MLP-ALAPÚ ELEMIESEMÉNY-ELŐREJELZÉS

MILEFF PÉTER

A kutatás arra a kérdésre keresi a választ, hogy miként lehetséges egy már létező, adott eseményhalmaz alapján olyan neurális hálózatot készíteni, amely a megfelelő szintű betanulás után alkalmas az úgynevezett eseménypredikcióra. A predikció során egy adott eseménysor következő, várható elemét szeretnénk megjósolni. A kutatás ezen részében kizárólag elemiesemény-predikció lehetséges megvalósításának vizsgálata a cél, amely során az MLP- (többrétegű perceptron hálózat) hálózatokkal foglalkozunk részleteiben, a prognosztizálást végző neurális hálózatot MLP-alapokon készítjük el.

1. A kutatás célja és lépései

A mai modern információs rendszerek bonyolult környezetben működnek. A felhasználói és ügyviteli folyamatok évről évre bonyolultabbak lesznek, melyek megfelelő informatikai támogatás nélkül nem bonyolíthatók le hatékonyan. Az irodalomban számos publikáció és tanulmány foglalkozik a felhasználói aktivitások naplózásával, amely során valamilyen formátumú naplófájl létrehozása a cél. A napló (szerencsésebb esetben) tartalmazza a folyamatok elvégzésekor megjelenő elemi eseményeket, azok minden legfontosabb adataival. Persze sok függ attól, hogy a rendszert eleve úgy tervezték, hogy megfelelő minőségben naplózzon, vagy pedig a naplófunkciót csak később „ültették" rá a folyamatokra.

A komplex folyamatok egyik hatékonyságnövelési lehetősége, ha bizonyos részeit vagy akár az egészet automatizálni tudjuk. A feldolgozás felgyorsul, a rendszer szűk keresztmetszetei felderíthetők és megszüntethetők. Az ERPA kutatási projekt egyik fontos alappilléreként szintén megjelenik ez az igény. A naplófájlok feldolgozásával és a modern mesterséges intelligencia eszközeivel vélhetően készíthető olyan szoftver, amely képes a rendelkezésre álló inputsor (az aktuális folyamat első megvalósult lépései) alapján megbecsülni a vélhető következő felhasználói tevékenységet. Az események predikciójára több módszert is kidolgoztak már az irodalomban, jelen munkában az egyik legnépszerűbb eljárást, a neurális hálózatokkal való modellezést szeretnénk részletesebben megvizsgálni.

A kutatási munka célja egy alapvető megvalósíthatósági elemzés elvégzése, amely arra irányul, hogy MLP alapú hálózat segítségével miként lehetséges az események predikciójának megvalósítása. A predikció klasszikus MLP, nem pedig LSTM (Long short-term memory) megoldással való elkészítése több szempontból indokolt:

- Az MLP-alapú hálózatok alkalmazása széles körben elterjedt különféle folyamatokban. Magától értetődik, hogy a kutatás egészéből, mint lehetséges alternatíva nem maradhat ki.

- A komplex eseményeken alapuló későbbi, fejlettebb/bonyolultabb megoldás elkészítésében fontos első lépcsőfok lehet az adatstruktúra és egyéb részek megértésében.
- Az elkészült MLP-modell jó összehasonlíthatósági alapot nyújt a jövőben elkészülő LSTM-hálózathoz.

A kutatás fontosabb lépései a következők:

- Az MLP-alapú neurális hálózatok elméleti hátterének feltárása és megértése.
- Python- és Keras-környezetek megismerése: MLP-hálózatok létrehozása, parametrizálása és a tanítás folyamata.
- Atomi események előrejelzését megvalósító kezdeti MLP-hálózat tervezése és modellezése Python-/Keras-környezetben. Egy egyszerűbb kezdeti prototípus.
- XES-adatstruktúra részletes elemzése, eseménysor átkonvertálása tanítóhalmazra.
- XES-formátum alapján működő prototípusmodell elkészítése Python- és Keras-környezetben.
- Tesztek készítése és elvégzése különböző paraméterbeállítások mellett.
- Eredmények előzetes értékelése, bemutatása.

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

A felhasználói aktivitások figyelésére általában valamilyen aktivitásfigyelő szoftvert alkalmazunk. Az aktivitásfigyelő szoftver rögzíti az alkalmazások és programok használatát a felügyelt munkaállomáson. A képernyőn megjelenő felhasználói tevékenységek egy előre kidolgozott és jól strukturált naplóba kerülnek. A naplók tehát információs adatbázisok, amelyek minden olyan tevékenységet tárolnak, amelyek aznap történtek. A mai modern technológiának köszönhetően számos lehetőség, megoldás áll rendelkezésre a monitorozásra, a tevékenységek figyelemmel kísérésére és kezelésére.

Az elvégzett kutatómunka több nagyobb részre, lépésre bontható, melyet a dokumentum részletesen bemutat. Röviden bemutatásra kerülnek az MLP-hálózatok, majd az MLP-modell alkotás folyamatát Python- és Keras-környezetekben először egy egyszerűbb előrejelzési példán mutatjuk be. Végül egy komplexebb eseménypredikció kerül bemutatásra, amely már a szabványos XES-formátumból dolgozik.

