Neurális hálózatok – Gyakorlat

Klasszikus hálók

A Neurális hálózatok c. tantárgyhoz tartozó elméleti előadásainak első részében megismerkedhetünk a napjainkban is használatos mély hálók alapját adó klasszikus neurális hálózatokkal. Jelen dokumentum célja az első gyakorlat jellegű demó feladatokat bemutató előadáson elhangzottak bővebb összegzése. Az előadás véges időtartama miatt számos dologra nincs idő. Ezen kimaradt részek (feladatok) itt megtalálhatóak rövid leírással kiegészítve, melyek az otthoni gyakorlás elősegítését hivatottak szolgálni.

Milyen feladatokat/példákat nézünk meg?

Alapvetően három elkülöníthető része lesz a demónak, melyek a következők:

- Elemi neuron összeállítása és tanítása. MLP összeállítása és tanítása.
- A túltanulás jelensége. Hogyan észlelhetjük, mit tehetünk ellene.
- Radiális bázisfüggvényes hálók (RBF) összeállítása és tanítása.

Az egyes feladatokat Tensorflowban mutatjuk be. A kiadott kódok az elegáns kód bemutatása helyett az egyes lépések jobb érthetőségének érdekében a (talán túlzott) tagolást helyezik előtérbe.

Az egyes fájlok felépítése.

Az elkészített feladatokat tartalmazó demókat érdemes az előre kialakított sorrendben végig zongorázni. Az első feladatokhoz tartozó kódok még csak minimális funkciókat hajtanak végre, több, a teljesnek mondható tanításokhoz szükséges lépések hiányoznak belőle. Ezekkel fájlról-fájlra fokozatosan bővülnek a kódok. Ennek praktikus – követhetőségi szempont szerinti okai vannak.

A kiadott notebookokon belüli fejlécek színezése az alábbiak szerinti:

- **ZÖLD FEJLÉC**: az egyes demó fájlokat érdemes többször végig futtatni különböző paraméterezések használatával. Ilyenkor a futtatást célszerű a kódban szereplő zöld fejléctől kezdődően újra futtatni. Ennél a résznél található ugyanis az a kódrészlet, ami visszaállítja a már korábban tanításon átesett hálót (azaz reset-eli a megtanult paramétereket), a jobb összehasonlítások érdekében.
- **PIROS FEJLÉC**: az egyes kódok (pár kivételtől eltekintve) csak néhány új részletet tartalmaznak a korábbiakhoz képest. Ezeket a fontosabb különbségeket (az adott demók lényegét) a piros fejlécű kódok tartalmazzák.

Az egyes demó kódok jellemző felépítése.

A kiadott kódok jellemzően az alábbi struktúrába szerveződnek.

A tanuláshoz szükséges mintahalmaz

1. A tanító mintahalmaz előállítása.

   Minden egyes példa esetén a kód legelején történik az osztályozandó mintahalmaz kialakítása. Ennek köszönhetően lehetőség van az egyes paraméterek (például várhatóérték, szórás) módosításával különböző eseteket is vizsgálni.

A háló felépítése

2. A bemenet(ek) és a (szabad) paraméterek létrehozása.

   Legyen szó tetszőleges neurális hálózatról, mindegyiknek rendelkeznie kell bemenettel, amelyeken keresztül a hálóba „tölthető" az osztályozandó minta, amely utána végighalad a hálón. A TensorFlowban a bemenetet ún. **placeholderek**kel lehet megadni. Ez egyfajta specifikáció arra vonatkozóan, hogy a tanítás (illetve a használat során) a megadott helyen biztosítani fogunk adatot. Egyik előnye, hogy a bemeneti tömb azon dimenziója, ami azt határozza meg, hogy hány tanítómintát használunk fel egyszerre (gondoljunk a fullbatch-re vagy a minibatch-re) később, a tanítás közben definiálódik.

   A paraméterek azok a **w** és **b** súlyok, amelyek az egyes neuronok közötti átmeneteknél súlyozó tényezőként szerepelnek. A háló tanítása során azokat a **w** és **b** értékeket keressük, amely optimális eredményt ad, azaz amelyek esetén a neurális háló kimenete a lehető legjobb eredményt adja a rendelkezésálló címkékhez viszonyítva.

3. A modell struktúrájának összeállítása.

   Ahhoz, hogy legyen egy hálónk, az egészet össze kell „drótozni". Egybe kell építeni az egyes rétegeket (pl. MLP esetén), valamint meg kell határozni, hogy egy adott rejtett rétegben hány darab neuron található, és ezen neuronok kimenetén milyen nemlinearitást akarunk alkalmazni.

   (Természetesen az előző pontban a **w** és **b** paraméterek megadásakor már tudnunk, ismernünk kellett a kialakítandó háló struktúráját, hiszen ez határozza meg, hogy hol és hány súlyunk van.)

A háló tanítástípusának definiálása

4. A minimalizálandó célfüggvény (veszteségfüggvény) meghatározása.

   Ahhoz, hogy a hálónkat tanítani tudjuk, szükségünk van egy kifejezésre, ami meghatározza, hogy az aktuális **w** és **b** súlyvektorok esetén milyen minőségű (jóságú) a hálónk válasza a tanítómintákon. Ehhez definiálnunk kell egy olyan célfüggvényt (veszteségfüggvényt / „loss function"-t), ami az elvárt kimenet (a tanítómintahalmazban szereplő címkék) és a háló kimenete közötti eltérést méri. Ebben a részben azt kell megadnunk, hogy miként definiáljuk az eltérést. Ezt a célfüggvényt felhasználva fogjuk a hálót tanítani, aszerint, hogy az egyes **w**

súlymódosítások végrehajtásakor a veszteségfüggvény értéke csökkenjen (tapasztalati kockázat minimalizáció elve).

5. Az optimalizáló eljárás kiválasztása.

   Az előző pontban tárgyalt célfüggvény (veszteségfüggvény) csökkentésére több módszer is létezik. Ebben a pontban a Tensorflow által támogatott lehetséges optimalizáló eljárások közül kell egyet kiválasztanunk.

A háló tanításának végrehajtása

6. A tanítási ciklus elkészítése.

   Most már megvannak a hálónk bemenetei és a tanítandó szabad paraméterei. A struktúrájának összeállításával felépítettük a neurális hálózatunkat, kiválasztottuk az háló válaszait minősítő célfüggvényt, valamint, hogy milyen módszerrel akarjuk optimalizálni ezt a hibát.

   Szükségünk van még egy olyan ciklus megírására, ami végrehajt egy-egy tanítási lépést, amely során a háló **w** súlyainak módosítása megtörténik.

