Bevezetés a Deep Learningbe Python alapon

Michaletzky Tamás, Témavezető: Kurics Tamás

2020. december 8.

Kivonat

A félév során a Matematika Intézet szemináriumára volt lehetőségem járni. Az előadásokat Kurics Tamás, volt egyetemi tanársegéd tartotta, akitől az Appercell jelenlegi programozójaként elsőkézből kaphattunk információkat, hogyan is áll az ipar a sokat emlegett MI területén. A képzés erősen gyakorlat orientált volt, a motivációk felvázolását a matematikai háttér ismertetése követte, végül a módszerek implementálása Pythonban. Ezt követem én is a beszámolómban. Az elején az érdekességekre helyezem a hangsúlyt, a végén pedig egy példán keresztül próbálom érzékeltetni deep learningben rejlő lehetőségeket.

Mi a mesterséges intelligencia?

Mint minden organikusan fejlődő tudományágnak, úgy a mesterséges intelligenciának (továbbiakban MI) sincs egységes definíciója, hiába alkamazzák világszerte sikerrel legkülönbözőbb problémák megoldására. Egyik elfogadott definíció szerint olyan ember építette gép vagy algoritmus, mely képes az ember kognitív képességeit utánozni, úgy mint tanulás, döntés, probléma megoldás, automatizált viselkedés és válaszkészség. Az MI-t már az 50-es évektől kezdődően kutatták, ekkor alkotta meg John McCarthy az *artificial intelligence*, azaz a máig használt MI kifejezést. Mivel azonban az MI gépigénye számottevő, újabb jelentős áttörést csak a 80-as években érhettek el, a számítógépek fejlődésével és a *machine learning*, a gépi tanulás elterjedésével. Később, a kognitív tudomány hatására, a neruális hálózatos, a *deep learning* megközelítés vált általánossá.

A paradigma váltás

Gépi tanulás alatt azokat a számítógépes algoritmusokat értjük, melyek képesek "tapasztalat" útján tanulni. Másként szólva a számítástudomány azon ága, ahol az algoritmus anélkül képes a teljesítményét növelni, hogy erre külön, *explicit* programozva lett volna. Képzeljük el ugyanis, hogyan programoznánk be az önvezető autónkat hogy egy embert megkülönböztessen egy lámpa oszloptól, vagy hogy automatikus képaláírás generálónk felismerje a széket és az asztalt és IKEA katalógusunkat automatikusan kitöltse; vagy éppen kézírásunkat "begépelje". Tradícionális programozáskor ugyanis az adatokból az explicit lekódolt szabályok alapján várjuk az algoritmustól a megoldást. MI-programozáskor azonban azt mondhatjuk, hogy az adatok és válaszok alapján a gép maga *találja ki a szabályokat*, és hoz döntéseket az alapján. Ezt nevezzük a gépi tanulás *paradigmaváltásának*. Ez nem jelenti azt természetesen, hogy ne programozhatnánk bele szabályokat előre, sőt: bármely játék esetén, például egy egyszerű amőba játék is (mely egyébként hagyományosan is programozható mondjuk 3x3-as esetben), megadjuk a megengedett lépések halmazát, tehát definálják a játékteret, amin belül a gép maga tanulja meg az optimális döntéseket. Fontos még, hogy ezek miatt bevett zsargon, hogy a modellt *tréneljük*, tanítjuk, nem csak programozzuk. Ez a tanítás nagy vonalakban azt jelenti, hogy a meghatározandó mennyiséget iteratívan keressünk, leggyakrabban az ún. súlyokat, és minden iterációban (neurális hálók esetén *epoch*-ban) az előző eredményeken javítunk valamilyen módon.