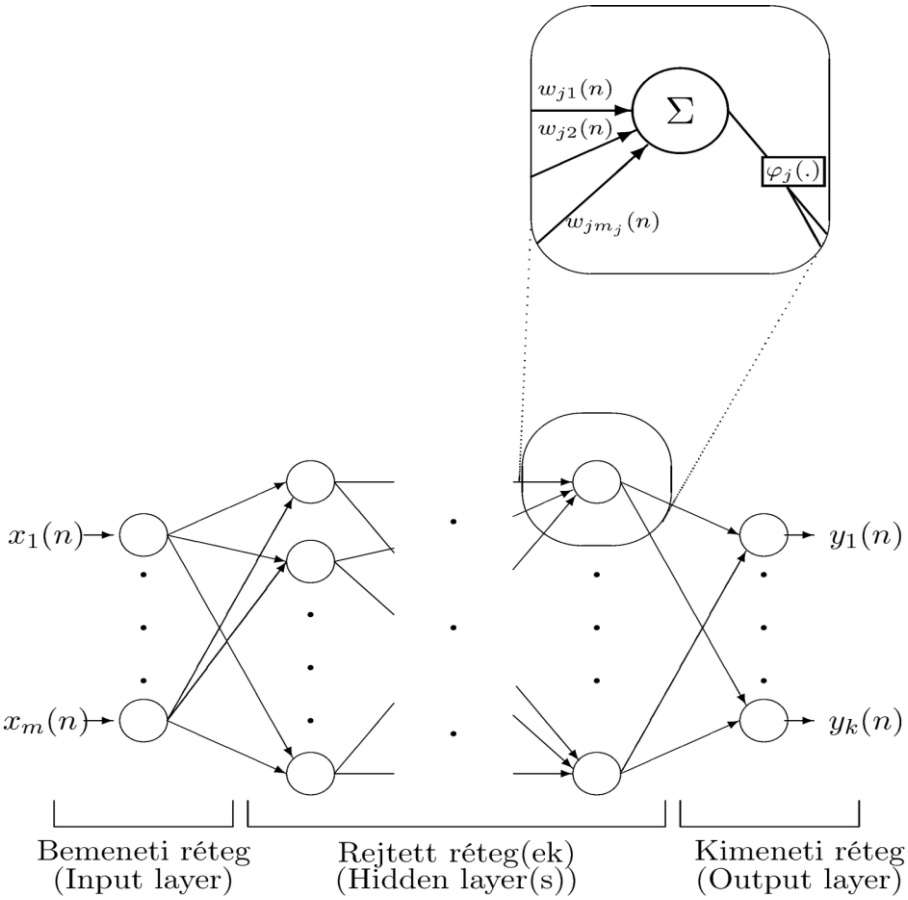
2.1.1. A többrétegű perceptron hálózat

Neurális hálózatokat széles körben alkalmaznak tudományos és műszaki feladatok megoldására. Többek között karakterfelismerésre, képfeldolgozásra, jelfeldolgozásra, adatbányászatra, bioinformatikai problémákra, méréstechnikai és szabályozástechnikai feladatokra használnak különböző neurális hálózatokat. Meg lehet velük oldani olyan összetett problémákat, amelyek visszavezethetők két alapvető feladatra: osztályozásra (azaz szeparálásra) és függvényközelítésre (azaz regressziószámításra). A mesterséges neurális hálózat az idegrendszer felépítése és működése analógiájára kialakított számítási mechanizmus. A neurális hálózatok közismert kezdeti típusa az egyetlen neuronból (idegsejtből) álló perceptron volt. A *Rosenblatt–Novikoff*-féle perceptron konvergencia tétel állítása alapján a perceptron képes elválasztani két lineárisan szeparálható halmazt. A következő lépés az *Adaline* megalkotása volt. Ez úgy tekinthető, mint egy lineáris függvény illesztésére alkalmas eszköz. Ennek tanítása a *Widrow–Hoff-algoritmus*, más néven a *Least mean square* eljárás. Kiderült azonban, hogy több neuront egy rétegbe rendezve sem oldható meg lineárisnál bonyolultabb feladat (Minsky és Papert, 1972). Bonyolultabb elrendezést pedig nem tudtak betanítani. Áttörést a többrétegű perceptron (Multi Layer Perceptron, MLP) tanítására szolgáló eljárás és a hiba visszaáramoltatása (hiba visszaterjesztése, error back-propagation) felfedezése hozta (Rumelhart, Hinton, Williams – 1986). Azóta a neurális hálózatok elmélete és alkalmazásai hatalmas fejlődésen mentek keresztül.