   Azt, hogy pontosan hány ilyen lépést szeretnénk végrehajtani, többféleképpen is meghatározhatjuk. Vagy előre megadunk egy adott értéket, vagy pedig a tanítás során figyeljük a veszteségfüggvényt, és adott küszöb elérése esetén leállunk.

A kapott háló eredményének szemléltetése

7. A megtanult osztályozás ábrázolása.

   Most már, hogy van egy betanított hálónk, kíváncsiak vagyunk arra, hogy az mit tanult meg. Ebben a részben felrajzoljuk a bemeneti tér felosztását (azaz, hogy a bemeneti teret milyen tartományokra osztja az osztályozás során). Ezenkívül, ha voltak olyan mintapontjaink, amelyeket nem használtunk fel a tanítás során (ez az ún. teszthalmaz), akkor megnézhetjük, hogy ezeken hogyan teljesít a háló.

I. rész: Elemi neuron (Adaline) és MLP összeállítása és tanítás

01_Single neuron – Linear classification.ipynb

Ebben a példában egy egyszerű elemei neuront építünk fel. Ahogy az előadáson elhangzott, egy ilyen processzáló elem lineárisan szeparálásra képes. Ábrázolási szempontok miatt jelen példáknál a hálók bemenetére érkező jellemzők mérete (azaz az **x** bemeneti vektor dimenziója) 2. Ez azt jelenti, hogy minden egyes tanítómintapont két koordinátával fog rendelkezni, amelyek az x1 és az x2. (Itt most az eltolást (bias) nem vesszük hozzá a **w** súlyokhoz, így az **x** vektort sem egészítjük ki egy konstans 1-es értékű koordinátával.)

Futtassuk a kód első részét (1), amely legenerálja a mintahalmazt. Ekkor egy lineárisan szeparálható osztályozási feladatot kapunk. Ez a kétdimenziós térben (vagy ha úgy tetszik síkban) azt jelenti, a síkon fel tudunk venni úgy egy egyenest, hogy az egyik osztály összes mintapontja az egyenes egyik oldalára essen, míg a másik osztály összes mintapontja az egyenes ellenkező oldalára kerüljön.

(2) Ebben a részben elsőként reseteljük a számítási gráfot. Ha egy háló tanítása után újabb paraméterezéssel akarunk próbálkozni, akkor érdemes ettől a ponttól futtatni a blokkokat, mert így nem a korábban megtanult súlyokat használjuk fel, hanem ismételten egy véletlenszerűen inicializált hálónk lesz.

A következő blokkban definiáljuk az ún. placeholdereket. Ezek reprezentálják a háló bemenetét, azaz az **X** tanító mintahalmazt (design matrix). Ezzel lényegében egy ígéretet (promise) teszünk, hogy a tanítás során (valamint a kiértékelés során), amikor a hálónak szüksége van a bemenetén mintapontokra, akkor biztosítani fogjuk azokat. Később a *feed_dict{ … }* függvény hívással tudjuk „megetetni” a hálót, azaz rendelkezésére bocsátani az ígért bemenetet.

Ezt követően meghatározzuk a **W** súlymátrixot. Itt, mivel kétdimenziós a bemeneti tér, egy [2, 1] méretű mátrixról (tehát kétdimenziós oszlopvektorról) van szó. Létrehozzuk a **b** eltolás paramétert is. Ez pedig egy egyelemű mátrix lesz, hiszen csak egyetlen neuronunk van.

(3) Ebben a részben összeállítjuk magát az egyszerű neuront. Ehhez meg kell adni azt a leképezést (műveletsorozatot), amely a bemenettől kiindulva meghatározza a kimenetet, felhasználva a **W** és a **b** paramétereket. Itt kell kiválasztanunk, hogy milyen nemlinearitást használunk. Ebben a feladatban a logisztikus szigmoidra esik a választás.

A kódban található *linear* változóba kerül az előadásokon s-sel jelölt súlyozott összegzés, míg a *model_output* változóban magának a neuronnak a kimenete. (Ez a notebookban található ábra segítségével jól követhető.)

Ezenkívül meghatározzuk, hogy adott bemeneti minta esetén a predikciót (azaz, amikor egy mintaponthoz a háló egy osztályba sorolási címkét határoz meg) hogyan kell kiszámítani. Ez kerül a *prediction*-be.

(4) Ebben a részben meghatározzuk a veszteségfüggvényt, valamint kiválasztjuk a használandó optimalizálót. Elsőként állítsuk be a tanításkor alkalmazott bátorsági tényezőt (*learning_rate*).

Ezt követően definiáljuk a hálónk által elvégzett osztályozást minősítő veszteségfüggvényt, ami most a bináris osztályozási feladatoknál előszeretettel használt keresztentrópia lesz (*xentropy*).

Ezt követően megadjuk, hogy optimalizáló eljárásként az ún. Gradient Descent (GD) módszert használjuk. (Ez tehát a paramétertérben a háló aktuális **W** és **b** paraméterei szerinti pontban meghatározza, hogy melyik irányba „lejt" leginkább a hibafelület és abba az irányba (ez a gradiens mínusz egyszerese) elmozdítja a paramétereket (azt, hogy mekkorát lépünk a bátorsági tényező (learning rate) határozza meg).

Végül definiáljuk a *train_step*-et, amelyet meghívva/kiértékelve végre hajtódik egyetlen egy tanítási lépés. Itt még meg kell adnunk, hogy az általunk létrehozott veszteségfüggvényt (ami a keresztentrópia volt) minimalizálni akarjuk. (Ha például veszteségfüggvény helyett hasznosságfüggvényt használnánk, akkor ennél a lépésnél a maximalizálást kellene megadnunk.)

(5) Ez az a kódrészlet, ami a dokumentum elején nem szerepel az főbb lépéseket áttekintő pontok között, mégis mindegyik demókódnak részét képezi. A Tensorflow a háttérben az általunk definiált változókból és struktúrából egy számítási gráfot épít. Ebben a gráfban a csúcsok az egyes műveleteket, míg a csúcsok között futó irányított élek az adatokat (többdimenziós adattömbök, azaz ún. tenzorok) reprezentálják.

Ebben a részben először indítunk egy Session-t, ami létrehozza (felépíti) a számítási gráfunkat. Ezt követően inicializáljuk a globális változókat. Korábban a háló paramétereinél megadott **W** és **b** változók *tf.Variable*-ként lettek létrehozva. Ezeket a változókat az első használatuk előtt inicializálni kell. Mi a létrehozásukkor megadtuk, hogy az értéke standard normál eloszlásból véletlenszerűen generálódjon (*tf.random_normal(…)*). Ezt az inicializáló random értéket most kapja meg.

