Szegedi Tudományegyetem

Informatikai Intézet

Diplomamunka

Készítette:

Kis-Szabó Norbert

programtervező informatika

szakos hallgató

Témavezető:

Berend Gábor

Számítógépes Algoritmusok és

Mesterséges Intelligencia Tanszék

Szeged

2018

Tartalomjegyzék

Feladatkiírás

A rekurrens neurális hálók igen sikeres és népszerű megoldásnak számítanak számos nyelvtechnológiai probléma megoldása során. A hallgató feladata (zene)szövegek generálására képes karakterszintű nyelvi modellek létrehozása rekurrens neurális hálók segítségével Keras környezetben Tensorflow backend használata mellett. A szakdolgozat további célja annak vizsgálata, hogy a nyelvi modellezés kontextusában milyen multi-task tanulási (*multi-task learning*) feladatok fogalmazhatók meg, illetve hogy ezek milyen eredmény elérésére képesek.

Tartalmi összefoglaló

A rekurrens neurális hálók már a 80-as években megjelentek. Ezek olyan neurális hálók, melyek figyelembe veszik az előző állapotokat a döntéshozatalban. A ma használt rekurrens hálók közül az lstm azaz a long shot-term memory a legkedveltebb mind közül, mert megoldást talál az rnn-ek egy alapvető problémájára, a gradiensek drasztikus növekedésére vagy csökkenésére, melyek ellehetetlenítik a hosszútávú tanulást. Ezt a hálótípust alkalmazom dolgozatomban.

Dolgozatom célja lstm rétegekkel létrehozni egy modellt, amely képes megtanulni egy adott előadó zenei stílusát karakterek sorozatát nézve. Pontosabban felteszi magában a kérdést: "ha ezt az x hosszú szöveget látom, vajon az előadó mit írna x+1. karakternek?". A model létrehozásában a python nyelven elérhető keras és annak hátterében a tensorflow keretrendszereket használom. Keras egy API amely elfedi a neurális hálókhoz szükséges matematikát, így átláthatóbbá téve a kódot, tensorflow pedig egy eszköz mellyel gépi tanuló szoftvereket könnyedén tudsz tanítani gyorsasága miatt, valamint átláthatóvá teszi a fejlesztést a tensorboard segítségével, mely egy vizualizációs eszköz.

Szakdolgozatomban először ismertetem az egyszerű neurális hálókat, működésüket, majd ismertetem a rekurrens hálókat azok hasznát, és kitérek a problémájukra melyet az lstm old meg. Ezután ismertetem a keras keretrendszerét, a tensorflow működését és ezen belül a tensorboard-ot. Ezek ismeretében már olvasható a tensorboard vizualizációja, így megmutatom a tanítások eredményeit.

Bevezetés

A neurális hálók egyre több figyelmet kapnak a mindennapokban, elképesztő teljesítményekre képesek mint például az AlphaGo, mesterséges intelligencia képes volt legyőzni a világ legjobb go játékosát. Ezt az eseményt nem várták a következő évtized távlatában.

Egyesek úgy látják, hogy a neurális hálók leváltják a tradicionális programozást és az emberek csak felügyelni fogják az algoritmusok működését, finomhangolják a hiperparamétereket. Rengeteg helyen már sikerült is áttörést elérnie az imperatív programozás ellen. Ilyenek például a gépi látás, beszéd felismerés, robotika de rengeteg más terület vár a hálók hódítására konstans számításigénye és memóriaigénye, könnyű áramkörbe égethetősége és agilissága miatt.

Mások úgy gondolják ez is csak egy eszköz, és a mesterséges intelligencia öregebb mint a számítástudomány. Hiszen egy egysoros bash kód erősebb tud lenni mint egy Hadoop klaszterek csoportja.

Bármi is vár a mesterséges intelligenciára az biztos, hogy egyre több jelentősége lesz az életünkben. A mesterséges intelligencia egy fontos kutatási területe a természetesnyelv-feldolgozás (angolul : natural language processing, röviden NLP). Vannak imperatív algoritmusok hasonló feladatokra, de egyre nagyobb jelentőséget kap a feladat neurális hálókkal történő megvalósítása. Ilyen feladatok például a gépi fordítás, chatbotok, text-to-speech. Ezek mind olyan feladatok amelyek a gépek és az emberek közötti kommunikációt könnyítik meg.

Dolgozatom témája ebbe a témakörbe tartozik, célja automatikus dalszöveg generálás az előző karakterek megfigyelése alapján. A modell próbál értelmezhető magyar dalszöveget gyártani úgy, hogy közben követi az adott zenész/zenei egyesület stílusát. Mivel a karaktereket választottuk absztrakciós rétegnek az inputnak, így nem várható el érhető magyar szöveg az outputban, jó eredménynek számít viszont, ha érthetőnek, természetesnek hat a generált

szöveg, bár a validálást az utolsó fejezetben statisztikailag végezzük, nem példák segítségével.

1. fejezet

Neurális hálók

A neurális hálók az agyunkban előforduló neuronok hálózata. Ezek határozzák meg a gondolatainkat, ez alapján hozunk döntést a minket körülvevő világról. Ahhoz, hogy ezt számítani lehessen szükség van egy matematikai formulára, egy modellre. A neurológia jelenleg elfogadott elméletére építjük a mesterséges neurális hálókat, és ezeknek az alapjait ebben a fejezetben fogom ismertetni. Minden mély háló őse az előrecsatolt mesterséges neurális háló, így a dolgozatban használt rekurrens, azaz hosszú távú memóriával rendelkező háló az LSTM alapja is. A fejezet ezeket a témákat fogja jobban szemügyre venni. A továbbiakban neurális hálók alatt a mesterséges neurális hálókat értem.

1.1. Neurális hálók és a gépi tanulás

A mélytanulás és a gépi tanulás kifejezések a köznyelvben gyakran összemosódnak, viszont lényeges különbség van a kettő között. A gépi tanulás feldolgozza az adatot, tanul a megfigyeltekből, és ezt később felhasználja ha hasonló helyzetbe kerül. Ezt úgy éri el hogy a programozó egy adott programozási nyelven leimplementálja az adott lépéseket: feldolgozás, tanulás, predikálás. Szóval ha egy problémát észlel a rendszerben, megkeresi a probléma forrását és kijavítja az adott metódust/osztályt, amely a problémát előidézte.

Ezzel szemben a mélytanulásban nem a részegységeit készíti elő a programozó, hanem megadja az input adatot, valamint a várt eredményt, és beállítja a modell hiperparamétereit, úgy hogy a tanulás hatásos legyen. Tulajdonképpen a mélytanulás részhalmaza a gépi tanulásnak, mert ugyanazt a célt szolgálják, viszont a mélytanulásban nem azt kell meghatározni,

hogy milyen alrendszerei vannak az adott rendszernek, hanem megadjuk az (x, y) párokat úgy, hogy $f(x) = y$ legyen, és az f függvényt a gép próbálja meg közelíteni úgy hogy nem ismert (x_1, y_1) párra is jól tippeljen a háló.

1.2. Felügyelt és felügyelet mentes tanulás

A tanulás egy másik csoportosítási szempontja, hogy felügyelt-e a modell tanulása. A felügyelet azt jelenti jelent esetben, hogy az input adat egy címkét kap, azaz megmondjuk mi a várt output. Ezzel szemben a felügyelet mentes modell célja automatikus információkinyerés az input adatból.

Ilyen például a SOM (azaz Self Organizing Map), mely egy adathalmaz klaszterizálását viszi véghez, vagy az autoencoder, mely célja, hogy kódolja az inputot, és visszaalakítsa azt. Ennek egy felhasználási módja például az input zajmentesítése.

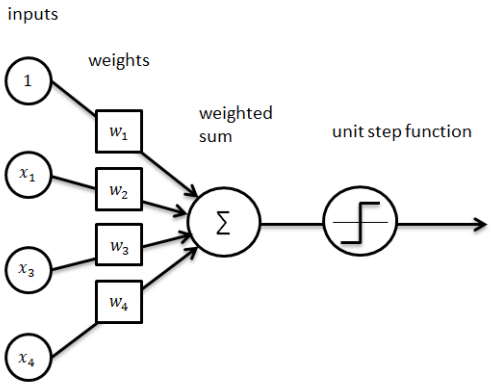
A rekurrens neurális hálók, amit én is fogok alkalmazni a dolgozatomban a felügyelt tanulás csoportjába tartoznak, mivel definíció szerint a rekurrencia azt mondja meg, hogy *x* lépés után mi lesz az *x+1*. lépés, ehhez az inputban meg kell adnunk az *x+1*. lépést, ekkor ezek lesznek a címkék, melyekre a modell optimalizálja magát. Ebbe a kategóriába tatoznak például az előrecsatolt valamint a konvolúciós neurális hálók.

