.

Ahogy készen vannak az súlyok azeket felhasználva az attention rész kimenete:

$$= \sum_{i=1}^{n} {}_{t} \quad h_t$$

Itt n a szekvencia hossza, láthatjuk hogy $2 \mathbb{R}^{2\,dim(h)}$, azaz nagysága fix dimenzióval rendelkezik, és megegyezik a kétirányú LSTM réteg belső állapotának dimenziójának kétszeresével, ez lesz az állandó dimenziós reprezentánsunk. A prediktálást egy sűrű réteg segítségével képezzük, amelynek kimenete két érték. Amennyiben változó dimenziójú lenne a reprezentáns mérete, akkor nem tudnánk tanítani hozzá sűrű réteget, hogy két értékbe képezzen, ezért van szükség az állandó dimenzióra. A predikcióhoz alkalmazunk egy aktivációs függvény nélküli sűrű hálót A-t, amely olyan hosszúságú vektorba képezi az attention rész kimenetét -t mint a lehetséges kimenetek.

$$k = A$$

Ahol $A \; 2 \; \mathbb{R}^{2 \;\; 2\,dim(h)}$, így k egy két elemű vektor lesz. Legyen a modell kimenete y, célunk, hogy y egy olyan kételemű vektor legyen, ahol $y[0] = P(x^{0} \; 2 \; normaljx; \quad)$ és $y[1] = P(x^{0} \; 2 \; anomalyjx; \quad)$, ahol x az bemeneti log szekvencia, x^{0} a szekvencia anomália címkéje, pedig a modell aktuális paraméterei. Mint látható y első eleme annak a valószínűsége, hogy a log szekvencia normális, második eleme pedig annak a valószínűsége, hogy anomália. Az hogy egy log szekvencia anomália vagy sem, egy teljes esemény rendszert alkot, így $y[0] + y[1] = 1$. Ahhoz hogy k-ból megkapjuk y-ont a soft-max függvényt használjuk, ami képlettel a következőképpen írható le:

$$y_i = \frac{k_i}{\sum_{j=1}^{dim(k)} (k_j)}$$

Vegyük észre, hogy ez eleget tesz a valószínűségi kritériumoknak.

A háló tanulásához szükségünk van egy veszteség függvényre, amely meghatározza mennyire pontos a predikciónk. Ennek a függvénynek azt kell mérnie mekkora eltérés van a modell kimenete, illetve a tényleges valós eredmény között. Természetesen minél kisebb ez a különbség átlagosan, annál jobban teljesít a modell. Azaz a feladatunk, hogy ezt a függvényt minimalizáljuk. A LogRobust esetében a Bináris Cross-Entropy-t használjuk,

amely egy a gépi tanulásos módszerekben gyakran használt függvény annak kifejezésére, hogy a modell által prediktált valószínűségek, és a tényleges valószínűségek között mennyi az eltérés. Bináris esetben két címkére generálunk valószínűséget p, $1 \quad p$, ezek a valószínűségek a modellünk softmax rétegének kimenete. Bináris Cross Entropy képletben:

$$\text{Loss} = \frac{1}{N} \sum_{i=1}^{N} (l \quad log(p) + (1 \quad l) \quad log(1 \quad p))$$

Itt l a valós érték, 1 normális esetben 0 egyébként, p annak a modell által mért valószínűsége, hogy a log blokk normális. N-el a mini batch-ek elemszámát jelöltem. Mini batcheket a tanulásnál használjuk, amikor át iterálunk a teljes adathalmazon, nem minden egyes rekord után változtatunk a modell változóinak értékein a rá kiszámolt veszteség függvény alapján, hanem N változóra számolt veszteség függvény alapján, ez gyorsabb konvergenciát eredményezhet, és segít hogy a gradiensek ne szálljanak el.

4.3. LSTM Autoenkóder

A második modell, amelyet használunk a log üzenetekben lévő anomáliák detektálásához egy autoenkóder lesz. Struktúrájában a [17] cikkben bemutatott anomália keresőn alapul. Ebben a cikkben ezt videófelvételeken használták az anomáliák felkutatása érdekében. A LogRobust-al ellentétben ez egy unsupervised módszer, így tanítható akár nem címkézett adathalmazokon is. A megértés érdekében először szükséges tisztázni azt, hogy mi is azaz autoenkóder.

4.3.1. Autoenkóder

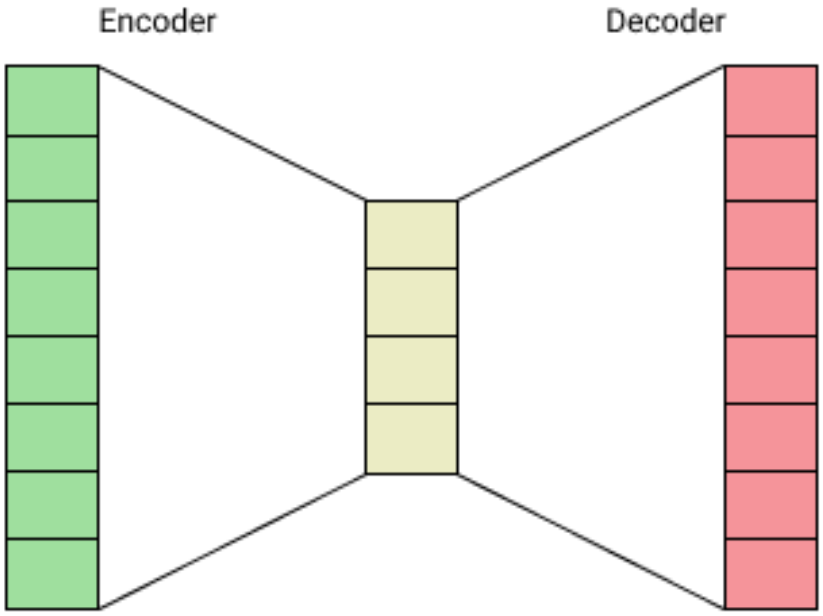
Az autoenkóder egy olyan unsupervised mély tanulásos módszer, amely elkészít egy alacsony dimenziós belső reprezentációt, majd ezt felhasználva újra rekonstruálja a bemeneti adatot. Alapvetően két részből áll, egy enkóderből A, amely létrehozza a belső reprezentációt

$$A(x; \quad) = b$$

illetve egy dekóderből B, amely ebből a reprezentációbol létrehozza a bemeneti adat rekonstrukcióját

$$B(b; \quad) = x'$$

Itt $dim(b) << dim(x) = dim(x^0)$. ; a hálók paraméterei. A, és B általában sűrű rétegek, vagy sűrű rétegek sorozata. Ahhoz hogy a háló tanulni tudjon a bemeneti adat és a rekonstrukció között használunk egy veszteség függvényt $\mathsf{Loss}(x; x^0)$, célunk hogy ezt minimalizáljuk, általában a gradiens leereszkedés valamelyik variánsával.



