Eötvös Loránd Tudományegyetem

Természettudományi Kar

Alkalmazott Analízis és Számításmatematikai Tanszék

Differenciálegyenletek megoldása neurális hálókkal

Témavezető:
Dr. Csomós Petra
adjunktus

Szerző:
Pozsár Attila
Matematika BSc hallgató

Budapest, 2023

NYILATKOZAT

Név: Pozsár Attila

ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: A2GB7I

Szakdolgozat címe:
Differenciálegyenletek megoldása neurális hálókkal

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023.06.06.

Pozsár Attila

a hallgató aláírása

Tartalomjegyzék

Bevezetés

Az elmúlt években a mesterséges neurális hálók jelentős fejlődésen mentek keresztül, alkalmazásuk mára számos területen széles körben elterjedt. A szakdolgozat célja, hogy bemutassa a neurális hálózatok általános működését, és felhasználását egyszerű differenciálegyenletek megoldására.

Az első fejezetben rövid áttekintést adunk az agyi neuronok, valamint a belőlük felépülő idegrendszer felépítéséről és működéséről, az [1, 2, 3] források felhasználásával.

A második fejezetben a [4] forrás alapján a mesterséges neurális hálózatok modellezését mutatjuk be. Részletesen tárgyaljuk a neurális hálózatok viselkedését alapvetően meghatározó aktivációs függvényeket és veszteségfüggvényeket. Az [5, 6, 7] források ismeretében bemutatjuk a leggyakrabban használt aktivációs függvényeket, elemezzük előnyeiket és hátrányaikat. Emellett felhasználva a [8, 9, 10, 11] forrásokat, személetesen bemutatjuk a különböző veszteségfüggvényeket, melyek a hálózat kimenetének hibáját hivatottak mérni.

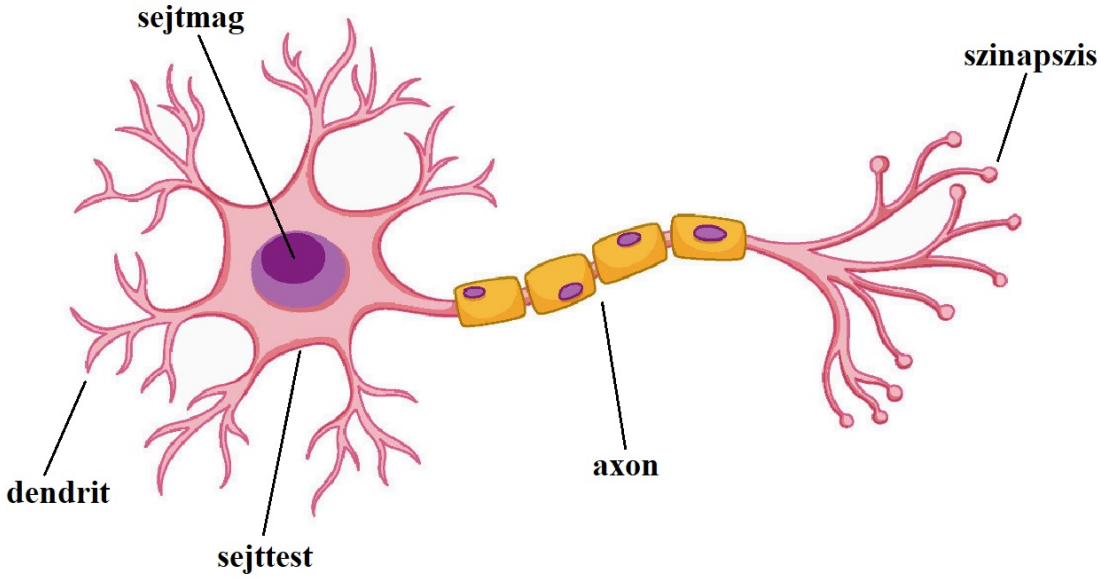
A hálózati modell felépítése után a [12, 13, 14, 15, 16] források alapján a harmadik fejezetben ismertetjük a működést, tárgyaljuk azokat a folyamatokat, amelyek a rendszer kimenetének előállításához, valamint a hibájának visszaterjesztéséhez szükségesek. Ebben a fejezetben egy egyszerű példán keresztül bemutatva átfogó képet kap az olvasó arról, hogy miként működik egy mesterséges neurális hálózat tanulási folyamata és végső kimenetének előállítása.

A szakdolgozat negyedik fejezete a [17, 18] források alapján közönséges differenciálegyenletek megoldását mutatja be Python programozási nyelv használatával. Adunk két konkrét példát, ahol a felhasznált modell hatékonyan és pontosan oldja meg a differenciálegyenletet.

1. fejezet

Biológiai neurális hálózatok

Az emberi idegrendszer irányítóközpontja az agy, melynek feldolgozó egységei az idegsejtek, az ún. neuronok. A neuronok ingerlékeny sejtek, fő alkotórészeik a drendit, a sejttest és az axon. A drenditek az idegsejtek „bemenetei", a rajtuk keresztül érkező ingert a sejttest és az abba ágyazódó sejtmag dolgozzák fel. Az idegsejtek „kimenetei" az axonok, amik a sejttestből indulnak és más sejtekhez csatlakoznak. Ezek a csatlakozások szinapszisok révén valósulnak meg a neuronok között. A neuronokat és az őket összekötő szinapszisok együttesét hívjuk neurális hálózatnak. Egy neuron vázlatos képét mutatja az 1.1 ábra.



1.1. ábra. Egy neuron vázlatos képe

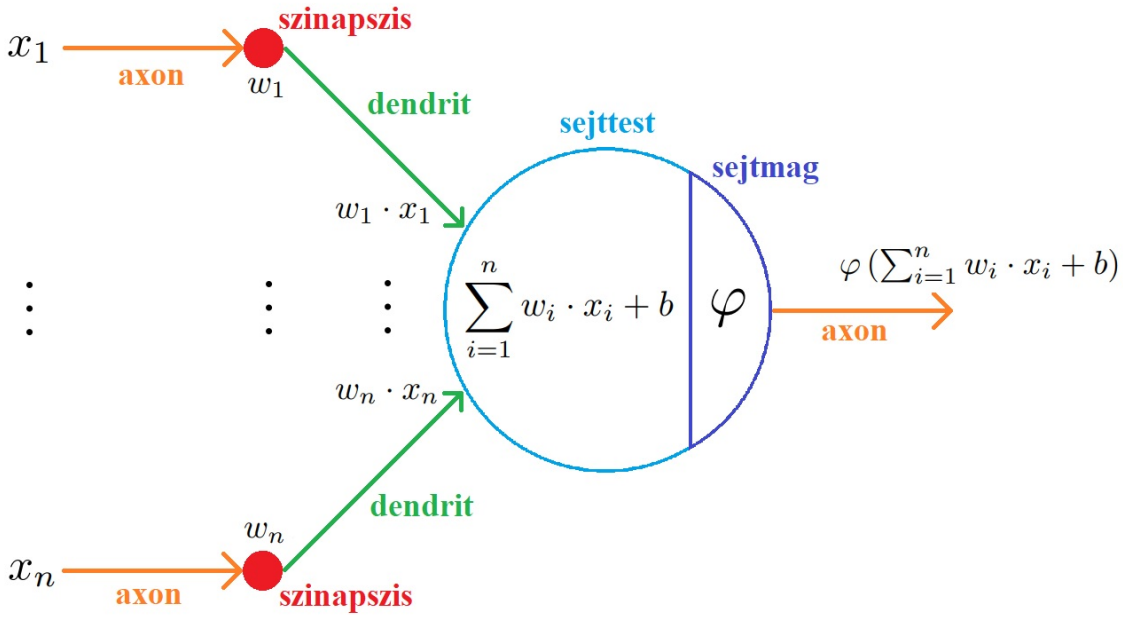
Ha egy neuron ingert kap, aktiválódik és továbbítja az ingert. Ekkor a kibocsátott

információ (inger) hatására a szinapszis másik oldalán található neuron is aktiválódik, és így tovább. Az agyban lévő neurális hálózatok képesek a tapasztalatok és élmények alapján változni. Az agy folyamatosan tanul és alkalmazkodik az új helyzetekhez, így a neuronok közötti kapcsolatok erősödhetnek vagy gyengülhetnek az ismételt aktivációk hatására. Az agy neurális hálózataiban számos szinten történik az információ feldolgozása. Az alsóbb szintek a bejövő információk feldolgozásával foglalkoznak, míg a magasabb szintek az összetettebb észlelések és gondolkodási folyamatokért felelősek.

2. fejezet

Mesterséges neurális hálózatok modellezése

A mesterséges neurális hálózatok gráf alapú modellek, melyek a biológiai idegrendszer működéséből inspirálódtak, és az emberi agy működésének mintájára épülnek fel. A mesterséges neuronok a mesterséges neurális hálózat elemi számítási egységei, hasonlóan az idegsejtekhez az emberi agyban. A mesterséges neurális hálózat modellezése a következő:

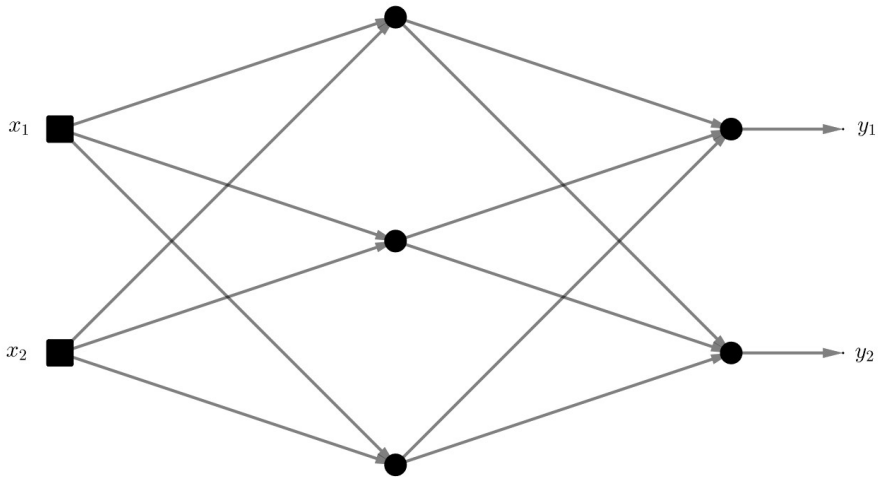


2.1. ábra. Egy neuron modellje

- axon $\longrightarrow$ neuron kimeneti éle: továbbítja a neuron kimenetét a következő neuron bemenete felé

- szinapszis $\longrightarrow$ w élsúly: két szomszédos neuron kapcsolatának erőssége
- dendrit $\longrightarrow$ neuron bemeneti éle: előtte lévő szomszédos neuron kimeneti értéke a szinapszisukkal súlyozva
- sejttest $\longrightarrow$ csomópont: összegzi a kapott bemeneti értékeket és hozzáad egy, az adott neuronhoz tartozó b számot
- sejtmag $\longrightarrow$ aktivációs függvény: előállítja a neuron kimeneti értékét a sejttest által meghatározott értékre

Egy neuronhálózat modelljének megtervezése a megoldani kívánt feladat típusától és bonyolultságától függ. Az agyban található ideghálózat rettenetesen bonyolult, a modellezés során törekedni kell egy egyszerűbb megvalósíthatóságra, amely könnyen menedzselhető és programozás során karbantartható. Így a hálózati modellben rendezett rétegekbe rendezzük a neuronokat: az első réteg neuronjai kapnak egy $\mathbf{x} \in \mathbb{R}^n$ bemeneti vektort, végül a kimeneti réteg egy $\mathbf{y} \in \mathbb{R}^m$ vektort ad eredményül. A köztes rétegek, illetve a bennük elhelyezkedő neuronok számának eldöntése nagyon nehéz feladat. Az optimális réteg- és neuronszám megtalálása sok tapasztalatot igényel és gyakran csak empirikus úton lehet megtalálni. Általánosan elmondható, hogy a hálózat rétegeinek és neuronszámának növelésével komplexebb problémák oldhatóak meg hatékonyan. A szomszédos rétegeket balról jobbra irányított élekkel kötjük össze úgy, hogy minden bal oldali neuron kimenetét minden jobb oldali neuron megkapja bemenetként. A 2.2. ábra egy két darab bemeneti értékre két kimeneti adatot adó modellt mutat be, egy köztes rétegen belül három neuronnal:



2.2. ábra. Egy mesterséges neurális hálózat vázlata

A hálózati paraméterek, vagyis a $w \in \mathbb{R}$ élsúlyok és a $b \in \mathbb{R}$ eltolások kezdeti értékeinek definiálására több elterjedt szokás létezik. Az élsúlyok megadása leggyakrabban random számok generálásával történik, vagy esetleg standard normális eloszlásból vett értékekkel. A b számok kezdőértékét sokszor nullára állítják. A modell a tanulás során ezeket a hálózati paramétereket változtatja majd annak érdekében, hogy megfelelő előrejelzéseket adjon. Ahhoz, hogy mérni lehessen a modell kimenetének hibáját, definiálni kell egy hibafüggvényt. Nagyon fontos szerepet játszanak továbbá a φ aktivációs függvények, hiszen meghatározzák az egyes neuronok, és velük együtt az egész réteg kimenetét. Feladattól függően, a különböző rétegek neuronjaihoz rendelhetőek különböző aktivációs függvények.