1.3. Előrecsatolt neurális hálók

Az előrecsatolt neurális hálók olyan adatszerkezetek, melyek képesek megbecsülni az f függvényt úgy, hogy $y = f(x, T)$ ahol T a hálónak adott legjobb eredménnyel becslő hiperparamétereket tartalmazza. A nevét onnan kapta, hogy az információ áramlásának iránya az inputtól az outputig tart, és nincsenek benne hurkok, ciklusok.

A legegyszerűbb működő neurális háló egyetlen perceptronból áll, melynek van egy súlya (W) és egy ferdesége (b). Ezekből a neurális háló előállítja a $\hat{y} = Wx+b$ egyenletet, és ezt optimalizálja a tanuló inputra és outputra hiba-visszaterjesztés alkalmazásával.

Sajnos ennek a megoldásnak van egy limitációja, csak lineáris regresszió előállítására képes. Erre hozták létre az aktivációs függvényeket, melyek nem lineáris függvények, és minden perceptron számításának a eredményére egy nem lineáris függvény hívódik



1.1. ábra. A perceptron

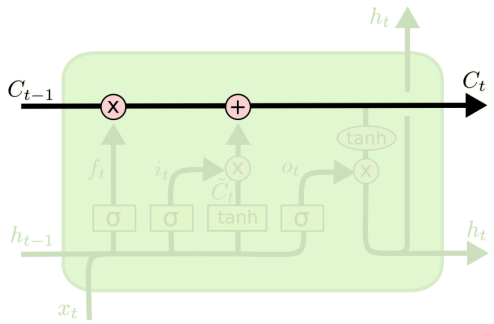
meg. Ezeket a függvényeket aktivációs függvényeknek nevezik. Rengeteg létezik belőle, de itt van néhány példa a leggyakrabban használtak közül: *reLU*, *sigmoid*, *tanh*. Szóval ha hozzáadjuk az aktivációs függvény *g*-t az előző példánkhoz a következő egyenlőségrendszert kapjuk.

$$\hat{y} = g(z),\ ahol\ z = Wx + b$$

A neurális háló attól lesz mély neurális háló, hogy több perceptron réteget egymás után kötünk. Ekkor lesz egy input réteg, lesznek köztes, azaz láthatatlan rétegek és lesz egy output réteg. Több hálóval mélyebb tudást tud szerezni egy modell, viszont feladatfüggő, mivel van olyan feladat amely jobban teljesít egy rejtett hálóval mint többel.

1.4. Rekurrens neurális hálók

Az emberek új gondolatai nem törlik ki az előzőeket. Nem az adott pillanatból ítéljük meg a helyzetünket, vagy hozunk döntést azzal kapcsolatban. Ez egy hasznos tulajdonság, hiszen kevés dolog van az életben, ami nem egy folyamat része. A rekurrens neurális hálók célja ezen folyamatok lemodellezése, így mivel tudja mi történt a múltban hosszútávú kapcsolatokkal is számításba venni modelljében. Minden egyes *t* időpillanatban az adott állapot megkapja saját inputját x_t valamint az előző időpillanat rejtett állapotának eredményét h_{t-1} és ez alapján a következő egyenlettel végzi el a predikciót az adott perceptron: $h_t = = f(Wx_t + h_{t-1} + b)$. Jól látható hogy a h_t egy az *t*-től függő egyenlet, mely egy általános problémához vezet minket. Az eltűnő és kirobbanó gradiens problémájához.



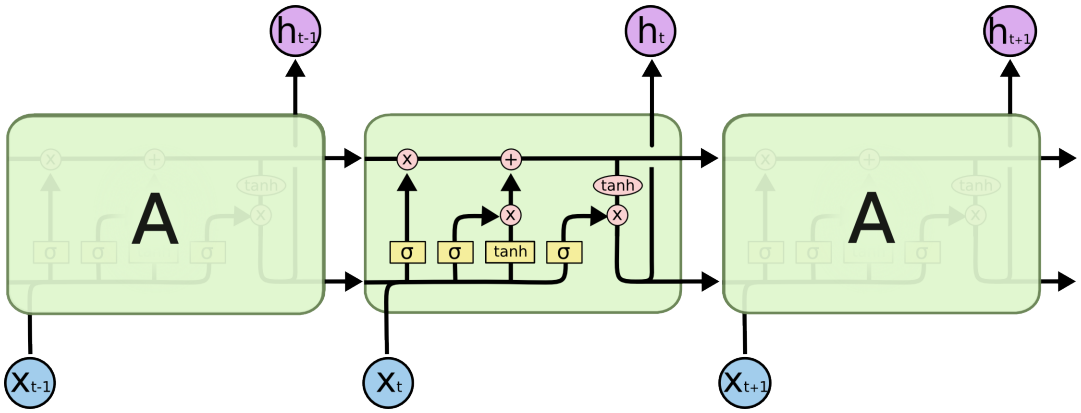
1.2. ábra. Az LSTM megoldása a gradiens kritikus szélsőértékének elkerülésére

1.4.1. Az RNN-ek problémája

A rekurrens neurális hálók képesek nagyon nagy pontossággal dönteni, de hihetetlenül nehéz őket jól betanítani, elsősorban az eltűnő és kirobbanó gradiens problémája miatt. Ugyanúgy ahogy az előrecsatolt neurális hálók, a rekurrens hálók is hiba-visszaterjesztéssel optimalizálják perceptronjaikat. A probléma ott mutatkozik, hogy a rejtett rétegek összeköttetésben vannak egymással, így ha frissíteni akarjuk az egyik réteget akkor az összes előtte lévőt frissíteni kell. Ekkor a háló első rétegei rengetegszer változnak a tanulás során, és ha a gradiens, mellyel beszorozzuk túl nagy vagy kicsi lesz, az értéktelenné teszi az adott perceptront, mivel az összes utána lévő neuron visszaterjeszti a hibáját rá, és ha rengetegszer szorzunk egy kis számmal, vagy egy naggyal akkor az megközelíti a nullát vagy a végtelent. Ekkor az első neuronjaink használhatatlanná válnak, de mivel minden réteg megkapja az előző réteg outputját, így az utolsó neuronok is értéküket vesztik. Ezekre a problémákra született rengetek megoldás. Kirobbanó gradiens : levágott hiba-visszaterjesztés, büntetőfüggvények, gradiens megfékezése. Eltűnő gradiens : súlyok inicializálása, valamit az rnn-ek egy altípusa, amelyet a következő alfejezetben fejtek ki, az LSTM.

1.4.2. LSTM

A Long Short Term Memory (LSTM) kulcsa az úgynevezett cella, egy hosszú egyenes vezeték, melyen csak lineáris transzformációkat hajtunk végre, így nem lesz radikális a változás a kezdő és végállapot között, ezzel elkerülve a parciális derivált kiugróan magas vagy alacsony értékeit. Ennek a vizuális szemléltetése a 1.4.2 ábrán látható. Ez a vezeték felel a hosszútávú memóriáért, ez a hidden state amit minden réteg az előző rétegtől kap.



1.3. ábra. Az lstm belső állapotai balról jobbra sorban látható kapuk: elfelejt, megtanul, output

Ezt a vezetéket módosítja az elfelejtő kapu (forget gate) mely egy *sigmoid* aktivációs függvényt tartalmazó réteg, valamint az új információt szolgáltató cső, mely két részből épül fel. Az egyik egy *sigmoid* aktivációs függvénnyel ellátott réteg, a másik pedig *tanh*-val van ellátva. Először a *tanh* kiszámolja az új információt amelyet továbbítani szeretne outputként majd átadja a *sigmoid*-nak. A *sigmoid* 0 és 1 közé képzi le a az inputját, minden *sigmoid* után áll egy elemenkénti szorzás operátor. Ez a konstrukció képes eldönteni, hogy mely elemnek milyen jelentősége van, mennyire releváns az adott kontextusban. Ez után a szűrt outputot hozzáadja a szűrt hosszútávú memóriához, amelyet majd a következő iteráció fog megkapni a későbbiekben. Az adott réteg outputja viszont úgy generálódik, hogy ezt a csövet, mely az új hosszútávú memóriát tartalmazza átvezetjük egy *tanh*-s rétegen, hogy -1 és 1 közé kerüljenek az értékek, valamint alkalmazunk még egy *sigmoid* szűrőt a kapott értéken ezzel megkapva az outputot, ami a következő réteg inputja is egyben.