4.4. ábra. Az ábrán az autoencoder struktúra illusztrációja látható

A belső reprezentáció fontos, hogy kisebb tér legyen ezzel kikényszerítve a modellből, hogy ne a bemeneti adatot tanulja meg, hanem az elemei közötti összefüggéseket. Abban az esetben, ha túl kicsi reprezentációt használunk nem lesz képes megfelelően rekonstruálni a bemenetet. A veszteség függvény választható a kimeneti és bemeneti vektorok között. Például valamilyen távolság megfelel, én $L1$ vagy $L2$ normát használtam. A megtanult összefüggéseket fel tudjuk használni anomália keresésre, mert feltételezhetjük, hogy a normálistól eltérő rekordok nem rendelkeznek ezekkel az összefüggésekkel, és az autoenkóder nem képes megfelelően prediktálni a bemeneti adatot. Az autoenkódereket, gyakran használják dimenzó csökkentésre, a belső reprezentációik segítségével.

4.3.2. Autoenkóder log üzenetekre

Az esetünkben változó hosszúságú listát kell rekonstruálni, amelynek az elemei vektorok. A szekvenciális adat kezelésére a LogRobustnál bevált LSTM háló struktúrát fogom használni. Legyen $x = [x_1; x_2; :::; x_N]$ a bemenő szekvencia, $x_t \in \mathbb{R}^m$ a szekvencia t.-eleme. Első lépésben az enkóder résszel létrehozzunk egy alacsony dimenziós reprezentánst.

$$A_E(x; \ _E) = b \tag{4.1}$$

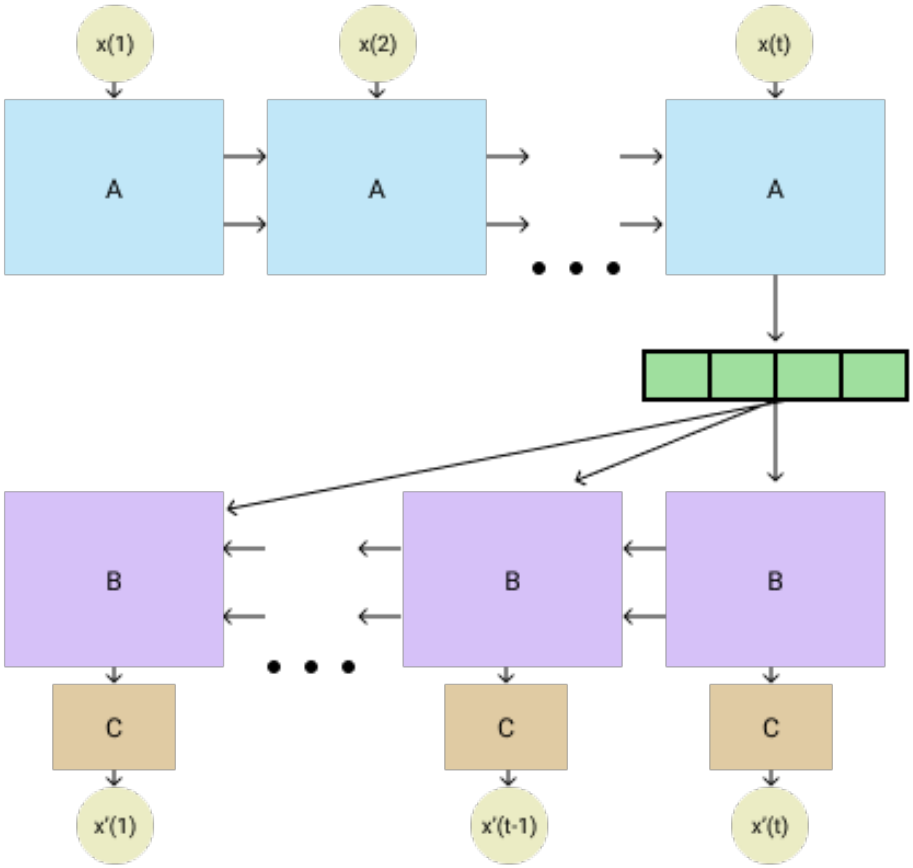
Itt A_E egy egyirányú LSTM réteg $_E$ paraméterekkel, amelynek minden időpontban van kimenete, így b felírható úgy mint ezeknek a kimeneteknek a listája $b = [b_1; b_2; :::; b_n]$. Az alacsony fix dimenziós reprezentáns b_n lesz, azaz az utolsó időpontban keletkezett kimenet, ez tartalmaz információt az egész szekvenciáról, így megfelel. Ennek a reprezentánsnak $N \ m$ dimenzióval rendelkező bemenet összefüggéseit kell eltárolnia magában, így hosszú bemeneti szekvenciáknál érdemes megfelelően nagy reprezentáns méretet választani. A dekóder részben b_n reprezentánst felhasználva, akarjuk rekonstruálni x-et. Változó hosszú szekvenciát akarunk készíteni, egy állandó dimenziós vektorból, ezt szintén egy LSTM rétegre bízzuk. Ennek az LSTM rétegnek n modulja lesz, így a kimenetek száma megegyezik majd a bemeneti szekvencia hosszal. Annak érdekében, hogy a dekóder LSTM megfelelően működjön szükségünk van n db bemenetre, ezt úgy oldjuk meg, hogy minden időpontban b_n-t adjuk meg. A dekóder bemenete így $[b_n; b_n; :::; b_n]$ egy n hosszú vektor lista, melynek elemei azonosan b_n vektorok, ezt jelöljük g-vel. A dekóder LSTM úgy írható fel, mint:

$$A_D(g; \ _D) = y \tag{4.2}$$

Itt A_D egy LSTM háló, $y = [y_1; y_2; :::; y_n]$ pedig egy n hosszú vektor sorozat, $y_i \in \mathbb{R}^k$, k az autoenkóder egy hiperparamétere, a dekóder LSTM belső dimenziója. Mivel a dekódernek minden időpontban a bemenete ugyanaz a vektor, a megfelelő predikcióhoz a reprezentációban eltárolt sorrendhez tartozó összefüggéseket használja a dekóder rész. Következő lépésben, y vektorból létrehozzuk a bemenet rekonstrukcióját, ehhez egy azonos sűrű rétegeket fogok alkalmazni, minden időpontban. Arra, hogy a sűrű réteg minden időpontban azonos legyen, a változó szekvencia hossz miatt van szükség. Predikciónk, a következőképpen írható fel:

$$x^{'} = [By_N + c; By_{N \ 1} + c; :::; By_1 + c] \tag{4.3}$$

Itt $B \in \mathbb{R}^{m \times k}$ a sűrű réteg mátrixa, $c \in \mathbb{R}^{m}$ pedig az eltolás vektora. Láthatjuk, hogy az y_i-ket fordított sorrendben alkalmazzuk, azaz y_n-t használjuk x_1 becslésére, y_{n-1}-et pedig x_2-becslésére. Ennek a megfordításnak az alkalmazása, azért fontos, mivel a dekóder LSTM első bemenete, kizárólag az LSTM enkóder utolsó kimenete, érezhető hogy minél később jött be egy bemenet, annál tisztább információt tartalmaz róla. A későbbi bemeneteket így könyebben rekonstruálhatjuk, az LSTM réteg elején, és tisztább információt tudunk küldeni tovább a korai bemenetek megkonstruálásához. Elkészítettük a rekonstrukciónkat x'-t, most az következik, hogy a modell súlyait megtanítsuk, hogy képes legyen minél pontosabb predikciót létrehozni a bemenetre x-re.



