EFOP - Technikai összefoglaló

Tossenberger Tamás

January 2019

1 A projekt célja

A projekt célja egy új fajta autóenkóder alapú generatív modell létrehozása volt amelynek generálás során a generálandó adat természetes paramétere vagy paraméterei bizonyos mértékig kiszabhatóak, a variational autoencoderekkel szemben. Továbbá ideális esetben az előbbinél generatív modellként is jobban teljesít. Valamint kiköszöböljük azt a VAE-ben előforduló kellemetlenséget hogy az egyes batch-ek pontjainak eloszlásával közelítjuk a kívánt (normál) eloszlást. Ehelyett inkább a tanítás során látens pontfelhő eloszlását közelítjük.

A projekt során az MNIST adathalmazon dolgoztam, így ezen az adatlhalmazon a feladat számjegyek generálása volt, oly módon, hogy az egyes számjegyek külön is genrálhassuk. Vagyis képesek legyünk olyan számjegyképeket generálni amelyek túlnyomó többsége ugyanazt a számjegyet ábrázolja.

2 Az IGMN algoritmus

A projekt alapját az Incremental Gaussian Mixture Network algoritmus alkotja. Ennek részletes matematikai leírása megtalálható (1) cikkben. A projekt egyik részét képezte ennek az algoritmusnak a lekódolása és paramétereinek beállítása a konkrét feladatra. A kódolás során a legtöbb paramétert ezen cikk elnevezései alapján neveztem el.

3 Konvolúciós autóenkóder

Első lépésként létrehoztam egy közönséges konvolúciós autóenkódert (file: *cae_mnist.py*), melyet az MNIST adathalmazon tanítottam, és a látens dimenziók számát $D = 6$-ra állítottam be. A háló betanítását követően a modelleket és a háló súlyozását elmentettem későbbi használatra (*autoencoder.json*, *encoder.json*, *autoencoder.h5*, valamint az enkódert felhasználva az adathalmazra legeneráltam az adathalmaz látens pontfelhőjét, és ezt lementettem (*latent_points.npy*).

4 IGMN használata a látens pontfelhőn

Ahhoz hogy a látens pontfelhőből és a háló dekóder részének segítségével tudjunk számjegyeketképeket generálni, a generált látens pontfelhőn futtattam az IGMN klaszterező algoritmust. Így tehát a látens pontfelhő eloszlását közelítettem néhány gauss eloszlás összegével (*IGMN.py*).

Feltételeztem hogy a látens pontfelhőben kellően elkülönülnek a különböző számjegyekhez tartozó pontok ahhoz hogy az egyes gauss komponensek különböző számjegyekhez tartozzanak. Természetesen mivel többek közt ez a 6 dimenziós tér már az adatpontok elég absztrakt ábrázolása volt, ez nem teljesült tökéletesen.

5 Paraméterválasztás

Az IGMN konkrét alkalmazásához meg kellett választani a *c*, *beta*, *v_min* és *sp_min* paramétereket, és ki kellett számolni *sigmasq* értékét. *sigmasq* értékének meghatározására (tehát a látens ponthalmaz átlagos négyzetes távolságának kiszámítására) megírtam a *LengthDev.py* (on-line, így nagyon gyors) programot.

A *beta* paraméter megadásához azt vettem figyelembe hogy nem sokkal több mint 10 komponenst szeretnénk létrehozni ideális esetben, így *beta* értékét érdemes $\frac{1}{10}$-nél lényegesen kisebbre választani, viszont nem is túl kicsire hogy különböző számjegyek lehetőleg ne kerüljenek azonos komponensbe. A *v_min* és *sp_min* paraméterek megválasztásánál azt kell figyelembe venni, hogy arányuk kicsit több legyen a jósolt komponenssszámnál, mivel így ha *v_min* kellően nagy, akkor nagy valószínűséggel csak a sporiadikus pontok által alkotott komponensek lesznek eliminálva.