Fontos megjegyezni, hogy mind a *sigmoid*-ok, mind a *tanh*-k neuronok, melyek tanulás alatt optimalizálják a változóikat (W és b) ezzel megtanulva mit kell továbbítani a következő rétegeknek, valamint a globális információból (a cella) mely információ releváns számára. Hiba-visszaterjesztéskor ezek változnak radikálisan, hisz ezeknek jóval kisebb vagy nagyobb tud lenni a gradiense, mint a lineáris transzformációknak.

2. fejezet

Használt eszközök

Rengeteg könyvtárat hoztak létre melyek segítik a mélytanulás leimplementálását. Először is meg kell említeni, hogy az elmúlt években a Python nyelv vált a mélytanulás de facto standardjává. Létezik más nyelvre is ismert könyvtár pl: Java - DeepLearning4J, lua - torch, Matlab - Neural network toolbox, de kétségtelen a Python előnye a piacon.

Ott van például a Caffe, melyet a Berkeley egyetem egy tanulója, Yangqing Jia PhD tanulmányai során készített, és az óta is közkedvelt keretrendszer. Másik példa a Microsoft Cognitive toolkit, melyet az óriás vállalat tart karban, így plusz funkcióként az Azure-t is támogatja. Hasonló helyzet áll fent a PyTorch-al kapcsolatban is, melyet a Facebook kezel és a Torch nevű lua framework python-ra való portolása, mely támogatja a dinamikus hálókat. Úgy mint a PyTorch, a dynet, mely a Carnegie Mellon Egyetemen született, is támogatja a dinamikus modellt.

Amikor döntöttem keretrendszer választás tekintetében, akkor ezekről le kellett mondanom, mivel szerettem volna használni a magas szintű API-t amelyet a Keras nyújt, és amelyre a következő szekcióban kitérek. Amikor a kódot elkezdtem leimplementálni csak a TensorFlow és a Theano volt kompatibilis a Keras-sal(, igaz azóta a Microsoft Cognitive Toolkit is kompatibilis vele).

Theano könyvtár egy matematikai segédeszköz, mely gyors és pontos számításokra képes, mátrix műveleteket optimalizálja, valamint CPU-n és GPU-n is futtatható. Egyetlen probléma vele, hogy a fő karbantartója MILA (Montreal Institute for Learning Algorithms) abba hagyta a fejlesztését, és jelentősen visszaesett a fejlesztői hozzájárulások száma az elmúlt fél évben.

2.1. ábra. A hármas if-then-else operátor pythonban

```
a = b if x else c
```

TensorFlow a ma használt egyik legkedveltebb mélytanuló könyvtár, jól dokumentált, minden megtalálható benne ami egy gépi tanulást alkalmazó programnak kell. Igaz csak statikus modelleket képes létrehozni, de könnyű debuggolni, a hozzá lefejlesztett segédeszköz, a TensorBoard segítségével. így a választásom a TensorFlow-ra esett.

2.1. Python

A Python egy interpretált nyelv, csak úgy mint a böngészőkben futó JavaScript nyelv, nagy előnyük, hogy lehetőség van érthetőbb kód írására velük, viszont ezek a nem olyan gyorsak mint a fordított nyelvek, például: c, c++.

A nyelv szintaxisa megközelíti a pszeudokódét, például a hármas operátort a 2.1 forráskódban látható módon definiáljuk, valamint sok vezérlési szerkezet helyettesítve lett. Például az && valamint a || ki lett cserélve az *and* és *or* kulcsszóra, a kapcsos zárójelek helyett indentálással, azaz szóközök vagy tabulátor alkalmazásával látható hogy az adott sor melyik függvény vagy vezérlési szerkezet látóköréhez tartozik.

A választott nyelv nagy ellenfelei az R és a Matlab nyelvek, hiszen kutatók legkedveltebb nyelvei könnyű szintaktikájuk miatt, viszont ezek nem általános programozási nyelvek, a Pythonnal ellentétben. Itt a webszerver épülhet részben ugyazokból az entitásokból, mint amit a tanító kód használt. Ezt még az is segíti hogy a nyelv objektum orientált, így a fejlesztőknek nem is kell foglalkoznia a tanítás részfolyamataival, elég csak integrálniuk a rendszerbe.

Vannak funkciók melyek nyelvi szinten bele kerültek a Pythonba ami hasonlít a matlab szintaxishoz. Ilyen a tömb indexelés, ha leakarjuk kérni a tömb 2., 3. és 4. indexét, csak beírjuk hogy tomb[2:5], vagy ha az utolsóra vagyunk kíváncsiak akkor lekérjük így: tomb[-1].

Sajnos a Python általános programozási nyelv, ezért nem várhatjuk el hogy minden szintaxis, amely mátrix specifikus bekerüljön a nyelvbe, erre feltudjuk használni a Numpy könyvtárat mely a Python mátrixszámolás hiányosságait hivatott orvosolni (ezt a 2.1 táblázat szemlélteti).

2.1. táblázat. Numpy és Matlab

	Numpy	Matlab
Transzponálás	Y=X.T	Y=X'
Mátrixszorzás	C=A.dot(B)	C=A*B
Elemenkénti szorzás	D=A*B	D=A.*B

2.2. Keras

A Keras, mint már említésre került, egy magas szintű API, melynek fő célja a gyors kísérletezés. Ahogy a honlapjukon található idézet írja: 'Az egyik legfontosabb dolog a kutatásban, hogy minél gyorsabban elérjünk az ötlettől az eredményig.'. Kulcsfontosságú volt a fejlesztése során a modularitás és a felhasználó barátság, cserébe nem kapunk olyan gyors algoritmust, melyet mondjuk akkor kapnánk, ha TensorFlow-t használnánk, Keras nélkül.

Az API alapja a *Model* osztály. Ebből funkcionális programozás segítségével komplex gráfokat tudunk létrehozni, de ha csak sor-folytonosan szeretnénk rétegeket hozzáadni akkor az ebből leszármazó *Sequential* osztályt kell alkalmazni, melynek az *add* metódusa paraméterül egy *Layer* objektumot vár, és hozzáadja következő elemként a modellhez. Ez után a *compile* függvénnyel megadhatjuk mit használjon, hibafüggvénynek, ami alapján megmondja hogy teljesít a modell, valamint mi legyen az optimalizáló függvény, ami a hiba-visszaterjesztést vezérli, valamint milyen metrikákat használjon pl: loss(hibaérték), accuracy(pontosság).

A compile után a modell kész a tanulásra. *Numpy* tömböket vár inputként és azt is ad vissza outputként. A tanulásnak több módja van, például a *fit* a legegyszerűbb, két kötelező paramétere van: *x*, azaz az input amiből tanul a modell és *y*, az output melyet képesnek kell lennie predikálni. Az én esetemben viszont ez nem volt járható út, mivel a korpusz (így nevezik a szöveg alapú adatot, melyből a modell tanulni képes,) amelyből tanul elérheti a $(730441 - 100) * 101 = 73764441$ elemű multidimenziós tömböt (ezt a rock.txt választással, 100-as szekvencián tudjuk elérni). Ebben az esetben használható az alacsonyabb szintű *train_on_batch* függvény, így futásidőben képes voltam az inputot kigenerálni a Keras számára.

Attól függően hogy milyen módon adjuk meg az inputot, a tesztelés is másképp működik. A *fit*-hez hasonlóan tesztadatra megnézhetjük hogyan teljesít modellünk még nem látott adaton a *evaluate* és a *train_on_batch*-hez hasonlóan működik a *test_on_batch*. Ha szeretnénk egy konkrét predikciót kapni akkor a *predict* függvényre van szükségünk, ahol egy adott x-re visszakapjuk az output y-t., ahol az x megfelelően formázott input, az y pedig a predikció eredménye.

2.3. TensorFlow

A TensorFlow egy a Google által karbantartott keretrendszer, mely az elmúlt időben egyre jobban fejlődik. Alapból c++ -hoz készítették, valamint egy Python interfészen keresztül kényelmesen fejleszthető, gyorsaság romlása nélkül, mert a Python is a motorháztető alatt ezt a c++ -os optimalizált könyvtárat használja. Viszont már létezik JavaScript-es és telefon kompatibilis (TensorFlow Lite) változata is. Az alap verzió is fejlődött, hiszen már nem csak CPU-val tudunk számítani, hanem GPU-val és TPU-val, azaz tensor processing unit, mely egy konkrétan a TensorFlow-hoz készített hardver. Ezen felül elosztott rendszereket is támogatja. A TensorFlow egy matematikai optimalizációért felelős könyvtár, mely rendkívüli gyorsasága miatt lett közkedvelt a mesterséges intelligenciával foglalkozók körében. A Python programozási nyelv egy szkriptnyelv. Ezzel nagyon sokat nyerünk, hiszen egy sor kód felelős lehet több sornyi c kódért, mivel a Python rendelkezik magas szintű adatszerkezetek pl : generátor, inline for ciklus, trenary operátor egy emberileg olvashatóbb formája. Viszont ezzel gyorsaságot veszítünk, hiszen a Python sorról sorra fordítja át a kódot c kóddá, majd ezt hívja a python interpreter, ha szükség van az adott kódrészletre. Ennek a lassúságnak elkerülése érdekében találták ki a TensorFlow-t mely a háttérben előre legenerálja a c kódot, optimalizálva a műveleteket, így a neurális hálókhoz szükséges intenzív matematikai számítások napokról órákra csökkenhetnek.

