Korszerű számítástechnikai módszerek a fizikában

1.

Általános függvényközelítés C++-ban implementált neurális hálóval

Mocskonyi Mirkó György

Fizika MSc I. évfolyam

1. Bevezető

A projektemben C++ programnyelv felhasználásával implementáltam egy olyan környezetet, mely lehetővé teszi, hogy teljesen kapcsolt, előrecsatolt neurális hálókat lehessen betanítani különféle függvények közelítésére. A háló tetszőleges architektúrátval rendelkezhet, így a rejtett rétegek, valamint a neuronok száma szabadon változtatható, illetve beállítható. A projektben elvégzett feladataim a következők: különféle egyváltozós, valós szám értékű függvények, nevezetesen az x^2, $sin(x)$, $sin(x)/x$, e^{-x^2} leképezések közelítése, valamint a tanítás zajra való érzékenységének és a betanított háló extrapolációs képességének vizsgálata.

1.1. Neurális háló

Nagyon röviden egy előrecsatolt, teljesen kapcsolt neurális háló működéséről általánosságban. Feltesszük, hogy valamilyen bemeneti értékekhez ismerjük valamilyen leképezés értékét, a hálóval pedig ezt a leképezést szeretnénk megtanulni. Ehhez definiálhatunk egy alapvető transzformációt, amelynek valamilyen egymásutánjától várjuk el, hogy képes legyen közelíteni az alulfekvő leképezést.

Legyen egy adott ponton a bemenetünk (az i-edik transzformáció kimenete, $N > i \geq 0$) $\mathbf{x}^i$, k^i dimenziós valós vektor. Ekkor az $i+1$-edik transzformáció:

$$\mathbf{x}^{i+1} = g^{i+1}\left(\mathbf{W}^{i+1}\mathbf{x}^i + \mathbf{b}^{i+1}\right), \tag{1}$$

ahol $\mathbf{W}^{i+1}$ egy $k^{i+1} \times k^i$ dimenziós mátrix, $\mathbf{b}^{i+1}$ egy k^{i+1} dimenziós vektor, g^{i+1} pedig egy nemlineáris differenciálható függvény. Ilyen transzformációk egymásutánja tulajdonképpen a neurális háló, ahol a k^i dimenziót tulajdonképpen az i-edik rétegében található neuronok számával feleltehetjük meg, k^0 a bemenet dimenziója, k^N pedig a kimenet dimenziója. Ekkor azt mondjuk, hogy a hálónak összesen N rétege van, melyek közül az elsőt bemeneti rétegnek, az utolsót kimeneti rétegnek, az összes többit pedig rejtett rétegnek nevezzük.

Miután az $\mathbf{x}^0$ bemenettől eljutottunk az $\mathbf{x}^N$ kimenetig, az igazi kimenet ($\mathbf{y}$) imseretében kiszámíthatjuk a veszteségértéket valamilyen erre speciálisan kiválasztott függvény szerint, például az átlagos négyzetes hiba gyökét:

$$L = \sqrt{\frac{1}{k^N}\sum_{i=1}^{k^N}(\mathbf{x}_i^N - \mathbf{y}_i)} \tag{2}$$

Mivel minden eddig felhasznált transzformáció differenciálható, vagy feltettük korábban, hogy differenciálható, így a veszteségértéket differenciálhatjuk a $\mathbf{W}^{i+1}$ mátrixok, valamint a $\mathbf{b}^{i+1}$ vektorok szerint. Ezután példának okáért a $\mathbf{b}^{i+1}$ vektort frissíteni tudjuk a gradiens

ereszkedés módszerével a következő módon:

$$\mathbf{b}^{i+1} \Rightarrow \mathbf{b}^{i+1} - \alpha \frac{\mathrm{d}L}{\mathrm{d}\mathbf{b}^{i+1}}, \tag{3}$$

ahol α az úgynevezett tanulási ráta, amely a felhasználó által beállított hiperparamétere a modellnek. Miután minden réteg minden paramétere frissítésre került, azt mondjuk, hogy egy teljes tanítási ciklus (epoch) befejeződött, másképpen mondva a háló az összes bemenetet egyszer látta és a gradiens ereszkedések megtörténtek. Nagyon fontos hangsúlyozni, hogy a g^{i+1} függvények nemlineárisak, azok ugyanis az egyetlen forrásai a nemlinearitásnak ebben az egyszerű modellben. Ha nem lennének azok, akkor az egész neurális háló csupán egy egyszerű lineáris leképezésre lenne redukálható.

A fenti modellnek és a tanításának a célja az, hogy a modell a megtanult leképezésnek megfelelően számára tanítási idóben nem látott bemenetekre is jól közelítse a helyes kimenetet. Matematikailag belátható, hogy elegendő számú rejtett réteggel és neuronnal tetszőleges nemlineáris leképezés tetszőlegesen pontosan közelíthető. Innen ered jelen projekt motivációja, mely ennek az állításnak a demonstrálására tesz kísérletet, azonban emellett talán nagyobb hangsúlyt helyez a C++ programnyelvben történő implementáció létrehozására és az abból származó tapasztalatszerzésre.

2. Az implementáció

Az implementáció megtalálható a `https://github.com/mmgy/MMCTP1/tree/master/Project` GitHub oldalon, ahol is az *nn.h* fájl tartalmazza a hozzá tartozó kódot. Alapvetően a felhasznált standard objektum az *std::vector*, erre építettem föl az egész környezetet. Maga a kód több osztály definiálásával indul, amelyek után a szükséges függvénydefiníciók következnek. Összességében egy tetszőleges architektúrájú, uniform random inicializált súlyokkal rendelkező neurális hálót definiálnak, amelyben ReLU ($ReLU(x) = \max(0, x)$), valamint szigmoid ($S(x) = 1/(1 + \mathrm{e}^{-x})$) aktivációs függvények érhetőek el. A tanítási algoritmus Mini-Batch Gradient Descent [1], ahol a gradiens a felhasználó által definiált méretű adatocsoportokon (batch-eken) átlagolódik, és aszerint történik meg a gradiens ereszkedés, a gradienseket pedig a modell a jól ismert hibavisszaterjedés (Backpropagation [3]) algoritmussal számítja ki. Veszteségfüggvényként a háló kimenete, valamint az igazi értékek közötti átlagos négyzetes hiba gyökét (Root Mean Squared Error, RMSE) használja.

A háló tanítását megkönnyítendő a tanító adatpontokat és a hozzájuk tartozó igazi kimeneteket is a minimális és maximális értékük alapján eltranszformáltam a $[0, 1]$ tartományra, majd a tanítás és kiértékelés végeztével visszatranszformáltam őket. Ebből adódik az, hogy a tanítás során a veszteségértékek más nagyságrendbe esnek, mint ahogy arra az eredeti bemeneti és kimeneti adatok alapján számítani lehetne. A validációs,

illetve a teszt adathalmazokon figyelmesnek kell lenni és a tanítóhalmaz szerint kell azokat is eltranszformálni, hogy a háló a tanítással összhangban álló értékeket kapjon. A transzformáció:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}} \cdot (max - min) + min, \tag{4}$$

ahol X az eredeti adat, X_{min}, X_{max} annak a minimuma, illetve maximuma, min, max pedig az átskálázott adat minimuma, illetve maximuma.