4.5. ábra. Az ábrán az általunk használt LSTM autoencoder struktúrája látható, $x(t)$ a bemeneti vektor a t időpontban, $x'(t)$ pedig $x(t)$ predikciója. A, B LSTM blokkok, C sűrű réteg.

A Tanuláshoz számolunk egy veszteség függvényt, amelyen gradiens módszerrel tanul-

hatunk. Én a veszteség meghatározására $L1$ távolságokat átlagoltam a valós, és prediktált vektorok között

$$\frac{\sum_{i=1}^{N} \sum_{j=1}^{M} |x_{i;j} - x'_{i;j}|}{N}$$

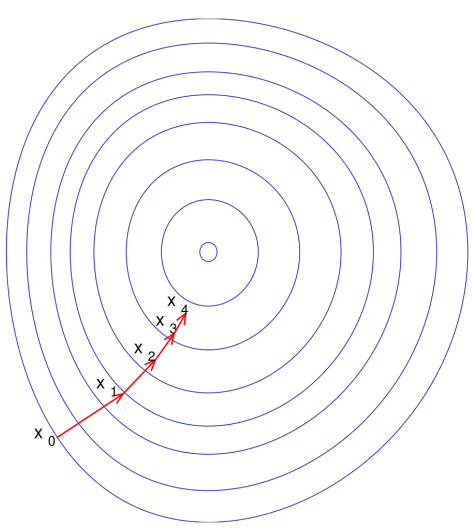
ahol N egy szekvencia hossza, M pedig egy bemeneti vektor dimenziója, $x_{i;j}$ a bemeneti lista i-edik vektorának j-edik eleme, $x'_{i;j}$ pedig a kimeneti lista i-edik vektorának j-edik eleme. Ez helyettesíthető más távolság metrikákkal is, például $L2$ normával. Célunk ennek a függvénynek a minimalizálása. Ahhoz hogy anomália címkét prediktájunk a log szekvenciákra, szintén ezt a veszteség függvényt használjuk. Amennyiben ez a veszteség érték alacsony, akkor feltételezhetjük, hogy a bemenet egy hasonló eloszlásból keletkezett, mint az átlagos log szekvenciák, ebben az esetben normális-nak prediktáljuk a log szekvenciát. Abban az esetben, amennyiben a veszteség magas, akkor az adat rekord nem rendelkezik azokkal a tulajdonságokkal, amelyet a modell megtanult, így anomáliának bélyegezzük. A döntés határt, hogy milyen veszteség felett prediktáljuk, a szekvenciát anomáliának a modell ROC-görbéje alapján határozzuk meg, egy validációs halmaz segítségével.

4.4. Modellek tanitása

A neurális hálókat inicializálás után be kell tanítani egy veszteség függvényen keresztül. Feladattól függően cél az optimális megoldás megtalálása, általában egy veszteség függvény minimalizálása. A LogRobust, és az LSTM autoenkóder esetében, is használtam, mind sztochasztikus gradiens leereszkedést, mind Adam módszert is. Általánosan elmondható, hogy az Adam mindkét modell esetében gyorsabban konvergált, mint az egyszerű sztochasztikus gradiens leereszkedés módszerrel.

4.4.1. SGD

A gradiens leereszkedés célja, hogy megtaláljuk, a veszteség függvényünk minimumát. A veszteség függvényt jelöljük $\mathrm{Loss}(\theta; x)$-el, ahol θ a modell paramétereit, x a bemenetet jelöli. Szükséges kritérium, hogy a veszteség függvény a paraméterei szerint deriválható legyen.



4.6. ábra. Ilusztráció a gradiens módszer működésére

Ahhoz, hogy megtaláljuk a minimumot a súly paramétereket a függvényen vett parciális deriváltak segítségével frissítjük az alábbi módon:

$$w_n = w_{n\ 1} \qquad \frac{@}{@w}\,\mathrm{Loss}(\ _{n\ 1}; x_n)$$

ahol w_n az a modell egy paramétere az n-edik iterációban, $\frac{@}{@w}\,\mathrm{Loss}(\ _{n\ 1}; x_n)$ pedig az n-edik iterációban kiértékelt veszteség függvényünk w által vett parciális deriváltja. A tanulási ráta jele , ezt mi választjuk, általában egy kicsi érték 0.3 , 0.2, 0.03. Fontos, paraméter a tanulás szempontjából, ha túl kicsit veszünk, megnőhet a tanulás időtartama, és könnyebben kerülünk egy lokális minimum-ba, míg ha túl nagy, könnyen előfordulhat, hogy sosem közelítünk meg semmilyen minimumot. Mivel az egyszerű gradiens leereszkedés számítási kapacitása igen magas lenne a teljes adathalmazon, ezért a sztochasztikus gradiens leereszkedést (SGD) használjuk általában. Az SGD ben, az algoritmus hasonló a gradiens leereszkedéshez, de amíg az egyszerű leereszkedésnél a teljes adathalmaz rekordjainak vett veszteségével frissítjük a gradienst, addig az SGD esetében választunk egy rekordot, és annak a vesztesége alapján frissítjük a gradienst. Gyakran nem egy, hanem több adat rekordot választunk, egy ilyen adatrekord halmazt mini-batch-nek hívunk.

4.4.2. Adam

Az Adam módszer az egyik legfelkapottabb optimalizációs módszer, 2014-ben publikálták [18], az ott bemutatott mérésekben több esetben jóval túlteljesítette az SGD momentumos variánsait. Alapvető probléma az SGD módszerrel, hogy ugyanazzal a lépésközzel frissíti a súlyokat minden beérkező gradiens esetében, de a gradiensek nagysága akár nagyságrendekben is eltérhet. Az Adam módszer erőssége abban rejlik, hogy minden paraméterre külön súlyokat tanul amellyel azt módosíthatja. A paraméterek tanulási súlyait, a hozzájuk tartozó gradiens első, és második momentumának becslésével számolja a módszer. A módszer a következőképpen írható le:

$$m_t = \quad _1 \quad m_{t\ 1} + (1 \qquad _1) \quad g_t \tag{4.4}$$

$$v_t = \quad _2 \quad v_{t\ 1} + (1 \qquad _2) \quad g_t^2 \tag{4.5}$$

$$\hat{m}_t = \frac{m_t}{(1 \qquad _1^t)} \tag{4.6}$$

$$\hat{v}_t = \frac{v_t}{(1 \qquad _2^t)} \tag{4.7}$$

$$_t = \quad _{t\ 1} \qquad p\frac{\hat{m}_t}{\hat{v}_t +} \tag{4.8}$$

itt m_0-t és v_0-t 0-nak választjuk, $g_t = \frac{@}{@}Loss(\ _{t\ 1}; x_t)$ azaz a t.-edik iteráció beli gradiens. $_1; \ _2; \ ; \ $ az algoritmus hiperparaméterei, általában a legtöbb feladathoz $_1 = 0{:}9, \ _2 = 0{:}999, \ = 0{:}001$ illetve $= 10^{\ 8}$ megfelel. A cél hogy m_t-vel g_t első momentumát, míg v_t vel pedig a g_t második momentumát közelíteni. 4.6, és 4.7 a közelítés korrigálására szolgálnak, a nevező levezethető az alábbi módon:

$$m_0 = 0$$

$$m_1 = \quad _1 \quad m_0 + (1 \qquad _1) \quad g_1 = (1 \qquad _1) \quad g_1$$

$$m_2 = \quad _1 \quad m_1 + (1 \qquad _1) \quad g_2 = \quad _1 \quad (1 \qquad _1) \quad g_1 + (1 \qquad _1) \quad g_2$$

$$m_3 = \quad _1 \quad m_1 + (1 \qquad _1) \quad g_2 = \quad _1^2 \quad (1 \qquad _1) \quad g_1 + \quad _1 \quad (1 \qquad _1) \quad g_2 + (1 \qquad _1) \quad g_3$$

Ezt tovább folytatva a látjuk, hogy a következőképpen tudjuk felírni az egyenletet

$$m_t = (1 \qquad _1) \sum_{i=1}^{t} \quad _1^{t\ 1} \quad g_i \tag{4.9}$$

Ahhoz hogy g első momentumát közelítsü m_t vel a következőnek kell teljesülnie: $E[m_t] = E[g_t]$. 4.9-ból kiindulva a következőt láthatjuk

$$\begin{aligned} E[m_t] &= E[(1 \quad _1) \sum_{i=1}^{t} {}_1^{t} {}^{1} g_i] \\ &= E[g_t] \quad (1 \quad _1) \sum_{i=1}^{t} {}_1^{t} {}^{i} + \\ &= E[g_t] \quad (1 \quad {}_1^{t}) + \end{aligned} \tag{4.10}$$

Itt az első lépésben g_i-t becsüljük g_t-vel, és így ki tudjuk szedni a szummából, viszont keletkezik egy hiba. A v_t levezetése analóg módon keletkezik annyi különbséggel, hogy v_t-nek a második momentumot kell közelítenie azaz a $E[v_t] = E[g_t^2]$ egyenletnek kell teljesülnie. 4.10 ből pedig következik hogy 4.6 megfelelő közelítés. A hiperparaméterek választhatók, de az általános beállításoktól nem nagyon szoktak eltérni. Az m_t módosítására használt $_1$, illetve a v_t módosítására használt $_2$ exponenciálisan szabályozzák, hogy az adott időpontbeli gradiensek, mekkora súllyal jelenjenek meg, az m_t, v_t elkészítésében, ahogy az 4.9-ben látható. Ezeket általában egy körüli számnak választjuk, így elérhető, hogy az aktuális, és azt közvetlenül megelőző gradiensek magas arányba járuljanak hozzá a súly frissítéshez. minden esetben egy kis szám, erre azért van szükség, hogy az első iterációban elkerüljük, a nullával való osztást. A gradiens léptetés nagyságát az hiperparaméterrel szabályozhatjuk. Ebben az esetben, mivel egy lépés nagysága paramétertől függően változik, könnyebben választható megfelelő érték *alpha*-ra, viszont figyelembe kell venni, hogy *alpha* egy felső határt adhat a lépés nagyságára.

5. fejezet

Kiértékelés

5.1. Az adat

A log üzenetek vizsgálatára egy jól ismert adatbázis csomag áll rendelkezésre, a Loghub cikk [19] által. Kifejezetten kutatási célra vannak kigyűjtve rendszer log-ok. Különféle forrású, típusú előre feldolgozott log adathalmaz áll rendelkezésünkre. Szakdolgozatom szempontjából egy olyanra volt szükség, amely cimkézett, hogy az eredményeket validálni lehessen. Fontos kritérium volt még, hogy a log üzenetek csoportonként legyenek címkézve, mivel nem azt akarjuk osztályozni, hogy egy adott üzenet mennyire súlyos hibát jelez, hanem hogy a különböző üzenet típusok időbeli eloszlása, mennyire utal arra, hogy az adott folyamat hibásan működik. Erre a célra leginkább a HDFS log adathalmaz felelt meg.

A HDFS (Hadoop Distributed File System) egy a kutatásban népszerű adatbázis. Több log fájl vizsgálattal foglalkozó cikkben is használják, mint kizárólagos adathalmaz [8]. A log üzenetek folyamat azonosítóit, ebben az esetben a „blokk ID”-k jelölik, ezek az ID-k azonosítják be, hogy milyen folyamat következményeképpen született meg a log üzenet. Ezen blokkok alapján tudjuk majd a logokat csoportosítani, és kiértékelni. A Loghub által közzétett adat, minden blokk-ra tartalmaz egy címkét, hogy anomália vagy sem, ennek alapján tudjuk majd kiértékelni és tanítani a modellünket. Az adatbázis mintegy 500 000 blokkot tartalmaz, ebből 16 838 anomália. A felsorolt tulajdonságai miatt tökéletesen megfelel analitikai céljainkra.

5.2. Előfeldolgozás

Az adathalmaz előfeldolgozásában a 3-dik fejezetben bemutatott Drain paramétereit úgy állítottam, hogy a fa mélysége négy legyen, azaz az első 3 konstans elem ellenőrzése alapján állítja elő az eseményeket. A hasonlósági küszöböt 0.2-re állítottam, így keletkezett megfelelő mennyiségű különböző log esemény, és a log események nem tartalmaztak túl sok * konstans elemet. A hasonlósági küszöb apró változtatásával, vagy a fa mélységének feljebb állításával lehet hangolni a log események számát. A hasonlósági küszöb növelésével jellemzően több esemény születik, mivel így a log üzenetnek nagyobb hasonlóságot kell mutatnia a létező eseményekkel, hogy ne hozzon létre újat. A hasonlósági küszöböt túlságosan megnövelve, az események száma közel egyenlő lesz a log üzenetek számával, így használhatatlanná válnak. Abban az esetben, ha ez a küszöb alacsonyabb, pedig kevesebb esemény születik, de ha túl alacsony előfordulhat, hogy nagyon kevés esemény születik, és egy esemény elemei zömében * karakterek lesznek. A fa mélységének növelésével, az események száma nő, viszont fennáll a veszélye, hogy hasonló log üzenetek eltérő eseményeket kapnak, és így túl sok fölösleges esemény születik, a szemantikus vektorizálás ezt a hibát ugyan áthidalhatja, viszont az adat előállításának ideje megnő. Reguláris kifejezésekkel kivettem a blokk számokat, idő bélyegeket, illetve egyéb azonosításra szolgáló log üzenet részeket, mivel ezek változó értékek, és a log üzenet típusáról nem tartalmaznak információt. Ezen beállításokkal 43 különböző log esemény keletkezett, ezeket a 3-dik fejezetben ismertetett módszerrel vektorizáltam, párosítottam a hozzájuk tartozó log üzenettel, illetve csoportosítottam blokk számuk szerint.