A keretrendszer elméleti háttere, hogy a modellünket egy adatfolyam gráfnak tekinti, melyekben a folyamok találkozási pontjánál transzformációk történnek. Az adat amely keresztül megy a gráfon, egy tensor. A tensor egy geometriai objektum, mely más tensorok, skalárisok és vektorok között ír le lineáris relációkat (mint például mátrixszorzat, skaláris szorzat). Mivel matematikai számításokra fejlesztették hasznos mélytanuláshoz, sőt még

nagyon sok mélytanuló tartalom is van benne, hibafüggvényektől és optimalizáló függvényektől kezdve kész modelleken át. Dolgozatom kezdete óta a Keras is része lett a TensorFlow könyvtárnak.

2.3.1. TensorBoard

Az előző szekcióban említett mélytanuló könyvtár egy nagyon nagy előnye ellenfeleivel szemben a beépített vizualizációs eszköz, a TensorBoard. Ennek segítségével kevés kóddal nagyszerű vizualizációkat figyelhetünk meg tanulás közben és után is. A TensorFlow számokon keresztül képes tanulni, számot vár inputként, számot ad outputként, mivel ez egy optimalizálási feladat. Ellenben az embereknek nehéz felfogni hatalmas adatok táblázatát, erre jó a vizualizáció, hogyha valami nincs rendben a modellel egy gyakorlott adattudós könnyen észreveheti hol rontotta el a modelljét, valamint egy kezdő is meglátja hogyha valami probláma van a modellel. Ilyen eszközök a skaláris vektorok, melyek az idő függvényében mutatják meg egy tensor értékének változását, hisztogramok, melyek az eloszlását nézik az adott tensor értékeinek, a projektorral képes vagy egy vektorteret kirajzolni, melyet PCA (Principle Component Analysis) vagy T-SNE (T-distributed Stochastic Neighbour Rendering), esetleg egyéni leképezéssel. A modell architektúráját is közelebbről szemügyre vehetjük, hiszen van egy olyan menüpont, mely egy gráfot generál nekünk az elkészült modellből amelyben láthatjuk hogy a tensorok hogyan folynak végig a számítási gráfon, azaz egy statikus képet kapunk, hogy futásidőben hogyan fog végigfolyni az adat. Viszont dolgozatom megkezdése óta a TensorBoard egy újabb verziója már támogatja a dinamikus hibakeresést is.

3. fejezet

Karakter alapú szöveg modellezés

A természetes nyelvfeldolgozás egy olyan tudományág, melynek célja az emberek és a gépek közötti kommunikáció létrehozása, valamint az adatbányászatra használt eszköz. Az utóbbihoz rendkívül hasznos, hisz rettentő nagy mennyiségű ember által írt forrás található meg az interneten, mely bányászatához sokszor elengedhetetlen a nyelvtudás, ekkor tud segíteni a nyelvfeldolgozás. A beszédfelismerés, természetes nyelv megértése valamint generálása is ezen tudományterület szerepkörei közé tartozik. Ezekre a célokra nagyon sok algoritmus született az elmúlt évtizedben, viszont az idő azt igazolta, hogy a neurális hálók legalább olyan jók erre a célra mint a létező NLP algoritmusok.

Dolgozatomban egy karakter alapú neurális hálót hoztam létre, mely az előző karakterek alapján képes megjósolni, hogy a tanult korpusz (egy zeneszerző) mely karakterrel folytatná a kapott sztringet (dalszöveget).

Az írott szöveg modellezésének több absztrakciója is létezik: karakter, token, szó és mondat alapú. Ezek a balról jobbra egyre komplexebb egységek, ami azt jelenti, hogy több adatra van szükség, hogy a modell releváns értéket adjon vissza, hiszen például karakterből sokkal kevesebb van mint szóból, így ha szó alapú modellt használnánk rengeteg irreleváns információt kapnánk ritka szavakról (bár erre is van megoldás, melyre az első szekcióban kitérek). Ez azt jelenti, hogy nagyon nagy adathalmazzal és számítókapacitással megéri magasabb szintű absztrakciót használni, viszont én az egyszerűség és a könnyű fejlesztés (kevés várakozás a modellek futtatása között,) miatt a karakteralapú modellezést választottam.

3.1. Preprocesszálás

Minden neurális modell első és az egyik legfontosabb része a preprocesszálás, azaz az adatok előkészítése a tanulásra. Jelen esetben a először is szükségünk van egy leképezésre, mely a karakternek egy egyéni azonosítót ad 0 és az összes eltérő karakter száma (továbbiakban *N*) között. Ez után átalakítjuk a számokat one-hot kódolást használva, azaz minden karakter kap egy *N* dimenziós vektort, és az *n*-edik azonosító lineárisan független az *n+1*-ediktől. Ezzel azt érjük el, hogy miközben a modellünk tanul, nem próbál meg relációt vonni a karakterek között például ha nem használunk kódolást, és azt mondjuk hogy a indexe 1 és b indexe 2, azt is gondolhatja a modellünk hogy b kétszer olyan értékes mint a, ezért van szükségünk a lineáris függetlenségre, hisz akkor nem állhat fenn rezonancia.

Az újabb megvalósításhoz egy másik módszert használunk, melyhez nincs szükségünk a one-hot kódolásra, mert a modell meg fogja tanítani az adott karaktert reprezentáló vektorokat minden vektorra, de ezek nem lesznek lineárisan függetlenek, relációba állíthatóak a karakterek, viszont ezek a relációk várhatóan relevánsak lesznek, nem úgy mint az előbb felhozott példám. Ezt a megvalósítást a keras *Embedding* rétege fogja megvalósítani.

A preprocesszálást a *scikit-learn* csomag által definiált encoder séma szerint csináltam. Lényegében van egy *fit* függvénye, mely beállítja az encoder-t az input alapján, egy *transform*, mely visszaadja a megváltoztatott inputot, ezeknek a kombinációja a *fit_transform*, mely sor-folytonosan meghívja a két függvényt. Az *inverse_transform* melynek célja hogyha f a transform függvény és g az inverse transform, akkor $g(f(x)) = x$ bármely *x*-re.

Célom volt továbbra az, hogyha egy olyan karaktert kap az encoder amellyel még nem találkozott, akkor azt ritka karakterként kezelje, ahelyett, hogy leállna a predikáló program.

Kialakítottam egy rendszert arra, hogy transzformációban extra lépések is megtehetőek legyenek, például az általam is használt *lowercase*, azaz kisbetűs transzformáció.

Ez a kód megtekinthető a Függelékben a 6.1 szekcióban.

3.2. A modell

A modell a Keras keretrendszer objektumaiból épül fel, a szekció ezekre tér ki. Mivel dolgozatom során nem volt szükségem komplex modellre, elég volt a *Sequential*-t használnom, melyhez a *add* függvénnyel lehet hozzáadni réteget. A legutoljára felhasznált osztály, melyre már említést tettem az *Embedding* réteg, melyről a dokumentáció is írja, hogy csak a modell első rétegként alkalmazható. Ennek 3 paraméterét adjuk meg, a szótár mérete, az elvárt embedding egységek száma, azaz mennyi dimenzióra szeretnénk levetíteni az inputot. Ezt a számot magunknak kell megtapasztalnunk hogyan szeretnénk alkalmazni, ajánlott 50 és 1000 között megkeresni az optimálisat. A lényeg az hogy a hasonló szavak közelebb kerüljenek egymáshoz a kialakult vektortérben.

Az *Embedding* a Keras keretrendszeren belül valójában egy *Dense* olyan réteg, mely paraméterül vár egy one-hot kódolással ellátott vektort. Ez csak egy segítség a fejlesztő számára, hogy levegye a terhet a válláról, valamint a TensorBoard használatával vizualizálhatjuk az szótár elemei közötti kapcsolatokat.