(6) Most következik magának a tanítási ciklusnak a megírása. Először meghatározzuk a tanítási iterációk számát megadó *num_epoch* paramétert. Ez után egy for ciklus következik, amely meghívja/kiértékeli a *train_step*-et, aminek hatására végrehajtódik egy tanítási lépés.

Az implementáció úgy értelmezhető, hogy amikor kiértékeljük a *train_step*-et, akkor a 4-c blokkban szereplő kód miatt a Gradient Descent optimalizáló végrehajt egy minimalizálási lépést az *xentropy* veszteségfüggvényünkön. Ekkor tehát ki kell értékelni az *xentropy*-t, ami pedig a 4-b blokkban lévő kód miatt kiszámolja a tanítómintapontokhoz tartozó címkék és a modellünk kimenete közötti keresztentrópiát. Itt tehát kiértékelődik a *model_output*, ami a 3-a blokkban lévő kód miatt kiértékeli a *linear*-t, amelyben már szerepel az *x_data* placeholderünk. Ezenkívül a 4-b blokkban lévő kódban is szerepelt már az *y_target* placeholder. Ahogy az már korábban szóba került, a placeholder az egyfajta ígéret, hogy a megfelelő időben itt rendelkezésre fog állni a kívánt adat: jelen esetben az adott tanítómintákhoz tartozó **X** bemeneti vektorok (ezek kerülnek az *x_data*-ba) és a hozzájuk tartozó helyes osztályozási címkék (ezek kerülnek az y_target-be). Így tehát visszajutottunk oda, ahonnan indultunk, a 6-b blokkba.

```
sess.run(train_step, feed_dict={x_data: inputs, y_target: targets})
```

Az előző kódsor a *sess* Sessiont (számítási gráfot) meghívva végrehajtja a *train_step* tanítási lépést, és mivel ennek végrehajtásához adattal kell „etetni” a modellt, így a *feed_dict{ … }* résszel ez a betöltés meg is történik. Itt most a teljes tanítóhalmaz (azaz az *inputs*) és a hozzátartozó összes címke (*targets*) kerül felhasználásra (tehát az előadáson tanult fullbatch-nek felel meg). Később nézünk olyan példát, ahol a teljes mintahalmaznak csak egy véletlenszerűen kiválasztott részhalmazát fogjuk (általában a teljes mintahalmaz túlságosan nagy mérete miatt) felhasználni. Ezt nevezzük majd minibatch-nek, vagy Stochastic Gradient Descent-nek (SGD).

(8) Ebben az utolsó részben a már megtanított háló ábrázolása történik meg. A kód mögött meghúzódó ötlet, hogy a kiértékelni kívánt tartományt osszuk fel kis négyzetrácsokra. A négyzetrácsok metszési pontjaiban értékeljük ki a hálónkat (azaz, mintha az adott pont egy bemenete lenne a hálónak) – azaz a bemeneti teret egy grid felett mintavételezzük. A háló válasza alapján pedig aszerint színezzük ki a négyzetrácsokat a megfelelő színnel, hogy melyik osztályba kerültek.

(9) Zárjuk be a létrehozott számítási gráfunkat. Ha ezt a sort végrehajtjuk és utána valamilyen korábbi kódot akarunk futtatni, akkor hibát kapunk. Ilyenkor újra fel kell építeni a számítási gráfot (5-a). Érdemes tehát ilyenkor is a **zöld fejlécű** blokktól futtatni a kódot.

Lehet próbálkozni…

- Állítsuk át az 1-a részben található *variance* paramétert nagyobb értékre. Ezzel megnöveljük az osztályokat alkotó mintahalmaz szórását, így elérhető a közöttük való átfedés (azaz lineárisan nem szeparálható osztályozási feladatot kapunk). Végezzük el így is a tanítást. Mit tanul meg a háló?
- Hajtsunk végre különböző *learning_rate* értékekkel tanítást. Mikor hány iteráció szükséges ahhoz, hogy jól megtanulja a háló az osztályázást?

01_MLP – NonLinear.ipynb

Ebben a feladatban az előző demóban szereplő egyszerű neuronokból építkezünk. Lényegében az előző példában vett elemi neuronnak tekinthető (kisebb módosításokkal) az előadásokon vett Rosenblatt féle Perceptron és a Widrow féle Adaline. Ezekkel lineárisan szeparábilis osztályozási problémákat oldhatunk meg.

A gyakorlatban azonban a problémák aligha ilyenek. Nemlineáris feladatok esetén az egyik lehetőségünk, hogy ezeket a neuronokat elkezdjük rétegekbe szervezni. Az így kialakított architektúrát nevezik Multilayer Perceptronnak, azaz MLP-nek.

Egyelőre az egy rejtett réteget tartalmazó MLP alkalmazásával fogjuk a demókódok elején kigenerált mintahalmazokat osztályozni. Ennek az architektúrának egyik előnye, hogy az általa megtanult szeparálási felületet (osztályozási tartományokat) könnyedén értelmezni tudjuk az egyszerű neuronokhoz tartozó szeparáló hipersíkok alapján.

Mivel az alkalmazott egy rejtett rétegű MLP a kimeneti rétegében az általunk beállított számú neuronok válaszának súlyozott összegét állítja elő (jelen példában a kimeneti rétegen nem definiálunk nemlinearitást), az előző részben látott hipersíkok kombinálásával előállítható osztályozási tartományokat kapunk.

A kód megértése

Első körben a (2)-es blokkban található kódokkal még ne foglalkozzunk.

A kódot megnyitva elsőként a (3) részt érdemes végignézni. A 3-a blokkban meghatározzuk a MLP struktúráját leíró hiperparamétereket. A *num_hidden_1* változóval adjuk meg a rejtett rétegben lévő neuronok számát.

A (4)-es blokkban létre kell hozni a rejtett réteg neuronjaihoz tartozó és a kimeneti lineárisan súlyozó réteg (erre is tekinthetünk neuronként) súly, valamint eltolás paramétereit.

Az (5)-ös blokkban definiáljuk azt a függvényt, amiben felépítjük a hálónkat. Ez a függvény adott bemenetekből (melyek a bemeneti mintapontok, a háló súlyai és eltolás paraméterei) határozza meg az háló válaszát.

Első futtatás

Futtassuk le a kódot az elejétől egészen a 9-b blokkig. A (2)-es részben lévő blokkokat is lefuttathatjuk, ez egyelőre még nem végez semmilyen változtatást a bemeneti adatunkon. Később még visszatérünk ide.