5.3. Anomália érték vizsgálat

Miután elvégeztük a feldolgozást választunk egy teszt adathalmazt a tanításra, és egy validációs adathalmazt a eredmények kiértékelésére. A választás a két modellnél eltér különböző jellegük miatt. A tanitás végeztével kiértékeljük a validációs halmazban található rekordokat, minden esetben egy anomália értéket kapunk. A LogRobust esetében, ez egy valószínűség, az autoenkóder esetében, pedig egy rekonstrukciós veszteség. Az anomália értékek között választhatunk egy döntés határt, egy c értéket, és ha $y > c$, akkor anomáliának cimkézzük, ahol y a választott modellel vett aktuális rekord kimenete. A döntés küszöb meghatározásában, a ROC (Receiver Operating Characteristics) görbe segít. A

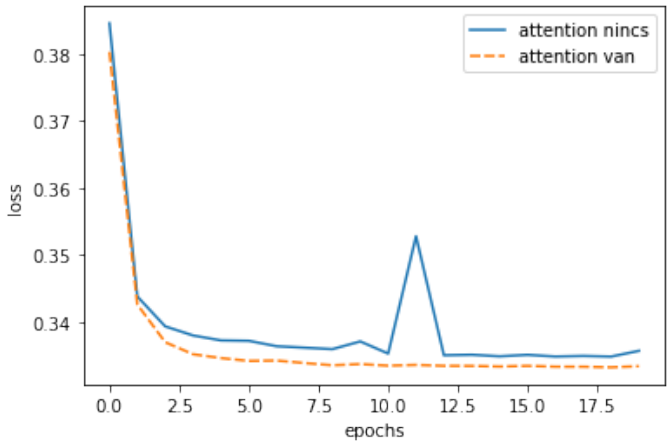
görbe azt mutatja, hogy egyes döntési határok mellett, milyen true positive rate-tel (*tpr*), illetve milyen false positive rate-tel (*fpr*) rendelkezik a modell. Az y tengelyen jelöljük a *tpr*-t, x-tengelyen pedig az *fpr*-t. Ebben az esetben az anomáliát választottam „pozitív" címkének. Az eredmények összegzésénél a modellek összehasonlítására minden esetben azt a döntési határt használtam, ahol a legalacsonyabb a *tpr-fpr* érték, ez a legjobb pontossággal rendelkező döntési határ. Az összehasonlításnál három metrikával dolgoztam. A recall (*tpr*) azt határozza meg, mennyi anomáliát detektálunk helyesen, az összes valós anomáliához képest, minél alacsonyabb annál több az olyan érték amelyet normálisnak értékelünk, pedig anomália, ezt szeretnénk leginkább elkerülni. A sensitivity (1 *fpr*) azt határozza meg, mennyi normálisat detektálunk helyesen az összes valós normálishoz képest. Amennyiben a döntési küszöbbünkkel növeljük a recall értékét, annál kevesebb valóban anomáliát bélyegzzünk normálisnak, viszont jellemzően annál több valóban normálist anomáliának. Végül az accuracy egy általános metrika, azt jelzi mennyi címkét találtunk el helyesen, az összeshez képest.

5.4. Eredmények

5.4.1. LogRobust

Mivel nem sok helyen tértem el az eredeti cikkben közölt [2] modelltől, így az eredmények is hasonlóak lettek. A HDFS adathalmazon nagyon egyenetlen a címkék eloszlása, mivel a modell supervised módon tanul, ez eredményezheti azt, hogy minden log adat rekordot normálisnak értékeljen, és így minimalizálja a veszteséget. Ezen probléma kiküszöbölésére nem a teljes adathalmazt használtam, hanem kiválasztottam az összes anomáliát 16838-at, illetve véletlenszerűen kiválasztottam hozzá ugyanennyi normálisat, így kiegyensúlyozott adathalmazon dolgozhattam. A tanuláshoz két részre bontottam a már lecsökkentett adathalmazt. Egy 25000 példányt tartalmazó tanuló adathalmazra, ezt használom a modell tanulási fázisában. Illetve egy 8676 példányt tartalmazó validációs adathalmazra, ezt felhasználva, pedig a már betanított modell teljesítményét mérem. Tanulási rátának 0.03-at választottam ,amikor a legjobb eredményeket elértem az SGD optimalizálásnál, illetve 0.0001 -et az Adam-nél. Ezek általános választások ezen módszereknél, az Adam módszer valamivel gyorsabban konvergált. A batch-ek mérete minden esetben 256 volt. Az LSTM rész belső dimenzióját 64 re állítva képes volt az anomália detektáláshoz szükséges

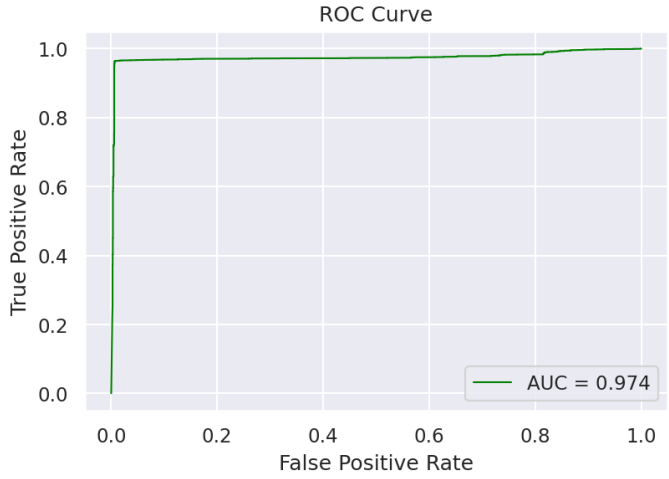
információkat megtanulni, 128-ra vagy 256-ra állítva a modell teljesítménye nem javult, viszont a tanítási idő nőtt.



5.1. ábra. Különböző LogRobust variánsok vesztesége tanulás közben

A 5.1 ábrán látható, hogy az általam legyártott attention rész kivételével, a LogRobust modell bizonyos mértékben lassabban tanul, és a tanulás is instabilabbá válik. Látható, hogy a plusz sűrű réteg segít jobban rátanulni arra, milyen súlyokkal érdemes összegezni az egyes kimeneti állapotokat. A teljes adathalmazon a tanulás során 20 szor iteráltunk át, a modell viszonylag gyorsabban tanult az Adam esetében, mint az egyszerű SGD-vel. A modellünk tényleges kimenete egy blokkra két valószínűség, p annak a valószínűsége, hogy a blokk anomália, és $1 \quad p$, arra hogy nem anomália.