Az embedding tulajdonképpen egy adathalmaz elemeit próbálja relációba állítani. Főleg szavak relációba állítására használják, de a fent említett példából látható, hogy az általam létrehozott karakter alapú embedding-nek is van emberileg felfogható értéke. Léteznek nem mélytanuló algoritmusok szóbeágyazásokra, valamint vannak előre tanított adathalmazok, ha olyan programot szeretnénk írni amiben szó alapú nyelvi felismerésre van szükség. Ilyen például a Google által karban tartott *word2vec* valamint a nagy riválisa a *GloVe*. A kettő között a lényegi különbség hogy a *word2vec* predikciókat végez, ameddig a *GloVe* statisztikai alapon végzi a számolást, viszont a motorháztető alatt mindkettő a szöveg kontextusából próbál meg rájönni az adott szó jelentésére.

A már említett *Dense* réteg egy teljesen összekapcsolt réteg, az input mindegyik eleme hozzá van kapcsolva az output összes részéhez. Ha sima neurális hálóra lenne szükségünk ezeket kellene használnunk, különböző mennyiségben, különböző számú output egységekkel.

A következő réteg az *LSTM*, azaz a rekurrencia részét képző háló/hálók. Egy olyan háló, mely önmagába csatolódik vissza minden input karakter után. A következő paramétereket használom: *units*, azaz hány egység legyen benne, hasonlóan működik mint a *Dense* egységek, *return_sequences*, mely ha igazra van állítva, mint jelen esetben, azt mondja

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

3.1. ábra. A softmax függvény matematikai leírása

meg hogy az összes állapotot adja-e vissza, valamint a *statefulness*, azaz a következő állapot az előzőt vegye-e paraméterül, ami szintén igazra van állítva.

Utoljára egy *Dense* réteget adunk meg, *softmax* függvénnyel, melynek output dimenzióinak meg kell egyeznie az *y* dimenzióival, mivel ez a modellünk outputja.

A *softmax*-al azt érjük el, hogy az inputot 0 és 1 közé szorítjuk, úgy hogy ha inputnak a *Dense* réteg outputját vesszük, akkor az visszaad egy tensort, mely elemei összegének a transzformáció után 1-et kell visszaadna. Így a mi esetünkben megkapjuk, hogy melyik tensor milyen valószínűséggel aktiválódik, ami felhasználható később predikálásra.A 3.2 ábra mutatja a leírását, valójában ez egy normalizált exponenciális függvény ahol z_k a *Dense* réteg $k.$ outputja. Ezzel elkészül a magas szintű keras modellünk, ez után a *compile* függvénnyel elkészül ebből a TensorFlow modellünk. De ehhez előbb még meg kell említenünk a *compile* metódus összes szükséges paraméterét. Ezek a hibafüggvény, az optimalizáló és opcionális a metrikák.

3.2.1. Hibafüggvény

A hibafüggvény a matematikai optimalizációban, statisztikában, gépi tanulásnál és mélytanulásnál használt kifejezés. Tulajdonképp vektorokat, mátrixokat (a mi esetünkben tensorokat) egy valós számra vetít le, melyet a fent említett területeken használnak, hogy számszerűsíteni tudják a tanulás állapotait, azaz meg tudja nézni hogy javult-e a tanulás az adott iterációban (más néven kisebb-e a hibafüggvény értéke).

A hibafüggvényt a programozó feladata meghatározni, hiszen rossz hibafüggvénnyel a tanulás értelmetlen. Kerasban a kész modellünk compile függvényében szerepel a hibafüggvény, kötelező paraméterként. Mi magunk is létrehozhatunk hibafüggvényeket, melyben paraméterként kapunk egy *y_true* és egy *y_pred* tensort és ez alapján kell kiszámolni a hibát. Ha általános hibafüggvényre van szükség a Keras magába foglal rengeteget közülük. Ezeket jelentésükkel együtt a következő táblázat fogja szemléltetni. Ezeket begépelhetjük sztringként az alábbi táblázat alapján, vagy a *keras.losses* csomagban található rájuk a referencia, mellyel saját

paraméterekkel is meghívhatjuk őket, de a dokumentációban ki van emelve, hogy ez nem ajánlott.

3.1. táblázat. Keras keretrendszer előredefiniált hibafüggvényei

Név	Leírás
mean_squared_error	n
mean_absolute_error	n
mean_absolute_percentage_error	n
mean_squared_logarithmic_error	n
squared_hinge	n
hinge	n
categorical_hinge	n
logcosh	n
categorical_crossentropy	n
sparse_categorical_crossentropy	n
binary_crossentropy	n
kullback_leibler_divergence	n
poisson	n
cosine_proximity	n

A mi esetünkben a megfelelő választás a categorical_crossentropy lesz, mivel minden karakter egy különálló kategória és azt akarjuk megmondani hogy az adott kategóriának mekkora esélye van a következőnek lenni az input alapján.

3.2.2. Optimalizáló függvény

A compile függvény másik kötelező paramétere az optimalizáló függvény, melyet megadhatunk sztringként is valamint egy callback-el is hozzáadhatjuk a modellünkhöz. A keras előredefiniált optimalizálói a *keras.optimizers*-ben találhatóak. Az optimalizáló célja minimalizálni a hibafüggvény értékét. Ezek a hibafüggvény deriváltját veszik alapul, célja megtalálni azt a pontot, ahol ha a hibafüggvényünk $f(x) = y$, akkor megkeresi azt az értéket ahol $f'(x)$ értéke negatív, azaz y x viszonylatában csökken. Vannak esetek, mikor a második deriváltat is érdemes felhasználni, de ez ritka, mivel a második deriváltat, azaz a függvény *Hesse* mátrixát számításköltséges kiszámolni.

3.2. táblázat. Keras keretrendszer előredefiniált optimalizáló függvényei

Név	Leírás
SGD	n
RMSProp	n
Adagrad	n
Adadelta	n
Adam	n
Adamax	n
Nadam	n
TFOptimizer	n

3.2.3. Metrikák

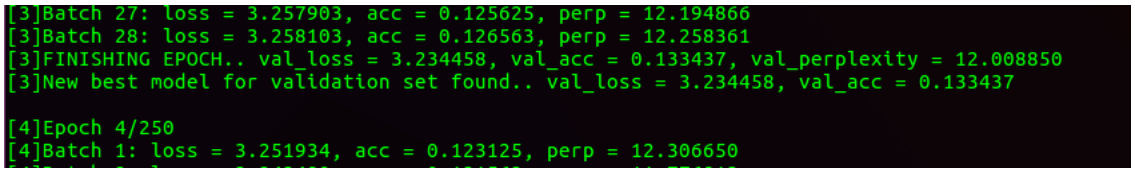
Szintén megadható paraméterként a compile-nak a *metrics*, mely egy tömböt vár, melynek elemei vagy sztringek, vagy egy callback függvény. A kódban ez az egyetlen példa, ahol kihasználom a callback függvénnyel való definiálást, mivel a keras alapból nem tartalmazza a perplexitás metrikát. A perplexitás a természetes nyelvfeldolgozás egy alap metrikája, jelentése az, hogy a modell milyen jól másolja a valódi korupusz eloszlását. A másik felhasznált metrika a pontosság, lényegében visszakérjük a kerastól hogy a modell milyen pontosan határozta meg a várt értéket a tanítás során.

A metrika egy olyan egysége a modellnek, melyet nem használ fel a tanulás során, ez csak a fejlesztő számára nyújt segítséget, hogy megértse hogyan viselkedik a modellje.

3.3. A tanulás menete

Ott tartunk hogy van egy modellünk, az input tanításra kész állapotban van, és a modell le lett compile-olva az adott alacsony szintű nyelvre, adott esetben tensorflow-ra.