A többrétegű perceptron (multi-layer perceptron, MLP) a gyakorlati feladatok megoldásánál talán a leggyakrabban alkalmazott hálózatarchitektúra. Az MLP egy előrecsatolt neurális hálózat, amely rétegekbe szervezett neuronokból áll, ahol annak biztosítására, hogy a hálózat kimenete a súlyok folytonos, differenciálható függvénye legyen, a neuronok differenciálható kimeneti nemlinearitással rendelkeznek.

A hálózatban háromféle réteget (layer) különböztetünk meg: ***bemeneti, rejtett***, valamint ***kimeneti réteg***ből. A rétegek angol nevei: *input layer, hidden layer, output layer*. Rejtett rétegből tetszőleges számú lehet, viszont bemenetiből és kimenetiből csak egy-egy. A rejtett réteg hozzáadásának az az előnye, hogy kiterjeszti a háló által reprezentálható hipotézisek terét, ez fogja definiálni a hálózat mélységét. Gondoljunk minden egyes rejtett neuronra úgy, mint ami egy lágy küszöbfüggvényt reprezentál a bemeneti térben.

Az alábbi ábra a többrétegű perceptron általános hálózatát mutatja be.

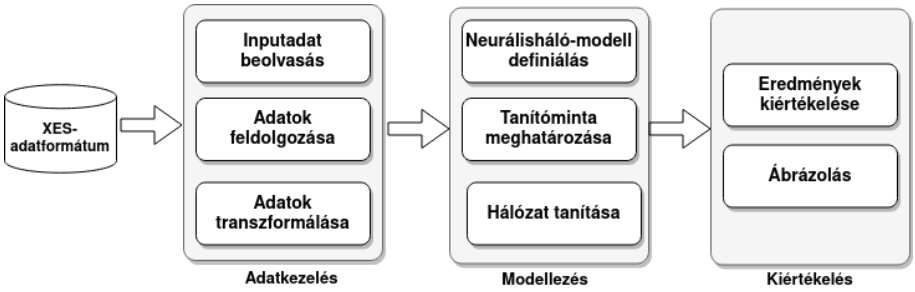


1. ábra. *Többrétegű perceptron hálózat általános modellje*

Bal oldalon van a bemeneti réteg, jobb oldalon a kimeneti, közöttük pedig egy vagy több rejtett réteg. A jel balról jobbra áramlik, azaz egy adott rétegbeli neuron bemenete (inputja) a tőle balra lévő rétegbeli neuronok kimenete (outputja). Az általunk tárgyalt modell esetén nincs kapcsolat rétegen belül és távolabbi rétegek között sem. Viszont minden neuron kapcsolatban van a vele közvetlenül szomszédos rétegek minden neuronjával. A többrétegű perceptron fontos tulajdonsága, hogy minden neuronjának saját aktivációs függvénye és saját súlyai vannak.

A rejtett neuronok számának előzetes meghatározása még napjainkban sem jól megoldott probléma. Számos kutatás foglalkozik az egzakt meghatározásával, vagy valamilyen megfelelő becslés adásával, de a gyakorlatban jelenleg empirikus alapokon történik a rejtett neuronok számának a meghatározása.

2.1.2. A többrétegű perceptron hálózat tanítása



2. ábra. *A technológiai folyamat lépései*

A többrétegű perceptron egy előrecsatolt hálózat (feedforward network). Azaz az inputjel rétegről rétegre halad előre, az outputréteg pedig megadja a kimenő jelet. A többrétegű perceptron tanításánál nehézséget jelent, hogy egy neuron kimenetét nem mindig tudjuk közvetlenül minősíteni, hiszen csak a kimeneti réteg esetében van elvárásunk a kimenetre, így csak ezeknél a neuronoknál tudjuk közvetlenül értelmezni a hibát. Erre jelent megoldást a hiba-visszaterjesztéses módszer, amely kihasználja, hogy egy adott neuron bemenete az előző réteg kimenete így a rétegeken visszafelé haladva tudja minősíteni a neuronok kimenetét és meghatározni a korrekciót.

A tanítás fő lépései:

- Megadjuk a kezdeti súlyokat.
- A bemeneti jelet (azaz a tanítópontot) végigáramoltatjuk a hálózaton, de a súlyokat nem változtatjuk meg.
- Az így kapott kimeneti jelet összevetjük a tényleges kimeneti jellel.
- A hibát visszaáramoltatjuk a hálózaton, a súlyokat pedig megváltoztatjuk a hiba csökkentése érdekében.

A többrétegű perceptron tanítása a *hiba visszaáramoltatása* módszerrel (hiba visszaterjesztése, error/back-propagation algorithm) történik.

2.1.3. A többrétegű perceptron hálózat a gyakorlatban

A gyakorlatban a többrétegű perceptron hálózatok széleskörűen alkalmazhatók számos különböző problémák megoldásában. Egyik leggyakoribb alkalmazási és a projekt szempontjából releváns területe az úgyneverett idősorok / atomi eseménysorok várható jövőbeli értékeinek előrejelzése. Az eseménysorok várható következő elemének predikciója kihívást jelent, különösen akkor, ha hosszú sorozatokkal, zajos adatokkal, többlépcsős előrejelzésekkel és több bemeneti és kimeneti változóval kell dolgozni. A mesterséges neurális hálózatok, leginkább a mély tanulási módszerek ígéretes lehetőséget kínálnak az idősorok előrejelzésére, mint például az időbeli függőség automatikus megtanulása és az időbeli struktúrák, például a trendek és a szezonalitás automatikus kezelése.