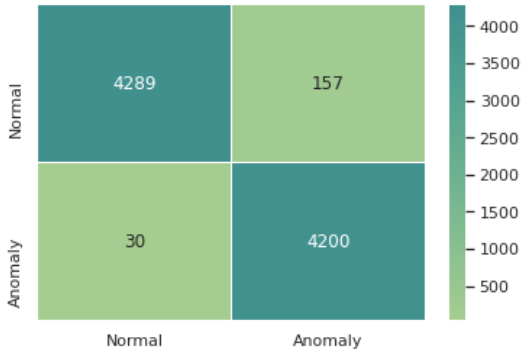
Megfontolhatjuk ezen a ponton, hogy milyen küszöb értéket választunk a döntésünk meghozása érdekében, azaz hogy mi az a valószínűségi küszöb p-re amelyre azt mondjuk hogy e felett anomáliának bélyegezzük a blokkot. Célunk az, hogy olyan döntési határt válasszunk, ahol minél kevesebb anomáliát értékelünk hibásan normálisnak, ennek kifejezésére a true positive rate szolgál. Az 5.2 ábrán látható ROC görbén látható, hogyha a döntési határ meghalad egy értéket, akkor a true positive rate növelése csak nagymértékű false positive rate növekedéssel lehetséges.



5.2. ábra. A legjobb LogRobust modell ROC görbéje látható

A következő táblázatban látható a modell teljesítménye különböző paraméter beállításokkal. Az attention oszlop a modell attention rétegét jelöli, 1 ha tartalmazott, - ha nem. Az lr a tanulási rátát, az lstm dim pedig az lstm belő dimenzióját jelöli.

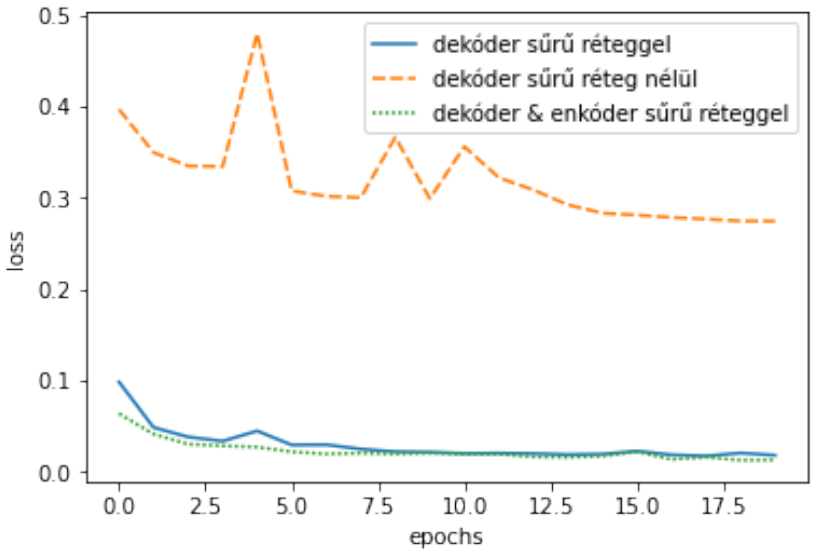
accuracy	recall	specificity	optim	lstm dim	lr	attnetion
97.70	96.81	98.69	SGD	128	0.03	1
98.00	96.81	99.23	SGD	64	0.01	1
97.91	96.47	99.36	SGD	256	0.01	1
98.12	96.88	99.35	Adam	64	0.0001	1
97.89	96.29	99.47	SGD	128	0.01	-



5.3. ábra. A legjobb LogRobust modell tévszetés mátrixa látható, x tengelyen a valós, y tengelyen a prediktált értékekkel.

5.4.2. LSTM autoenkóder

Azonos LSTM autoenkóderrel, nem találkoztam a kapcsolódó cikkekben, amelyet log anomália keresésre használtak volna. Alapvetően kétséges is lehet, hogy mennyire képes rekonstruálni változó hosszúságú nagy dimenziós adatot egy fix méretű reprezentánsból a modell, leginkább a sorrendben eltárolt összefüggésekre hagyatkozva. Cél, hogy a LogRobusthoz hasonló eredményeket érjen el a módszer a HDFS adathalmazon. Unsupervised jellegéből adódóan a tanítási adathalmaz itt eltér a LogRobust-ban kiválasztottól. A modellnek, a normális adatoknak az eloszlását kell megtanulnia, anomáliának pedig azokat osztályozni amelyek ettől az eloszlástól eltérnek. Ennek érdekében kizárólag normálisnak címkézett adattal tanul a modell, összesen 90 000 normális adatrekordot használtam tanuláshoz. A teljes adathalmaz óriási, így minél több adatot felhasználtam, hogy teljes kép legyen a log üzenetek várható alakjáról. Ezenfelül 20 000 adatot választottam véletlenszerűen a modell validálására, ebből 10 000 anomália, 10 000 pedig nem.

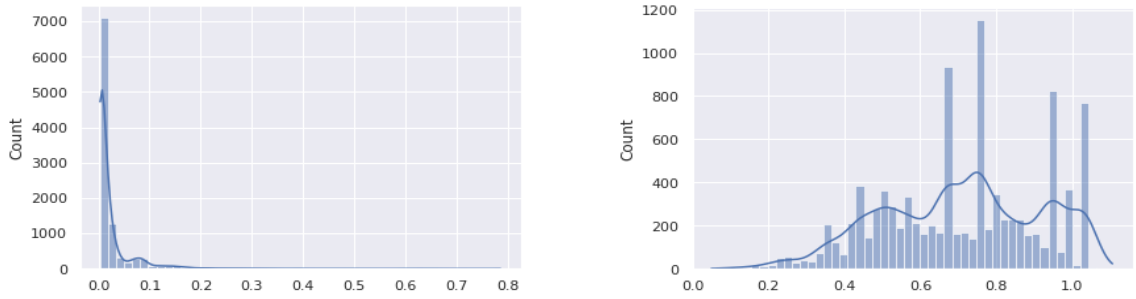


5.4. ábra. Különböző LSTM autoenkóder variánsok vesztesége tanulás közben.