A következő alfejezetekben az aktivációs- és a veszteségfüggvényeket mutatjuk be.

2.1. Aktivációs függvények

A mesterséges neurális hálózatok aktivációs függvényei azok a matematikai függvények, amelyek meghatározzák az egyes neuronok aktiválódásának mértékét a hálózatban. Hasznuk abban rejlik, hogy korlátozzák a neuronok kimenetének amplitúdóját, ami a hálózat stabilitásához szükséges. Ugyanis ha az aktivációs függvény nem korlátozná az egyes neuronok kimenetét, akkor az értékük „túl nagy” is lehetne. Ez azért jelent problémát, mert csupán egy bementi jel is hatalmas változásokat okozhatna a hálózat paramétereiben, és a tanulás hatékonyságát veszélyeztetné. Alkalmazástól függően persze használhatóak nem korlátos függvények is. A hálózat tanulásához szükséges gradiens módszer miatt ajánlott a deriválható aktivációs függvények használata. Sok modellhez viszont nagyon gyorsan számolható, egyszerű aktivációs függvényeket használnak, ahol valamilyen úton biztosított, hogy ne tegyenek kísérletet egy nemlétező derivált kiszámítására. Alább bemutatunk pár gyakori aktivációs függvényt.

2.1.1. Identitás függvény

$\varphi_{\text{Identitás}} \colon \mathbb{R} \to \mathbb{R}$

$$\varphi_{\text{Identitás}}(x) = x$$

$$\varphi'_{\text{Identitás}}(x) = 1$$

Az identitás a legegyszerűbb aktivációs függvény. Nem korlátos, így a kimeneti értéket sem korlátozza. Egyszerűsége miatt nagyobb neurális hálózatokban kevésbé használják, de praktikus lehet, ha fontos a be- és kimeneti adatok közötti lineáris összefüggés.

2.1.2. ReLU függvény

$\varphi_{\text{ReLU}} \colon \mathbb{R} \to \mathbb{R}$

$$\varphi_{\text{ReLU}}(x) = \max\{0, x\} = \begin{cases} x, & \text{ha } x > 0 \\ 0, & \text{különben} \end{cases}$$

$$\varphi'_{\text{ReLU}}(x) = \begin{cases} 1, & \text{ha } x > 0 \\ 0, & \text{ha } x < 0 \\ \text{nem létezik}, & \text{ha } x = 0 \end{cases}$$

A ReLU aktivációs függvény hatékony a képfeldolgozási problémák megoldásában. Amikor egy adott típusú jellemzőt tanul a hálózat, a hozzá tartozó rész a modellben aktiválódik, míg más részek nem, így ezekre nem lesz hatással. Tehát ha a hálózat megtanul egy adott objektumot felismerni, utána ugyanezt a hálózatot megtaníthatjuk más objektumok detektálására is. Nagyobb hálózatokban is hatékonyan alkalmazható gyorsasága miatt, mivel a függvényérték egyszerűen számolható. Ha ezt az aktivációs függvényt használjuk, akkor a teljes hálózati modellben csupán számok összehasonlítására, szorzására és összeadására van szükség.

Nagy hátránya azonban a „halott neuron” problémája. Ha az a jelenség lépne fel, hogy egy neuron mindig negatív értékeket kap, akkor az eredmény mindig nulla lenne. Ez azt jelenti, hogy a neuron nem tud tovább tanulni, mert a gradiens nullára csökken, és a továbbiakban nem változtatja meg a paramétereit.

2.1.3. Szivárgó ReLU függvény

$\varphi \colon \mathbb{R} \to \mathbb{R}$, $\lambda \in \mathbb{R}$

$$\varphi_{\text{Szivárgó ReLU}}(x) = \max\{\lambda \cdot x, x\} = \begin{cases} x, & \text{ha } x > 0 \\ \lambda \cdot x, & \text{különben} \end{cases}$$

$$\varphi'_{\text{Szivárgó ReLU}}(x) = \begin{cases} 1, & \text{ha } x > 0 \\ \lambda, & \text{ha } x < 0 \\ \text{nem létezik}, & \text{ha } x = 0 \end{cases}$$

A ReLU függvény hibáját kompenzálja, a neuronok örökös inaktív állapotba kerülésének lehetőségét küszöböli ki. A szivárgó ReLU függvény ezt a problémát úgy oldja meg, hogy a negatív bemeneteket kicsi pozitív, jellemzően egy $\lambda = 0{,}01$ értékekkel szorozza meg, így a neuronok a negatív bemenetekre is képesek reagálni.

Habár könnyen számolható, kicsit összetettebb függvény, mint az egyszerű ReLU, így számításigényesebb. Mivel a negatív bemenetekre is aktiválódik a függvény, így nem tud olyan hatékonyan tanulni, mint elődje.

2.1.4. Szoftplusz függvény

$\varphi_{\text{Szoftplusz}} \colon \mathbb{R} \to (0, +\infty)$

$$\varphi_{\text{Szoftplusz}}(x) = \ln(1 + e^x)$$

$$\varphi'_{\text{Szoftplusz}}(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

A ReLU aktivációs függvény $x = 0$ körüli simítása. A két függvény nagyrészt hasonló, de a Szoftplusz a ReLU függvénnyel ellentétben differenciálható a nullában. Kiszámítása időigényesebb, mint a ReLU függvényé.

2.1.5. Logisztikus függvény (Szigmoid)

$\varphi_{\text{Logisztikus}} : \mathbb{R} \to (0, 1)$

$$\varphi_{\text{Logisztikus}}(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x} = 1 - \varphi_{\text{Logisztikus}}(-x)$$

$$\varphi'_{\text{Logisztikus}}(x) = \frac{e^x}{1 + e^x} - \left(\frac{e^x}{1 + e^x}\right)^2 = \varphi_{\text{Logisztikus}}(x) \cdot (1 - \varphi_{\text{Logisztikus}}(x))$$

Tulajdonságaiból adódóan ez a függvény neurális hálózatokban jól alkalmazható 0 és 1 közötti valószínűségek előállítására. Így főleg bináris osztályozási feladatokra használják, amikor a kimenetben egy *igen* vagy egy *nem* döntést kell meghozni.

Hátránya, hogy ha a bemenet nagyon kicsi vagy nagyon nagy, akkor a gradiens nagyon lapos lesz és nehézkessé válik a tanulás. Ez a jelenség az angol irodalom-

ban „vanishing gradient - eltűnő gradiens” problémaként szerepel. Emiatt nagyobb méretű hálózatokban nem javasolt a használata.

Megjegyzés. Habár a szakirodalomban a Logisztikus aktivációs függvényt inkább Szigmoidnak nevezik, a „szigmoid” jelzőt az ‚S’ alakú görbével rendelkező függvényekre használják. Ezek a függvények hasonló viselkedést mutatnak az értéktartományuk két végén, azaz az alacsony és a magas értékeknél. Három szakaszuk van: egy monoton lassan növekedő szakasz, majd egy hirtelen változású, a függvény egyetlen inflexiós pontját tartalmazó monoton növő rész, végül ismét egy monoton lassan növő szakasz. Egy szigmoid függvényt két vízszintes aszimptota korlátozza.

2.1.6. Tangens hiperbolikus függvény

$\varphi : \mathbb{R} \to (-1, 1)$

$$\varphi_{\text{Tangens hiperbolikus}}(x) = \text{th}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$

$$\varphi'_{\text{Tangens hiperbolikus}}(x) = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2 = 1 - \varphi^2_{\text{Tangens hiperbolikus}}(x)$$

Hasonlóan a logisztikus függvényhez, többnyire döntési feladatok megoldására használják. Páratlan függvény, tehát $\varphi_{\text{Tangens hiperbolikus}}(x) = -\varphi_{\text{Tangens hiperbolikus}}(-x)$, így a pozitív bemenet mellett a negatív értékeket is ugyanúgy képes osztályozni, és a nulla közeli bemenetekez nulla közeli függvényérték fog tartozni.

2.1.7. Szoftmax függvény

$\varphi_{\text{Szoftmax}} : \mathbb{R}^n \to (0, 1]^n$, ahol $n \geq 1$

$$\varphi_{\text{Szoftmax}}(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{n} e^{\mathbf{x}_i}}; \qquad i = 1, \ldots, n; \qquad \mathbf{x} = [x_1, \ldots, x_n] \in \mathbb{R}^n$$

A szoftmax aktivációs függvényt általában akkor használják, ha a hálózatnak többkategóriás osztályozási problémát kell megoldania. A szigmoid függvény több dimenzióra vonatkozó általánosítása, a neurális hálózatok utolsó aktivációs függvényeként használják. Mivel a függvény eredménye egy olyan vektor, amelynek elemei 0 és 1 közötti értékek, valamint összegük 1-re normalizált, így a kimenet valószínűségeloszlásként értelmezhető. Tehát a szoftmax függvény az egyes kategóriákba való

tartozás valószínűségét adja meg, a maximum érték indexe pedig a legmagasabb valószínűségű kategória.

Például az $\mathbf{x} = [2; 3; 9; 5]$ bemenetre alkalmazva a szoftmax függvényt:

$$\varphi_{\text{Szoftmax}}([2; 3; 9; 5]) = \frac{[e^2; e^3; e^9; e^5]}{e^2 + e^3 + e^9 + e^5} \approx [0{,}001; 0{,}002; 0{,}979; 0{,}018].$$

Jól látható, hogy kiemeli a nagyobb értékeket és elnyomja azokat, melyek jelentősen elmaradnak a maximum értéktől. Vegyük észre azonban, hogy a 0 és 1 közötti értékek esetén a szoftmax függvény elnyomja a maximális értéket:

$$\varphi_{\text{Szoftmax}}([0{,}2; 0{,}3; 0{,}9; 0{,}5]) = \frac{[e^{0{,}2}; e^{0{,}3}; e^{0{,}9}; e^{0{,}5}]}{e^{0{,}2} + e^{0{,}3} + e^{0{,}9} + e^{0{,}5}} \approx [0{,}183; 0{,}202; 0{,}368; 0{,}247].$$

2.2. Veszteségfüggvények

Neurális hálózatokban a gradienssereszkedés megvalósíthatósága miatt a veszteségfüggvények olyan differenciálható függvények, amelyek egy skalárérték formájában meghatározzák, hogy mennyire pontosak a hálózat előrejelzései a célváltozó értékeivel összehasonlítva. A neurális hálózatok témakörében alapvetően két típusát különböztetjük meg a veszteségfüggvényeknek.

Regressziós típusú veszteségfüggvények: Regressziós feladatokat megoldó hálózatokban használják, tehát olyan problémák megoldásához, ahol az egymáshoz tartozó értékekre legjobban illeszkedő görbét keressük. Ehhez először definiálunk egy hibafüggvényt, amellyel meghatározzuk, hogy az illesztett görbe milyen mértékben tér el a ponthalmaztól, majd ezt az eltérést minimalizáljuk. Alább ismertetjük a négyzetes és az abszolút eltérést, valamint bemutatjuk a Pszeudo-Huber veszteségfüggvényt.