Világos tehát, hogy meg kell különböztetni néhány esetet. Ha az adataink olyanok, hogy ismerjük a helyes választ, címkézett, *labelled* adathalmazról beszélünk. Ekkor a tanulásforma *supervised*, felügyelt. Az adathalmazt tanuló és teszt részekre bontjuk, a tanulón betanuljuk az összefüggéseket, a teszt halmazon értékeljük. Amint elértük a kívánt hatékonyságot, az algoritmus képes megfelelően pontos előrejelzésekre ismeretlen, addig nem látott adathalmazon is. Klasszikus esetei a regressziós és klasszifikációs problémák, ahol a (rendre) folytonos vagy diszkrét (igen-nem) cél mennyiséget a prediktorok alapján kell becsüljük, például házak értékesítésekor paraméterei alapján becsülni, árazni a házakat.

Unsupervised, felügyeletlen tanulás esetén a jó válaszok nem világosak vagy nem definiáltak előre. Nagy mennyiségű adathalmazból, pl. online aktivitásuk alapján kategorzáljuk a felhasználókat

valamilyen rendszerezés szerint (klaszterizációs probléma és anomália felismerés), vagy faktorizációs, dimenzó csökkentési problémák képtömörítéskor tárolóhely felszabadítás végett. Egy mókás alkalmzást már-már klasszikussá vált idézni: egy kutatás során kiderült, hogy a sör és a pelenka egyszerre fogy[1] (fogyásuk pozitívan korrelál), így a hasznunkat úgy növelhetjük, a gyanútlan vásárlót megtévesztve szinte észrevétlen, hogy az egyik árának csökkentésekor (akciókor) a másikat megfelelően növeljük. A jelenségre egyébként azóta sem találtak kielégítő magyarázatot, de hát úgy is tudjuk: a korreláció nem garantáltan létező oksági viszonyokat fed fel!

Ezen két fő területen kívül a harmadik legjelentősebb a *reinforcement learning*, ahol valamilyen célfüggvény maximalizálását várjuk el az algoritmustól. Léteznek ezenkívül természetesen még más területek, adatbányászattól kezdve online ajánlórendszerekig sok minden, mint a már fent említett *computer vision* vagy dokumentum hasonlóság keresés. A szeminárium során a sokról egy-egy példát is megnéztünk, itt azonban csak egyet részletezek majd ki.

Mély tanulási paradigma

Mint később látni fogjuk, még így sem fedtük le az említett lehetséges problémahalmaz egy jelentős részét. Ehhez szükségünk lesz a mély tanulási paradigmára, mely során az algoritmus nem csak imitálja a kognitív funkciókat, hanem implementálásában is hajaz rá. *Deep learning* megoldásforma esetén a gépi tanulást egy ún. neurális hálón hajtjuk végre. A neuronok rétegekbe rendeződnek és össze vannak kötve, típustól függően valamilyen módon. Minden neuron a bejövő inputok lineáris kombinációja alapján (az agyi neuronok szinoptikus tüzeléséhez hasonlóan) az aktivációs függvénye szerint aktiválódik, mely aktivitás lehet diszkrét (aktív, nem aktív) vagy egy folytonos output. Bármely neurális háló minimum két rétegből áll, egy input és egy output rétegből. A további rétegeket rejtett rétegeknek hívjuk. Bonyolultságuk, összekötésük módja szerint többféle neurális hálót különböztetünk meg, mindegyik más és más problémák megoldására alkalmas.

A következő példán bemutatom, hogy milyen további, tisztán gépi tanulással nem jól megoldható problémát küszöböltünk ki ezzel a megközelítéssel. Ehhez azonban szükségünk lesz az alábbi definíciókra.

Egy klasszikus probléma

Klasszikus lineáris regresszióban egy y skalár változóról azt feltételezzük, hogy

$$y = w^T x + b + \epsilon$$

alakban írható fel, tehát azt tételezzük fel, hogy az n-változós x prediktor (magyarázó) és az y magyarázott változó között lineráis összefüggés van, nulla várható értékű ingadozást (hibát) figyelembe nem véve. Adott x, y mellett ugyan az $y = w^T x$ lineáris egyenletnek ismerjük a megoldását (Moore–Penrose-féle pszeudoinverz), mi azonban elsősorban nem a teszt halmazon kívánunk jó eredményt elérni (jelen esetben a hibánk várható értéke nulla lenne), hanem szeretnénk jól előrejelezni, azaz a modellnek ismeretlen halmazon is jól kell szerepelnie. Szakszóval el kell kerüljük a *túltanulást*, amikor a modell rátanul a teszthalmaz véletlen hibáira is. Klasszikus példa, hogy n pontra mindig illeszthető $(n-1)$-ed fokú görbe, mely így hibátlan a teszt példán, de egyrészt nem feltétlen indokolt az $(n-1)$-ed fokú összefüggés, másrészt a modell nem *tanult* várhatóan semmit.