A keras több módot is felkínál a modellünk tanítására, melyekről már szó esett a kerasról szóló 2.2 fejezetben. Személy szerint az alacsony szintű *train_on_batch* megvalósítást választottam, mivel az első próbálkozás alkalmával a fit nem bizonyult használhatónak a hatalmas tármennyiségnek amit a preprocesszált input adat megkövetel. Erre használhattam volna a *fit_generator*-t, viszont mikor ezt észrevettem már hatalmas refaktorálásra lett volna szükségem a *lyrics.py* fájlban, mivel az összes callback-et, melyet a *fit* támogat meg



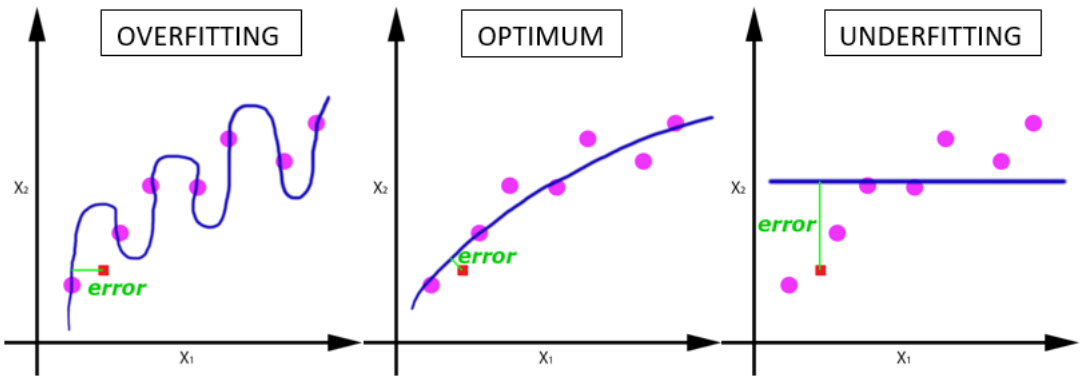
3.2. ábra. Itt látható a program tanulás közben, az első szám az epoch száma, a többi a batch, valamit a metrikáké

kellett valósítanom a saját kódomban, ennek köszönhetően jobban megértettem a tanulás folyamatát. Ezek a tensorboard vizualizációi, valamint az korai megállás.

Az early stopping, azaz korai megállás azt jelenti, hogy habár a tanuló adathalmazon állandóan egyre jobb értékeket sikerül elérnie, a validációs halmazon látszik, hogy van egy pont, ahol elkezd romlani az érték. Ekkor ha hagyjuk a modellnek hogy tovább tanuljon egyre rosszabbul teljesít majd a validációs halmazon, és egyre jobb lesz a tanító inputon, ezt a jelenséget szeretnénk elkerülni. Ezt egyszerűen megtehetjük, de előbb pár a tanításhoz fontos fogalommal meg kell ismerkednünk.

3.3.1. Epoch, Validáció, Teszt

Már említettem, hogy először a *fit* függvényt használtam, mely a tanulás egészét elrejti előlünk. Viszont a *train_on_batch*-el való megvalósítással sokkal mélyebben belelátunk a tanulás folyamatába. A legkülső héja a tanulásnak az epoch, ez azt mondja meg, hányszor fusson át az egész adathalmazon a modell, mielőtt befejezi a tanulást. Erre azért van szükség, mert a *Gradient Descent*, az optimalizáló algoritmusok alapja iteratívan működik, így ha csak egyszer megyünk végig az adathalmazon nem látja eleget az adatot, nem tudja megtanulni a relációt az input és az output között, ekkor alultanulásról beszélünk. Ennek az ellentettje a túltanulás, ekkor túl sokszor nézte végig az adathalmazt a modell, bemagolta az abban lévő input-output párokat, de új adatra romlik az ítélőképessége. Ennek a vizualizációjára szolgál a 3.3.1 ábra, melyen látható a reláció a reprezentált tudás és az új adatokon való predikálás között. Az epochon belül, mint már említettem egyszer végig megy az összes adaton a program. Viszont ez hihetetlenül nagy memória igénnyel járhat, adathalmaztól függően. Ezért szokták batch-ekre osztani az adatot, melyeket a *read_batches* generátor állít elő, melyet egy for ciklus segítségével végig iterálunk és



3.3. ábra. Balról jobbra: Túltanulás, optimális tanulás, alultanulás. Észrevehető hogy az optimumnál a legkisebb a távolság a görbe és a piros négyzet, azaz az új adat között.

a már említett *train_on_batch* függvényen keresztül odaadunk a kerasnak processzálásra. Ekkor a keras az adott batch alapján egy gradiens frissítést végez, azaz frissíti a háló perceptronjainak súlyát és visszaadja nekünk a metrikákat amelyekre feliratkoztunk. Ez a mi esetünkben a hibaérték, pontosság és a perplexitás volt. A hibaérték és a pontosság között az a különbség, hogy az előbbi az összes lehetséges kimenetre megnézi mennyire tér el a várt eredmény a kapottól és ezt a *categorical_crossentropy* egy valós értékre transzformálja, pontosabban megmondja mekkora különbség van a várt és a kapott eloszlás között, az utóbbi pedig megmondja hány százalékban volt a kapott kimenet egyenlő a várt eredménnyel. A hibaérték láthatóan több információt tartalmaz, de később észrevehető, hogy magas a negatív korreláció a kettő között (, mivel minél kisebb a loss, annál nagyobb az accuracy). Minden egyes epoch végén validálom a modellem, azaz megnézem a javulását. Ehhez a *test_on_batch* függvényt használom. Ennek megegyezik a visszatérési értékének formája mint a *train_on_batch*-ével, viszont nem végez módosítást a háló súlyain. Nagyon fontos megjegyezni, hogy a validáció adaton kell elvégezni, amit nem használtunk tanítás folyamán, különben irreálisan nagy értéket kaphatunk, és később ismeretlen adatra rosszabb predikciót végez.

A tanítás legvégén hasonlóan a még nem látott adattal tesztelhetjük a modellünket, ugyanúgy a *test_on_batch* függvénnyel.

Az ebben a szekcióben leírtak megtalálhatóak a függelék 6.2 részében.

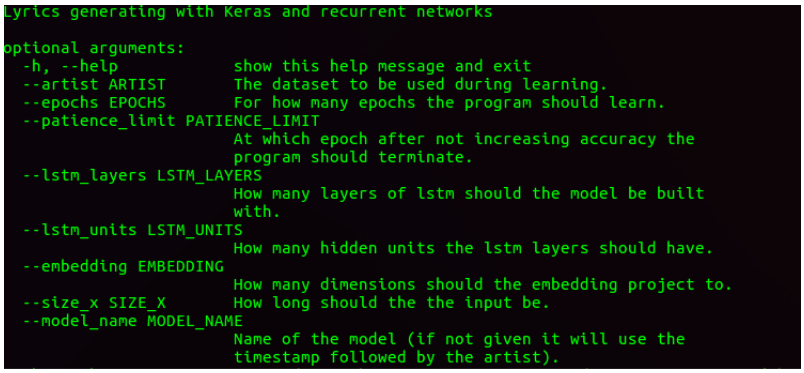
3.3.2. Eredmények feldolgozása

Most hogy megismerkedtünk a tanulás folyamatával megnézzük mit kell tennie egy epoch folyamán a rendszerünknek. Már említettem a korai megállást. Célja, hogy ne tanítsuk túl a modellünket, azaz meg kell néznünk hogy az új validáció során jobban viselkedett-e a modell mint az előtte levőben. Ha jobb volt archiváljuk későbbi felhasználásra, ha ezt nem tesszük meg akkor a program lefutása után a modell elveszik. Ha nem akkor megbüntetjük a modellt, ha túl sok büntetést szerzett, azaz elérte a türelmi szintet (*patience_limit*), akkor leállítjuk a tanulást, ezt a szintet mi állíthatjuk be ha parancssorról indítjuk a programot.

Másik fontos dolog az adatvizualizáció. A program összegyűjti a metrikákat melyet a keras visszaad számára, aggregálja azt, majd az epoch végén veszi az átlagait mind a tanulásnak, mind a validálásnak és átadja a tensorboard objektumnak archiválásra. Szemléltetés céljából a batch-en végzett módosításokat is elmenti a program, viszont erre legtöbb esetben nincs szükség.

3.4. Parancssori interfész

A lyrics.py futtatható parancssorból is, ahol a hiperparamétereit lehet szabályozni tanulás előtt. Első kötelező paramétere a –artist, mely arra utal hogy a dataset mappából mely zenefájlt használja tanításhoz. Ezen kívül választható paraméterként ott vannak a korábban már említett értékek. Ilyen az –epoch, hogy hány iteráción keresztül menjen végig az adathalmazon a program, –patience_limit, hogy hány olyan epoch után álljon le a tanítás melyben nem sikerült a validáción jobb eredményt elérni, ezekről a 3.3.1 szekcióban volt szó. Az –lstm_layers és –lstm_units pedig azt határozzák meg hogy hány darab lstm réteg legyen, és egy lstm rétegnek hány rejtett egységet adjunk. A –embedding az embedding egységek számára utal, melyet a 3.2 szekció tárgyalt ki bővebben. A -size_x azt határozza meg hány karakter után próbálja megítélni a modell hogy mi lesz a következő karakter. Az utolsó paraméter pedig azt kérdezi milyen néven mentse a modell-t, hogy aztán később beolvasható legyen, valamint hogy a tensorboard fel tudja dolgozni, ez a –model_name. Angol nyelvű segítséget tud nyújtani még a –help, mely látható a 3.4 ábrán.



3.4. ábra. A ‘python lyrics.py –help’ -et beütve a parancssorba ez jelenik meg.