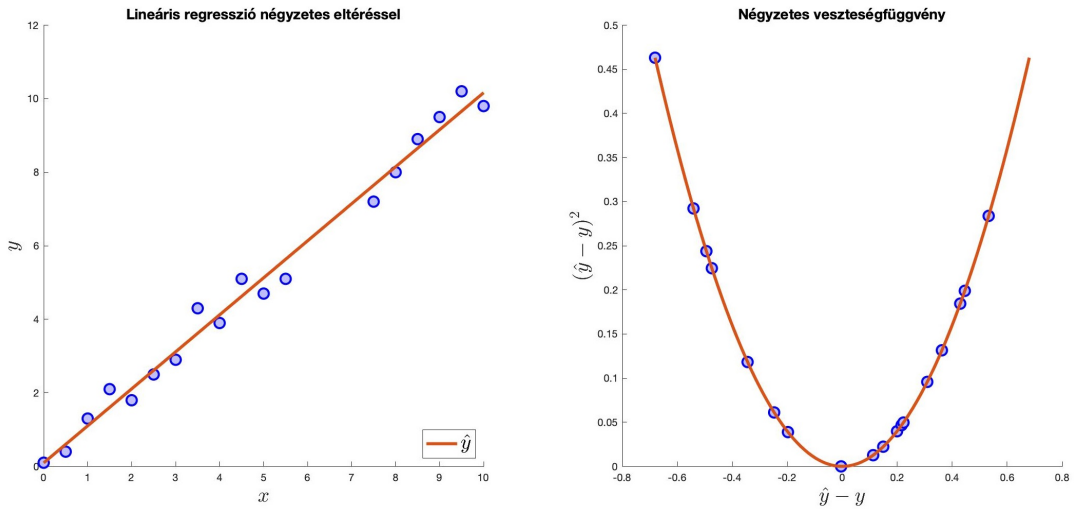
Kategorizáló típusú veszteségfüggvények: Kategorizáló neurális hálózatokban használják, ahol a hálózat egy adott bemenetre valószínűségi vektort állít elő. Ez a vektor tartalmazza a bemenet különböző osztályokba tartozásának valószínűségeit.

A veszteségfüggvények bemutatása során az y_i tényleges értékekhez tartozó predikciós értékeket $\hat{y}_i$ jelöli.

2.2.1. Négyzetes eltérés

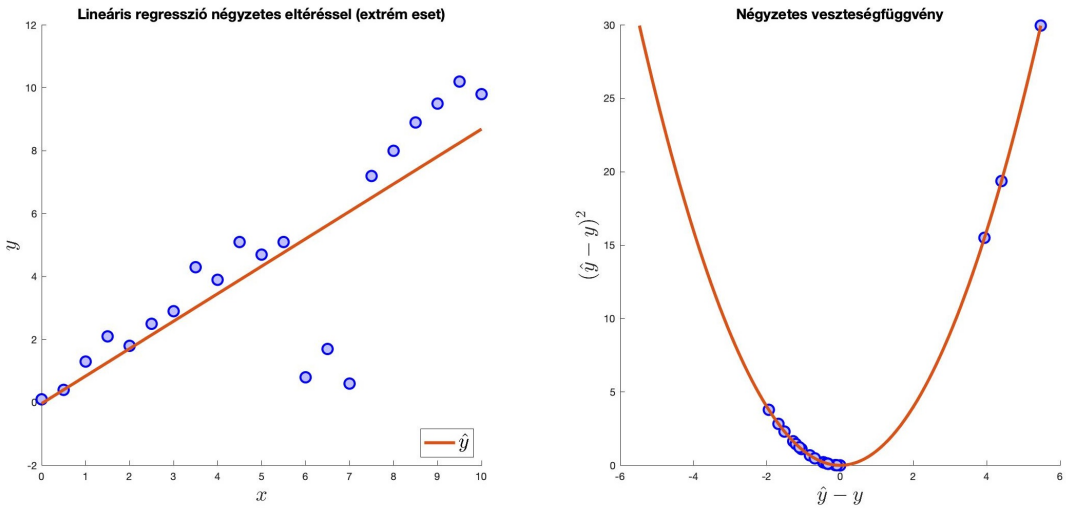
$$L_{\text{négyzetes}} = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Legáltalánosabb választás a legkisebb négyzetek módszere, amely az adathalmaz és a görbe közötti függőleges távolságok négyzetének összegét minimalizálja. Ilyen esetben tekintsünk a 2.3. ábrán egy lineáris regressziós modellt:



2.3. ábra. Lineáris regresszió négyzetes eltéréssel és a négyzetes veszteségfüggvény

Most úgy tűnik, szépen illeszkedik az illesztett regressziós egyenes az adathalmazra. Nézzük meg a 2.4. ábrán, mi történik, ha megjelenik néhány kiugró érték az adatok között:



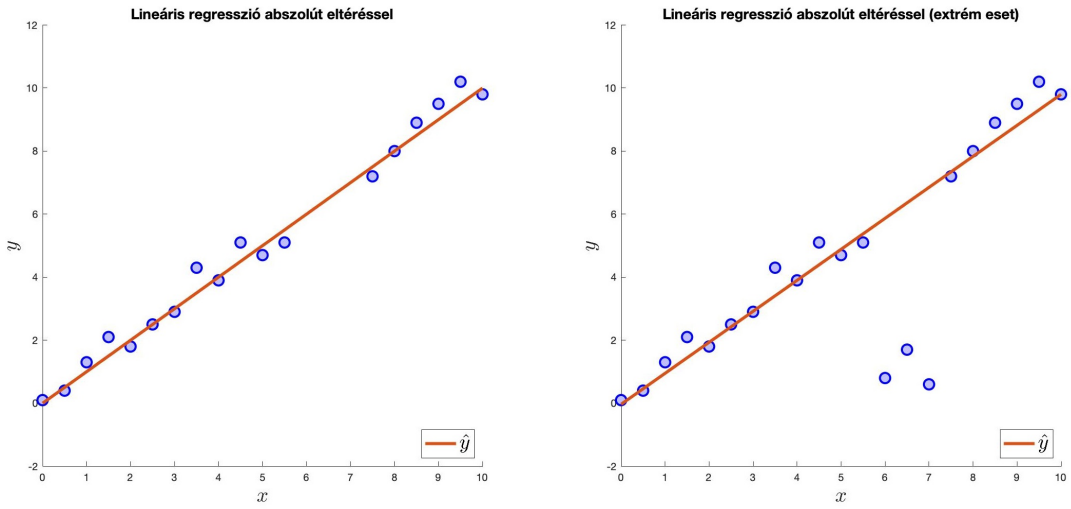
2.4. ábra. Lineáris regresszió négyzetes eltéréssel (extrém eset) és a négyzetes veszteségfüggvény

Látható, hogy ebben az esetben az egyenes nem illeszkedik túl jól az adathalmazra, pár új pont teljesen eltorzította a modellt. Ez azért van, mert a négyzetes hiba a kiugró értékeket fontosabbnak tartja kezelni, és emiatt őket próbálja meg a legjobban illeszteni. Ahhoz, hogy ezt elkerüljük, változtassuk meg a veszteségfüggvényt. Vizsgáljuk meg, hogyan viselkedik a modell, ha a hibák négyzete helyett az abszolútértéküket nézzük.

2.2.2. Abszolút eltérés

$$L_{\text{abszolút}} = \sum_{i=1}^{n} |\hat{y}_i - y_i|$$

A kiugró értékeket ugyanúgy kezeli, mint bármely más pontot, így nem tesz különbséget az adatok között. Az alábbi képen látható, hogy az extrém értékek nem okoznak nagy változást a modellben:



2.5. ábra. Lineáris regresszió abszolút eltéréssel normál és extrém esetben

Okozhat rossz előrejelzéseket, de ha az extrém eseteket figyelmen kívül szeretnénk hagyni, akkor ez a veszteségfüggvény megfelelő. Nagy hátránya viszont a matematikai analízisben való használhatósága, ugyanis mint tudjuk, az abszolútérték függvény nem differenciálható az $x = 0$ helyen. Ez azért probléma, mert az abszolút hiba nem optimalizálódik a gradiens csökkenésével: mivel a nullában veszi fel a minimumát és ott nem sima a függvény, így a gradiensmódszer nem fog jól konvergálni.

2.2.3. Pszeudo-Huber veszteség

Láttuk, hogy a négyzetes veszteség kiemeli az eltérő adatokat, míg az abszolút veszteség inkább figyelmen kívül hagyja azokat. Nagy adathalmazokhoz jó lenne egy olyan hibafüggvény, amely megfelelően figyelembe veszi a kiugró értékeket, de nem négyzetesen nagy súllyal. Az alapötlet az, hogy a kicsi, legfeljebb $\delta > 0$ abszolútértékű $x = \hat{y}_i - y_i$ hibákra alkalmazzuk a négyzetes hibafüggvényt, a nagyobbakra az abszolút veszteségfüggvényt. Az x^2 függvény δ helyen vett meredeksége $2 \cdot \delta$, tehát egy ilyen meredekségű abszolútérték függvényt kell illesztenünk hozzá. Ahhoz, hogy az $x = \delta$ helyen egyenlőek legyenek, az abszolút hibafüggvényt $-\delta^2$, értékkel kell eltolni a függőleges tengelyen, így a Huber veszteségfüggvény:

$$L_{Huber}(x) = \begin{cases} x^2, & \text{ha } |x| \leq \delta \\ 2 \cdot \delta \cdot |x| - \delta^2, & \text{különben} \end{cases}$$

A Huber hibafüggvény közelítéseként a Pszeudo-Huber veszteségfüggvényt alkalmazzák, ami a számítógépek számára egy könnyebben kezelhető kifejezés:

$$L_{Pszeudo-Huber}(x) = 2 \cdot \delta^2 \cdot \left(\sqrt{1 + \left(\frac{x}{\delta} \right)^2} - 1 \right)$$

Deriváltja könnyen kiszámítható és mindenhol létezik:

$$L'_{Pszeudo-Huber}(x) = \frac{2 \cdot x}{\sqrt{1 + \left(\frac{x}{\delta} \right)^2}}$$

A Pszeudo-Huber veszteségfüggvény kis x értékekre jól közelíti az x^2 értéket, míg nagyobb x-ekre a $2 \cdot \delta$ meredekségű abszolútérték függvény értékéhez tart. Fontos tudni, hogy más sima közelítések is léteznek a Huber veszteségfüggvényre.

2.2.4. Kereszt-entrópia (Logaritmikus veszteség)

$$L_{\text{logaritmikus}} = -\sum_{i=1}^{n} y_i \cdot \ln(\hat{y}_i); \qquad n: \text{ osztályok száma}$$

Ez a hibafüggvény klasszifikációs modellekben használatos, vagyis ha a bemenetet osztályozni szeretnénk valamilyen tulajdonság alapján. A kimeneti értékeket normáló Szoftmax aktivációs függvény mellett használatos, tehát $\hat{y}_i \in [0, 1]$. Mindig

arról kell döntést hozni, hogy egy adott osztálynak eleme-e az adott bemenet, tehát a kimeneti érték 1 (igen), vagy 0 (nem). Pontosabban, az osztályozás során a kimenet egy vektor, ami a bemenet különböző osztályokba tartozásának valószínűségeit tartalmazza, és a legmagasabb valószínűséghez tartozó osztály lesz a végleges kimenet.

Vegyünk példának egy olyan modellt, amely képeket kap bemenetként. Tudjuk, hogy az egyes ábrákon a következő gyümölcsök egyike szerepel: alma, narancs vagy kiwi. A bemenetre egy almáról készült kép érkezik, ekkor legyen a hálózat kimenete a Szoftmax aktivációs függvény után:

$$\hat{y} = [P(\text{alma}); P(\text{narancs}); P(\text{kiwi})] = [0{,}76; 0{,}18; 0{,}06].$$

Viszont tudjuk, hogy alma van a képen, tehát az $y = [1; 0; 0]$ eredményvektort várnánk. Ekkor a logaritmikus veszteség:

$$L_{\text{logaritmikus}} = -\sum_{i=1}^{3} y_i \cdot \ln(\hat{y}_i) = -(1 \cdot \ln(0{,}76) + 0 \cdot \ln(0{,}18) + 0 \cdot \ln(0{,}06)) \approx 0{,}2744.$$

Felmerülhet a kérdés, hogy a logaritmikus veszteség használata miért jó klasszifikációs problémák megoldására. A válasz abban rejlik, hogy az $\hat{y} \in [0, 1]^n$ intervallumon a valószínűségi értékeket logaritmikusan vesszük figyelembe. Ha az 1 valószínűségi értékhez tartozó $\hat{y}_i$ prediktív érték nagyon kicsi, vagyis közel van a 0-hoz, akkor a hibát egyre jobban „bünteti", mivel $\lim_{\hat{y}_i \to 0^+} -\ln(\hat{y}_i) = \infty$.