A klasszikus megoldás a sztochasztikus gradiens süllyedés (SGD) módszere. Adott w súlyok mellett a modell átlagos négyzetes hibáját szeretnénk minimalizálni:

$$J(w) = \frac{1}{2} MSE = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \to \min!,$$

ahol $\hat{y}_i = h_w(z_i)$ az i-edik, z_i teszt példára a predikciónk. Ne feledjük, hogy a teszthalmazon ismert a helyes, y_i érték. Az iteráció egy lépésében a $J(w)$ n változós konvex skalárfüggvényt csökkentjük minden koordináta irányban a derivált α-szorosával, ahol α az ún. *learning rate*, vagy bátorsági tényező, melynek alkalmas megválasztásával tovább optimalizálható az eljárás:

$$w_j^{(n)} = w_j^{(n-1)} - \alpha \frac{\partial J(w)}{\partial w_j} = w_j^{(n-1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)(z_i)_j.$$

Az ún. L_2-regularizáció során egy λ paraméter bevezetésével elkerülhetőek a magas együtthatók és így a magasabb fokú tagok, ami mint korábban tárgyaltuk, csökkenti a túltanulás lehetőségét.

[1] https://www.slideshare.net/mrm0/beer-diapers-and-correlation-a-tale-of-ambiguity

Legyen a módosított költségfüggvény $J_\lambda(w) = \frac{1}{2m}\sum_{i=1}^m(\hat{y}_i - y_i)^2 + \lambda\sum_{j=1}^n w_j^2$. Világos, hogy λ elegendően, de nem túl nagy választása esetén ez a magas eggyütthatókat jobban "bünteti".

Hasonlóképpen fontos lehet a bemenet standardizálása, megint csak a túltanulás elkerülése végett.

Másik probléma a futásidő. Jól látszik, hogy minden iteráció során az egész teszthalmazt használjuk. Minibatch-eket használva ezt a problémát kerülhetjük ki. Minden iteráció során válasszunk egyenletesen véletlen a teszthalmazból elemeket, és csak azokon végezzük el a (megszorított) költségfüggvény minimalizálását. Mindez azonban a költségfüggvény oszcillálását okozhatja. Erre ad megoldást a momentum módszeres batch gradiens módszer, mely során a költségfüggvény oszcillálását simítjuk.

Léteznek természetesen más, az órán is említett optimalizálási eljárások, úgy mint az RMSProp vagy ADAM.

Harmadik fontos megjegyzés a linearitásra vonatkozóan. Adott magyarázó változók esetén nem tilos, sőt, mint ahogy látni fogjuk, érdemes azokból további változókat nyerni és arra építeni a modellt. Ha nem lineáris összefüggést látunk, hanem mondjuk négyzeteset, vegyük bele a prediktor négyzetét, és ezen futtassunk regressziót. A modellépítés során az ilyen és ehhez hasonló változásokat nevezzük *feature engineering*-nek. Szokás ugyanis ezen a területen a prediktorokat *feature*-nek is hívni.

A továbblépéshez szükségünk van a bináris klasszifikáció fogalmára is.