A 9-a blokk kimenetén látható, hogy a háló még nem tanulta meg jól az osztályozási feladatot. (Mivel a hálót véletlenszerűen inicializáltuk, elképzelhető az is, hogy már az első futtatás során rögtön tökéltes osztályozás születik. Ekkor futtassuk újra az egészet, és remélhetőleg az itt leírtaknak megfelelő eredményt kapunk. A továbbiakban mindig igaz lesz, hogy adott futtatás esetén az eredmény eltérhet az itt leírtaktól a sztochasztikus inicializáció miatt. A hiperparaméterek kezdeti beállítása azonban igyekszik ezt a lehető legkisebb valószínűségre szorítani. Ha mégsem az itt leírt eredményhez hasonlót kapunk valahol, csak futtassuk újra a kódunkat.)

A kezdeti 10 000 epoch nem bizonyult elégnek a feladat megtanulására. Futtassuk még le párszor a 8-b rész második blokkjától, rátanítva a hálónkra. (Ha újra futtatjuk a *train_loss_vec = []* részt, akkor a tanulási görbénk (8-c rész) mindig csak az utolsó tanítás alatti hibát fogja tartalmazni. Ha azt szeretnénk, hogy a görbe a teljes tanítás során számolt hibát ábrázolja, akkor ennek a sor kódnak a futtatását hagyjuk ki.)

További kétszer-háromszor futtatva a 10 000 epochos tanítási ciklust, a hálónk már képes lesz megtanulni egy a tanító mintapontokkal konzisztens osztályozást.

A 9-b blokk kimenetén láthatjuk az MLP rejtett rétegében található neuronok által megtanult szeparáló hipersíkokat. A háló végső válasza e hipersíkok súlyozott összegzésével előálló szeparáló felület lesz. (Ez a fajta értelmezés csak az egy rejtett réteget tartalmazó MLP-k esetén adódik.)

Második futtatás

Az eddigiek során kihagytunk egy nagyon fontos lépést. Ez pedig a bemeneti adatok normalizálása (standardizálása). Ahhoz, hogy ennek az előfeldolgozási lépésnek megvizsgálhassuk a tanítás sebességére gyakorolt hatását, futtassuk le előröl a kódot a következő módosításokkal.

Írjuk át az (1)-es blokkban lévő kódban az *inputs += [-1.0, -1.0]* sort a következőre:

```
inputs += [2.0, 1.0]
```

Ezzel a teljes adathalmazt eltoltuk az origóból. Ezt követően futtassuk a teljes kódot ismételten a 9-b részig. Egyszer futtatva a 10000 epochot, most sem kapunk még jó eredményt, így futtassuk le többször a 8-b rész második blokkjában lévő tanítási ciklust annyiszor, amennyiszer szükséges, hogy a tanulómintapontokat helyesen osztályozza a háló.

Várhatóan azt tapasztaljuk, hogy az előző beállításhoz képest lényegesen több epochra volt szüksége a hálónak a hibátlan osztályozás kialakításához. Ennek oka az, hogy ebben az esetben a bemenő adatunk rendelkezik egy ofszettel, azaz a bemeneti mintahalmaz nem az origó környékén helyezkedik el. Ez a 2-c blokk kimenetén kapott ábrán is jól látható.

A tanulás lelassulása a következőképpen magyarázható: A rejtett rétegben található neuronokhoz tartozó súly és eltolás paramétereket 0 várhatóértékű 1 szórású normáleloszlással inicializáltuk. Ennek következményeként a végső szeparálást kialakító, a neuronokhoz köthető hipersíkok (9-b blokk kimeneti ábrája) az origó környékére lettek inicializálva. Ekkor tehát a tanulással nagyobb mértékű eltolásuk szükséges, amit a háló nehezebben, vagy ha úgy tetszik, lassabban tanul meg. Úgy is nézhetjük, hogy a kezdeti inicializálásunk a paramétertérben messzebb esik az optimális eredményt biztosító megoldások helyétől, mely problémához még szuperponálódik az alkalmazott nemlinearitás telítődő hatása miatt jelentkező „eltűnő gradiensek” problémája.

Harmadik futtatás

Ebben a részben megírjuk az adatok előfeldolgozását biztosító standardizálást, amihez ugorjunk a kód 2-b blokkjához. Ez alatt az előfeldolgozási lépés alatt a következőt érjük:

A tanítómintahalmazban lévő adatokat (amelyekre tekinthetünk valószínűségi vektorváltozókként is) nulla várhatóértékűvé és egységnyi szórásúvá normáljuk (standardizáljuk). Ehhez használjuk fel az adatunkhoz tartozó, a 2-a blokkban kiszámolt várhatóérték (*mean*) és szórást (*std*) közelítést. Ahhoz, hogy az adatunkat megfelelően transzformáljuk ezek felhasználásával, először ki kell vonni az összes elemből a teljes mintahalmaz várhatóértékét, ezt követően pedig le kell osztani a szórással. Egészítsük ki tehát a 2-b blokkot a következő kóddal:

```
inputs = (inputs - mean)/std
```

Ezt követően futtassuk a teljes kódot a korábbiaknak megfelelően. Várhatóan kevesebb epoch elég lesz a hálónak az adathalmaz osztályozásának megfelelő megtanulására.

A bemeneti mintapontok normalizálását a gyakorlatban érdemes jellemzőnként (azaz koordinátánként) végezni. Esetleg ha vannak olyan jellemzők (feature) amelyek szorosan összefüggnek, akkor azok várhatóértékének és szórásának számítása történhet együttesen.

02_MLP – Circles.ipynb

Ebben a feladatban is még az egy rejtett rétegű MLP-kel foglalkozunk. Most megnézzük, hogy a megtanult szeparálási felületre milyen hatással van a rejtett rétegben található neuronok száma. Valamint több különböző bátorsági tényező mellett megnézzük, hogy a hálónk tanulásának gyorsasága mennyiben változik.

Első futtatás

Futtassuk a kódot az elejétől, egészen a (2)-es blokk végéig. A tanítómintahalmaz két egymással koncentrikus köríven található zajos mintapontokból áll.

Kérdés: Amennyiben itt is egy rejtett rétegű MLP-t alkalmazunk, legalább hány neuronnak kell lennie ebben a rétegben ahhoz, hogy szeparálható legyen a két osztály?

Futtassuk végig a kódot. (Ismételten csak a 9-b blokk végéig, hiszen ha a (10) blokkban lévő kódot is lefuttatjuk, akkor törlődik a felépített számítási gráfunk és nem tudunk rátanítani a hálóra.)