Végül lehetőség van a tanító bemenetekhez, illetve a tanító valódi értékekhez standard normális eloszlású zajt adni külön-külön, a felhasználó által beállított várhatóértékkel és szórással. Az implementációmban a zaj hozzáadása úgy történik, hogy minden epochban ugyanolyan várható értékű és szórású standard normális eloszlású számokat generál, azonban különböző "seed"-del, azaz a generált számok minden alkalommal különbözőek.

Az implementáció meglehetősen hosszú, de minden osztály és függvény előtt komment jelzi, hogy mire való, mit hajt végre az adott része a kódnak. Továbbá megjegyzendő, hogy sok helyen a kód nem a legeffektívebben, hanem naiv módon került megírásra, így relatíve egyszerű hosszú futási időt elérni, úgy gondolom azoban, hogy a projekt nehézsége, valamint kiterjedtsége indokolja a részemről mutatott ezt a fajta naivitást.

3. Tanítás

3.1. Általános megjegyzések

Az alkalmazott háló architektúrájáról általában azt lehet elmondani, hogy az első réteg aktivációja mindig szigmoid függvény volt, mivel ellenkező esetben képtelen volt a háló tanulni (túl magas tanulási ráta viszont egyéb instabilitásokat szül). Ennek az oka az lehet, hogy mivel a ReLU negatív értékekre nullára értékelődik ki, valamint a háló kezdósúlyai a $[-1, 1]$ tartományról vett egyenletes eloszlásból kerülnek inicializálásra, ezért lehetséges, hogy túl sok negatív neuron vágodik nulla értékre jelentős információt veszítve. Mindenesetre a többi rétegben gond nélkül lehetett ReLU aktivációt használni, a tanítás úgy már megfelelően zajlott. A réteg- és neuronszám tekintetében a gyors tanítást lehetővé téve 4-5 réteget használtam a legtöbb futtatáshoz, melyekben (a bemenettől és a kimenettől eltekintve) 16-64 neuron volt megtalálható.

Jónéhány hiperparaméter állítható az implementált modellben, melyek közül is a legfontosabbak az epochok számát, illetve a tanulási rátát megadó paraméterek. A tanulási rátát lehetséges változtatni a tanítás alatt, ehhez a *milestone*, valamint a *gamma* hiperparaméterek állnak rendelkezésre. Ezek jelentése, hogy minden *milestone* számú epoch után az elején beállított tanulási ráta a *gamma*-adára csökken. Sokszor hasznos a tanulási rátát változtatni, hogy közelebb tudjon kerülni a háló a veszteségfüggvény minimumához.

Természetesen, általában a validációs halmazon mért veszteséget kell figyelembe venni, ezért a futtatás során az is az epochok számával egyetemben kiíratásra kerül.

Ezeken kívül beállítható még a batch méret is, mely segít a háló tanulásában oly módon, hogy nem minden egyes adatpont után vett gradiensekkel léptetünk, hanem egyenletesebb, több adatponton kiátlagolt gradiensekkel. A nagyobb batch méret általában gyorsabb tanítást is eredményez, azonban az én mostani implementációm erre nem képes. Ennek az oka abban rejlik, hogy az adatpontok egyenként vannak adagolva a hálónak, nem egyszerre a teljes batch. Ennek a megoldása természetesen egyáltalán nem lehetetlen, de a kód jelentős részét újra kellene gondolni ennek érdekében, azt pedig a projektre fordított eddigi időt tekintve és a további idő szűke miatt nem tettem meg. Az utolsó hiperparaméter egy amolyan kényelmi beállítás csak, nevezetesen, hogy hány epoch-onként értékelje ki a háló a validációs adathalmazt tanítási időben és számoljon abból validációs veszteségértéket. Azért mondható kényelminek a beállítás, ugyanis nagyon ritkán okoz a futási idóben lassulást ha minden epochban kiíratjuk a validációs veszteségértéket is. Azonban azon ritkább esetekben, amikor lassítana, ritkábbra lehet állítani a validációs adathalmaz kiértékelését.

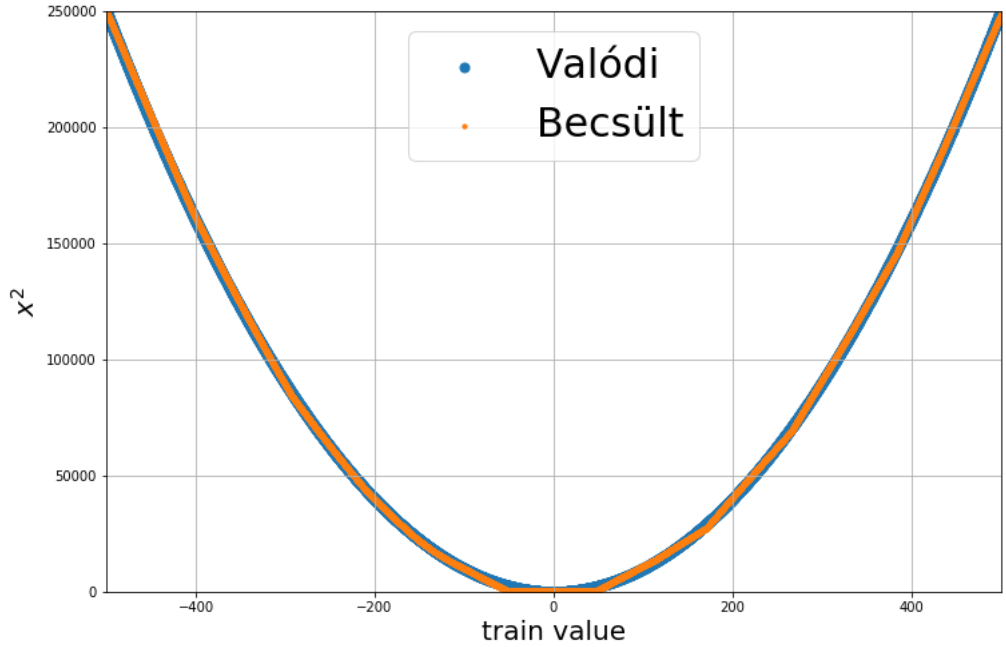
Az előzetes futtatási kísérletek alapján az alábbi architektúrát választottam ki és használtam a függvények közelítéséhez. Négy réteget használtam, melyeknek rendre 1, 16, 64, 1 dimenziók lettek beállítva. Az első és utolsó réteg 1 dimenziós, mivel egy bemenettel és 1 kimenettel dolgoztam ebben az esetben. A tanítás futtatásához 2000 epoch-ot, és 0.15 tanulási rátát használtam, melyet 400 epoch-onként az 1.5-ére csökkentettem. Az ezen beállításoktól való eltérést a későbbiekben minden alkalommal megemlítek. Emellet 16-ra állítottam a batch méretet és minden lépésben kiértékeltem a validációs halmazon. Ilyen módon tudtam monitorozni a validációs veszteségértékeket is.

A tanítási idő természetesen nagymértékben függ a háló méretétől és a neuronok számától, azonban a fenti választással az én esetemben a futási időt a tanító adatok mennyisége határozta meg leginkább. Mivel az implementációm nem a leghatékonyabb, valamint saját hardveren tanítottam a hálót, ezért úgy optimalizáltam a futási időre, hogy az 20-30 perc környékén legyen maximum.