2.1.3.1. Adatstruktúra-előkészítés

Ahhoz, hogy „bármilyen" jellegű problémát modellezni lehessen egy neurális hálózattal, ahhoz az adatok megfelelő előkészítése, a megfelelő struktúra kialakítása elengedhetetlen. A gyakorlati gépi tanulás többsége felügyelt tanulást használ. Jelen fejezetben az idősorok adatait alakítjuk át úgy, hogy az alkalmas legyen a felügyelt tanulási formátumnak.

A felügyelt tanulás az, ahol van bemeneti változó (X) és kimeneti változó (y), és egy algoritmus segítségével megtanulja a leképezési funkciót a bemenetről a kimenetre. A cél az, hogy olyan jól közelítsük meg a valódi mögöttes leképezést, hogy új bemeneti adatok birtokában megjósolhassuk az adatok kimeneti változóit.

Ha adott egy adathalmaz-sorozat, akkor az adatokat átalakíthatjuk úgy, hogy azok felügyelt tanulási problémának minősüljenek. Ezt úgy tehetjük meg, hogy a korábbi lépéseket használunk bemeneti változóként, és a következő lépéseket használjuk kimeneti változóként.

Példaadatsor: 1, 2, 3, 4, 5, ...

A sorozat bemeneti és kimeneti komponensekkel rendelkező olyan mintákká alakítható, amely a betanítási folyamat részeként felhasználható, például egy mély tanulási neurális hálózat oktatására.

X,	**y**
[1, 2, 3]	4
[2, 3, 4]	5

Ezt *csúszóablak-transzformáció*nak nevezik, mivel ez olyan, mint egy ablak elcsúsztatása olyan korábbi megfigyelések között, amelyeket a modell bemeneteként

használnak a sorozat következő értékének előrejelzésére. Ebben az esetben az ablak szélessége 3 időlépés.

Az MLP-modellek esetében a Keras ebben formátumban fogja várni az adatokat, ezért a modellalkotást mindig meg fogja előzni egy adat-előkészítés, transzformáció, amely a fenti vagy ahhoz hasonló formátumra alakítja az adatokat.

2.1.3.2. Egyváltozós MLP-modell

A mesterséges neurális hálózat építését egy egyszerű, úgynevezett egyváltozós eseménysoron alapuló modell építésével kezdjük. Az egyváltozós eseménysorok olyan adathalmazok, amelyek egyetlen megfigyelési sorozatból állnak időbeli sorrendben. Egy olyan modell kialakítására van szükség, amely képes lesz a korábbi megfigyelésekből fakadó sorozatok tanulására, majd pedig a sorozat következő értékének előrejelzéséhez. A következőkben ezt mutatjuk be részleteiben.

2.1.3.2.1. Adatok előkészítése

Az egyváltozós sorozat modellezése előtt az adatokat szintén a megfelelő módon elő kell készíteni, ahogyan már a korábbiakban említésre került. Az MLP-modell olyan tanulást fog végezni, amely során a múltbéli eseményeket, mint egy bemeneti értékek sorozatát leképezi kimeneti eseménnyé. Ahhoz, hogy ez megvalósulhasson, a megfigyelések sorozatát több tanulómintára kell bontani, amelyekből a modell tanulhat.

Tekintsük az alábbi egyváltozós sorozatot mintaként:

[10, 20, 30, 40, 50, 60, 70, 80, 90]

Ahhoz, hogy tanuló mintát hozzunk létre, a szekvenciát több bemeneti/kimeneti adatra kell szétosztani, amelyeket *minták*nak nevezünk. Három „időlépést”, azaz három sorozati értéket használunk bemenetként, és egy időlépést használunk kimenetként a tanuló egylépéses előrejelzéshez.

X,	y
10, 20, 30	40
20, 30, 40	50
30, 40, 50	60
...	

A sorozatból jól látszik, hogy két logikai egységet képezünk: egy adott inputmintának milyen kimenete lesz. Azaz jelen esetben mely sorozati értékek után minek

kell következnie. Az adatok egységessége kulcsfontosságú, különben a program nem lesz képes feldolgozni azokat.