4. fejezet

Tanulás eredményei

Mikor elkezdtem a keras-t használni, hogy kitanítsa a modelleket úgy validáltam a munkámat, hogy lefuttattam a *demo.py*-t mely vagy a legvalószínűbb értékű outputot választotta, vagy a softmax által létrehozott disztribúciót vette alapul, és ez alapján döntötte el a következő karaktert, így esélyt adva a kevésbé esélyeseknek is. Ez viszont nem egy olyan megoldás amellyel hosszú távon érdemes egy mélytanuló algoritmust ellenőrizni, így a karban tartását felfüggesztettem.

Ez után döntöttem úgy hogy a TensorBoard-ot használom munkám során a fejlesztés könnyítéséhez. A board lokális szerverként indul el, csak be kell írni a terminálba hogy tensorboard –logdir target/tran_log' amennyiben a tensorboard telepítve van. A target/train_log mappa az ahova ment a tensorflow a vizualizációs eszközben szükséges logfileokat. A tanításnak megadott –name paraméterrel, vagy az adott dátumot és előadót figyelembe véve egy generált nevet kap.

4.1. Skalárisok

Ha van legalább egy modellünk, akár kész akár éppen tanul, ha egy böngészőablakkal odanavigálunk az oldalra amit a konzol kiadott, akkor látni fogjuk a modell tanulásának fázisait.

Ha több modellünk van kiválaszthatjuk az általunk látni kívántat, valamint filterezhetünk rá reguláris kifejezéssel a *Runs* szekcióban. Ekkor megjelenik az összes tanulással kapcsolatos időben változó metrika.

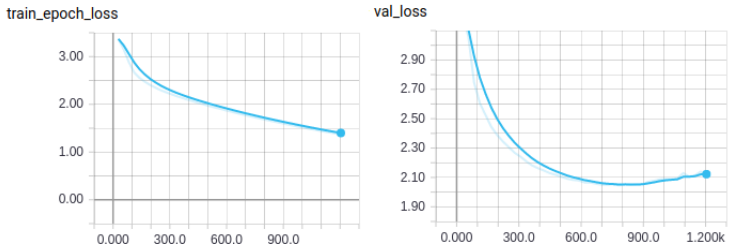
Rákattintva a keresés gombra és beírva a '.*' reguláris kifejezést az összes metrika megjelenik összecsoportosítva. Ezek jelentései a 4.1 táblázatban kerül leírásra.

4.1. táblázat. A tensorboard skalárként ábrázolt metrikái

Név	Jelentés
train_batch_accuracy	Az adott batch tanulása közben adott pontos találatok
train_batch_loss	Az adott batch tanulása közben számolt keresztentrópia
train_batch_perplexity	Az adott batch perplexitása
train_epoch_accuracy	Az adott epochban visszaadott pontosságok átlaga
train_epoch_loss	Az adott epochban visszaadott keresztentrópia átlaga
train_epoch_perplexity	Az adott epochban visszaadott perplexitás átlaga
val_accuracy	A validálás során adott jó találatok átlaga
val_loss	A validálás során számolt keresztentrópia átlaga
val_perplexity	A validálás során számolt perplexitás átlaga

Észre vehetjük hogy tanulás elején a hiba mind a validáción, mind a tanító halmazon nagyon magas. Ez a *Gradient Descent* miatt van, próbálja megközelíteni a tanuló halmaz által reprezentált halmazt, így azt vehetjük észre, hogy a tanuló halmaz hibaértéke valamivel gyorsabban csökken, a pontossága nő. Ezen kívül egy idő után elkerülhetetlen a már említett túltanulás jelensége, ekkor azt fogjuk látni, hogy a validációs halmazon elkezd nőni a hibaérték, még a tanulóhalmazon csökken. Erre egy konkrét esetet mutat be a 4.1 ábra.

A Modellünk pontossága 50-60% közötti értéket ér el becsléseivel a teszt halmazon. Ez a gépi tanulás területén nem számít kimagasló teljesítménynek. A modellt legalább 50% pontosra kellett volna optimalizálni. Ez az érték független attól, hogy dropout segítségével, vagy nélküle futtatjuk, bár már bebizonyították, hogy a dropout nem segíti az lstm tanulását. Viszont érdekes módon ha adam optimalizálóval tanítjuk a modellt a rekurrens hálókhoz ajánlott rmsprop helyett a preplexitást alacsonyan tudjuk tartani, habár a teszt halmazon rosszabb eredményt mutat vele a modell. Tanuláshoz általában száz hosszú inputhoz mértem az egy hosszúságú outputot. A kis-grofo korpuszhoz viszont szükségem volt egy kisebb input méretre, mivel túl kicsi az adatmennyiség, hogy rendesen szét lehessen osztani a tanuló-teszt-validáció között. Ezért egy harminc hosszú inputtal próbálkoztam, ekkor érte el a modell a 80% feletti *train_batch_accuracy*-t, viszont ekkor nagyon gyorsan észlelhető volt a túltanulás hiszen a hibafüggvény már 150-200 iterációnál megjelent (az



4.1. ábra. A túltanulás jelensége egy 1 lstm réteget, 250 rejtett reprezentációval tartalmazó 30y-t tanuló modell túltanulása.

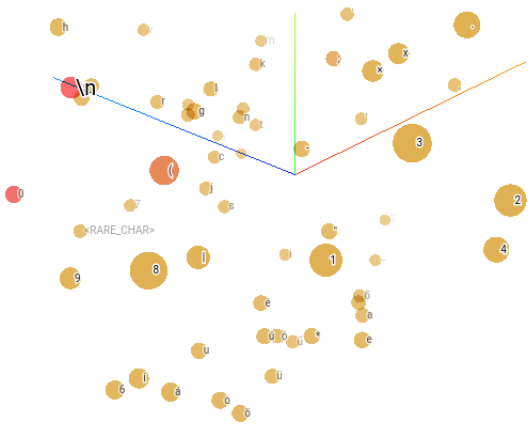
összes 400-ból). A perplexitás csökkenése érthető, hiszen a definíció szerint a kettő a kategorizált keresztentrópia értékére emelve, hiszen nagyobb korpuszban kevesebb karakter található, így a keresztentrópia értéke is csökken.

4.2. A modell

A modell gráf reprezentációja tekinthető meg a Graph fülön. Ez egy hasznos eszköz, hiszen ha nem látni hogy néz ki a modell, az ember hajlamos figyelmetlen módon elírni. Ez történt pontosan velem is, a TimeDistributed wrapper-t véletlenül az lstm-re helyeztem, így az összes lstm kapott egy wrapper-t és az egész gráp átláthatatlan volt. Komplexebb modellnél elengedhetetlennek tartom a gráf használatát, sőt akkor még name scope-ok használata is ajánlott nagyobb átláthatóság érdekében.

4.3. Projektor

A projektorral képesek vagyunk vizualizálni többdimenziós tereket, azaz levetíthetjük őket két vagy három dimenzióra úgy, hogy a vektorok közötti kapcsolatok konzisztensek maradnak. Ezt megtehetjük a tensorflow összes rétegének belső állapotaival, viszont nem mindegyikkel van értelme. Az Embedding egy különleges réteg, mely az inputot alakítja át, hogy ne lineárisan függetlenek legyenek, hanem viselkedésük alapján össze tudjanak csoportosulni. Ez nem elég ahhoz, hogy olvasható legyen az Embedding, hisz a tensorflow nem tudja hogy az adott kódolt szám milyen jelentéssel bír. Szükségünk van egy leképezésre a szám és a karakter között, ehhez a tensorboard egy tsv fájlt kér be, mely ha sikeres volt a beolvasás megjelennek a benne található címkék a vektortérben.



4.2. ábra. A PCA vizualizáció az egyik tanítás eredményére. Látható, hogy a számok egymás szomszédságában helyezkedtek fel.

Ekkor láthatjuk, mely karakter melyikhez hasonlít legjobban. Ahogyan a 4.3 ábra is mutatja a számok összecsoportosulnak. Ez azért van, mert a ugyanabban a kontextusban helyezkednek el. Ez különösen igaz zeneszövegekre, ahol a szám csak metaadatként jelenik meg (például : ‘(3x)’), és ez magyarázza azt is, hogy a különleges karakterek miért kerültek olyan közel a számokhoz.