Tanítsunk rá a hálóra a 8-b blokkban lévő ciklus lefuttatásával annyiszor, hogy a háló már jól megtanulja a feladatot.

Második futtatás

Állítsuk át a (3)-as blokkban lévő *num_hidden_1* változó értéket valamilyen más számra. Ez a változó határozza meg az MLP rejtett rétegében lévő neuronok számát. Próbálkozzunk több értékkel is, és futtassuk végig a kódot. Nézzük meg, hogy a neuronok számának növelésével miként változik a megtanult szeparáló felület alakja (9-a és 9-b blokkok kimenetei).

Harmadik futtatás

Vizsgáljuk meg a (6)-os blokkban lévő *learning_rate* bátorsági tényező paraméter hatását a tanulásra. Próbáljunk ki a kezdetinél lényegesen kisebb és nagyobb értékeket is. Nézzük meg,

hogy melyik esetben kb. hány epochra van szükség a helyes osztályozás eléréséhez. Ezenkívül nézzük meg a paraméter hatását a 8-c részben lévő tanulási görbére.

03_MLP – TwoSpirals.ipynb

Ebben a részben az eddig használt egy rejtett rétegű MLP-t kibővítjük több rejtett rétegűre.

Tanítómintahalmazként a demópéldákban gyakran megjelenő kettősspirál mintát használjuk. Előadáson elhangzott, hogy ez a feladat egy rejtett rétegű MLP-vel igencsak nehezen megoldható probléma.

Első futtatás

Futtassuk le először a kódunkat változtatás nélkül. Próbálkozzunk az epochok számának változtatásával (8-a blokk, *num_epochs*), valamint a rejtett rétegben lévő neuronok számának növelésével ((3)-as blokk, *num_hidden_1*).

Mit tapasztalunk? Képes a háló megtanulni a mintahalmaz hibátlan osztályozását? Az epochok számának növelésével és a rejtett rétegbeli neuronok számának növelésével mennyiben javul a dolog, látszik valamilyen fejlődés? Hogyan változik a tanulás sebessége?

Második futtatás

Bővítsük ki most a kódot úgy, hogy az MLP két rejtett réteget tartalmazzon. Ennek egy lehetséges megoldása a következő.

```
num_hidden_2 = 80
```

A (3)-as blokkba vezessük be a következő változót:

Ezen kívül, ha korábban az első rejtett réteg neuronszámát (*num_hidden_1*) magasabb értékre állítottuk, állítsuk most vissza ezt is 80-ra.

Írjuk át a 4-b blokk tartalmát a következőre:

```
# Creating model variables
# Weights
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, num_hidden_1])),
    'h2': tf.Variable(tf.random_normal([num_hidden_1, num_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_hidden_2, output_dim]))
}
# Biases
biases = {
    'b1': tf.Variable(tf.random_normal([num_hidden_1])),
    'b2': tf.Variable(tf.random_normal([num_hidden_2])),
    'out': tf.Variable(tf.random_normal([output_dim]))
}
```

Az 5-a blokk tartalmát is írjuk át a következőre:

```
# Creating the network's structure with the mapping below
def multilayer_perceptron(x_data, weights, biases):
    # First hidden layer with ReLU activation
    layer_1 = tf.add(tf.matmul(x_data, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

model_output = multilayer_perceptron(x_data, weights, biases)
```

Ha mindent átírtunk, akkor futtassuk le előről az egészet.

Változott a modell tanulási képessége? Érdemes itt is kísérletezni a háló rejtett rétegeiben található neuronok számával ((3)-as blokk, *num_hidden_1* és *num_hidden_2*).

Ha esetleg a (80, 80) rejtett neuron mellett nem tanulta meg a háló jól az osztályozást, akkor először tanítsunk rá egyszer-kétszer, ha így sem kaptunk jó eredményt, akkor inicializáljuk újra a hálót. Ha pedig ez sem segít, akkor növeljük a neuronok számát (100, 100)-ra.

II. rész: A túltanulás

Ebben a részben továbbra is a többrétegű perceptron modellt (MLP) használjuk, azonban a figyelem középpontjában az ún. túltanulás jelensége kerül.

Az előző példákban láthattunk már számos különböző összetettségű példát (bár az összes a nagyon egyszerűnek tekinthető kétdimenziós térben helyezkedett el, valamilyen struktúrában). Láttuk, hogy bonyolultabb mintahalmazhoz nagyobb komplexitású hálóra volt szükségünk. Általánosságban elmondható, hogy a háló komplexitásának növelésével (ami MLP esetén a rejtett rétegek számának növelését és a bennük található neuronok számának növelését jelenti) bonyolultabb osztályozások is végrehajthatóak.

Most arra következnek példák, hogy miért nem éri meg a tanítási idő túlságos megnövekedésén kívül a hálónkat adott komplexitású probléma esetén túlságosan nagynak (komplexnek) megválasztani. Abban az esetben, ha a hálóban található szabad paraméterek száma lényegesen nagyobb, mint ami az adott probléma megoldásához szükséges lenne, akkor a hálónk képes lesz a kívánt szeparáló felületnél lényegesen összetettebbet is megtanulni. Mivel általában a mintáink zajosak, ez ennek a zajnak a felesleges (és nemkívánt) megtanulását is jelenti.

Ebben a részben először megvizsgáljuk miként jelentkezik egyszerű mintahalmazok esetén a túltanulás. Miként jelenik meg ez a kétdimenziós bemeneti mintatérben, ha szemléltetjük az osztályokhoz megtanult tartományokat.

Ezt követően pedig több módszert is megvizsgálunk, amelyekkel csökkenthető a túltanulás mértéke. Megnézzük a hálóban alkalmazott nemlinearitások hatását, kibővítjük a használt költségfüggvényt (veszteségfüggvény) egy regularizációs taggal, valamint implementáljuk a korai leállás módszert.

04_MLP – Overfitting.ipynb

Ebben a feladatban a korábbiaknál is egyszerűbbnek mondható adathalmazzal fogunk foglalkozni, azonban ez bőven elég lesz ahhoz, hogy a túltanulás ábrázolható legyen.

Első futtatás

Futtassuk a kódot. Amennyiben az elején (a véletlenszerű generálás miatt) olyan mintahalmazt kapnánk, amelyben a kék mintapontok között nincs piros és fordítva, akkor generáljunk újat, erre ugyanis szükségünk lesz.

A tanítás során egy két rejtett rétegű MLP-t használunk, rétegenként 10 neuronnal. Ha lefuttatjuk a teljes kódot, akkor látható, hogy a háló képes megtanulni a mintapontok hibátlan osztályozását.