Fontos megjegyezni, hogy az $L_{\text{logaritmikus}}$ függvény meghívásakor a gyakorlatban az $\hat{y}$ vektor értékeihez egy kicsi, programozási nyelvtől függő, például 10^{-16} nagyságrendű ϵ számot adnak hozzá, hogy elkerüljék a 0 logaritmusának kiszámítását.

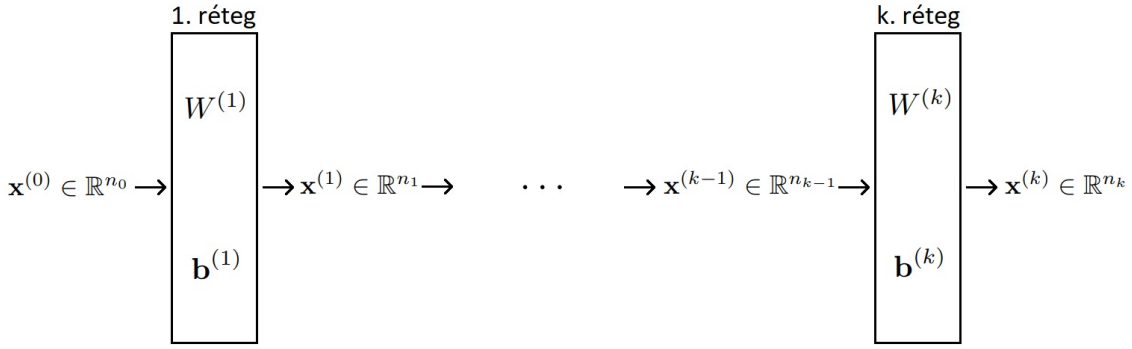
3. fejezet

Mesterséges neurális hálózatok működése

3.1. Kimenetképzés

Láttuk, hogyan néz ki egy mesterséges neurális hálózat modellje. Az alábbiakban megnézzük, miként áll elő egy ilyen modell kimenete. Tekintsünk egy k rétegű hálózatot, amelyben az i. réteg n_i darab, φ_i aktivációs függvénnyel rendelkező neuront tartalmaz, minden $1 \leq i \leq k$ egész számra.

Az i. réteg neuronjaihoz tartozó súlyértékeket a $W^{(i)} \in \mathbb{R}^{n_i \times n_{i-1}}$ súlymátrix, míg az eltoló értékeket a $\mathbf{b}^{(i)} \in \mathbb{R}^{n_i}$ eltoló vektor tartalmazza. Az i. réteg az $(i-1)$. bemenetre állítja elő az i. kimeneti vektort, amit a következő réteg kap meg bemenetként, és így tovább. A hálózat $\mathbf{x}^{(0)}$ bemenete (amit tehát egy 0. réteg kimenetének tekinthetünk) legyen n_0 méretű, vagyis $\mathbf{x}^{(0)} \in \mathbb{R}^{n_0}$. Ezt láthatjuk a 3.1. ábrán.



3.1. ábra. Neurális hálózat modellje

Az 1. réteg $\mathbf{x}^{(0)}$ bemenetre adott $\mathbf{x}^{(1)}$ kimenete:

$$\varphi_1(W^{(1)}\cdot\mathbf{x}^{(0)}+\mathbf{b}^{(1)}) = \varphi_1\left(\begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} & \dots & w_{1n_0}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} & \dots & w_{2n_0}^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n_11}^{(1)} & w_{n_12}^{(1)} & w_{n_13}^{(1)} & \dots & w_{n_1n_0}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n_0} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_1} \end{bmatrix}\right) = \mathbf{x}^{(1)}$$

A további rétegek kimeneteit hasonlóan sorban megkaphatjuk, a teljes hálózat $\mathbf{x}^{(0)}$ bemenetre adott predikciója az $\mathbf{x}^{(k)}$ vektor lesz.

3.2. Hibavisszaterjesztés

A visszaterjesztés egy olyan algoritmus a neurális hálózatok tanítására, amely visszaterjeszti a hálózat hibáját az összes rétegen keresztül, így a hálózat minden paraméterét módosítani tudja annak érdekében, hogy minimalizálja a modell kimenetének hibáját. Ennek köszönhetően a hálózat meg tudja tanulni a be- és kimenetek közötti összefüggéseket, így javítva az előrejelzési képességét. A folyamat megértéséhez először tekintsük át a témakörhöz kapcsolódó fontosabb matematikai alapismereteket.

1. Definíció. Lokális minimum:

Az $f\colon (D_f \subseteq \mathbb{R}^n) \to \mathbb{R}$ függvénynek a $p \in \mathbb{R}^n$ pontban lokális minimuma van, ha p-nek van olyan $U \subset \mathbb{R}^n$ környezete, amelyben f értelmezve van, és minden $x \in U$-ra $f(x) \geq f(p)$. Ekkor a p pontot az f függvény lokális minimumhelyének nevezzük.

1. Tétel. *Láncszabály:*

Ha a $g\colon \mathbb{R} \to \mathbb{R}$ függvény differenciálható $p \in \mathbb{R}$ pontban és az $f\colon \mathbb{R} \to \mathbb{R}$ függvény differenciálható $g(p)$-ben, akkor a $h = f \circ g$ függvény is differenciálható p-ben, és

$$h'(p) = f'(g(p)) \cdot g'(p).$$

Az $y = g(x)$ és $z = f(y)$ jelöléssel a tétel állítása a könnyen megjegyezhető

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

alakban írható.

2. Definíció. Skalármező gradiense:

Az $f\colon (D_f \subseteq \mathbb{R}^n) \to \mathbb{R}$ függvény gradiensvektora, más szóval gradiense a $\operatorname{grad} f = \nabla f\colon \mathbb{R}^n \to \mathbb{R}^n$ vektorfüggvény. A $\mathbf{p} = (x_1, x_2, \ldots, x_n) \in D_f$ pontban a gradiens:

$$\nabla f(\mathbf{p}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1}(\mathbf{p}) \\ \dfrac{\partial f}{\partial x_2}(\mathbf{p}) \\ \vdots \\ \dfrac{\partial f}{\partial x_n}(\mathbf{p}) \end{bmatrix},$$

vagyis f parciális deriváltjai a $\mathbf{p}$ pontban vannak számolva.

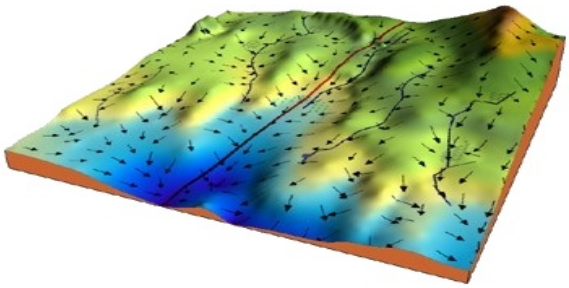
Megjegyzés. Az összetett (esetleg többszörösen összetett) f függvények gradiense hatékonyan számolható a láncszabály segítségével (ld. 1. Tétel).

Emlékeztető. Mint láttuk, neurális hálózatokban például az i. réteg $x^{(i)}$ kimenete függ az $(i-1)$. réteg kimenetétől (ami önmaga is egy függvényérték), valamint a réteg neuronjainak paramétereitől: $x^{(i)} = \varphi_i(W^{(i)} \cdot x^{(i-1)} + \mathbf{b}^{(i)})$.

A skalármező gradiensének geometriai jelentése az, hogy a $\mathbf{p}$ pontban megadja a legnagyobb meredekség irányát. A vektor hossza pedig ennek a legnagyobb meredekségnek a nagyságát mutatja meg. Ha a $\mathbf{p}$ pont lokális minimum, lokális maximum vagy nyeregpont, akkor a gradiensvektor a nullvektor lesz.

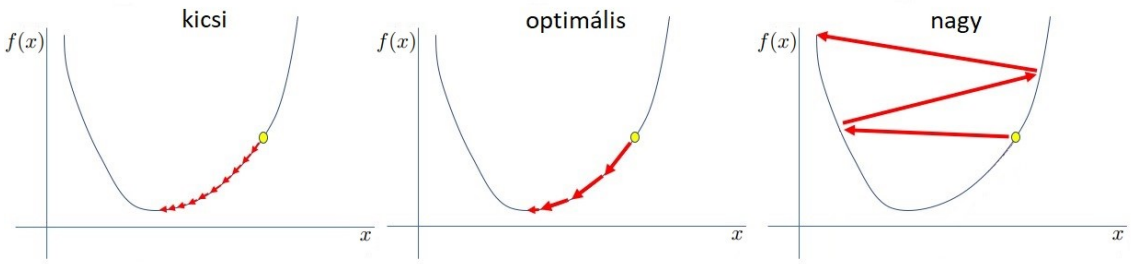
Gradiens süllyedés:

A gradiens süllyedés iteratív optimalizáló eljárás egy megfelelő simasági tulajdonságokkal rendelkező f függvény lokális minimumának megtalálására. Az f függvény egy $\mathbf{p}_0$ pontjából indul ki, ahol a legnagyobb meredekség irányát $\nabla f(\mathbf{p}_0)$ adja meg. Ekkor a $-\nabla f(\mathbf{p}_0)$ vektor a legmeredekebb ereszkedés irányába mutat. Az ötlet az, hogy ezt a vektort egy ún. tanulási ráta (vagy tanulási sebesség) skalárértékű paraméterrel szorozzuk meg, ami meghatározza, hogy mekkorát lép lefelé az algoritmus. Az így kapott új pontokból iterálva közelítjük f lokális minimumhelyét.



3.2. ábra. Gradiens egy domborzat különböző pontjaiban

Ha a tanulási sebesség értéke túl nagy, akkor az algoritmus instabillá válhat, ugyanis kicsi meredekségeknél könnyen túlléphet az f függvény minimumhelyén és az konvergenciaproblémához vezetne. Túl kicsi érték esetén viszont lassabb konvergencia mellett több iterációs lépés szükséges.



3.3. ábra. Lokális minimumhely keresése különböző nagyságú tanulási sebességek mellett

A tanulási ráta megfelelő értékének megválasztása nehéz feladat, függ f viselkedésétől és paramétereinek számától. Értéke általánosan 0,1-nél kisebb szám, de a gyakorlatban nem ritka a 0,001-nél kisebb értékű tanulási sebességek használata sem. Bizonyos esetekben az értéke változhat a már megtett lépésszám függvényében is: minél többször léptünk, értéke annál kisebb legyen. A gradiens süllyedés használatával tehát a neurális hálózatok hibafüggvényeinek minimumhely-keresése optimalizációs problémára vezethető vissza.