3.2. x^2

Először az összes vizsgált függvény közül a legelemibbel kezdtem a háló tanítását, tehát a bemenetek négyzetét tápláltam be, mint igaz kimenetek. A tanításhoz -500 és 500 között generáltam bemeneteket 1.0-ás közökkel, így összesen 1001 tanító adattal dolgoztam. Validációs adathalmazként -1000 és 1000 közötti számokat használtam, így bőségesen volt olyan adatpont is, mely kívül esett a tanítás tartományán. Az eredményeket ugyanezen a tartományon értékeltem ki, de nagyobb finomsággal mintavételeztem. Az 1. ábrán

látható, hogy a háló a tanítás tartományán figyelemre méltóan jól megtanulta a négyzetre emelést. Az egyetlen érdekes tartomány a 0 körüli értékek, mivel azoknál egy vízszintes

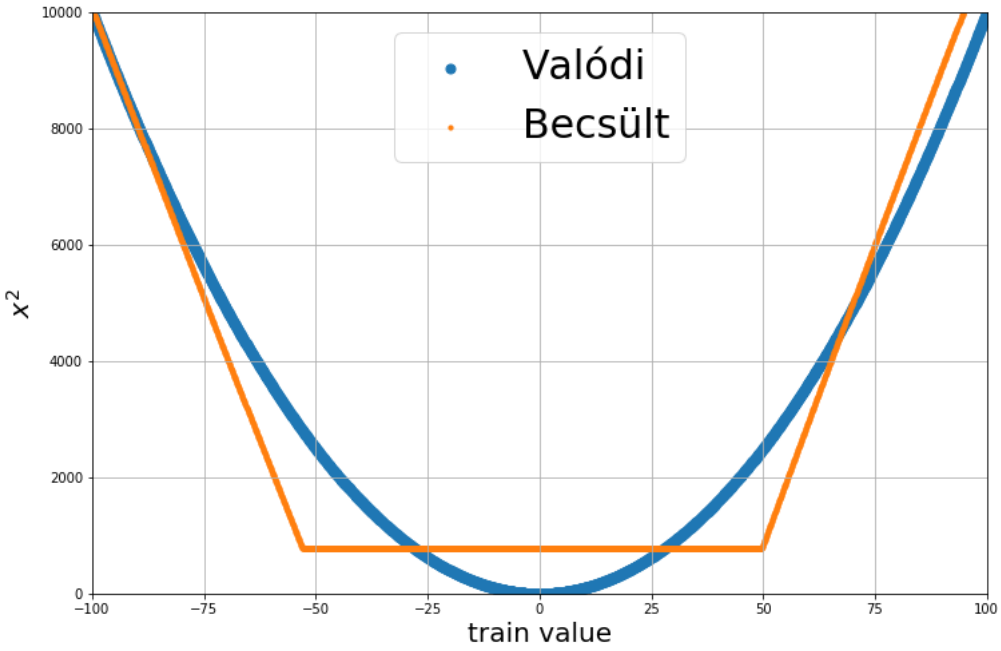


1. ábra. A négyzetre emelés tanítása a hálónak, az ábra a tanító adatok tartományát mutatja

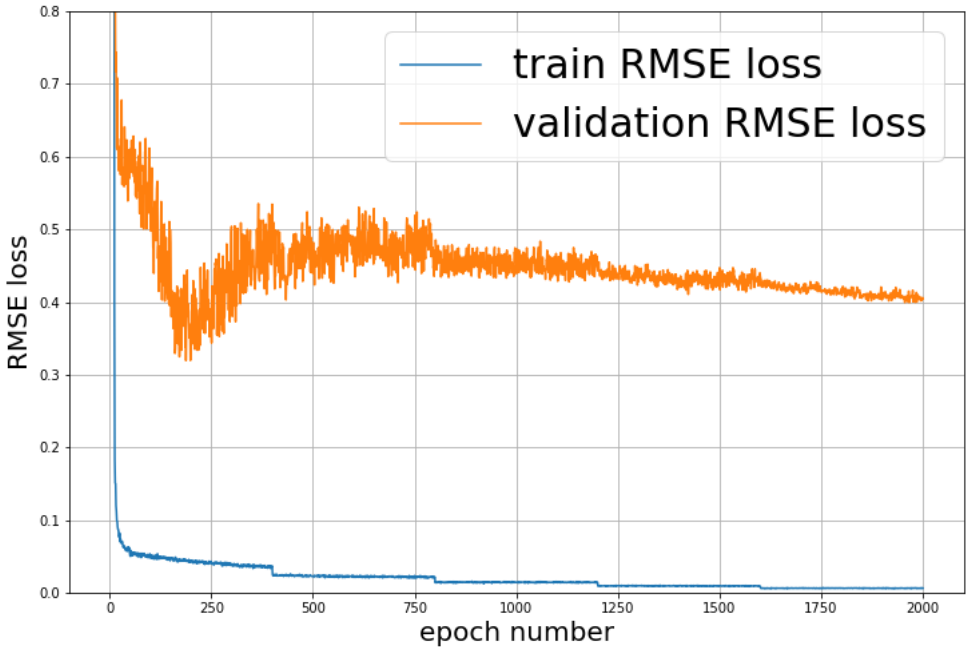
vonal látható, nagy eltéréssel a valódi értékektől. Ez a jelenség minden alkalommal jelentkezett, akkor is, ha szűkebb tartományon és nagyobb felbontással tanítottam a hálót - ekkor rövidült a vízszintes vonal hossza. Ezt mutatja nagyítva a 2. ábra.

A fent ismertetett tanítási eljárás természetesen fölöslegesen hosszú a négyzetre emelés tanításához, a validációs veszteségértékek a tanítás nagy részén nem csökkentek tovább és a tanulási ráta csökkentése sem segített a helyzeten. A tanító adathalmazon számolt veszteségérték azonban láthatóan mindig csökkent a tanulási ráta csökkentésével, jelen esetben valószínűsíthetően az történt, hogy túltanult a háló, azaz memorizálta az adathalmazt. Természetesen mivel relatíve kis hálóval dolgoztam, ez csak egy limitált mértékben történhetett meg. Szigorúan véve a validációs veszteségértékek minimumát jelentő modellnél kellett volna megállnom. A 3. ábrán láthatóak a két halmazon a tanítási időben számolt veszteségértékek.

A leképezésnél érdekes kérdés, hogy mennyire képes extrapolálni a háló a tanítási tartományon kívül. Itt jegyzendő meg, hogy mivel a négyzetre emelés nem korlátos függvény, így a hálónak a tanító halmaz pontjaitól távol levő pontokra sokkal nagyobb értékeket kell becsülnie, mint amilyenekkel tanítási időben találkozik. A 4. ábrán látható háló becslése a teljes teszt adathalmazon kiértékelve. Látható, hogy a háló extrapolációs képességei