Az adatok ilyen alakra való hozását valamilyen automatikus megoldással célszerű elvégezni. Az alábbi mintapélda és a benne szereplő *split_input_sequence*() függvény megvalósítja ezt a viselkedést, és egy adott egyváltozós sorozatot több mintára oszt fel, ahol minden minta meghatározott számú időlépéssel rendelkezik, és a kimenet egyetlen időlépés. A megvalósító Python-kód:

```
from numpy import array
# split a univariate sequence into samples
def split_input_sequence(sequence, n_steps):
	X, y = list(), list()
	for i in range(len(sequence)):
		# find the end of this pattern
		end_ix = i + n_steps
		# check if we are beyond the sequence
		if end_ix > len(sequence)-1:
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
		X.append(seq_x)
		y.append(seq_y)
	return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_input_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
	print(X[i], y[i])
```

A mintapélda futtatása során az egyváltozós sorozat hat mintára oszlik, ahol minden minta három bemeneti és egy kimeneti időlépéssel rendelkezik.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

2.1.3.2.2. MLP-modell-idősorok előrejelzéséhez

Az idősorok becsléséhez szükséges Keras-környezetben megvalósított kezdeti MLP-modell a következő. Jelen mintakód nem tartalmazza a *split_input_sequence*() függvényt, de a gyakorlatban annak szerves része kell legyen.

```
# univariate mlp example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_input_sequence(raw_seq, n_steps)
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

A modell elemeinek részletes magyarázata

A Keras-könyvtár központi eleme a modell, amelyet a könyvtárban *(keras.models)* a ***Sequential*** osztály reprezentál. Ez a modell egy alap a Keras-csomagban, működését tekintve a modell rétegek lineáris halmazaként valósul meg.

Egy ***Sequential*** modellt többféleképpen hozhatunk létre, többféleképpen adhatjuk meg a rétegeket. Legegyszerűbb megadás a konstruktorban való definiálás:

```
from keras.models import Sequential
model = Sequential(...)
```

Hasznosabb megoldás azonban az, ha a modell létrehozásakor a szükséges rétegeket a végrehajtani kívánt számítás sorrendjében adjuk meg, például:

```
from keras.models import Sequential
model = Sequential()
model.add(...)
model.add(...)
model.add(...)
```

Modellbemenetek

A modell első rétegében meg kell adni a bemenet alakját. Ezt egy úgynevezett *Dense* típusú réteggel valósítjuk meg.

A *Dense* réteg az alábbi műveletet valósítja meg: output = aktivációs_fgv (dot (input, kernel) + bias), ahol az aktivációs_fgv az elemek szerinti aktiválási függvény, a kernel a réteg által létrehozott súlymátrix, és a bias pedig a réteg által létrehozott egy eltolásvektor.

Egy alap Dense réteg létrehozásához a batch és a bemeneti attribútumok számának megadása szükséges.

Például egy ***Dense*** típusú réteghez, ha 8 bemeneti minta-darabszámot szeretnénk rendelni, akkor az alábbiak szerint adhatjuk meg:

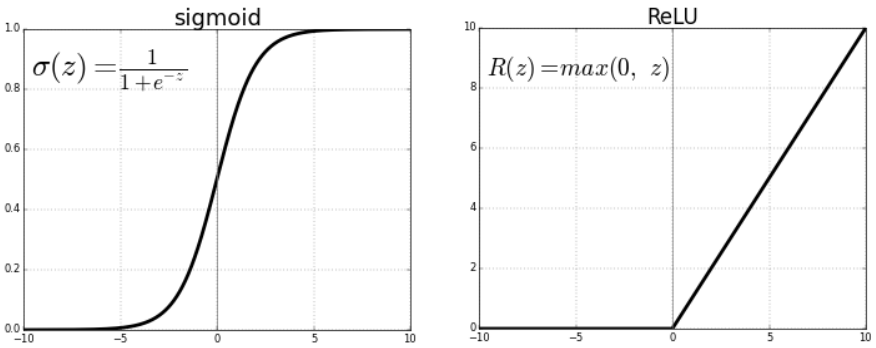
```
Dense(16, input_dim=8)
```

Aktivációs függvény

Az aktivációs függvény a neurális hálók működtetéséhez szükséges elem. A háló úgy működik, hogy a hálóban lévő egyes neuronok kapcsolatban állnak. Az egyes neuronok a bemenetük és a kimenetük között egy transzformációt végeznek. Minden egyes neuronkapcsolathoz tartozik egy numerikus súly. A neuron először veszi a bemeneteinek egy súlyozott összegét, majd ezek után ezt az értéket keresztülvezeti az aktivációs függvényen. A Keras számos szabványos neuronaktivációs függvényt támogat, például: *softmax, relu, rectifier, tanh* és *sigmoid*.

A ***Sigmoid*** és ***ReLU*** a világ leggyakrabban használt aktivációs függvényei. Szinte minden konvolúciós mély tanulási hálózatban használják. A bemutatott MLP-modellben a ReLU-függvény került alkalmazásra.

A modell definíciójában fontos a bemenet alakja, ezt várja a modell bemenetként az egyes mintákhoz az időlépések számát tekintve. Ez az a pont, ahol a létrehozandó modellt és a bemeneti mintasort egymáshoz igazítjuk. Ez a gyakorlatban azt jelenti, hogy *split_sequence()* függvény argumentumaként megadott felbontási számot (n_steps) a modell *input_dim* argumentumának is át kell adni, így a modell által várt adatstruktúra és a *split_sequence()* függvény által készített formátum egyezni fog.



3. ábra. *Sigmoid és ReLU aktivációs függvények.*

A modell definiálása után lehetőségünk van a hálót a tanító-adatsorra betanítani.