Logisztikus regresszió esetén diszkrét célváltozónk van, gyakran igen-nem osztályok. Ez a bináris klasszifikáció esete. A lineáris regresszióhoz hasonlóan most adott w súlyok és x paraméterérték mellett $h_w(x)$ becsülje annak a a valószínűségét, hogy x-re igen a válasz, tehát a $\mathbb{P}(y = 1|w, x)$ feltételes valószínűséget. Ehhez szükségünk lesz egy aktivációs függvényre, mely leggyakrabban a szigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$. Azt mondjuk, hogy

$$y = 1 \Leftrightarrow \sigma(h_w(x)) \geq \frac{1}{2} \Leftrightarrow \log\frac{h_w(x)}{1 - h_w(x)} = w^T x + b \geq 0,$$

tehát a klasztereket egy hipersík választja el. A költségfüggvényt maximum likelihood becslésből származtatjuk: az ún. *binary cross-entropy* függvényt, a $J(w, b) = \sum_{i=1}^m \big(- y_i \log h_w(z_i) - (1 - y_i)\log(1 - h_w(z))\big)$ függvényt szeretnénk minimalizálni. A modell pontosságát pedig valamilyen metrika szerint szokás mérni. Ennek legegyszerűbb változata a (metrikának csak mosolyogva nevezhető) *accuracy*, ahol megmondjuk, hogy a modell az egyes osztályokba tartozó elemeket milyen százalékkal találta el.

A mélytanulás a fentiek általánosítása. Ha a bemeneti x vektor koordinátáit neuronoknak képzelem el, akkor a logisztikus regresszió nem más, mint egy egyszerű, rejtett réteg nélküli, *fully connected, forward* neurális háló. Egyetlen output neuronunk ugyanis "begyűjti" a w súlyokból adódó aktivációkat és a szigmoid függvény szerint tüzel vagy sem. Rejtett rétegek beiktatásával nyerünk valódi neurális hálózatot, melynek magja a következő képlet: bármely z neuronra a bejövő inputok aktivációja általi értéke

$$h(z) = \sum_{x\in N(z)} w(x)x + b \to \sigma(h(z)),$$

alapján továbbtüzel vagy inaktív marad. A végén pedig az output neuronokon megtörténik a kiértékelés.

Fontos, hogy míg egyazon rétegen célszerű ugyanazt az aktivációs függvényt használni, rétegenként azonban eltérthetnek. Jellemző viszont az outputon a szigmoid használata. További aktivációs függvények például a tanh vagy a $relu(x) = \max x, 0$. Gyakori közös tulajdunságuk, hogy a számegyenest valahogy egy kompakt halmazra képezik. Érdemes észrevenni, hogy aktiváció nékül az összefüggés az adott neuronra lineáris, tehát aktiváció nélküli neurális háló egy túlbonyolított lineáris regresszió csupán.

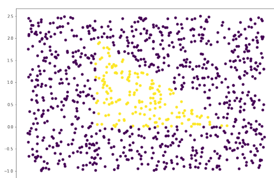
Háromszögezés

Tekintsük most a következő, felügyelt logisztikus regressziós példát: adott a síkon egy háromszög, detektáljuk a pontjait. Precízebben, egyenletesen szétszórva pontokat az egységnézeten tanítsuk meg az algoritmust, hogy a háromszögön belüli pontokra IGEN-t, kívülire NEM-et mondjon.

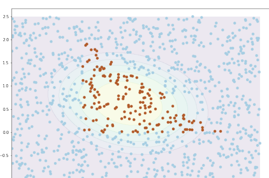
Először is világos, hogy triviális módon nem választható el hipersíkkal a két felület, hiszen a háromszög maga három félsík metszete. Magasabb rendű tagokat bevéve azonban már jobb eredmény érhető el. Természetesen egyáltalán nem kielégítő. Mit csináljunk akkor? Az ötlet a következő: mi lenne, ha neurális hálót használnánk, például egy egyszerűt, 3 neuronos rejtett réteggel, a három

hipersík (egyenes) detektálására. Természetesen a háló maga nem tudja, hogy a 3 neuron feladata ez. Ez is a *feature engineering* szépsége.