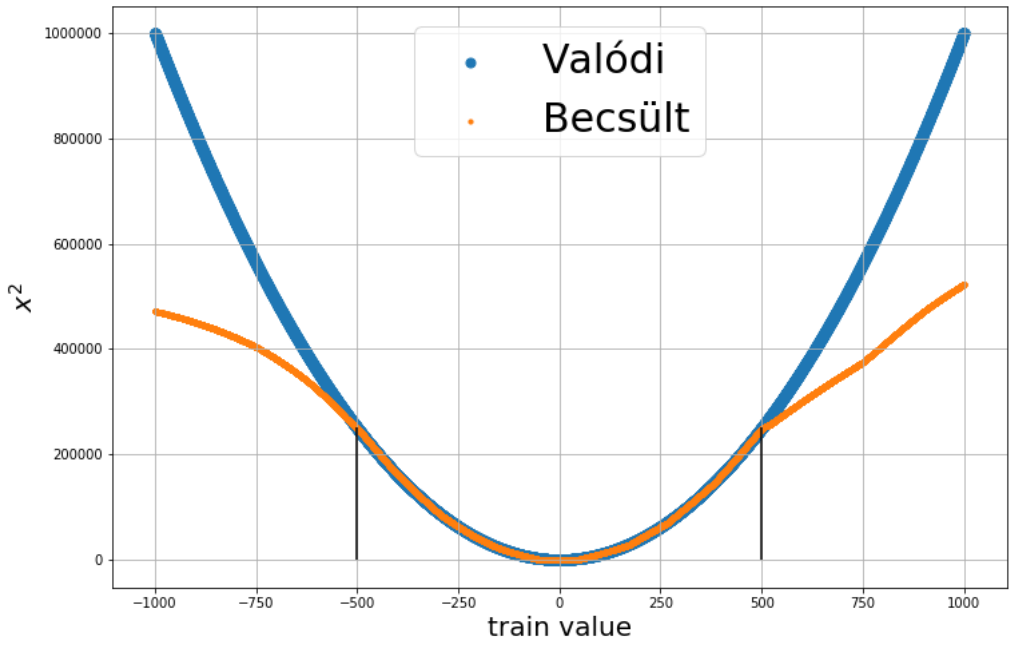


2. ábra. A nulla körüli vízsszintes vonal



3. ábra. Veszteségértékek a négyzetre emelés tanítása alatt

gyengének mondhatók, mivel amint kilépünk a tanító halmaz tartományából a becslések egyre inkább eltérnek a valódi értékektől. Ez azt jelenti, hogy a háló nem tudta álta-



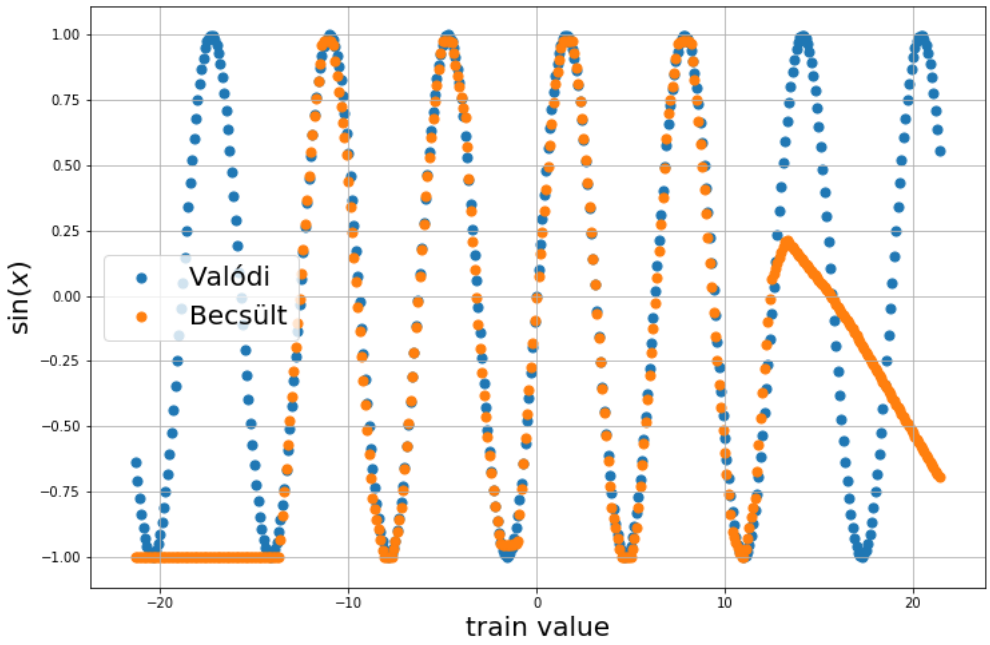
4. ábra. A négyzetre emelésre tanított háló a teljes teszthalmazon kiértékelve. Az ábrán a függőleges vonalak jelzik a tanítási tartomány határait

lánosítani a leképezést, mégis képes volt a tanító halmazon jó eredményeket elérni. Az implementációm biztosan tudna ennél jobban teljesíteni néhány változtatással és trükkel, mint például előnyösebb random inicializálás és regularizációs módszerek használata.

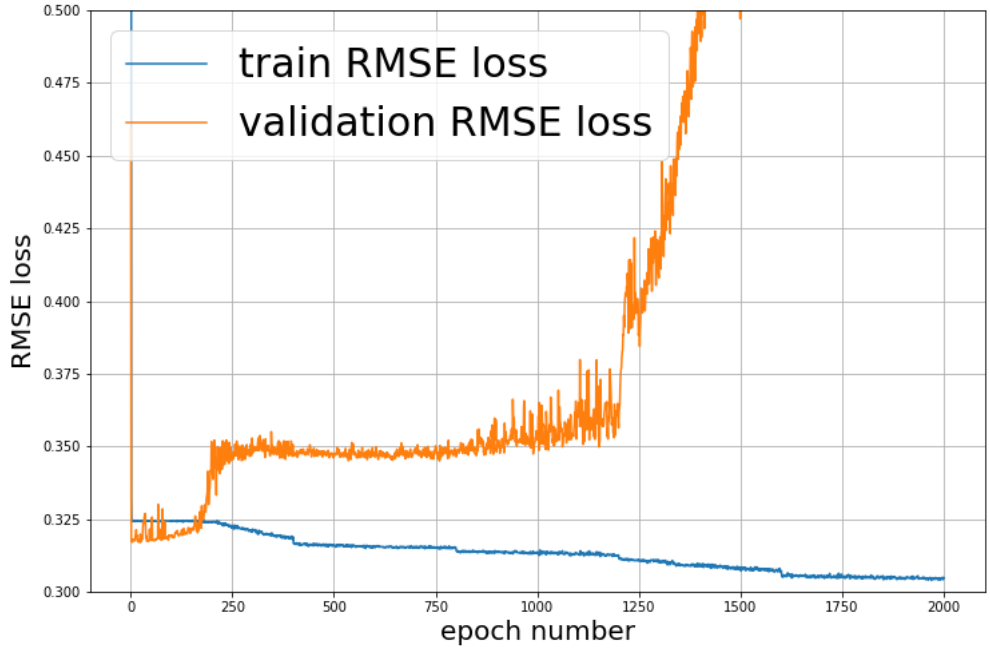
3.3. $\sin(x)$

A következő vizsgált leképezés a szinusz függvény volt, melyet a $[-12.8, 12.8]$ tartományon mintavételeztem a tanításhoz 0.05 felbontással. A validációhoz és a teszteléshez ugyanazt az adathalmazt, nevezetesen a $[-21.4, 21.4]$ tartományon 0.1 felbontással mintavételezett pontokat használtam. Az eredményeket ezen a tartományon összehasonlítva a valódi értékekkel mutatja az 5 ábra.