A tanulás folyamata:

1. A veszteségfüggvény gradiensének meghatározása a hálózati paraméterek (a w súlyértékek és b eltolóértékek) szerint.
2. A gradiens kiszámolása a hálózati paraméterek jelenlegi értékének behelyettesítésével.
3. A paraméterek lépésközeinek meghatározása:

   lépésköz = tanulási ráta · meredekség
4. Frissítjük a paraméterek értékeit:

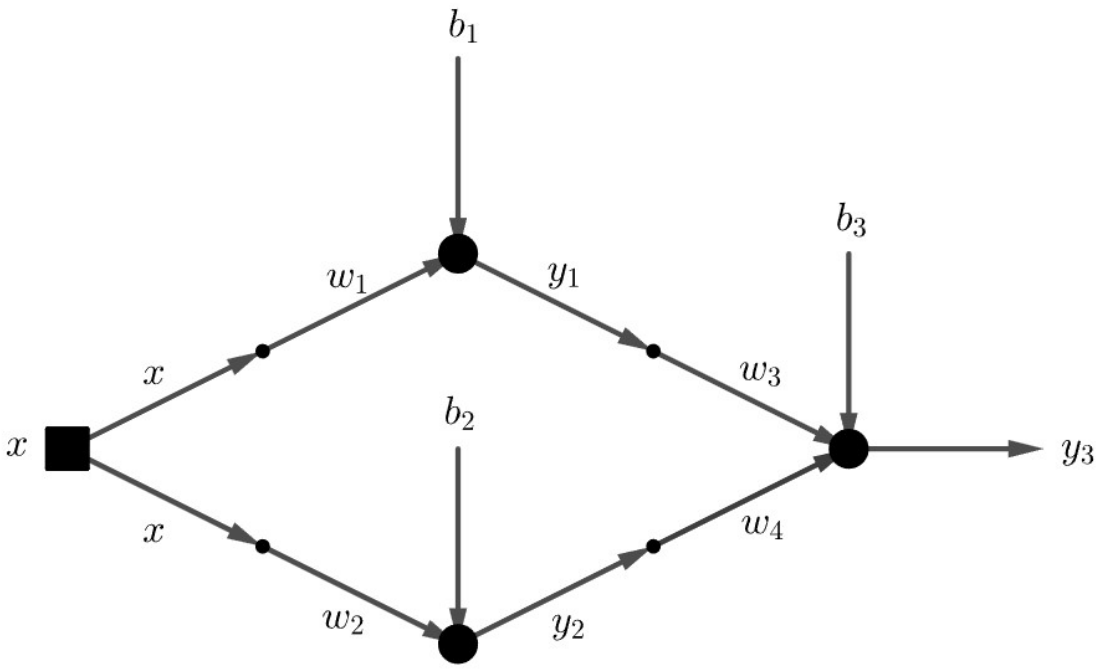
   új érték = előző érték - lépésköz

Általában akkor áll le egy neurális hálózat tanulása, amikor a lépésköz már nagyon kicsi lesz, vagy a modell elér egy előre meghatározott pontosságot a veszteségfüggvény szerint. Az is opció lehet, hogy K darab tanulási folyamat után nem tanítjuk tovább a hálózatot. Amíg nem áll le a tanulás, addig a 2. ponttól ismételjük az eljárást.

A következő alfejezetben egy neurális hálózat működése követhető végig lépésről lépésre.

3.3. Példa

A könnyebb megértés érdekében nézzük meg egy mesterséges neurális hálózat működését egy egyszerű példán keresztül. A modellnek legyen egy be- és kimenete, valamint egy köztes rétegen belül legyen két neuronja. Ehhez a két neuronhoz szoftplusz, míg a kimeneti neuronhoz lineáris aktivációs függvényt rendelünk. Az élek súlyai kezdetben standard normális eloszlásból vett értékek, az eltoló-vektorok pedig nulla értékűek legyenek. A tanulási sebesség legyen $LR = \frac{1}{16}$ és az várjuk, hogy az $x = 2$ bemenetre $y = \frac{1}{2}$ kimeneti értéket kapjunk. A hiba kiszámításához használjuk az $L = (\hat{y} - y)^2$ négyzetes veszteségfüggvényt. Ha a kapott érték 5%-os hibahatáron belüli, vagyis $L < (\frac{1}{2} \cdot 0{,}05)^2 = 0{,}000625$, akkor leállítjuk a hálózat tanítását.



3.4. ábra. A példa modell

A modell paraméterei kezdetben:

- $w_1 = 0{,}81$
- $w_2 = -1{,}38$
- $w_3 = -2{,}07$
- $w_4 = 0{,}19$
- $b_1 = b_2 = b_3 = 0$

1. kimenetképzés

Neurononként írjuk fel az x_1, x_2, x_3 bemenetek, és az y_1, y_2, y_3 kimenetek kiszámításának módját a hálózati paraméterek szerint, illetve a hibafüggvényt:

$x_1 = w_1 \cdot x + b_1 = 0{,}81 \cdot 2 + 0 = 1{,}62$

$x_2 = w_2 \cdot x + b_2 = -1{,}38 \cdot 2 + 0 = -2{,}76$

$y_1 = \varphi_{\text{Szoftplusz}}(x_1) = \ln(1 + e^{x_1}) = \ln(1 + e^{1{,}62}) \approx 1{,}8006$

$y_2 = \varphi_{\text{Szoftplusz}}(x_2) = \ln(1 + e^{x_2}) = \ln(1 + e^{-2{,}76}) \approx 0{,}0614$

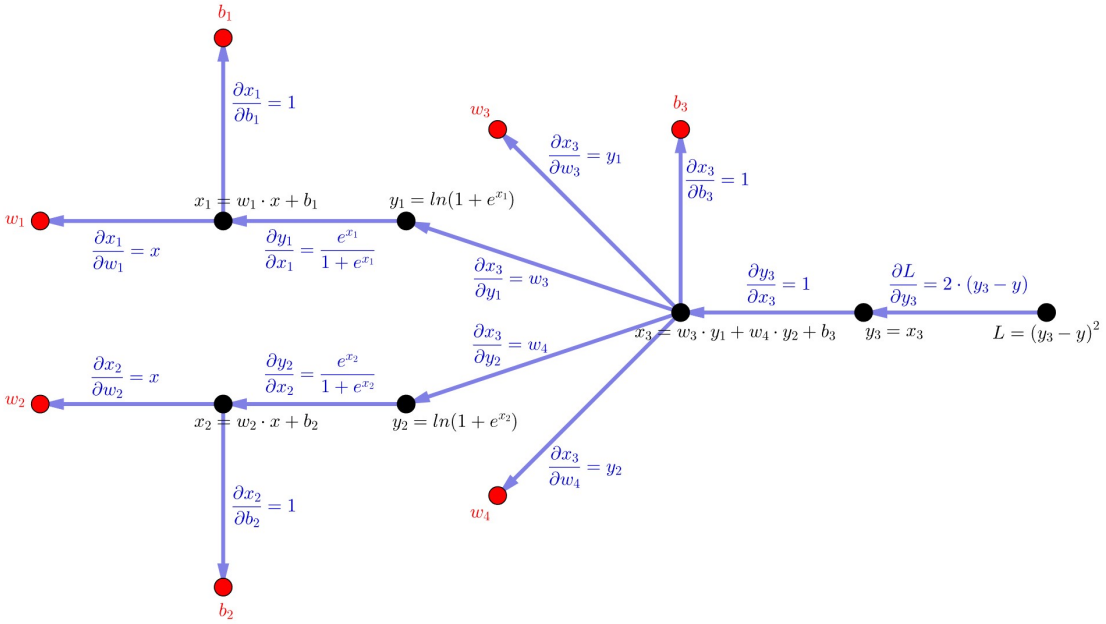
$x_3 = w_3 \cdot y_1 + w_4 \cdot y_2 + b_3 = -2{,}07 \cdot 1{,}8006 + 0{,}19 \cdot 0{,}0614 + 0 \approx -3{,}7156$

$y_3 = \varphi_{\text{Identitás}}(x_3) = x_3 \approx -3{,}7156$

$L = (y_3 - y)^2 \approx (-3{,}7156 - 0{,}5)^2 \approx 17{,}7713$

1. hibavisszaterjesztés:

A fenti, kimenetképzésben leírtaknak megfelelően a gradiens láncszabállyal való kiszámításához segítségül rajzoljunk fel egy segédgráfot:



3.5. ábra. Fagráf a láncszabály felírásához

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} \cdot \frac{\partial x_1}{\partial w_1} = 2(y_3 - y) \cdot 1 \cdot w_3 \cdot \frac{e^{x_1}}{1 + e^{x_1}} \cdot x \approx 29{,}1387$$
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_2} \cdot \frac{\partial x_2}{\partial w_2} = 2(y_3 - y) \cdot 1 \cdot w_4 \cdot \frac{e^{x_2}}{1 + e^{x_2}} \cdot x \approx -0{,}1907$$
$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial w_3} = 2(y_3 - y) \cdot 1 \cdot y_1 \approx -15{,}1812$$
$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial w_4} = 2(y_3 - y) \cdot 1 \cdot y_2 \approx -0{,}5177$$
$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} \cdot \frac{\partial x_1}{\partial b_1} = 2(y_3 - y) \cdot 1 \cdot w_3 \cdot \frac{e^{x_1}}{1 + e^{x_1}} \cdot 1 \approx 14{,}5693$$
$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_2} \cdot \frac{\partial x_2}{\partial b_2} = 2(y_3 - y) \cdot 1 \cdot w_4 \cdot \frac{e^{x_2}}{1 + e^{x_2}} \cdot 1 \approx -0{,}0954$$
$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial y_3} \cdot \frac{\partial y_3}{\partial x_3} \cdot \frac{\partial x_3}{\partial b_3} = 2(y_3 - y) \cdot 1 \cdot 1 \approx -8{,}4312$$

1. tanulás után frissítjük a paraméterek értékeit:

- $w_1 := w_1 - \left(LR \cdot \frac{\partial L}{\partial w_1} \right) \approx -1{,}0111$
- $w_2 := w_2 - \left(LR \cdot \frac{\partial L}{\partial w_2} \right) \approx -1{,}3681$
- $w_3 := w_3 - \left(LR \cdot \frac{\partial L}{\partial w_3} \right) \approx -1{,}1212$
- $w_4 := w_4 - \left(LR \cdot \frac{\partial L}{\partial w_4} \right) \approx 0{,}2224$

- $b_1 := b_1 - \left(LR \cdot \frac{\partial L}{\partial b_1} \right) \approx -0{,}9106$
- $b_2 := b_2 - \left(LR \cdot \frac{\partial L}{\partial b_2} \right) \approx 0{,}0060$
- $b_3 := b_3 - \left(LR \cdot \frac{\partial L}{\partial b_3} \right) \approx 0{,}5270$

2. kimenetképzés:

$x_1 = w_1 \cdot x + b_1 = -1{,}0111 \cdot 2 + (-0{,}9106) = -2{,}9328$

$x_2 = w_2 \cdot x + b_2 = -1{,}3681 \cdot 2 + 0{,}0060 = -2{,}7302$

$y_1 = \varphi_{szoftplusz}(x_1) = \ln(1 + e^{x_1}) = \ln(1 + e^{-2{,}9328}) \approx 0{,}0519$

$y_2 = \varphi_{szoftplusz}(x_2) = \ln(1 + e^{x_2}) = \ln(1 + e^{-2{,}7302}) \approx 0{,}0632$

$x_3 = w_3 \cdot y_1 + w_4 \cdot y_2 + b_3 = -1{,}1212 \cdot 0{,}0519 + 0{,}2224 \cdot 0{,}0632 + 0{,}5270 \approx 0{,}4829$

$y_3 = \varphi_{lineris}(x_3) = x_3 \approx 0{,}4829$

$L = (y_3 - y)^2 \approx (0{,}4829 - 0{,}5)^2 \approx 0{,}0003$

Látható, hogy az 1. tanulás után elértük a kívánt pontosságot, tehát a hálózat sikeresen el tudta végezni a kitűzött feladatot.

4. fejezet

Differenciálegyenletek megoldása neurális hálózatokkal

A differenciálegyenlet kapcsolatot ír le egy függvény, annak független változói, és az egyes független változók szerinti deriváltjai között. A közönséges differenciálegyenlet egy speciális esete a differenciálegyenletnek, ahol csak egy független változó van jelen. Ebben a fejezetben közönséges differenciálegyenletek neurális hálózatokon alapuló megoldását mutatjuk be Python programmal.

Legyen $f\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ a második változójában lokálisan Lipschitz-folytonos és $a, b, t_0, x_0 \in \mathbb{R}$. Ekkor keressük azt a folytonosan differenciálható $x\colon \mathbb{R} \to \mathbb{R}$ függvényt, melyre

$$\begin{cases} x'(t) = f(t, x(t)), & t \in [a; b] \\ x(t_0) = x_0. \end{cases} \tag{KDE}$$

4.1. Programkód

A kód kezdetben előkészíti a környezetet a neurális hálózat használatához. A szükséges könyvtárak importálása után definiáljuk a megoldani kívánt differenciálegyenletet a kezdeti értékével együtt, és annak analitikus megoldását az összehasonlításhoz. A bemutatott kódban a (KDE) rendszert oldjuk meg $f(t, x) = t \cdot x, a = 0, b = 1, t_0 = 0$ és $x_0 = 1$ választással, aminek megoldása az $x(t) = e^{\frac{t^2}{2}}$ függvény.