Az LSTM autoenkóder első körben nem tartalmazott sűrű réteget a dekóder rész végén, így kénytelen volt 100 dimenziós belső LSTM dimenzióval dolgozni, ezzel a tanulás sem volt olyan sikeres. Amint ez a sűrű réteg bekerült, és bővithetővé vált az LSTM réteg, a modell szignifikáns javuláson ment keresztül. Később egy sűrű réteg került az enkóder rész kimenete után is, de ez érdemben nem javított a modell hatékonyságán. A különböző beállításokkal mért tanítás közbeni veszteség az 5.4 ábrán látható. Megfigyelhető, hogy a

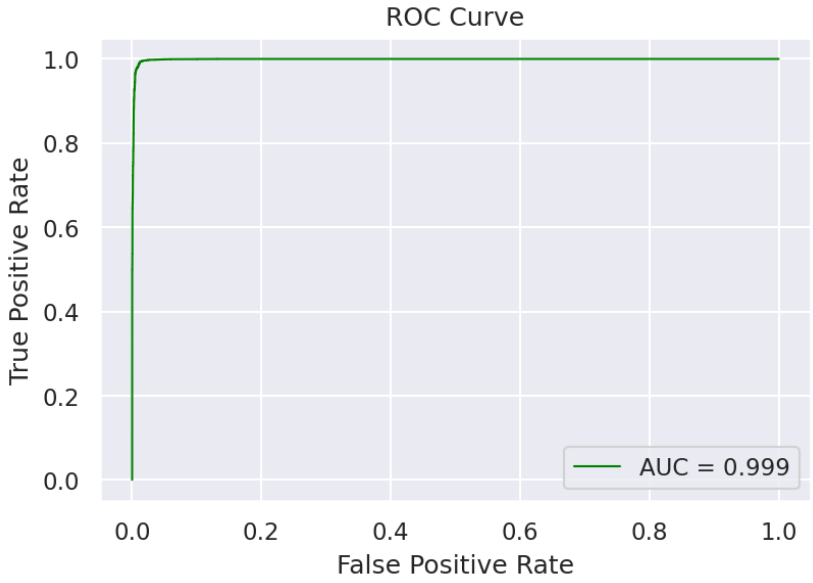
sűrű réteg nélküli dekódert használó LSTM autoenkóder egy jóval magasabb minimumba konvergál, mintsem a sűrű réteggel ellátott. Érdemes itt megjegyezni, hogy az enkóder LSTM rétegének belső dimenziója tetszőlegesen változtatható volt, igaz ezzel pontosan vele változott, a belső reprezentáns mérete is, míg a dekódernél ebben az esetben, mivel 100 dimenziós vektorokat állított elő minden időpontban 100 dimenzióval kellett rendelkeznie, ezt a problémát oldotta meg a dekóder LSTM utáni sűrű réteg.

A modell paraméterei, az enkóder, dekóder LSTM belső dimenziója, amennyiben használtam sűrű réteget az enkóderben, akkor a belső reprezentáns dimenziója. A legjobb eredményeket akkor kaptam, amikor a két LSTM belső dimenziója azonos 150 dimenziós volt. Ez a dimenzió szám elegendő volt, hogy megfelelően reprezentálja a bemenő log üzenet sorozatot, illetve elegendő volt ahhoz is, hogy a reprezentánst a dekóder megfelelően rekonstruálja bemenetté. Sűrű réteget használva az enkóder LSTM után hasonló eredmény született, mint nélküle. Optimalizálásnál az Adam módszer konvergált a leggyorsabban 0.0003-as tanulási rátával. Veszteség függvénynek, mind az átlagolt $L1$ illetve átlagolt $L2$ norma hasonló eredményeket hozott, végül $L2$ normát használtam a legjobb modell elérésénél. A teljes adathalmazon 20-szor iteráltam végig, a futási idő átlagban 3 és fél óra körül mozgott. A modellt a validációs halmazon kiértékelve, majd a rekonstrukciós veszteségeket kiszámolva láthatjuk 5.5 ábrán, hogy a normális adatokra a veszteség jól elkülöníthetően kisebb, mint az anomáliákra. Ezek alapján kell kiválasztanunk egy veszteség küszöböt, ha egy rekord ennél nagyobb értéket kap akkor anomáliának bélyegezzük.



(a) Normális (b) Anomália

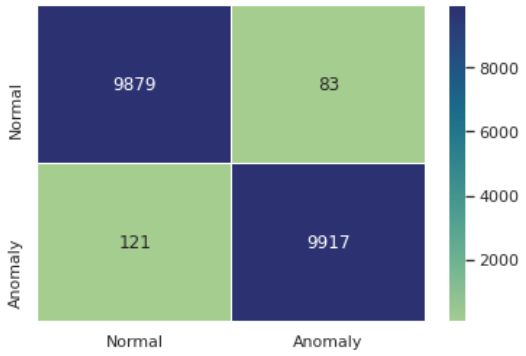
5.5. ábra. A hisztogrammokon a validációs adatok vesztesége látható, címke alapján elválasztva.



5.6. ábra. A legjobb LSTM autoenkóder modell ROC görbéje látható

Az 5.6 ROC görbét vizsgálva láthatjuk, hogy az AUC azaz a görbe alatti terület 0.999, ami annyit jelent, hogy a modellünk szinte tökéletesen prediktál, illetve azt is jelenti, hogy amikor meghatározzuk a veszteség küszöböt nagyobb szabadságot kapunk, ahhoz hogy a true positive rate-et növeljük, mivel ez nem fogja drasztikusan rontani a false positive rate-et. Ezáltal a modell döntési küszöbét meghatározhatjuk úgy, hogy minél érzékenyebb legyen az anomáliákra. A következő táblázat mutatja az eredményeket a hiperparaméterek függvényében, a küszöböt az összehasonlításhoz a legjobb accuracy eléréséhez állítottam. A táblázatban a dimE oszlop, az enkóder LSTM, míg a dimD a dekóder LSTM belső dimenzióját jelöli. Amennyiben a dekódernél ilyet nem adunk meg, akkor az azt jelenti, hogy nincs utána sűrű réteg, így az nem állítható, megegyezik a bemenet egy időpontban bejött vektor dimenziójával. A linearED, az enkódolt alacsony dimenziós reprezentáns dimenziója, amennyiben ilyet nem adunk meg akkor, az megegyezik az enkóder LSTM belső dimenziójával.

accuracy	recall	specificity	optim	lr	dimE	dimD	linearED
99.04	99.26	98.82	Adam	0.0001	150	150	-
99.19	99.51	98.87	Adam	0.0003	150	150	-
98.48	99.00	97.69	Adam	0.0003	150	150	128
98.51	99.20	97.82	Adam	0.0003	150	150	64
98.21	98.38	98.04	Adam	0.0003	64	64	-
98.65	99.15	98.16	Adam	0.0003	100	100	-
82.50	84.89	80.12	Adam	0.0003	150	-	-



5.7. ábra. A legjobb LSTM autoenkóder modell tévszetés mátrixa látható, x tengelyen a valós, y tengelyen a prediktált értékekkel

5.5. Összehasonlítás

A mindkét bemutatott modell LSTM tipusú hálót használ, hogy megküzdjön a változó szekvencia hosszal, és teszi ezt sikeresen, összehasonlítva a témában megtalálható cikkekben publikált eredményekkel. A két módszert egymással szembe állítva hasonló eredményeket láthattunk. Az autoenkóder valamivel sikeresebben becsülte meg az anomáliákat, de komplexebb hálóval is rendelkezik. A 5.6, 5.2 ROC görbéknél, látszik hogy mindkét modell teljesítménye majdnem hibátlan, ami a HDFS adathalmazon el is várható, viszont az AUC értékből is kiolvasható, hogy az autoenkóder valamivel jobb eredményt ért el, illetve kevesebb veszteséggel lehet úgy alakítani az eredmény döntési határát, hogy érzékenyebb legyen az anomáliákra. A következő táblázatban a két modell legjobb méréseinek accuracy, specificity és recall metrikái láthatóak.