Ezen tanítás alatt is monitoroztam, majd kiírattam a tanító és validáló halmazon számolt veszteségértékeket, amiket a 6. ábra mutat. Látható, hogy ebben az esetben is túl sokáig tanítottam a hálót, mivel a validéciós veszteségérték hirtelen nagymértékben megnőtt. Érdekes jelenség látható azonban az első epoch-oknál; a validációs veszteségértékek kisebbek, mint a tanító veszteségértékek. Ennek két oka lehet, vagy olyan jó kiinduló becslése volt a hálónak, amely a validációs halmazon összességében kisebb hibát eredményezett, vagy nagyon gyorsan tanult a háló és a tanító veszteségértékek, amelyek epoch közben voltak kiszámítva még nagyobbak voltak, mint az epoch végére teljesen frissített



5. ábra. A szinusz függvény tanítása. A háló extrapolációs képessége ebben az esetben is nagyon gyenge.



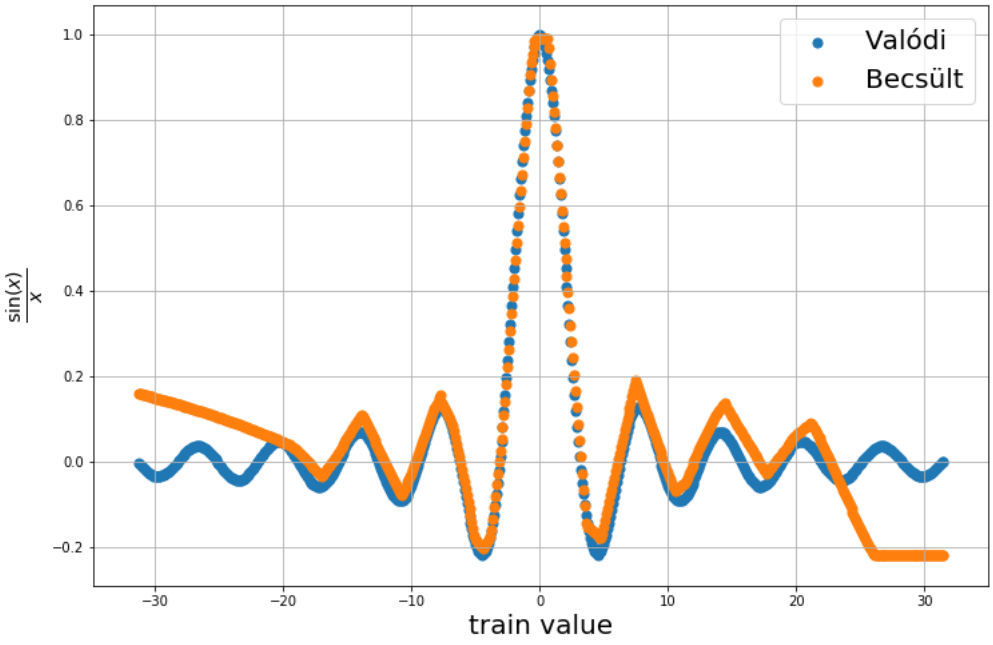
6. ábra. Veszteségértékek a szinusz függvény tanítása alatt

hálóval a validációs halmazon vett veszteségértékek. Elnézve azonban a veszteségértékek változásának mértékét valószínűleg az előbbi esetről van szó.

Itt a végén megjegyezném, hogy a szinusz függvényt tágabb tartományokon egyre nehezebbnek találtam betanítani. Ennek okát ott keresném, hogy nagyobb finomsággal kellett volna tanítani, azonban az az én esetemben több nehézséget is jelentett. Egyrészt olyan esetben nehezebb megtalálni az optimális tanulási rátát, valamint valószínűleg tovább kell tanítani egy mélyebb hálót, így nagy futási időt is igényelt volna.

3.4. $\frac{\sin(x)}{x}$

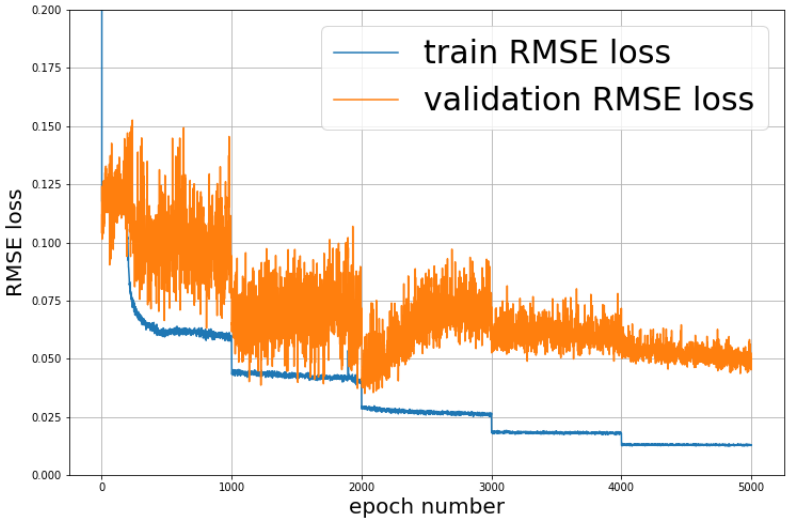
A harmadik leképezés, amit vizsgáltam és betanítottam a szintén nevezetes $\frac{\sin(x)}{x}$ függvény volt. Tanító halmazként a $[-16, 16]$ tartományt használtam, melyet 0.05 finomsággal osztottam fel. A validációs- és teszthalmaz a $[-31.4, 31.4]$ tartomány volt 0.1 finomsággal felbontva. Ebben az esetben eltértem a megszokottól és 5000 epoch-hal tanítottam be, míg a tanulási rátát 1000 epoch-onként csökkentettem. Ennek praktikus oka volt, meg akartam vizsgálni, hogy hogyan fog egy ilyen hosszú tanítás lezajlani egy kevésbé egyértelmű függvény esetében. A teszthalmazon történt kiértékelést mutatja a 7. ábra.



7. ábra. A $\sin(x)/x$ tanítása. A háló a tanítóhalmaz szélénél már instabil eredményeket ad, extrapolálni ez esetben sem képes.

A monitorozott tanító és validációs veszteségértékeket mutatja a 8. ábra.

Ezen rész végén azt jegyezném meg, hogy a szinusz függvényhez képest ezt egyszerűbb

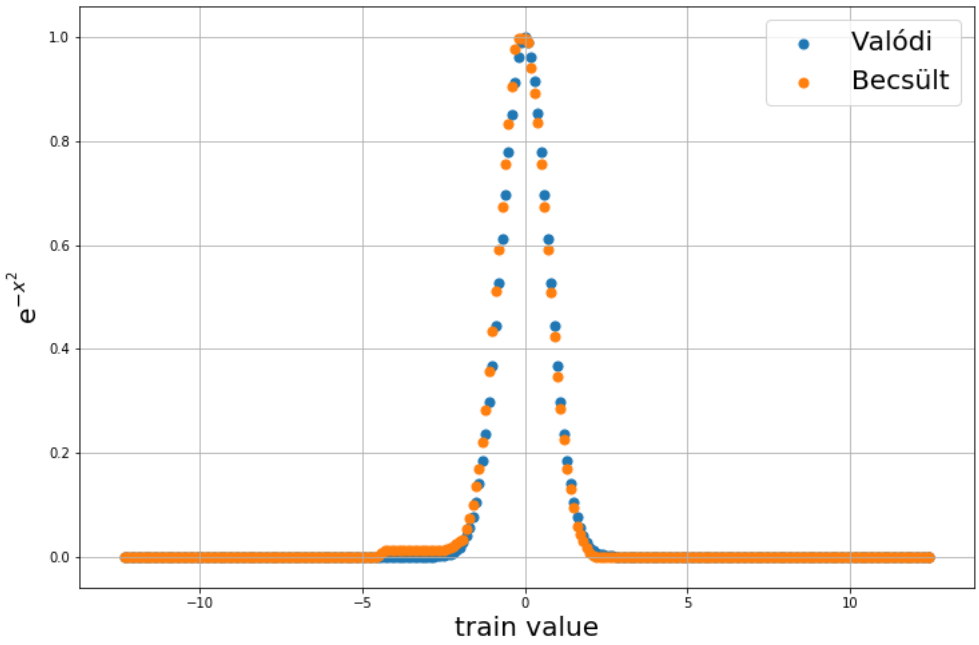


8. ábra. Veszteségértékek a $\sin(s)/x$ függvény tanítása alatt

volt tanítani ugyanolyan tág tartományon. Ennek az oka az lehet, hogy ez egy csökkenő amplitúdójú függvény, így a háló egyre könnyebben tud közel jó becslést adni.