```
# fit model
model.fit(X, y, epochs=2000, verbose=0)
```

Miután a hálózat betanult, már használhatjuk előrejelzésre. A bemenet megadásával megjósolhatjuk a sorozat következő értékét. Például ha a bemenet értéke:

[70, 80, 90]

A hálózat által elvárt és jósolt eredmény: [100]

A Kerasban a predikció elvárja, hogy a bemeneti adat kétdimenziós legyen [minták, jellemzők], ezért az előrejelzés előtt át kell alakítani a bemeneti mintát. Erre a *reshape()* Python-függvény ad lehetőséget.

```
demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
```

Megjegyzés: A kód futtatása során az eredmények eltérőek lehetnek, tekintettel az algoritmus vagy az értékelési eljárás sztochasztikus jellegére. Célszerű a példát többször futtatni és összehasonlítani az eredményeket, azok átlagát nézni.

2.1.4. MLP-alapú atomiesemény-predikció

A neurális hálózatok numerikus értékekkel dolgoznak. Ahhoz, hogy egy nem numerikus alapú problémát neurális hálózattal modellezni lehessen, ahhoz numerikus alakra kell leképezni. Az esemény-előrejelzési feladat is természetesen ebbe a csoportba tartozik, ugyanis a gyakorlatban az egymástól elkülönülő eseményeket általában valamilyen atomi azonosítóval modellezik. Míg a gyakorlatban, egy-egy valós működő információs rendszerben a számalapú eseményreprezentáció is elméletben megfelelő lenne, azonban a numerikus reprezentációtól az emberi olvashatóság miatt gyakran eltérnek.

Feltételezhetjük tehát, hogy az elemi eseménysor valamilyen szöveges azonosítóval van ellátva. Az egyszerűség kedvéért vizsgáljuk az alábbi eseménysorpéldát, ahol az egyes elemi események egy betűvel reprezentáltak:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

Ahhoz, hogy a neurális hálózat inputadathalmazaként a fenti adatsor megadható legyen, le kell képezni valamilyen numerikus értékké. Jelen egyszerű példában magától adódik az az érték hozzárendelési módszer, ahol az abécé betűin sorban haladva kerülnek az értékek társításra a betűkhöz. Az alábbi Python-mintakód ezt valósítja meg:

```
raw_seq_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
letters_dictionary = {chr(i+96):i for i in range(1,27)}
```

A kódrészlet egy úgynevezett „dictionary”-t készít el, amely minden betűhöz egy számot rendel hozzá a következőképpen:

Az elkészült adatsorból pedig már kialakítható az a megszokott input-adatstruktúra, amit a neurális hálózati modell elvár:

```
raw_seq = []
for char in raw_seq_char:
  raw_seq.append(letters_dictionary[char])
```

Ezzel megkaptuk a számsoralapú inputvektort, amelyet a fenti példába illesztve a hálózat betanítása elvégezhető.

2.1.4.1. MLP-alapú eseménypredikció XES-formátum esetén

A kutatási munka végső célja, hogy olyan neurális hálózati modell kerüljön kialakításra, amely képes XES állományokban szereplő adatok tanulására, majd az események predikciójára. A fejlesztéshez egy szabványos benchmark *pdc_2016_1.xes* adatállomány állt rendelkezésre, a kísérleteket ezen végeztük el.

Jellemzői:

- szimbólumkészlet mérete: 18
- a leghosszabb szekvencia hossza: 30
- adatminták darabszáma: 1000

A kutatási jelentés korábbi részeiben részletesen bemutatásra került az XES-formátum, ezért ebben a dokumentumban nem térünk erre ki. Ahhoz, hogy az XES-adatok az MLP számára kezelhetők legyenek, az adathalmazon beolvasás után számos transzformációt kell elvégezni. A beolvasást és transzformációt végző kódrész a következő:

```
def load_graphs(self, xmlfile="pdc_2016_1.xes"):
    t_lists = []
    f_dict = dict()

    tree = ElementTree.parse(xmlfile)
    root = tree.getroot()

    for item in root.findall('.//trace'):
        t_lists.append([])
        for eve in item.findall('event'):
            for vv in eve.findall('string[@key="concept:name"]'):
                t_lists[-1].append(vv.attrib['value'])

    n = len(t_lists)
    print("N=", n)

    for i in range(n):
        kk = ''.join(t_lists[i])
        if kk in f_dict.keys():
            f_dict[kk] += 1
        else:
            f_dict[kk] = 1

    self.c_dict = dict()
    self.data = []
    i = 0
```

```
    self.max_sequence_length = 0
    for kk in f_dict.keys():
        kl = []
        if len(kk) > self.max_sequence_length:
            self.max_sequence_length = len(kk)
        for c in kk:
            if c not in self.c_dict.keys():
                self.c_dict[c] = len(self.c_dict) + 1
            kl.append(self.c_dict[c])
        i += 1
        self.data.append(kl)