5. fejezet

Összegzés

6. fejezet

Függelék

6.1. CharEncoder

```
class CharEncoder:

    def __init__(self, formaters=dict()):
        self.onehot = OneHotEncoder()
        self._RARE = '<RARE_CHAR>'
        self.vocab = dict()
        self._onehotted_vocab = dict()
        self._formaters = formaters

    def _format(self, y):
        for _, formater in self._formaters.items():
            y = formater(y)
        return y

    def transform(self, y, with_onehot=True):
        y = self._format(y)
        if not with_onehot:
            return np.array([self.vocab[ch] if ch in self.
```

```
                vocab else self.vocab[self._RARE].flatten()
                for ch in y])
        else:
            return np.array([self._onehotted_vocab[ch] if
                ch in self.vocab else self.vocab[self._RARE
                ].flatten() for ch in y])

    def fit(self, y):
        y = self._format(y)
        vocab = sorted(list(set(y)))
        vocab.append(self._RARE)
        self.vocab = dict((c, i) for i, c in enumerate(
            vocab))

        self.onehot.fit(np.array(list(self.vocab.values()))
            .reshape(-1, 1))
        for key, value in self.vocab.items():
            self._onehotted_vocab[key] = self.onehot.
                transform(value).toarray().flatten()

    def fit_transform(self, y, with_onehot=True):
        self.fit(y)
        y = self._format(y)
        return self.transform(y, with_onehot)

    def inverse_transform(self, y):
        rev_vocab = {v: k for k, v in self.vocab.items()}
        return ''.join([rev_vocab[i] for i in y])

    def get_formaters_str(self):
        return get_formaters_str(self._formaters)
```

6.2. Training

```
# -*- coding: utf-8 -*-
from utils import *


def train(artist, epochs, patience_limit, lstm_layers,
    lstm_units, embedding, size_x, model_name):
    from sequentials import get_classifier
    from keras.callbacks import TensorBoard
    from metrics import perplexity
    import math

    if not artist:
        print('You need to pick an artist first')
        exit(-1)

    BATCH_SIZE=32
    data = read_file('dataset/%s.txt' % artist)
    DATA_SLICE = len(data) // 10
    artifact_params = (artist, size_x) if not model_name
        else (artist, size_x, model_name)
    artifact = ModelArtifact(*artifact_params)
    tensor_logger = artifact.get_tensorflow_logdir()
    encoder = artifact.load_or_create_encoder(data)

    os.makedirs(artifact.get_tensorflow_logdir(), exist_ok=
        True)
    metadata_file_name = os.path.abspath(os.path.join(
        artifact.get_tensorflow_logdir(), "metadata" + ".tsv
        "))
```

```
save_metadata_of_embedding(metadata_file_name, encoder.
    vocab)

# embeddings_freq=True is a hack for embeddings to be
    shown
tensorboard = TensorBoard(tensor_logger,
    embeddings_metadata=metadata_file_name,
    embeddings_freq=True)
# we need the callback to init the visualizer
train_log_per_batch_names = ['train_batch_loss', '
    train_batch_accuracy', 'train_batch_perplexity']
train_log_per_epoch_names = ['train_epoch_loss', '
    train_epoch_accuracy', 'train_epoch_perplexity']
val_log_names = ['val_loss', 'val_accuracy', '
    val_perplexity']
test_log_names = ['test_loss', 'test_accuracy']

# Split data for testing and validating purposes
val_data = encoder.transform(data[0: DATA_SLICE],
    with_onehot=False)
test_data = encoder.transform(data[DATA_SLICE:2*
    DATA_SLICE], with_onehot=False)
data = encoder.transform(data[DATA_SLICE:], with_onehot
    =False)
classifier = get_classifier(BATCH_SIZE, size_x, len(
    encoder.vocab), lstm_layers, embedding, lstm_units)
tensorboard.set_model(classifier)

min_loss = math.inf
patience = 0
global_steps = 0
```

```
for epoch in range(epochs):
    print('\n[%d]Epoch %d/%d' % (epoch + 1, epoch + 1,
        epochs))
    losses, accs, perps, v_losses, v_accs, v_perps =
        [], [], [], [], [], []

    for i, (X, Y) in enumerate(read_batches(data, len(
        encoder.vocab), BATCH_SIZE, size_x)):
        loss, acc, perp = classifier.train_on_batch(X,
            Y)
        print('[%d]Batch %d: loss = %f, acc = %f, perp
            = %f' % (epoch + 1, i + 1, loss, acc, perp))
        write_log_to_board(tensorboard,
            train_log_per_batch_names, (loss, acc, perp)
            , global_steps)
        losses.append(loss)
        accs.append(acc)
        perps.append(perp)
        global_steps += 1

    for (val_X, val_y) in read_batches(val_data, len(
        encoder.vocab), BATCH_SIZE, size_x):
        val_loss, val_acc, v_perp = classifier.
            test_on_batch(val_X, val_y)
        v_losses.append(val_loss)
        v_accs.append(val_acc)
        v_perps.append(v_perp)
    val_loss_avg = np.average(v_losses)
    val_acc_avg = np.average(v_accs)
    val_perp_avg = np.average(v_perps)
    train_loss_avg = np.average(losses)
```

```
train_acc_avg = np.average(accs)
train_perp_avg = np.average(perps)
write_log_to_board(tensorboard, val_log_names, (
    val_loss_avg, val_acc_avg, val_perp_avg),
    global_steps)
write_log_to_board(tensorboard,
    train_log_per_epoch_names,
                    (train_loss_avg, train_acc_avg,
                        train_perp_avg), global_steps
                        )
if epoch % 20 == 0:
    save_embedding_to_board(tensorboard.
        embeddings_ckpt_path, epoch)
print('[%d]FINISHING EPOCH.. val_loss = %f, val_acc
    = %f, val_perplexity = %f' %
        (epoch + 1, val_loss_avg, val_acc_avg,
            val_perp_avg))

if val_loss_avg <= min_loss:
    min_loss = val_loss_avg
    patience = 0
    artifact.persist_model(classifier)
    print('[%d]New best model for validation set
        found.. val_loss = %f, val_acc = %f' %
            (epoch + 1, val_loss_avg, np.average(
                v_accs)))
elif patience >= patience_limit:
    print('[%d]Patience limit (%d) reached stopping
        iteration. Best validation loss found was:
        %f' %
            (epoch + 1, patience_limit, min_loss))
```

```
                break
            else:
                patience += 1

    classifier = artifact.load_model()
    classifier.compile(optimizer="rmsprop", loss="
        categorical_crossentropy", metrics=['accuracy',
        perplexity])

    t_losses, t_accs = [], []
    for i, (test_X, test_y) in enumerate(read_batches(
        test_data, len(encoder.vocab), BATCH_SIZE, size_x)):
        test_loss, test_acc, _ = classifier.test_on_batch(
            test_X, test_y)
        write_log_to_board(tensorboard, test_log_names, (
            test_loss, test_acc), global_steps+i)
        t_losses.append(test_loss)
        t_accs.append(test_acc)

    print('Best model\'s test_loss = %f, test_acc = %f' % (
        np.average(t_losses), np.average(t_accs)))


def cli():
    import argparse
    parser = argparse.ArgumentParser(description='Lyrics
        generating with Keras and recurrent networks')
    parser.add_argument('--artist', type=str, help='The
        dataset to be used during learning.')
    parser.add_argument('--epochs', type=int, help='For how
        many epochs the program should learn.', default
```

```
        =250)
    parser.add_argument('--patience_limit', type=int,
                        help='At which epoch after not
                            increasing accuracy the program
                            should terminate.', default=25)
    parser.add_argument('--lstm_layers', type=int, help='
        How many layers of lstm should the model be built
        with.',
                        default=3)
    parser.add_argument('--lstm_units', type=int, help='How
        many hidden units the lstm layers should have.',
        default=64)
    parser.add_argument('--embedding', type=int, help='How
        many dimensions should the embedding project to.',
                        default=32)
    parser.add_argument('--size_x', type=int, help='How
        long should the the input be.', default=100)
    parser.add_argument('--model_name', type=str,
                        help='Name of the model (if not
                            given it will use the timestamp
                            followed by the artist).',
                        default=None)

    args = vars(parser.parse_args())

    train(**args)


if __name__ == '__main__':
    cli()
```

Nyilatkozat

Alulírott ..... szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet ..... Tanszékén készítettem, ... diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2018. május 16. .....

aláírás

Alulírott ..... szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet ..... Tanszékén készítettem, ... diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a TVSZ 4. sz. mellékletében leírtak szerint kezelik.

Szeged, 2018. május 16. . . . . .

aláírás

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani **X. Y-nak** ezért és ezért ...

Irodalomjegyzék

[1] J. L. Gischer, The equational theory of pomsets. *Theoret. Comput. Sci.*, **61**(1988), 199–224.

[2] J.-E. Pin, *Varieties of Formal Languages*, Plenum Publishing Corp., New York, 1986.