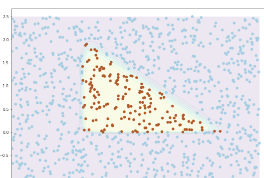
A gépi tanulás, és így a mély tanulás alapvető programozási nyelve jelenleg a Python. A terület felfutásakor ez volt ugyanis az a nyelv, mely könynen volt tanulható, tipizálható és amelyben, külső könyvtárak használatával hatékonyan lehetett és lehet máig is dolgozni. Egyszerűen szólva a lényeg nem a kódoláson, hanem a modell építésén van. Kedvelt platformja ennek a Keras, mely a Tensorflow egy ráépülő, interface-egyszerűsítő modulja. Mi is itt dolgoztunk. Segítségével, mint látni fogjuk, játszi könnyedséggel oldhatunk meg bonyolult feladatokat. Jelen esetben pár sorral háromszöget detektálni.



(a) Az adathalmaz.



(b) Logisztikus regresszióval.



(c) Neurális hálóval.

Első esetben négyzetes logisztikus regresszióval a 830 külső és 170 belső pontból 60-at rossz helyre soroltunk (0,94 pontosság), 24 FP eset és 36 FN (tehát amikor külsőt belsőnek és fordítva detektáltunk). Másodjára, neurális hálóval, mint az ábra is mutatja jobbak voltunk: 0,99 pontossággal találtuk el, mindössze 6 FP és 4 FN eset volt. Ránézésre talán a két eredmény eltérése nem olyan feltűnő (valóban nem az), ám az ábrákra tekintve világos, míg a második random mintavételezésre képes lesz várhatóan tartani az eredményét, úgy az első érezhetően romlani fog, kiváltképp a háromszög sarkainál. Négyzetes összefüggéssel ugyanis gyakorlatilag a háromszögbe írható legnagyobb ellipszist közelíti legjobb esetben is.

Elsőre talán nem világos, miért érdekelnek minket háromszögek, valójában megtettünk egy fontos első lépést egy újfajta megoldási módszer felé.

Zárásként tegyünk egy bátor kísérletet megjósolni a jövőt.

Az MI (várható) jövője

A Gartner nemzetközi elemző cég érdekes megállapítást tett az MI jövőjét illetően[2]. Ahogy minden újonnan megjelenő módszerrel történik, eleinte mindenki a megváltást várja tőle a saját poblémájára: így milliárdos cégek és vállakozók fektetnek tőkét a kutatásba, az önvezető autók és az automatizált társadalom, az általuk vágyálomként megjelölt cél felé vezető út következő lépéseként; ám a kiábrándulás mindig megtörténik. A kezdeti *hype*-ot követően, miután a befektetett energia és pénz nem térül meg, az ún. nyerészkedők új lehetőségek felé kacsintgatnak, így miután a tudományágat csúcsra járatták, azon lassú egyenletességgel már csak az elszántabb (és kevésbé pénzéhes) kutatók folytatják a munkát. Napjaink beláthatatlan adatmennyiségét, a *big data*-t is talán pont ezért válthatja le (vissza) rövid időn belül a *small data*, olcsóbb tárolással hasonló eredményeket elérve.

Hasonlóképpen, realista hozzáállással nem holnap, és nem is 10 év múlva fogunk a Körúton lámpák és jogosítvány nélkül utazni. Ez az utópia még messzebb van.

[2]https://medium.com/machine-learning-in-practice/deep-learnings-permanent-peak-on-gartner-s-hype-cycle-96157a1736e

Forráskód

```
import numpy as np
import sklearn.metrics as mcs
import tensorflow.keras as keras

#Read data to variables X and y

#Logistic regression
model = keras.Sequential([
    keras.layers.Dense(units=1, activation='sigmoid', input_shape=(5,))
])
optimizer = keras.optimizers.SGD(learning_rate=0.01)

#Neural network
model = keras.Sequential([
    keras.layers.Input(shape=(2,)),
    keras.layers.Dense(units=3, activation='sigmoid'),
    keras.layers.Dense(units=1, activation='sigmoid')
])
optimizer = keras.optimizers.Adam(learning_rate=0.01)

#Same from here
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics='accuracy')
history = model.fit(X, y, verbose=0, epochs=200)

predictions = np.round(model.predict(X))
print(mcs.accuracy_score(predictions, y))
mcs.confusion_matrix(y, predictions)
```