Azonban ebben az esetben feltételezhető, hogy nem ez a célunk. Az eredeti eloszlása a két osztálynak (azaz a megoldandó probléma) valami olyasmi lehetett, amelyet egy lineáris szeparálásra képes eszközzel osztályozni lehet. Az, hogy a ránézésre jól elkülöníthető két

osztályban mégis található a másik osztályba sorolt mintapont, valamilyen hiba (vagy zaj) eredménye lehet.

Második futtatás

Írjuk át a (3)-as blokkban lévő kódban a rejtett rétegekben található neuronok számát 2-2-re (*num_hidden_1*, *num_hidden_2*). Futtassuk újra a teljes kódot. Amennyiben a megoldandó feladatnak megfelelő komplexitású hálót használunk, a túltanulás nem jelentkezik. A további kódok futtatásához írjuk vissza a rejtett rétegekben lévő neuronok számát a kezdeti 10-10-re.

Harmadik futtatás

Az, hogy a háló mennyire képes a komplex szeparáló felületek megtanulására (és így a túltanulásra), függhet az alkalmazott nemlinearitásoktól. Az előző futtatás során az ún. ReLU-kat használtuk. Írjuk most át az 5-a blokkban lévő nemlinearitásokat a logisztikus szigmoidra.

Ehhez cseréljük le az alábbi két sort:

```
layer_1 = tf.nn.relu(layer_1)

layer_2 = tf.nn.relu(layer_2)
```

arra, hogy:

```
layer_1 = tf.nn.sigmoid (layer_1)

layer_2 = tf.nn.sigmoid (layer_2)
```

Tanítsuk újra előröl a hálónkat. Jól láthatóan csökkent a túltanulás mértéke, azaz kevésbé vannak ugrások és nyúlványok a szeparáló felületben. Ez lényegében a logisztikus szigmoid simító hatásának a következménye.

Negyedik futtatás

Először is írjuk vissza az előzőpontban átírt nemlinearitásokat az eredetileg használt ReLU-ra.

Ezt követően vegyük ki a kommentet a 6-a blokk alábbi kódrészletéből:

```
lambda_l2 = 0.05
```

Ezt hajtsuk végre a 6-b blokkban lévő kód alábbi sorával is:

```
regularizer = tf.nn.l2_loss(weights['h1']) + tf.nn.l2_loss(weights['h2'])
```

Valamint tegyük ezt meg a 6-c blokkban lévő alábbi kódrészlettel is:

```
xentropy = tf.reduce_mean(xentropy + lambda_l2*regularizer)
```

Futtassuk ezután újra a kódot. A tanítás végrehajtása után kapott szeparáló felület (akár több rátanítást követően is) egyszerűbb. Érdemes kísérletezni a *lambda_l2* regularizáció „erősségét” szabályozó paraméterrel.

05_MLP – Overfitting2.ipynb

Az előző feladatban kis betekintést kaphattunk a regularizáció világába. Ebben a részben további megközelítésekkel bővítjük ismereteinket.

Az előző részben ún. explicit regularizációt alkalmaztunk (amikor a veszteségfüggvényt explicite kibővítjük egy regularizációs taggal), most megismerkedünk az ún. korai leállással, amire implicit regularizációként is tekinthetünk. Az eljárás lényege, hogy a rendelkezésünkre álló teljes adathalmazt három részre bontjuk: tanítómintahalmazra, validációs halmazra és teszt halmazra.

A tanítómintahalmazon végezzük a háló tanítását. Minden egyes tanítási lépést végrehajtó iterációban a validációs halmazon (amit tehát közvetlenül a tanítás során alkalmazott súlymódosításnál nem lát a háló) mérjük a háló jóságát. Ezen a mintahalmazon tehát minden lépés során kiértékeljük a hálón válaszának hibája felett definiált költségfüggvényt. Abban az esetben, ha azt tapasztaljuk, hogy a validációs halmazon mért ún. validációs hiba elkezd nőni, mikor még a tanítómintahalmazon számolt hiba csökken, túltanulásról beszélhetünk. Ekkor érdemes leállítani a tanulást, és visszatölteni a legjobb eredményt adó paraméterezést.

Mivel a tanítás során a tanítómintahalmaz elemeit használjuk fel a háló szabad paramétereinek módosítására, így érthető módon a rajta számolt hiba folyamatosan csökkenni fog (legalább is fullbatch esetén). Ugyanez a tanítás során közvetlenül fel nem használt validációs mintára is elmondható. Azonban egy ponton túl a háló elkezd túltanulni, aminek következtében a tanítómintahalmazban rendelkezésre álló mintapontokon található zajt tanulja meg a rendszer, veszítve ezzel a számunkra fontos általánosítóképességéből.

Érdemes tehát a tanítást ekkor leállítani és a legjobb eredményt biztosító paraméterezést visszatölteni. A felosztott teljes mintahalmaz között szerepelt az ún. teszthalmaz, amelyről eddig még nem esett szó. Ezt a mintahalmazt használhatjuk a tanítás leállásakor kapott végleges hálónk minősítésére. Fontos, hogy ennek a minősítésnek olyan mintahalmazzal kell történnie, amelyet a hálónk még impliciten sem látott. A tanítómintahalmaz kizárása egyértelmű, azonban a validációs mintahalmazt sem használhatjuk, hiszen a leálláshoz a hiba rajta történő kiértékelését is felhasználtuk, így implicit módon belekerült a tanítási eljárásba. A teljesen elkülönített teszthalmazt azonban ekkor látja először a háló, így a rajta értelmezett hiba (amennyiben a valós probléma eloszlása és a mintahalmazunk eloszlása azonos) nem fog megtéveszteni minket.

A három mintahalmazt a következő arányban szokás (megfelelő mintaszám esetén) felbontani:

- tanító mintahalmaz: 70%, esetleg 70%
- validációs mintahalmaz: 20%, esetleg 15%
- teszt mintahalmaz: 10%, esetleg 15%

A kiadott demókód több helyen is tartalmaz új részeket. A (2)-es blokkban történik meg a mintahalmaz három részre bontása. Bár a mintahalmazunk egy mátrixban található, a részekre bontása nem három összefüggő tartományra bontással történik, hanem véletlenszerű sorok kiválasztásával, biztosítva ezzel az egyenletes szelektálást.

A 2-c blokk kimenetén látható a felbontás eredménye.

A (9)-es blokkban található a másik része a korai leállásnak. Először is a 9-a rész első blokkjában definiáltunk egy változót, amelyben a validációs mintahalmazon számított hibát mentjük el (*validation_loss_vec*).