A megoldani kívánt differencálegyenlet tehát:

$$\begin{cases} x'(t) = t \cdot x(t), & t \in [0; 1] \\ x(0) = 1 \end{cases}$$

```python
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import math
from matplotlib.pyplot import figure

# KDE és az x0 kezdeti érték:
def f(t, x_t):
    return t * x_t
x0 = 1

# KDE analitikus megoldása:
def analitikus_megoldas(t):
    return tf.math.exp(0.5 * t**2)
```

A modell különböző paramétereit változókban rögzítjük. Az $[a; b]$ értelmezési tartományt 100 egyenlő részre felosztva beállítjuk az $\mathbf{X} \in \mathbb{R}^{101}$ bemeneti vektort, amelynek minden elemére egy közelítő megoldást ad majd a rendszer, szintén egy vektor formájában. A be- és kimenet között két köztes réteget definiálunk, mindkettőt 32 darab neuronnal. A reprodukálhatóság céljából a `tf.random.set_seed(0)` kódsor segítségével beállítjuk a TensorFlow véletlenszám-generátorának kezdőállapotát.

```python
# Paraméterek és az X bemeneti vektor:
tf.random.set_seed(0)
reteg_1_neuronszam = 32
reteg_2_neuronszam = 32
tanulasi_sebesseg = 0.001
lepesek_szama = 2**7
a = 0
b = 1
X = np.linspace(a, b, 101)
```

Inicializáljuk a hálózati paramétereket. Az egyes rétegekhez tartozó súlymátrixokat és eltolóvektorokat szótárként definiáljuk. Előbbieket standard normális eloszlásból vett véletlen értékekkel, utóbbiakat nullákkal töltjük fel.

```python
# Hálózati paraméterek:
sulymatrixok = {
    'W1': tf.Variable(tf.random.normal([1, reteg_1_neuronszam])),
    'W2': tf.Variable(tf.random.normal([reteg_1_neuronszam, reteg_2_neuronszam])),
    'W3': tf.Variable(tf.random.normal([reteg_2_neuronszam, 1]))
}
eltolovektorok = {
```

```
    'b1': tf.Variable(tf.zeros([reteg_1_neuronszam])),
    'b2': tf.Variable(tf.zeros([reteg_2_neuronszam])),
    'b3': tf.Variable(tf.zeros([1]))
}
```

Építsük fel a neurális hálózatot. A modellnek a bemeneti vektort kell az egyes rétegeken keresztül feldolgoznia, majd előállítani a kimeneti vektort. Mint már láttuk, a bemenetet feldolgozza az első réteg, annak kimenetét feldolgozza a második réteg, és így tovább. A két köztes réteg neuronjai a $\varphi_{\text{Logisztikus}}$ (másik nevén Szigmoid) aktivációs függvényt alkalmazzák, a kimeneti réteg egy neuronja pedig a $\varphi_{\text{Identitás}}$ függvényt használja.

```
# Modell felépítése:
def modell(x):
    x = np.float32(np.array([[[x]]]))
    reteg_1 = tf.matmul(x, sulymatrixok['W1']) + eltolovektorok['b1']
    reteg_1_kimenet = tf.nn.sigmoid(reteg_1)
    reteg_2 = tf.matmul(reteg_1_kimenet, sulymatrixok['W2']) + eltolovektorok['b2']
    reteg_2_kimenet = tf.nn.sigmoid(reteg_2)
    kimenet = tf.matmul(reteg_2_kimenet, sulymatrixok['W3']) + eltolovektorok['b3']
    return kimenet
```

A hálózat kimenetének előállítása után szükség van egy veszteségfüggvényre, amivel a hibát mérni tudjuk. A modellt a megadott differenciálegyenlet megoldásának közelítésére használjuk, tehát a cél:

$$\text{modell}(t) \approx x(t).$$

Az $x(t)$ függvényt nem ismerjük, máshogyan kell előállítani a hibafüggvényt. Az ötlet az, hogy ha a modell(t) jól közelíti az $x(t)$ megoldást, akkor deriváltjaik is jól közelítik egymást, így:

$$\text{modell}'(t) \approx x'(t) = f(t, x(t)) \approx f(t, \text{modell}(t)).$$

Ebből f függvény adott, modell$'(t)$ pedig meghatározható. A veszteségfüggvény létrehozásakor természetesen figyelembe kell venni a megadott kezdetiértéket is, vagyis a hibafüggvény legyen:

$$L = \sum_i \left(\frac{d}{dt}\text{modell}(t_i) - f(t_i, \text{modell}(t_i)) \right)^2 + (\text{modell}(0) - x_0)^2.$$

A gyakorlatban sok esetben az így definiált veszteségfüggvény használata nem vezetne megoldásra, instabil tanulási folyamatot eredményezne. E probléma kiküszöbölésére létezik egy hatékonyabb tanulást eredményező hibafüggvény. Először definiálni kell egy új függvényt, ami a modell(t) helyett alkalmazható:

$$\text{közelítő megoldás}(t) = t \cdot \text{modell}(t) + x_0.$$

Triviális, hogy ez az új függvény mindig kielégíti majd a kezdeti feltételt, hiszen $t = 0$-ra x_0 értéket ad, és ez a tulajdonsága nagyon fontos szerepet játszik majd abban, hogy a tanulási folyamat stabilabb legyen. Így az új veszteségfüggvény:

$$L = \sum_i \left(\frac{d}{dt}\text{közelítő megoldás}(t_i) - f(t_i, \text{közelítő megoldás}(t_i)) \right)^2.$$

Az alábbi kódrészlet az imént leírtakat valósítja meg. A `kozelito_megoldas` függvény t pontjaiban a derivált értékének kiszámítása numerikus úton történik, a differenciálhányados definíciója szerint. Ehhez h-t, egy kis gépi epszilont adunk meg.

```
# KDE közelítő megoldása:
def kozelito_megoldas(x):
    return x * modell(x) + x0

# Négyzetes veszteségfüggvény:
h = np.sqrt(np.finfo(np.float32).eps)
def vesztesegfuggveny():
    negyzetosszeg = 0
    for i in X:
        t = np.float32(i)
        dNN = (kozelito_megoldas(t + h) - kozelito_megoldas(t)) / h
        negyzetosszeg += (dNN - f(t, kozelito_megoldas(t)))**2
    return negyzetosszeg
```

A továbbiakban meg kell határozni a veszteségfüggvény gradiensét a hálózati paraméterek szerint, majd értéküket frissíteni. A gradiens süllyedés algoritmust, mint optimalizáló eljárást meghívhatjuk a TensorFlow könyvtár segítségével.

```
# Kimenetképzés + Hibavisszaterjesztés:
gradiens_sullyedes = tf.optimizers.SGD(tanulasi_sebesseg)
def lepes():
    with tf.GradientTape() as tape:
        veszteseg = vesztesegfuggveny()
    halozati_parameterek = list(sulymatrixok.values()) + list(eltolovektorok.values())
```

```
    gradiens = tape.gradient(veszteseg, halozati_parameterek)
    gradiens_sullyedes.apply_gradients(zip(gradiens, halozati_parameterek))
```

A kód végén a bemutatott függvényekkel megvalósítjuk a működési folyamatot. Az első ábra megmutatja, hogy a hálózati paraméterek kezdőértékével milyen közelítést ad a modell. A további grafikonok minden 2^i. tanulás után ($i = 0, 1, \ldots$) illusztrálják a hálózat által adott predikciós függvényt és a tényleges megoldást. Az utolsó diagram a veszteség alakulását ábrázolja, a tanulások számának függvényében.

```
# Működés:
kovetkezo_2hatvany = 1
vesztesegek = []
for i in range(lepesek_szama + 1):
    vesztesegek.append(vesztesegfuggveny().numpy()[0][0][0])

    if (i) == 0:
        y_kalap_kezdoallapot = []
        for j in X:
            y_kalap_kezdoallapot.append(kozelito_megoldas(j).numpy()[0][0][0])

        figure(figsize = (12, 8))
        plt.plot(X, analitikus_megoldas(X), label = "Analitikus megoldás", color = "green")
        plt.plot(X, y_kalap_kezdoallapot, label = "Közelítő megoldás", color = "red")
        plt.legend(loc = 2, prop = {'size': 15})
        plt.title("Veszteség a kezdőállapotban: %f " % vesztesegfuggveny(), fontsize = 20)
        plt.show()

    elif (i) == kovetkezo_2hatvany:
        kovetkezo_2hatvany *= 2
        y_kalap_2hatvany_lepesek = []
        for j in X:
            y_kalap_2hatvany_lepesek.append(kozelito_megoldas(j).numpy()[0][0][0])

        figure(figsize = (12, 8))
        plt.plot(X, analitikus_megoldas(X), label = "Analitikus megoldás", color = "green")
        plt.plot(X, y_kalap_2hatvany_lepesek, label = "Közelítő megoldás", color = "red")
        plt.legend(loc = 2, prop = {'size': 15})
        plt.title("Veszteség a(z) %i. lépés után: %f " % (i, vesztesegfuggveny()), fontsize = 20)
        plt.show()

    lepes()

figure(figsize = (12, 8))
plt.xlabel("tanulások száma")
plt.ylabel("veszteség")
plt.plot(vesztesegek, color = "blue")
```

```
38 plt.title("Veszteség alakulása a tanulásszám függvényében", fontsize = 20)
39 plt.show()
```

4.2. Szinusz függvény közelítése

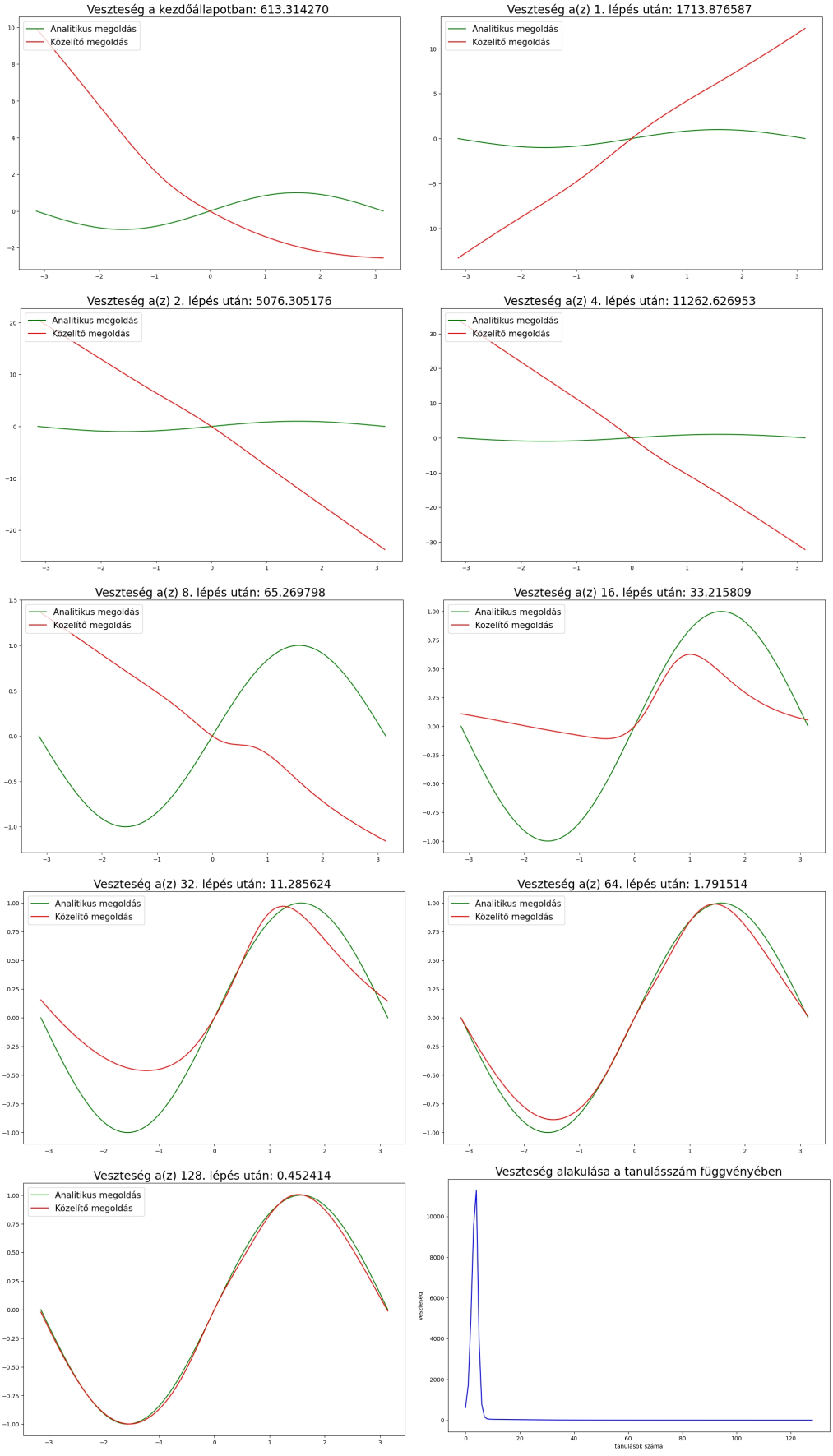
Oldjuk meg a (KDE) rendszert $f(t,x) = \cos(t), a = -\pi, b = \pi, t_0 = 0$ és $x_0 = 0$ választással, azaz keressük

$$\begin{cases} x'(t) = \cos(t), & t \in [-\pi; \pi] \\ x(0) = 0 \end{cases}$$

megoldását, ami az $x(t) = \sin(t)$ függvény. A kódban az alábbi beállításokat tegyük:

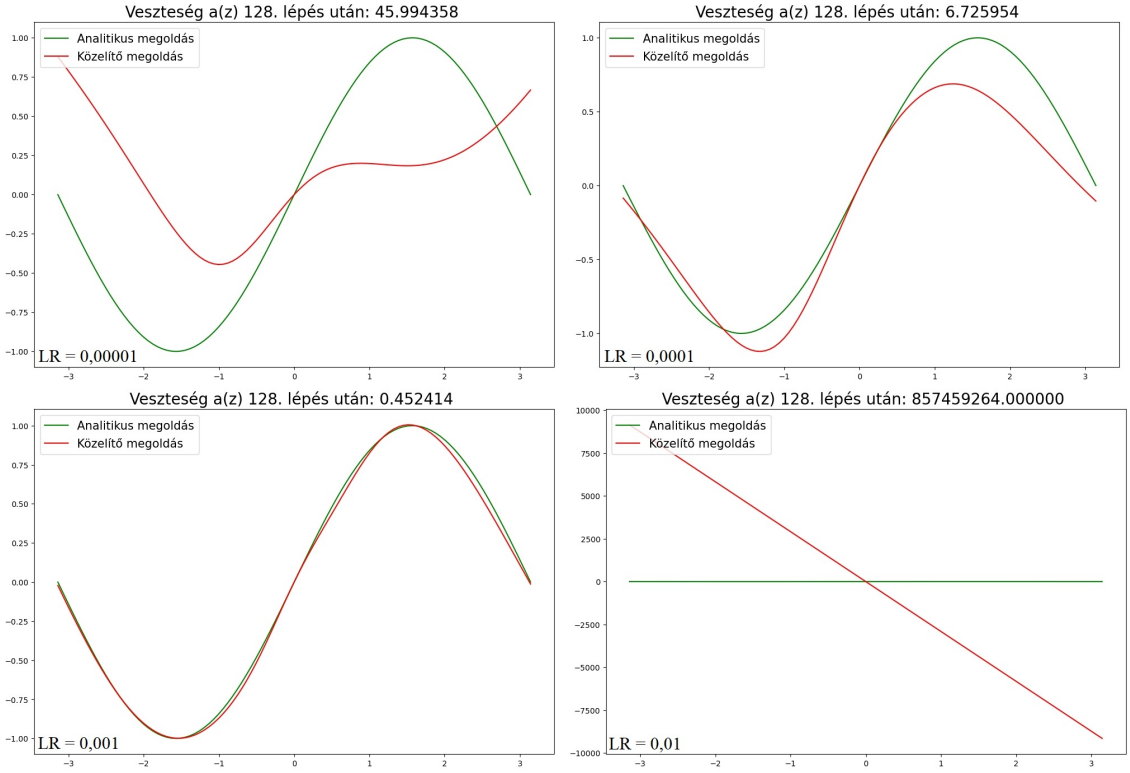
```
def f(t, x_t):
    return tf.math.cos(t)
x0 = 0

def analitikus_megoldas(t):
    return tf.math.sin(t)

a = -math.pi
b = math.pi
```

A működési folyamatot 0,001 tanulási sebesség mellett a 4.1. ábra szemlélteti a következő oldalon.



4.1. ábra. A $\sin(t)$ függvény közelítése

Vizsgáljuk meg a következő ábrán, hogy a tanulási sebesség megváltoztatásával hogyan módosul a rendszer predikciós képessége az utolsó tanulási ciklus után.



4.2. ábra. Közelítés különböző LR tanulási sebességek mellett

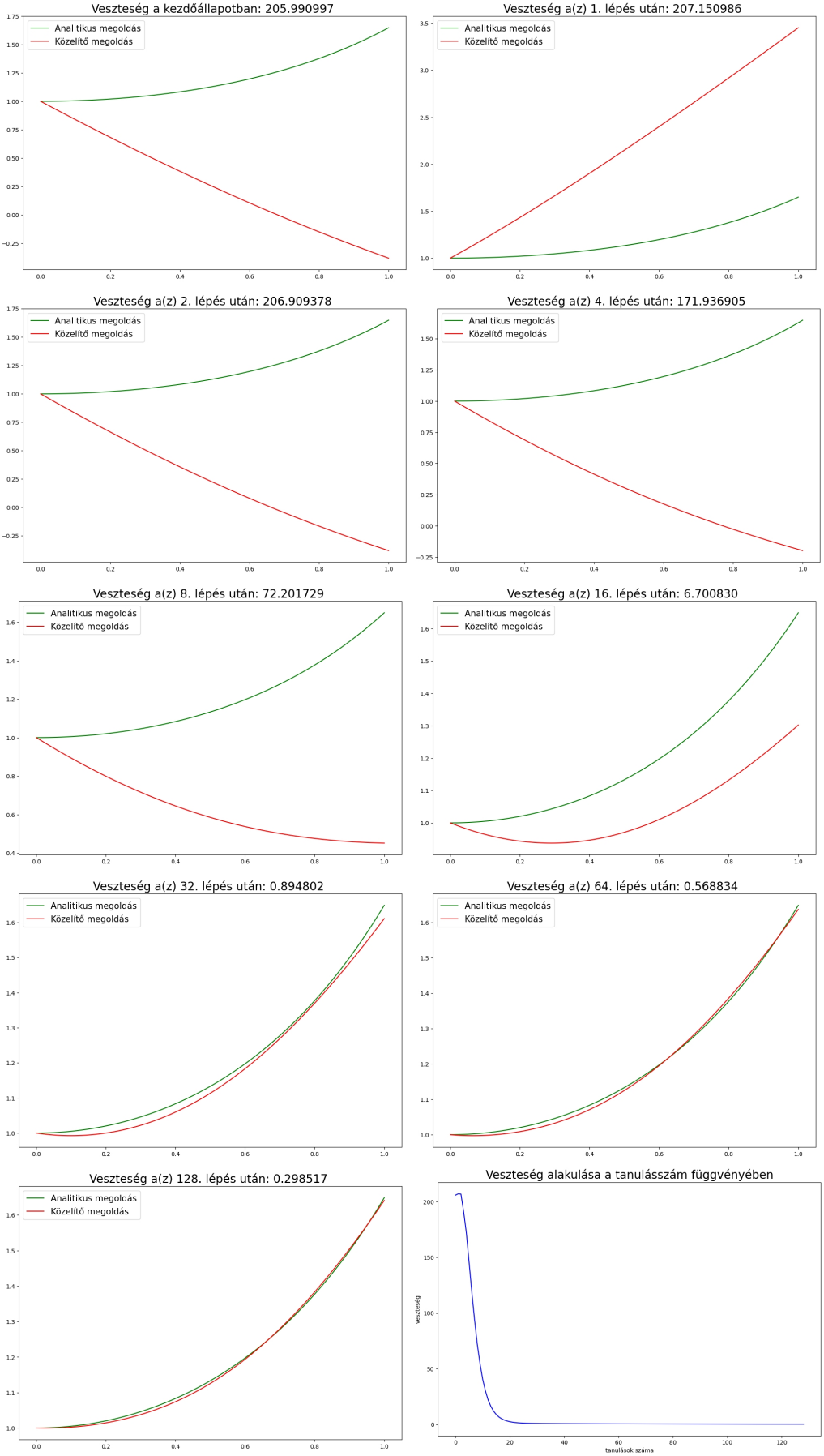
Mint látható, az $LR = 0{,}001$ érték vezetett a legpontosabb eredményre. A két kisebb sebesség esetén a modell lassan, de szépen elkezdte közelíteni a $\sin(t)$ függvényt. Több számítási erőforrással és tanulási ciklus megtételével a közelítés jól konvergálna a megoldáshoz. Az $LR = 0{,}01$ tanulási sebesség egyértelműen instabil működéshez vezetett.

4.3. Exponenciális függvény közelítése

Vizsgáljuk meg a 4.1. fejezetben bemutatott kódban szereplő

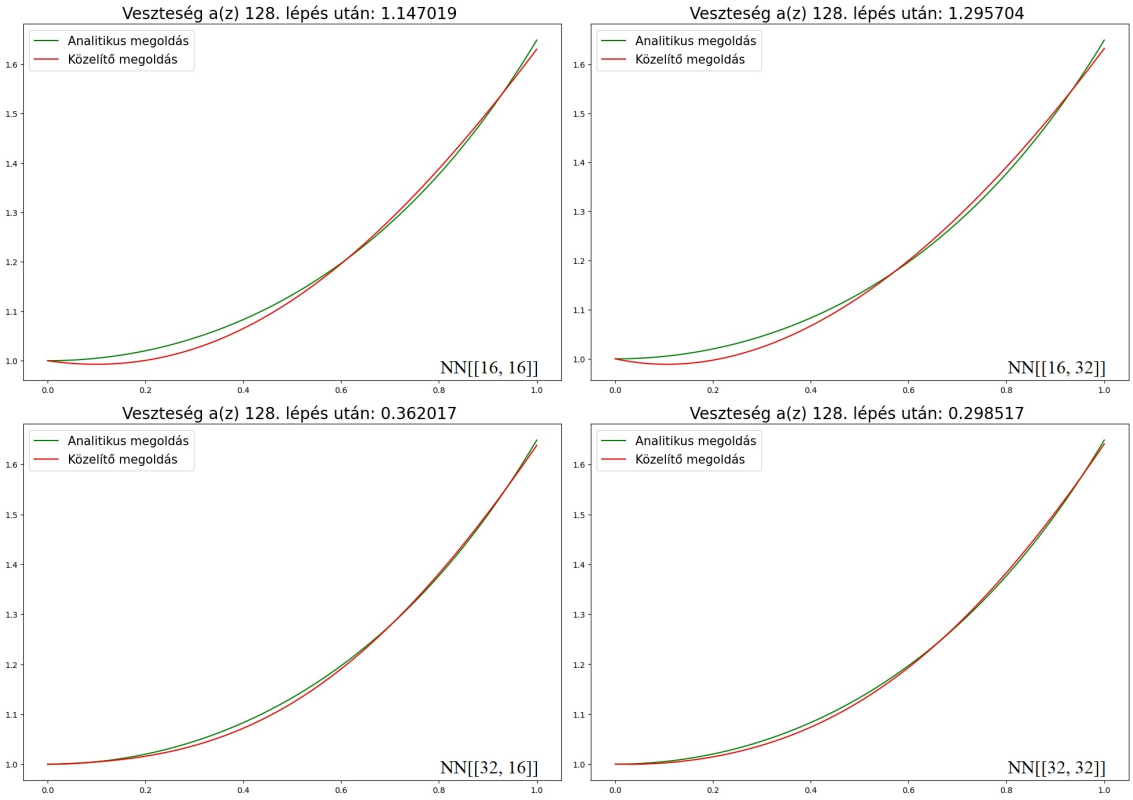
$$\begin{cases} x'(t) = t \cdot x(t), & t \in [0;1] \\ x(0) = 1 \end{cases}$$

rendszert. A kezdőértékével adott differenciálegyenlet analitikus megoldása az $x(t) = e^{\frac{t^2}{2}}$ függvény. A tanulási folyamat a 4.3. ábrán látható.