model	accuracy	recall	specificity
LSTM autoenkóder	99.19	99.51	98.87
LogRobust	98.12	96.88	99.35

A modellek összetételében akadnak különbségek. Amíg az LSTM autoenkóder unsupervised, addig a LogRobust supervised, ahogy a 2-dik fejezetben is említettem, a log adatok címkéjinek aránytalansága miatt, a valós felhasználásban jobban tudunk dolgozni unsupervised módszerekkel ebben a feladat körben. A LogRobust esetében, mivel két osztályt tanult, mind az anomáliát, mind a normálisat, könnyen előfordulhat, hogy ha egy új az eddigi anomáliáktól különböző anomália keletkezik, akkor arra nem lesz elég érzékeny a helyes döntés meghozatalához. A LogRobusttal szemben az LSTM autoenkóder, amely a normális osztályok struktúráját tanulja, robusztus lesz az új anomáliákra is, amennyiben azok szerkezete eltér a megtanult normális szerkezettől. A LogRobust mellett szól viszont, hogy „end-to-end" módszer révén teljesen összefügve a különbséget tanulja a modell a normális és az anomália adatok között. Ezzel ellentétben az LSTM autoenkóder esetében pedig rekonstruáljuk a változó hosszú log sorokat, amely egy kevésbé tiszta adatbázis esetén, ahol az idősorok alakjában nem játszik ekkora szerepet az elemek sorrendje lehetséges, hogy nem vezetne eredményre. A probléma kiküszöbölésére az LSTM autoenkóder átalakítható, egy „end-to-end" modellé, amellyel, közvetlenül tanulhatjuk az anomáliák meghatározását, a szakdolgozatomban ezekre az átalakításokra már nem térek ki.

6. fejezet

Összefoglalás

A szakdolgozatban ismertettem az anomália keresés témakörét, majd ezt a tudást a számítógépes napló fájlok elemzésénél használtam fel. Összességében a log fájlok elemzése egy kritikus témakör, leginkább az adatok formájából adódóan, de mint láthattuk a tudomány jelenlegi állásának megfelelő log feldolgozási módszerek, ha nem is teljesen automatikusan, de képesek egy olyan formába képezni az adatot, amely használható arra, hogy gépi tanulásos módszerek eredményeket érjenek el. Láthattuk azt is, hogy a mély tanulásos módszerek milyen erőt képviselnek, képesek változó hosszúságú adat rekordokon is az anomáliákat megfelelően prediktálni. Az LSTM autoenkóder esetében a modell tovább fejleszthető, a belső eltárolt reprezentáció segítségével képesek lehetnénk, egy „end to end" modellt is készíteni belőle, illetve több modell összekötésével, egy robosztusabb háló szerkezet jöhetne létre. Egy kutatásra elő feldolgozott bár való életből származó log adatbázison történtek a kiértékelések ugyan, de az eredmények alapján kijelenthető, hogy a hálók kisebb átalakításokkal, az előfeldolgozás módosításával, képesek lehetnek a való életben is számitógépes rendszerek monitorozására. Következő lépés lehet az, hogy ezek a hálók nemcsak az anomáliát tudják meghatározni, hanem annak az okát is prediktálni, ezzel sok időt és energiát spórolva a használójuknak.

A mérésekhez szükséges kód elkészítéséhez Python 3-at használtam és annak az adatbányászatban népszerű package-it mint numpy, pandas, a plot-ok elkészítéséhez seaborn-t, illetve modellek elkészítéséhez, tanításához és az adatok előfeldolgozásához pytorch, és sklearn könyvtárakat. A modell tanításához egy GeForce RTX 2080 Ti videókártyát, amelyet az ELTE biztosított. A forráskód megtalálható az alábbi github repositoryban: https://github.com/RetfalviBence/LogAnalyzer

Irodalom

[1] *Hadoop Distributed File System*. 2021. url: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.

[2] Xu Zhang és tsai. „Robust Log-Based Anomaly Detection on Unstable Log Data”. (2019).

[3] Hongzhi Wang, Mohamed Jaward Bah és Mohamed Hammad. „Progress in outlier detection techniques: A survey”. *IEEE Access* 7 (2019), 107964–108000. old.

[4] Guansong Pang és tsai. „Deep Learning for Anomaly Detection: A Review”. *ACM Computing Surveys (CSUR)* 54.2 (2021), 1–38. old.

[5] Raghavendra Chalapathy és Sanjay Chawla. „Deep learning for anomaly detection: A survey”. *arXiv preprint arXiv:1901.03407* (2019).

[6] Wei Xu és tsai. „Detecting large-scale system problems by mining console logs”. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, 117–132. old.

[7] Pinjia He és tsai. „Drain: An online log parsing approach with fixed depth tree”. *2017 IEEE International Conference on Web Services (ICWS)*. IEEE. 2017, 33–40. old.

[8] Shayan Hashemi és Mika Mäntylä. „Detecting Anomalies in Software Execution Logs with Siamese Network”. *arXiv preprint arXiv:2102.01452* (2021).

[9] Tomas Mikolov és tsai. „Efficient estimation of word representations in vector space”. *arXiv preprint arXiv:1301.3781* (2013).

[10] Jeffrey Pennington, Richard Socher és Christopher D. Manning. „GloVe: Global Vectors for Word Representation”. *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, 1532–1543. old. url: http://www.aclweb.org/anthology/D14-1162.

[11] *GigaWord5*. 2021. url: https://catalog.ldc.upenn.edu/LDC2011T07.

[12] Ilya Sutskever, Oriol Vinyals és Quoc V Le. „Sequence to sequence learning with neural networks". *arXiv preprint arXiv:1409.3215* (2014).

[13] Sepp Hochreiter és Jürgen Schmidhuber. „Long short-term memory". *Neural computation* 9.8 (1997), 1735–1780. old.

[14] Zhiheng Huang, Wei Xu és Kai Yu. „Bidirectional LSTM-CRF models for sequence tagging". *arXiv preprint arXiv:1508.01991* (2015).

[15] Alex Graves, Abdel-rahman Mohamed és Geoffrey Hinton. „Speech recognition with deep recurrent neural networks". *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee. 2013, 6645–6649. old.

[16] Dzmitry Bahdanau, Kyunghyun Cho és Yoshua Bengio. „Neural machine translation by jointly learning to align and translate". *arXiv preprint arXiv:1409.0473* (2014).

[17] Nitish Srivastava, Elman Mansimov és Ruslan Salakhudinov. „Unsupervised learning of video representations using lstms". *International conference on machine learning*. PMLR. 2015, 843–852. old.

[18] Diederik P Kingma és Jimmy Ba. „Adam: A method for stochastic optimization". *arXiv preprint arXiv:1412.6980* (2014).

[19] Shilin He és tsai. „Loghub: a large collection of system log datasets towards automated log analytics". *arXiv preprint arXiv:2008.06448* (2020).