A 9-b rész több mindennel is bővült. A korai leállás implementálásához a következő dolgokra lesz szükségünk. Először is kell egy változó, amiben a mindenkori legjobb validációs hibát elmentjük (*best_validation_error*). Ez lesz az az érték, amellyel az aktuális validációs hibát összehasonlítjuk minden egyes tanítási iterációban. Abban az esetben, ha az aktuális validációs hibánk értéke nagyobb az eddigi legkisebb értéknél (azaz a validációs hiba elkezd nőni), akkor a *counter* változó értékét elkezdjük növelni. Előre meghatározunk a *patience* változóban egy ún. türelmi időt, mellyel a korai leállás hangolható. Nem akarjuk rögtön az első alkalommal leállítani a tanulást, ahogy elkezd nőni a validációs hiba. Szeretnénk várni egy előre megadott értékű lépésszámot, mely során egyszer sem csökken a hiba. Ha a türelmi időn belül újra elkezd csökkenni a hiba, akkor a *counter* számláló értékét nullázzuk. Ha a *counter* értéke eléri a *patience*-ben definiált értéket, akkor a tanítást leállítjuk. Ekkor még a legjobb eredményhez tartozó paramétereket vissza kell állítanunk. Ehhez minden egyes tanítási iterációban a *W_best* és a *b_best* változókba elmentettük a legjobb eredményt biztosító paramétereket.

A visszatöltés (mivel a célváltozónak számító *weights[…]* változók TensorFlow változók), a kódban található formában tehetjük meg az *assign* használatával.

Első futtatás

A példa első futtatása során nem alkalmazunk még semmilyen regularizációt. Futtassuk végig a kódot, esetleg többször is rátanítva a hálóra, és nézzük meg a kapott eredményt.

A hálónk a mintahalmazon lévő zaj miatt a két osztály mintáinak találkozásánál lévő „zavaros” sávban a kívánt lineáris szeparálás helyett rátanult a zajra, és egy komplexebb szeparáló felületet tanult meg.

Második futtatás

Ebben a részben a már megismert explicit regularizáció hatását vizsgáljuk meg. A 7-a blokkban lévő *lambda_l2* értékét állítsuk a következőre:

```
lambda_l2 = 0.05
```

Ezt követően futtassuk a kódot a korábbi tanítást törlő (5)-ös blokktól. Remélhetőleg a regularizációt nélkülöző eredményhez képest egy simább szeparáló felületet megtanult háló eredményét láthatjuk a (10)-es blokk kimenetein. Érdemes több lambda értéket is kipróbálni.

Harmadik futtatás

Ebben a részben az implicit regularizációt végrehajtó korai leállás hatását vizsgáljuk meg. Ehhez először is nullázzuk ki az előző részben beállított lambda értékét a 7-a blokkban található kódban.

```
lambda_l2 = 0.00
```

Ahhoz, hogy a tanítás leálljon a fent részletezett túltanulás esetén, írjuk át a 9-b blokkban található *patience* változó értékét a következőképpen:

```
patience = 50
```

Ezenkívül ugyanebben a részben lévő, az információk kiírásának gyakoriságát meghatározó értéket is változtassuk meg. (Ez a 9-b blokk kódjának alján látható if (…) rész.)

```
if (i+1)%50==0:
```

Futtassuk így újra az (5)-ös résztől a kódunkat.

Értékeljük a 9-b rész második blokkjának kimenetén látható tanulási görbéket.

Nézzük meg a (10)-es blokk kimenetein látható szeparálás eredményét. Valószínűleg a kapott eredmény nem lesz annyira lineáris jellegű, mint az explicit regularizáció alkalmazása mellett kapott eredmény. Azonban remélhetőleg látható, hogy a háló tanítása annak ellenére leállt, hogy még tudott volna a módosítani a szeparáló felület komplexitásán kisebb nagyobb ugrások belevitelével.

Negyedik futtatás

Utolsó lépésként még egészítsük ki a kódunkat, hogy az eddigi fullbatch-es tanítás helyett minibatch-el történjen a tanítás. Igazán nagy mintahalmazok esetén nehézséget jelenthet a tanítási iterációk során a teljes mintahalmazon (fullbatch) számítani a hibát és ez alapján végezni a paraméter módosítást.

A minibatch használata ekkor azt jelenti, hogy a teljes tanítóhalmaznak csak egy részhalmazát használjuk fel egy iterációban. Itt perszer több kérdés is felvetődhet a felbontást illetően. A mintapontok véletlenszerű kiválasztását végezhetjük úgy, hogy a tanítópontban lévő mintaszám felhasználását követően az összes mintahalmazt legalább egyszer alkalmazzuk. Egy lehetséges másik megközelítés, hogy minden részhalmaz kiválasztásakor teljesen véletlenül választjuk ki az adott részhalmazt. Ez utóbbi esetben elképzelhető, hogy egy 1000 tanítópontot tartalmazó mintahalmaz esetén összességében már 13 darab 100 mintapontos minibatchen végeztünk tanítást mégsem került sorra mindegyik mintapont legalább egyszer.

Azt, hogy milyen módszert használunk a részhalmazok kiválasztásakor, ránk van bízva.

Írjuk át a 9-b rész első blokkjának kódját a következőre.

```
best_validation_loss = np.inf
counter = 0
minibatch_size = 100

patience = 30

W_best = sess.run(weights)
b_best = sess.run(biases)

# The training loop
for i in range(num_epochs):
    rand_index = np.random.choice(num_samples, size=minibatch_size)
    inputs_minibatch = inputs[rand_index]
    targets_minibatch = targets[rand_index]

    # Calling one train_step
    sess.run(train_step, feed_dict={x_data: inputs_minibatch, y_target: targets_minibatch})

    # Calculating and saving the actual loss over the training set
    train_loss = sess.run(xentropy, feed_dict={x_data: inputs_minibatch, y_target: targets_minibatch})
    train_loss_vec.append(train_loss)

    # Calculating and saving the actual loss over the validation set
    validation_loss = sess.run(xentropy, feed_dict={x_data: inputs_validation, y_target: targets_validation})
    validation_loss_vec.append(validation_loss)

#*********************************
#       Early stopping
#*********************************

    if(validation_loss < best_validation_loss - 0.1):
        best_validation_loss = validation_loss
        counter = 0
        W_best = sess.run(weights)
        b_best = sess.run(biases)
    else:
        counter += 1

    if(counter > patience):
        print("Early stopping - Load back the best model")

        assign_op1 = weights['h1'].assign(W_best['h1'])
        assign_op2 = weights['h2'].assign(W_best['h2'])
        assign_op3 = weights['out'].assign(W_best['out'])

        assign_op4 = biases['b1'].assign(b_best['b1'])
        assign_op5 = biases['b2'].assign(b_best['b2'])
        assign_op6 = biases['out'].assign(b_best['out'])

        sess.run(assign_op1)
        sess.run(assign_op2)
        sess.run(assign_op3)
        sess.run(assign_op4)
        sess.run(assign_op5)
        sess.run(assign_op6)

        break

#*********************************
#*********************************

    # Logging to screen number of epochs and current loss
    if (i+1)%50==0:
        print('Step #' + str(i+1))
        print('Train Loss = ' + str(train_loss))
        print('Validation Loss = ' + str(validation_loss))
        print()
```