3.5. e^{-x^2}

Az utolsó függvény, amit ennek a projektnek a keretében megpróbáltam betanítani a neurális hálónak, egy egyszerű Gauss-függvény volt, e^{-x^2}. Ebben az esetben ismét az előbbiekben ismertetett hiperparaméterekkel dolgoztam, tanító halmazként a $[-7, 7]$ tartományt használtam 0.01 finomsággal felosztva, a validációs-, valamint a teszt adathalmaz pedig -12.4-től 12.4-ig terjedt 0.1 finomsággal felbontva. A tanítás eredményét mutatja a 9. ábra. Ebben az esetben az megfigyelhető, hogy a becslések a tanítóhalmaz tartományán kívül sem pontatlanok, sőt, a háló egészen pontos becslést tud azzal adni, hogy nulla közeli értékeket becsül. Ez teszi így utólag talán meglepő módon - számomra legalábbis - ezt a leképezést a legkönnyebben tanulhatóvá. Érezhető is, hogy a hálónak gyakorlatilag csak egy szűk tartományon szükséges nem nullához közeli értékeket becsülnie, ez pedig viszonylag egyszerűen tanulható egy neurális hálózat számára. Természetesen, ha a nullához közeli értékekre is nagyon pontos becslést szeretnénk kapni, akkor nehezebb és körülményesebb lehet a tanítás, így igazándiból nem mondhatjuk ki, hogy jól extrapolál numerikusan a háló, csak a lineáris skálán kielégítő közelítést nyújt. Éppen ezért ábrázoltam a teszthalmaz eredményeit logaritmikus függőleges tengely használatával is, melyet a 10. ábra mutat. Látható, hogy ezen a skálán a becslések már közel sem egyeznek a valódi értékekkel, a nullától eltávolodva egy levágás látható, amely után a becslés mindig körülbelül ugyanaz a kicsi szám. Így megerősíthető, hogy numerikusan csakis nullához



9. ábra. A e^{-x^2} függvény tanítása. A háló képes megtanulni, hogy egy bizonyos távolságra a nullától nullához közeli értékeket érdemes becsülnie.

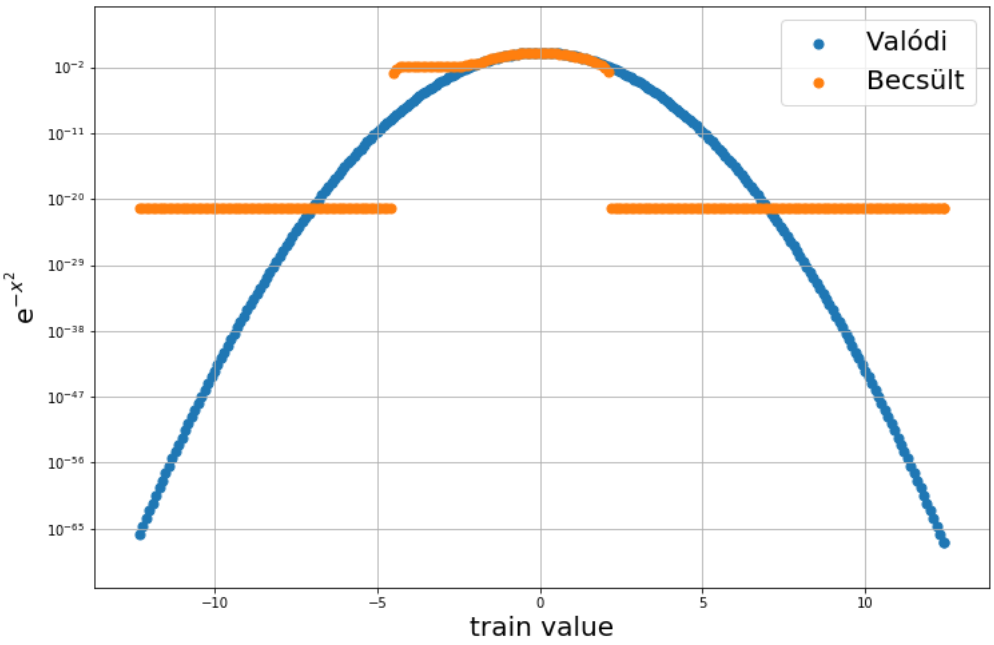
közeli bemenetekre pontosak a becslések.

Végül az ennél a tanításnál is monitorozott tanító és validációs veszteségértékeket mutatja a 11. ábra. Látható, hogy ebben a esetben a validációs veszteségértékek is folyamatosan csökkentek a tanulási rátával, azonban folyamatosan nagy oszcilláció fedezhető fel bennük.

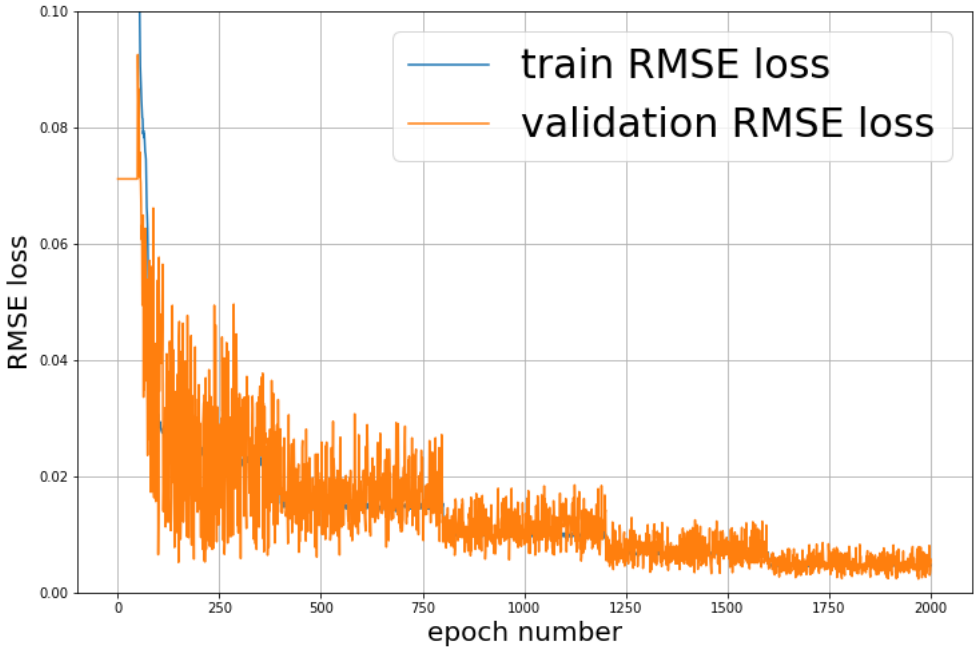
4. A zaj befolyásoló hatása

Érdekes vizsgálatok végezhetők, hogy a neurális háló hogyan képes zajos adatokon tanulni. Ettől előzetesen az várható, hogy mivel minden epoch-ban valamilyen eltérés keletkezik a bemenetekben, vagy a kimenetekben, ezért a háló képes lehet megragadni a leképezéseket egy általánosabb módon, így elősegítve, hogy ne csak a tanító halmazon adjon jó becsléseket a háló. Ilyen szempontból tehát a zaj egyfajta regularizációnak tekinthető.