4.3. ábra. Az $e^{\frac{t^2}{2}}$ függvény közelítése

Az előző példában megvizsgáltuk, hogyan módosul a tanulási sebesség megváltoztatásával a modell viselkedése. Most figyeljük meg a működést különböző neuronszámú köztes rétegek mellett a 4.4. ábrán. Az NN[[m, n]] azt a modellt jelöli, melynek első köztes rétegében m, második köztes rétegében n darab neuron van.



4.4. ábra. Közelítés különböző neuronszámú köztes rétegek mellett

Nem túl nagy hibahatár mellett minden esetben elfogadható eredményt kaptunk. Az a modell, amelynek csak az első köztes rétege tartalmazott 32 darab neuront, meglepően jól teljesített az eredeti, NN[[32, 32]] modellhez képest.

Összegzés

A szakdolgozat első felében megismertük, miként ihlette az agyi idegrendszer a mesterséges neurális hálózatok lemodellezésének ötletét. Bemutattuk a hálózat legfontosabb elemeit, a modell felépítésének folyamatát. A dolgozat második felében leírtuk, hogyan működik a létrehozott neurális hálózat, valamint hogy milyen eljárással képes tanítani magát egy adott feladat elvégzésére. Végül bemutattunk egy konkrét gyakorlati alkalmazást, a differenciálegyenletek neurális hálózatokkal történő közelítő megoldását.

Láttuk, hogy a modell tanulási sebességének megválasztása döntően befolyásolja a közelítés konvergenciáját a megoldáshoz. Kisebb sebesség mellett lassabb, de stabilabb konvergencia érhető el a tanulásban. Rétegenként több neuron alkalmazásával bonyolultabb feladatok oldhatók meg, nagyobb hálózat használatával pontosabb közelítéseket kaptunk. Az a következtetés biztosan levonható, hogy az optimális tanulási sebesség és a hálózatot felépítő neuronok számának meghatározása nehéz feladat, ami sok gyakorlati tudást igényel.

Köszönetnyilvánítás

Szeretném megköszönni konzulensemnek, Csomós Petrának az egész féléves munkáját. Nagyon hálás vagyok neki, hogy mindig külön időt és energiát fordított rám, mikor segítségre volt szükségem. Értékes javaslatai és ötletei nagyban hozzájárultak a szakdolgozatom elkészítéséhez.

Irodalomjegyzék

[1] *Idegsejt.* https://hu.wikipedia.org/wiki/Idegsejt. Hozzáférés dátuma: 2023.06.06.

[2] *NEURÁLIS HÁLÓZATOK: A KAPCSOLAT AZ EMBERI IDEGRENDSZER ÉS A MESTERSÉGES INTELLIGENCIA KÖZT.* https://netmasters.hu/blog/neuralis-halozatok-a-kapcsolat-az-emberi-idegrendszer-es-a-mesterseges-intelligencia-kozt/. Hozzáférés dátuma: 2023.06.06.

[3] *Importance of Artificial Neural Networks in Artificial Intelligence.* https://www.turing.com/kb/importance-of-artificial-neural-networks-in-artificial-intelligence. Hozzáférés dátuma: 2023.06.06.

[4] *Mesterséges neurális hálózatok.* https://semmelweis.hu/kepalkotas/files/ANN_BenyoB.pdf. Hozzáférés dátuma: 2023.06.06.

[5] *Activation Functions in Neural Networks.* https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. Hozzáférés dátuma: 2023.06.06.

[6] *Logistic function.* https://en.wikipedia.org/wiki/Logistic_function. Hozzáférés dátuma: 2023.06.06.

[7] *Softmax function.* https://en.wikipedia.org/wiki/Softmax_function. Hozzáférés dátuma: 2023.06.06.

[8] *Loss Functions and Their Use In Neural Networks.* https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9. Hozzáférés dátuma: 2023.06.06.

[9] *Loss Functions - EXPLAINED!* https://www.youtube.com/watch?v=QBbC3Cjsnjg. Hozzáférés dátuma: 2023.06.06.

[10] *Huber loss*. https://en.wikipedia.org/wiki/Huber_loss. Hozzáférés dátuma: 2023.06.06.

[11] *Cross-Entropy Loss Function*. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e. Hozzáférés dátuma: 2023.06.06.

[12] Laczkovich Miklós T. Sós Vera, szerk. *Valós analízis I.* 2012. ISBN: 978-963-279-732-8.

[13] Laczkovich Miklós T. Sós Vera, szerk. *Valós analízis II.* 2013. ISBN: 978-963-279-733-5.

[14] *Gradient Descent, Step-by-Step*. https://www.youtube.com/watch?v=sDv4f4s2SB8. Hozzáférés dátuma: 2023.06.06.

[15] *Gradient descent*. https://duchesnay.github.io/pystatsml/optimization/optim_gradient_descent.html. Hozzáférés dátuma: 2023.06.06.

[16] *Backpropagation Details Pt. 1: Optimizing 3 parameters simultaneously.* https://www.youtube.com/watch?v=iyn2zdALii8. Hozzáférés dátuma: 2023.06.06.

[17] *Using Neural Networks to solve Ordinary Differential Equations*. https://towardsdatascience.com/using-neural-networks-to-solve-ordinary-differential-equations-a7806de99cdd. Hozzáférés dátuma: 2023.06.06.

[18] *tf.keras.optimizers.experimental.SGD*. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/experimental/SGD. Hozzáférés dátuma: 2023.06.06.

Ábrák jegyzéke

Függelék

Négyzetes eltérés MATLAB kódja:

```
% Adatok inicializálása
x = [0; 0.5; 1; 1.5; 2; 2.5; 3; 3.5; 4; 4.5; 5; 5.5;...
    %6; 6.5; 7;...
    7.5; 8; 8.5; 9; 9.5; 10];
y = [0.1; 0.4; 1.3; 2.1; 1.8; 2.5; 2.9; 4.3; 3.9; 5.1; 4.7; 5.1;...
    %0.8; 1.7; 0.6;...
    7.2; 8; 8.9; 9.5; 10.2; 9.8];

felosztas = 101;
karika_meret = 100;

% Lineáris regresszió kiszámítása négyzetes veszteséggel
regresszio_felosztas = linspace(min(x), max(x), felosztas);
p = polyfit(x, y, 1);
meredekseg = p(1);
tengelymetszet = p(2);
regresszios_egyenes = @(x) meredekseg * x + tengelymetszet;

% 1280x720 méretű ábra
figure('Renderer', 'painters', 'Position', [0 0 1280 720])

% 1. ábra
subplot(1, 2, 1)
scatter(x, y, karika_meret,...
    'MarkerEdgeColor', [0 0 1.0],...
    'MarkerFaceColor', [0.75 0.75 1.0],...
    'LineWidth', 2.0);
hold on;
plot(regresszio_felosztas, regresszios_egyenes(regresszio_felosztas),...
    'MarkerEdgeColor', [0.5 0 0], 'LineWidth', 3.0);
legend('', '$$\hat{y}$$', 'Interpreter', 'latex', 'FontSize', 20, 'Location','southeast');
axis square;
title('Lineáris regresszió négyzetes eltéréssel', FontSize = 14);
xlabel('$$x$$', 'Interpreter', 'latex', 'FontSize', 20);
ylabel('$$y$$', 'Interpreter', 'latex', 'FontSize', 20);
hold off;

```

```
%Veszteségfüggvény
hibak = regresszios_egyenes(x) - y;
veszteseg_felosztas = linspace(-max(abs(hibak)), max(abs(hibak)), felosztas);
negyzetes_veszteseg_fuggveny = @(x) x.^2;

% 2. ábra
subplot(1, 2, 2)
scatter(hibak, negyzetes_veszteseg_fuggveny(hibak), karika_meret,...
    'MarkerEdgeColor', [0 0 1.0],...
    'MarkerFaceColor', [0.75 0.75 1.0],...
    'LineWidth', 2.0);
hold on
plot(veszteseg_felosztas, negyzetes_veszteseg_fuggveny(veszteseg_felosztas),...
    'MarkerEdgeColor', [0.5 0 0], 'LineWidth', 3.0);
axis square;
title('Négyzetes veszteségfüggvény', 'FontSize', 14);
xlabel('$$\hat{y} - y$$', 'Interpreter', 'latex', 'FontSize', 20);
ylabel('$$(\hat{y} - y)^2$$', 'Interpreter', 'latex', 'FontSize', 20);
hold off;
```

Abszolút eltérés MATLAB kódja:

```
% Adatok inicializálása
x = [0; 0.5; 1; 1.5; 2; 2.5; 3; 3.5; 4; 4.5; 5; 5.5;...
    % 6; 6.5; 7;...
    7.5; 8; 8.5; 9; 9.5; 10];
y = [0.1; 0.4; 1.3; 2.1; 1.8; 2.5; 2.9; 4.3; 3.9; 5.1; 4.7; 5.1;...
    % 0.8; 1.7; 0.6;...
    7.2; 8; 8.9; 9.5; 10.2; 9.8];

felosztas = 101;
karika_meret = 100;

% Lineáris regresszió kiszámítása abszolút veszteséggel
regresszio_felosztas = linspace(min(x), max(x), felosztas);
p = polyfit(x, y, 1);
abszolut_vesztesegfuggveny = @(v)sum(abs(v(1) * x + v(2) - y));
mb = fminsearch(abszolut_vesztesegfuggveny, p); % az optimális m és b kiszámítása
meredekseg = mb(1);
tengelymetszet = mb(2);
regresszios_egyenes = @(x) meredekseg * x + tengelymetszet;

% 1280x720 méretű ábra
figure('Renderer', 'painters', 'Position', [0 0 1280 720])

% 1. ábra
subplot(1, 2, 1)
scatter(x, y, karika_meret,...
    'MarkerEdgeColor', [0 0 1.0],...
    'MarkerFaceColor', [0.75 0.75 1.0],...
```

```
    'LineWidth', 2.0);
hold on;
plot(regresszio_felosztas, regresszios_egyenes(regresszio_felosztas),...
    'MarkerEdgeColor', [0.5 0 0], 'LineWidth', 3.0);
legend('', '$$\hat{y}$$', 'Interpreter', 'latex', 'FontSize', 20, 'Location','southeast');
axis square;
title('Lineáris regresszió abszolút eltéréssel', FontSize = 14);
xlabel('$$x$$', 'Interpreter', 'latex', 'FontSize', 20);
ylabel('$$y$$', 'Interpreter', 'latex', 'FontSize', 20);
hold off;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adatok inicializálása
x = [0; 0.5; 1; 1.5; 2; 2.5; 3; 3.5; 4; 4.5; 5; 5.5;...
    6; 6.5; 7;...
    7.5; 8; 8.5; 9; 9.5; 10];
y = [0.1; 0.4; 1.3; 2.1; 1.8; 2.5; 2.9; 4.3; 3.9; 5.1; 4.7; 5.1;...
    0.8; 1.7; 0.6;...
    7.2; 8; 8.9; 9.5; 10.2; 9.8];

% Lineáris regresszió kiszámítása abszolút veszteséggel
regresszio_felosztas = linspace(min(x), max(x), felosztas);
p = polyfit(x, y, 1);
abszolut_vesztesegfuggveny = @(v)sum(abs(v(1) * x + v(2) - y));
mb = fminsearch(abszolut_vesztesegfuggveny, p); % az optimális m és b kiszámítása
meredekseg = mb(1);
tengelymetszet = mb(2);
regresszios_egyenes = @(x) meredekseg * x + tengelymetszet;

% 2. ábra
subplot(1, 2, 2)
scatter(x, y, karika_meret,...
    'MarkerEdgeColor', [0 0 1.0],...
    'MarkerFaceColor', [0.75 0.75 1.0],...
    'LineWidth', 2.0);
hold on;
plot(regresszio_felosztas, regresszios_egyenes(regresszio_felosztas),...
    'MarkerEdgeColor', [0.5 0 0], 'LineWidth', 3.0);
legend('', '$$\hat{y}$$', 'Interpreter', 'latex', 'FontSize', 20, 'Location','southeast');
axis square;
title('Lineáris regresszió abszolút eltéréssel (extrém eset)', FontSize = 14);
xlabel('$$x$$', 'Interpreter', 'latex', 'FontSize', 20);
ylabel('$$y$$', 'Interpreter', 'latex', 'FontSize', 20);
hold off;
```