A *c* értékét érdemes 1-nél kisebbre választani, viszont azt figyelembe kell vennünk, hogy ha túl kicsire választjuk, akkor az első lépéseknél a torzítás mértéke olyan nagy lehet, hogy az egy nem szemidefinit mátrixot eredményezne a kovariancia mátrixnak. De azt tapasztaltam hogy a kovariancia mátrix viszonylag gyorsan konvergál, így nem gond ha *c* nem sokkal kisebb 1-nél, és így érdemes az új komponensek kezdeti kovariancai mátrixát viszonylag nagyra választani.

6 Esernyők

Az IGMN algoritmust először egy általam generált pontfelhőn teszteltem. Ezt az *IGMN_test.py* programmal tettem meg. Generáltam adott paraméterű gauss eloszlások összegéből pontokat, és erre alkalmaztam az IGMN algoritmust. Azt figyeltem meg hogy létrejöhetnek "esernyők", azaz előfordulhat hogy létezik olyan komponens ami közel azonos várható értékű egy másikkal, viszont a szórása nagyobb annál. Ez akkor jöhet létre ha egy sporiadikus pontot nem eliminálunk, hanem az végül a hozzá közeli komponens felé torzul.

Ennek megoldására hoztam létre az *umbrellas.py* programot, amely eliminálja az ilyen esernyőket. Az MNIST adathalmazon viszont azt tapasztaltam hogy erre a konkrét paraméterválasztással nem volt szükség, de valószínű hogy olyan

paramétervválasztás mellett amely több komponens eredményez, ezen az adathalmazon is tapasztalunk esernyőket. Ezek eliminálása fontos a továbbhaladáshoz.

7 Rátanítás a komponensekre

Eddig úgy állunk hogy egy már betanított konvolúciós autóenkóder látens pontfelhőjének az eloszlását közelítettük gauss eloszlások összegével. Azt szeretnénk hogy a látens ponthalmaz komponensei jobban elkülönüljenek, így célunk hogy a gauss eloszlások várható értékeihez bevonzzuk a látens pontokat, és újraszámoljuk a komponenseket.

8 Veszteségfüggvény

Ez előbbi megvalósításához egy saját veszteségfüggvénnyel tanítottam a korábbi hálót. Ennek a veszteségfüggvénynek az egyik tagja a korábbi *binary_crossentropy*, a másik tagja pedig a batch pontjainak a komponensekre vett a posteriori valószínűségösszeg (pontosabban ennek $-\mu$-szöröse, ahol μ paraméter értéke megválasztható).

A gondolat emögött hogy ha vennénk a sűrűségfüggvények összegének ellentettjét, akkor ezen véve az egyes pontokat minden pont (nagyjából) azon eloszlás várható értékéhez vonzódna be amelyre legnagyobb a likelihood-ja. Így tehát ezekbe a komponensekbe széthúzzuk a látens pontfelhőt, elkerülve bármilyen éles vágást amely feltételezné hogy az egyes pontok melyik komponensbe tartoznak.

Mivel a veszteségfüggvény második tagja jóval nagyobb értékeket adott az elsőnél, μ-t kicsinek kellett választani.

9 Mu beállítása

Azt tapasztaltam hogy ha μ nem kellően kicsi, akkor mivel a második tag dominál, ezért a látens pontfelhő összehúzódik egy pontba. Mű csökkentésével elérhető hogy *sigmasq* nagyobb legyen. Az lenne ideális ha megközelíteni az eredeti értékét, mivel azt szeretnénk hogy az egyes komponensek összehúzódnának, de amúgy megmaradnának. Sajnos az általam beállított paraméterek esetén nem ez volt a helyzet, az eredeti *sigmasq* 6.65 volt, a legjobb elért pedig a $\mu = 10^{-6}$-ra végzett hálózati tanítást követően 4.54. Tehát néhány komponens a tanítás ezen fázisában összehúzódik. Ez nem feltétlenül gond ha ezek azonos számjegyekhez tartozó komponensek, de mindenesetre ez is mutatja hogy érdemes lenne közben epoch-onként frissíteni a közelítő eloszlást (új komponenseket már nem hozva létre). Ez a jelenlegi algoritmussal kb. 2-3 óra tanítási időt igényelne. Ezen várhatóan rengeteget lehetne javítani az *IGMN_keras.py* program batch-re történő átírásával, kihasználva itt is a GPU-s párhuzamosításból eredő gyorsítást.