Ehhez kétféle módszert implementáltam, így a standard normális eloszlású zajt mind a bemenetekhez, mind pedig a valódi kimenetekhez hozzá lehet adni. Az első kísérletben a négyzetre emelés tanítását végeztem el ugyanazon a tartományon, mint amin eredetileg is tanítottam ($[-500, 500]$ tartományban az egész számok), azonban a $[0, 1]$ tartományra átskálázott bemenetekhez egy 0 várható értékű és 0.001 szórású zajt adtam. A szórás

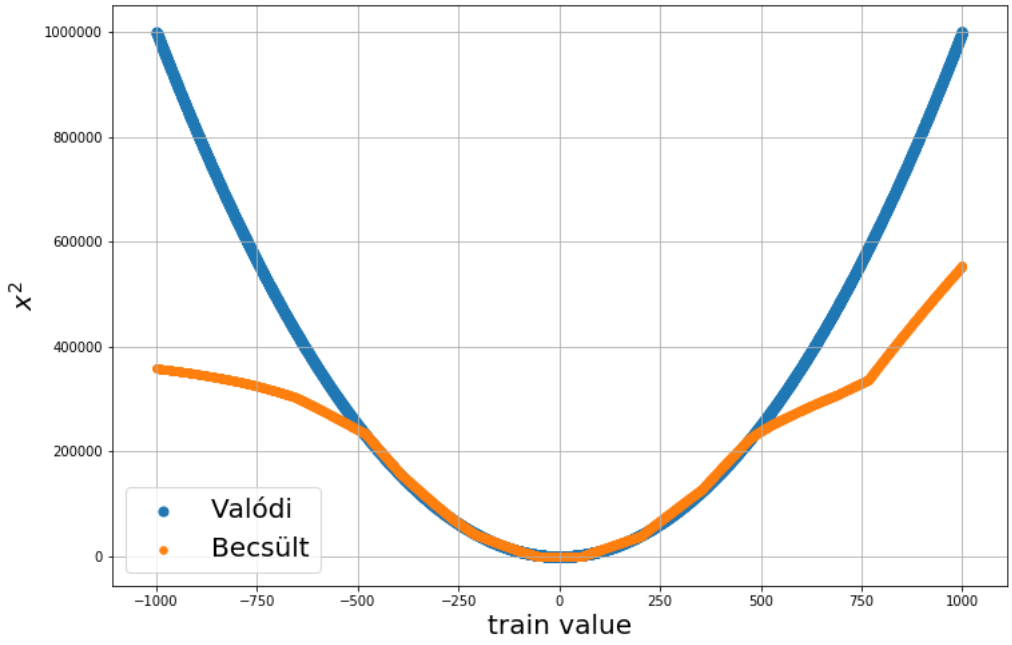


10. ábra. Látható, hogy ezen a skálán a becslések már közel sem egyeznek a valódi értékekkel a nullától eltávolodva, egy levágás látható, amely után a becslés mindig körülbelül ugyanaz a kicsi szám



11. ábra. Veszteségértékek a e^{-x^2} függvény tanítása alatt

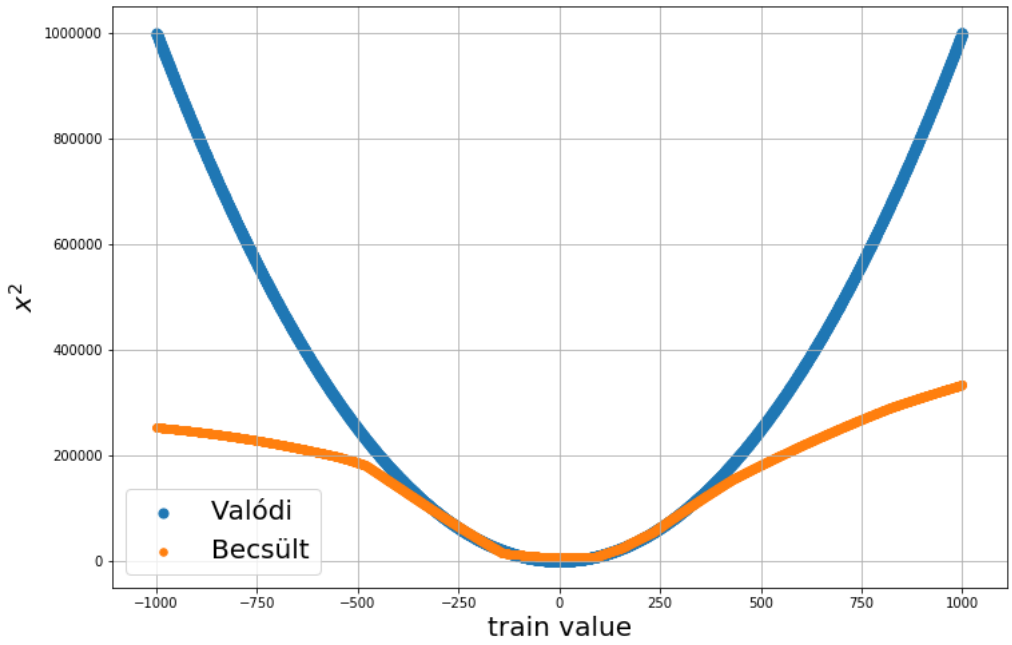
értéke ekkor az eredeti tartmányon 1-nek felelt meg. Az epoch-ok számát továbbá 500-ra állítottam, míg a tanulási rátát 100 epoch-onként csökkentettem. Az eredményeket mutatja a 12. ábra. Látható, hogy nagy változás nem történt a zaj nélküli tanításhoz képest,