Amit érdemes megvizsgálni, az a 9-b rész második blokkjának kimenetén ábrázolt tanulási görbe. Mivel a hibamódosítás nem a teljes tanítómintahalmazon történt, hanem csak egy részhalmazán, a korábbi simábbnak mondható fekete görbe most kisebb nagyobb ugrásokat tartalmaz. Ennek oka az, hogy a részhalmazon számított gradiens módosítása nem feltétlen egyezik meg a teljes halmazon számítottal, így a módosítás nem feltétlen javítja a háló teljesítményét minden egyes lépésben. Ez azonban arra is jó lehet, hogy nemkonvex hibafelületek esetén a hálót kiugrasszuk a lokális minimumokból.

III. rész: Radiális bázisfüggvényes hálók (RBF)

Ebben a részben a Radiális bázisfüggvényt alkalmazó hálókkal foglalkozunk. Ennek a megközelítésnek az egyik alapötlete, hogy a rendelkezésreálló bemeneti mintatérbeli lineárisan nem szeparálható feladatot Gauss bázisfüggvények alkalmazásával áttranszformáljuk egy ún. jellemzőtérbe, ahol a feladat már lineárisan szeparálható. Ekkor már használhatjuk a legelső demóban bemutatott egyszerű neuronunkat az osztályozásra.

A jellemzőtér dimenzióját az egyes esetekben felhasznált Gauss függvények száma határozza meg. Az alább következő feladatok tartalmazzák a kipróbálandó paraméterezéseket, illetve a hozzájuk tartozó eredmények rövid jellemzését.

01_RBF – Circles.ipynb

Ebben a feladatban egyetlen bázisfüggvényt használunk. Azt vizsgáljuk, hogy a Gauss bázisfüggvényhez tartozó különböző szórás (*sigma*) paraméterek használata milyen hatással van az osztályozás eredményére.

Első futtatás

Első futtatás során használjuk a 3-a részben megadott 0.6-os szigma (szórás) értéket. Nézzük meg a 3-e blokk kimenetén kapott ábrát. Ez az 1 dimenziós jellemzőterünkben ábrázolja a mintapontokat. Látható, hogy ebben a térben egy lineárisan szeparálható feladatot kell megoldanunk. Ezért használhatjuk a legelső demóban összeállított egyszerű neuronunkat.

Hajtsuk végre az osztályozást. Láthatjuk, hogy a kapott eredmény tökéletesen osztályozza a mintapontokat.

Második futtatás

Állítsuk a 3-a részben található *sigma* paraméter értékét 0.3-ra. Tanítsuk újra a hálót. A kisebb szórás paraméter hatása jól látható a 3-e blokk kimenetén. Az alkalmazott Gauss függvényünk hamarabb lecseng, ezért a bázisfüggvény középpontjától távolabb lévő mintapontok értéke (azaz a kék osztályba tartozó mintapontok) lényegesen kisebb, mint az előző paraméterezés esetén.

Végrehajtva a tanítást, a (10)-es blokk kimenetén kapott eredményen látható az alkalmazott Gauss függvény gyorsabb lecsengése. A piros mintapontok körül kialakított régió mérete kisebb lett.

Harmadik futtatás

Állítsuk a 3-a részben található *sigma* paraméter értékét 0.85-re. Itt a nagyobb szórás miatt a Gauss bázisfüggvény csak a középponttól nagyobb távolságra lévő mintapontok esetén cseng le. Ez jól látható a 3-e blokk kimenetén látható jellemzőtérbeli alakon. A (10)-es blokk kimenetén a piros régió területe megnőtt.

02_RBF – EXOR.ipynb

Ebben a feladatban két bázisfüggvényt használunk, így a kapott jellemzőterünk kétdimenziós lesz.

Első futtatás

Futtassuk a kódot a kezdeti 0.6-os szórású Gauss bázisfüggvényekkel. A 3-e részben látható a kapott kétdimenziós jellemzőtérbeli mintareprezentáció. Látható, hogy lineárisan szeparálható feladatot kaptunk.

Futtassuk a tanítást. Amennyiben nem tanulja meg elsőre a háló a tökéletes osztályozást, tanítsunk rá a hálóra a tanítási ciklus többszöri lefuttatásával.

Második futtatás

Állítsuk át a 3-a részben lévő szigma paraméter értékét 0.1-re. A 3-e blokk kimenetén látható, hogy a két osztály mintapontjai összecsúsznak a jellemzőtérben. A kisebb szórás miatt, hamarabb lecseng a Gauss függvény, és így a piros osztályba tartozó mintapontok szélén lévő bemenetek esetén már csak kis értéket vesz fel az ott elhelyezett Gauss függvény. Ezek a piros mintapontok a jellemzőtérben nagyon közel kerülnek a kék mintapontokhoz.

A tanítást végrehajtva nem is kapunk jó eredményt.

Harmadik futtatás

Állítsuk át a 3-a részben lévő szigma paraméter értékét 2.0-ra. Ekkor a 3-e rész kimenetén megjelenő jellemzőtérbeli osztályozás már nem is lineárisan szeparábilis.

A tanítás ebben az esetben sem működik.

03_RBF – TwoSpirals.ipynb

Ebben a részben a korábban már szerepelt kettősspirál feladattal foglalkozunk. A korábbi RBF-es példákkal ellentétben a komplexebb mintapont struktúra miatt lényegesebb nehezebb lenne a megfelelő bázisfüggvények helyének és szórásának kialakítása. Emiatt ebben a demóban minden mintapontra helyezünk egy Gauss függvényt.

Futtassuk a kódot a 3-a részben található szigma paraméter állítgatásával. Próbáljuk ki a 0.2, a 0.075, a 0.75 és az 1.0 paramétereket. Figyeljük meg az osztályozás eredményét a (10)-es blokk kimenetén.