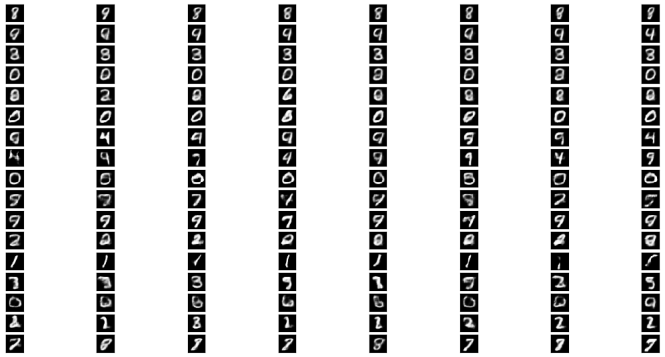

Figure 1: $\mu = 10^{-6}$

10 Eredmények

A *picgen_mnist.py* programmommal nyomon követtem hogy a generált számjegyek milyen minőségűek. A fenti képen a $\mu = 10^{-6}$ választással futtatott háló eredménye látható.

Látható hogy a generálás minősége és a számjegyek szeparálása is hagy kívánnivalót maga után. Egyes számjegyek mint a 0-ás, 3-mas, 1-es, 7-es, $8 - as$ és 9-es viszonylag szépen lett generálva és jól elkülönült a többitől, azonban nem minden számjeggyel volt ilyen szerencsénk.

11 Környezet és futási idő

Az IGMN algoritmus az MNIST 60000 (6-dimenziós) pontból álló látens pontfelhőjén laptop CPU-n körülbelül 1 perc alatt futott le. (Ez várhatóan a fenti ötlettel sokszorosára gyorsítható.) A tanítás pedig epoch-onként körülbelül 10 másodpercet vett igénybe a Google Collab felületen a GPU gyorsítás mellett.

12 Hogyan tovább?

Fent már említettem hogy érdemes lenne az *IGMN_keras.py* programot batchekre átírni és a *network.py* hálóval párhuzamosan futtatni.

Azt tapasztaltam hogy a paramétereket úgy választva hogy több komponens szülessen, a generálás sokkal hatákonyabb, viszont ez annak rovására megy hogy az egyes számjegyekhez sok komponens fog tartozni. Érdemes lenne sok komponenssel is tesztelni a hálót, és így kiértékelni a generálás minőségét. Mivel a komponensek számával csak lineárisan nő az IGMN futásideje, ezért ez nem okoz jelentős futásidőbeli gondot.

Érdemes lenne továbbá megnézni hogy semi-supervised módon ha elérjük hogy az absztrakt látens térben ne follyanak össze ennyire a különböző számjegyekhez tartozó adatpontok, akkor mennyire jó eredményt tudunk elérni.

Ezen apró dolgokon kívül még a későbbiekben szeretném tesztelni egy új ötletemet, miszerint az IGMN algoritmus által kapott a poszteriori eloszlással generálnék random pontokat a látens térben, és viszgálnám hogy mekkora buborék fúlyható fel az adott pontból hogy az bizonyosnál kisebb sűrűségben tartalmazzon az adatok látens pontjaiból. A kapott legnagyobb buborékok elhagyásával azt remélném hogy gauss eloszlásokkal nem olyan jól közelítható pontfelhők jobban közelíthatőek, és így például lecsökkenthető a számjegyre nem hasonlító eredmények száma. Ez a módszer elsősorban kisebb látens dimenziókra lenne tesztelve, kérdéses hogy mennyire valósítható meg nagyobb látens dimenziókban, illetve más tervezett módosításával ez megvalósítható-e.

References

[1] Rafael Coimbra Pinto, Paulo Martins Engel. *Scalable and Incremental learning of Gaussian Mixture Models.* "1701.03940.pdf"