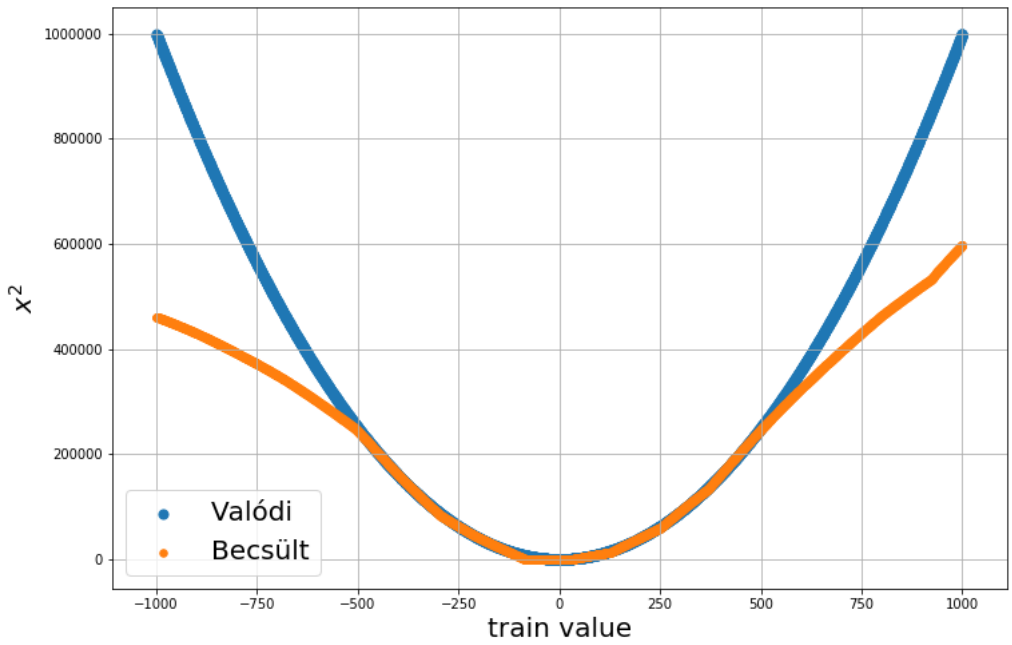
12. ábra. Zajos bemenetekkel tanított négyzetre emelés, 0.001 szórással

a 0 körüli viselkedés is változatlan maradt. Ezért tehét megpróbélkoztam egy nagyobb szórásértékkel, 0.005-tel. Továbbá a tanulási ráta változtatását is átállítottam úgy, hogy 50 epoch-onként csökkent az 500 epoch tanítás alatt. Az eredményt a 13. ábra mutatja. Látható, hogy ezen az ábrám sem látható lényeges eltérés, hacsak az nem, hogy a nagyobb szórású hiba miatt a tanító halmaz széleinél a becslések már pontatlanok, valamint a 0 körüli hiba is nagyobbnak látszik. Ezzel tehát kimondhattam, hogy az elvégzett tesztek nem segítettek a háló tanításában, inkább rontottak rajta. Természetesen alaposabb és részletesebb vizsgálat lenne szükséges, hogy az előzetes elvárás alapján a regularizáció segítő hatását látni lehessen.

Azonban elvégeztem egy olyan tesztet is, amelyben a kimenetre adtam zajt. A kimenet tartománya a bemenet tartományának megfelelően $[0, 250000]$ volt, így a zaj szórását 0.0004-re állítottam (a várható érték nulla maradt), ami a kimenetek nagyságrendjében 100-nak felelt meg. Itt is 50 epoch-onként csökkentettem a tanulási rátát. Az eredményt a 14. ábra mutatja. Látható, hogy az extrapolációs képesség még mindig nem jó, de javult a zajmentes tanításhoz képest, a 0 körüli becslések továbbra sem javultak, azonban érdekes módon a vízszintes vonal a negatív értékek felé eltolódott és meg is nyúlt. Ezt mutatja a 15. ábra.

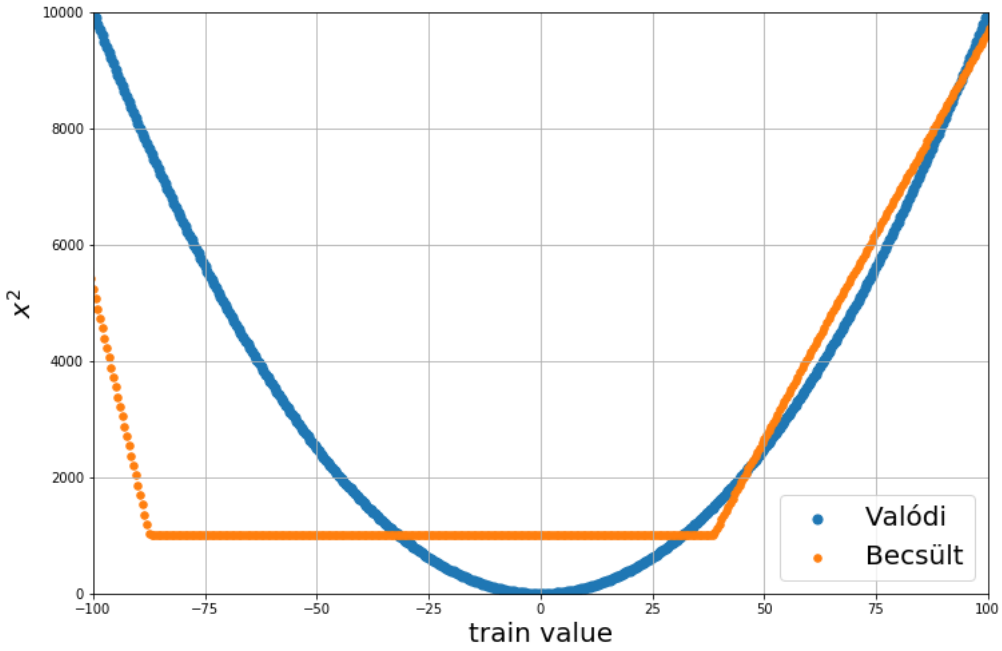


13. ábra. Zajos bemenetekkel tanított négyzetre emelés, 0.005 szórással



14. ábra. Zajos valódi értékekkel tanított négyzetre emelés, 0.0004 szórással

Összességében azt gondolom, hogy egy elegendően körültekintő és alaposan átgondolt módon történő zajos tanítás segíteni képes a háló tanításában egyfajta regularizációként,



15. ábra. A nulla körüli vízszintes vonal zajos valódi értékekkel való tanítás után

azonban az általam elvégzett néhány kísérlet és próbálkozás nem járt nagy sikerrel.

5. Kitekintés

A neurális hálók tanítása nagyon sok elemből áll és az én implementációmba is nagyon sok nem került be. Az egyik legfontosabb hiánynak az egyes batch-ek tanításának vektorosítását érzem, mivel az jelentősen felgyorsíthatja a tanulás folyamatát. Ezen kívül lehetne alkalmazni például további regularizációs eljárásokat (L1, L2), másfajta veszteségfüggvényt, valamint lehetne különböző előnyösebb eloszlásokból inicializálni a háló súlyait. Egy másik elem, amely jelentősen megnövelheti a teljesítményt, az a tanulási algoritmus, azon belül is az optimalizáló eljárás fejlesztése, amely a gradiensek alapján lépteti a háló súlyait. A gradiens ereszkedésre jónéhány jóval erősebb optimalizáló épül, melyek közül is talán a legelterjedtebb az Adam [2].

Végül tehát látható, hogy rengeteg helyen lehet még javítani és fejleszteni a kódot, azonban a Korszerű számítástechnikai módszerek a fizikában 1. kurzus projektjének megvalósításába számomra ennyi fért bele. Úgy hiszem, hogy a demonstráció lényegi része sikerült azzal, hogy különféle függvényalakokat láthatóan képes volt a háló közelíteni.

Hivatkozások

[1] *An overview of gradient descent optimization algorithms.* `https://ruder.io/optimizing-gradient-descent/index.html`. (Accessed on 05/07/2021).

[2] Diederik P Kingma és Jimmy Ba. "Adam: A method for stochastic optimization". *arXiv preprint arXiv:1412.6980* (2014).

[3] David E Rumelhart, Geoffrey E Hinton és Ronald J Williams. *Learning internal representations by error propagation.* Techn. jel. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.