    n2 = len(self.data)
    self.c = len(self.c_dict)
    print("C", self.c, "L", self.max_sequence_length, "N", n2)

    for i in range(n2):
        if len(self.data[i]) < self.max_sequence_length:
            self.data[i] = [0 for _ in range(self.max_sequence_length - len(self.data[i]))] + self.data[i]
```

Az algoritmus célja, hogy a fájlbetöltés után egységes formátumra hozza az egymás után következő összetartozó eseménysorozatot. Mivel az XES-en belül nem minden tevékenység eseményeinek darabszáma azonos, de a Keras-alapú MLP-modell viszont azonos méretű "szeletekben" várja az adatokat, így nem marad más lehetőség, mint az egységes formátumra hozás. Az algoritmus meghatározza a leghosszabb mintát, és ahhoz igazítja a többi tevékenység eseménysorát úgy, hogy a kevesebb darabszámú eseménysorokat balról nullákkal egészíti ki a leghosszabb méretűnek megfelelően. Példa:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 13, 13, 13, 13, 15, 14, 16, 1, 6, 3, 2, 4, 5, 7, 8, 10, 9, 11, 12]
```

A következő fontos elem az MLP-modell leírása. A következő kód ezt mutatja be:

```
def create_mlp_model(self, h):

    model = tf.keras.Sequential(
      [
        tf.keras.layers.Dense(h, activation='relu', input_dim=self.m),
  tf.keras.layers.Dense(h),
        tf.keras.layers.Dense(len(self.c_dict) + 1, activation='softmax'),
      ]
    )
     model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
return model
```

Konkrét, 100 darab ***h*** értékkel az alábbi hálózat jön létre a Kerasban:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 100)               1100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 19)                1919
=================================================================
```

Végül, de nem utolsósorban két fontos függvényt mutatunk be:

```
def create_dataset(self, dataset, look_back, data_x, data_y):
        c = len(self.c_dict)
        for i in range(len(dataset) - look_back - 1):
            if dataset[i + look_back] > 0:
                a = dataset[i:(i + look_back)]
                for j in range(len(a)):
                    a[j] = a[j] / c
                b = [0 for _ in range(c + 1)]
                b[dataset[i + look_back]] = 1
                data_x.append(a)
                data_y.append(b)
        return np.array(data_x), np.array(data_y)

    def gen_train_data(self, m):
        self.m = m
        train_x, train_y = [], []
        for i in range(len(self.data)):
            self.create_dataset(self.data[i], self.m, train_x, train_y)

        return train_x, train_y
```

A *gen_train_data* függvény szerepe, hogy egy megfelelő tanítási adathalmazt hozzon létre. Ehhez a betöltött XES-adatokat használja fel. A paraméterben megkapott érték alapján megadott méretű inputadat-szeletekből álló tömböt hoz létre a *create_dataset* függvény segítségével a korábbiakban bemutatott inputadatmintához hasonlóan.

A fenti függvények egy osztályban kaptak helyet. A működtetést végző kód az alábbi:

```
from MLPHelper import MLPHelper
import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    mlp = MLPHelper()
    mlp.load_graphs()
    (train_x, train_y) = mlp.gen_train_data(10)
    model = mlp.create_mlp_model(100)
    model.summary()

    train_x = np.array(train_x)
    train_y = np.array(train_y)

    history = model.fit(train_x, train_y, epochs=200, verbose=1)

    train_mae = history.history['accuracy']
    plt.plot(train_mae, label='train accuracy')
    plt.show()
```

Az adatok tanítását a *model.fit* függvény végzi, majd a tanítás közben mért hatékonyságot egy diagramban ábrázolja.

2.2. Kiértékelések eredményeinek bemutatása

A kutatás ezen fázisában a kiértékelés mérőszámaként a tanulás helyességét tudjuk értelmezni. A Keras a tanítási folyamat során beépített lehetőséget kínál arra, hogy a tanítási folyamat során keletkező ***„loss"*** és ***„accuracy"*** értékek mérhetők legyenek, melyek későbbi összehasonlításokhoz használhatók fel.

A fenti kódsorokban szereplő *„**categorical_crossentropy**"* definiálja a modellben a hibafüggvényt. A *categorical_crossentropy* mérőszámot a többosztályú osztályozás esetén szokás használni.

A tanítási folyamat hosszú és időigényes. A fejlesztő környezet ezt és a legfontosabb értékeket szövegesen mutatja. Példa:

```
Epoch 1/200
341/341 [==============================] - 0s 572us/step - loss: 2.5461 - accuracy: 0.1867
Epoch 2/200
341/341 [==============================] - 0s 791us/step - loss: 1.8668 - accuracy: 0.2925
```

Epoch 3/200
341/341 [==========] - 0s 790us/step - loss: 1.6797 - accuracy: 0.3251
Epoch 4/200
341/341 [==========] - 0s 787us/step - loss: 1.6208 - accuracy: 0.3484
Epoch 5/200
341/341 [==========] - 0s 808us/step - loss: 1.5994 - accuracy: 0.3552
Epoch 6/200
341/341 [==========] - 0s 765us/step - loss: 1.5832 - accuracy: 0.3594
Epoch 7/200
341/341 [==========] - 0s 569us/step - loss: 1.5487 - accuracy: 0.3696
Epoch 8/200
341/341 [==========] - 0s 548us/step - loss: 1.5287 - accuracy: 0.3761
Epoch 9/200
341/341 [==========] - 0s 566us/step - loss: 1.5229 - accuracy: 0.3904

A kísérletben több epoch értékkel végeztünk tanítást. Az epoch szám egy olyan hiperparaméter, amely azt a darabszámot definiálja, hogy hányszor haladjon végig a tanítóalgoritmus a teljes tanítómintán.

A neurális hálózatok nem determinisztikus rendszerek, amelyek ugyanazon inputra akár eltérő outputtal szolgálhatnak. Ebből kifolyólag a kutatási munka során egy-egy eredmény kiértékelését mindig több különböző futtatás alapján határoztuk meg.

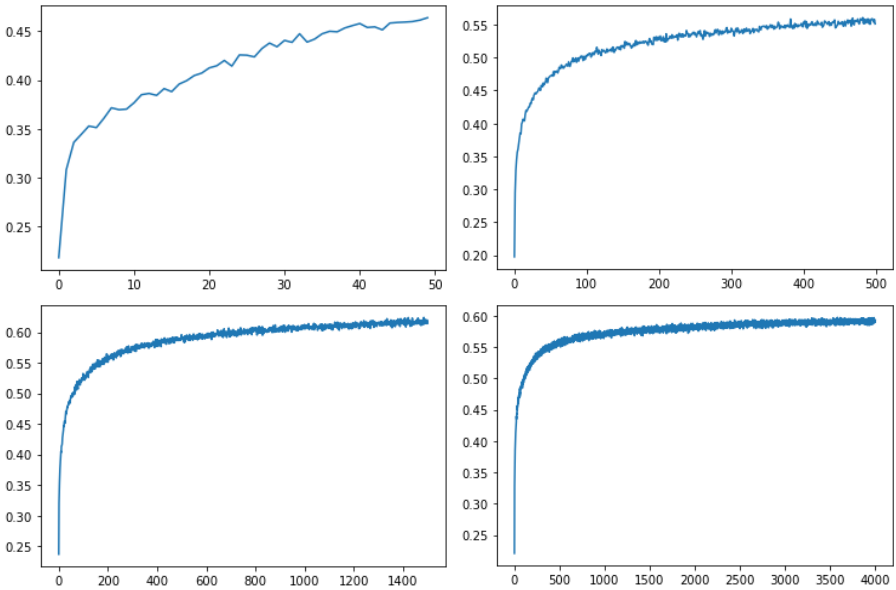
Az alábbi táblázat az epoch értékekhez tartozó loss és accuracy értékeket foglalja össze.

1. táblázat. *Futási eredmények összehasonlítása*

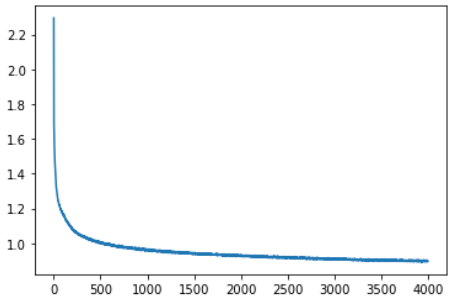
Epoch	Loss	Accuracy	Time
50	1,2645	0,4345	13 sec
200	1,1102	0,51	35 sec
500	1,0078	0,5586	1:55 min
1000	0,9360	0,5874	3:31 min
2000	0,9132	0,6018	7:23 min
4000	0,8642	0,6109	15:33 min

2.3. Eredményeket szemléltető diagramok

Az alábbi diagramok a különböző epoch számmal készített tanulási folyamatok eredményét, az „accuracy” értékének változását szemléltetik.



Veszteség függvény értékének változása 4000 epoch szám esetén:



3. Összegzés

A mesterséges intelligencia területei az utóbbi években nagymértékű fejlődésen mentek keresztül. Nem csupán újabb modellek születtek, hanem a rendelkezésre

álló szoftvereszközök egy sokkal szélesebb spektruma érhető ma már el mindenki számára. A bemutatott példákból jól látszik, hogy a Python- és Keras-környezetek alkalmasak a komolyabb feladatok elvégzésére is. A bemutatott MLP-hálózatok egy jól működő alternatívát kínálnak az események predikciójának megvalósítására. A tesztekben használt mintaadathalmazon megfelelő eredményt lehet elérni a modell alkalmazásával. A jövőben, az LSTM-modell mellett célszerű egy, a problémát megoldó MLP-modell-változatot is fenntartani és kidolgozni, amely egy lehetséges működő alternatívát kínál majd a futtatási eredmények összehasonlítására.