Gépi tanulás

egyetemi jegyzet

Bolgár Bence

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék
2021. augusztus

Tartalomjegyzék

Előszó

Ez a jegyzet a BME Gépi tanulás c. tárgyához készül. Elsősorban a 2020/21. tanév őszi félévének előadásaira épül, amelyeket a járvány miatt távoktatásban voltunk kénytelenek tartani. Az anyag feldolgozásához a BSc-s analízis, lineáris algebra és valószínűségszámítás tárgyak ismerete szükséges. Az elmélet tárgyalása során bayesi szemléletmódot követünk, amelybe a klasszikus és modern eredmények, algoritmusok egyaránt illeszkednek; áttekintünk egészen friss kutatásokat is. Mindazonáltal fontos megjegyezni, hogy ez nem egy *deep learning* kurzus, és bár megismerkedünk a neurális hálózatok alapjaival és módszereivel, a mélytanuló architektúrákat – néhány kivételtől eltekintve – nem részletezzük. A gyakorlatokhoz Python-t fogunk használni (a notebook-ok megtalálhatók a tárgy tanszéki honlapján), viszont a jegyzetben a példák és algoritmusok Julia nyelven szerepelnek. Azért döntöttünk így, mert amellett, hogy ez a nyelv kitűnően alkalmas a gépi tanulási számítások megvalósítására, a matematikai formalizmushoz is közel áll. Minden kód változtatás nélkül futtatható – olyannyira, hogy a jegyzetben szereplő ábrák is közvetlenül ezen kódrészletek hívásával jönnek létre a pdf generálása során. Kérdéseket, visszajelzéseket örömmel fogadok a tanszéki e-mail címemen.

Bolgár Bence
Budapest
2021. augusztus 31.

1. fejezet

Bayesi valószínűségelmélet

Ebben a fejezetben megismerkedünk a Bayes-tétellel, amely nagyon sok gépi tanulási módszer alapját képezi – így az itt megismert ötletek, számítások a későbbi fejezetekben is gyakran vissza fognak köszönni. Különböző eloszlásokkal, közelítésekkel és egyszerűsítésekkel, de végső soron mindvégig az lesz a cél, hogy a megfigyelt adatokból ismeretlen mennyiségekre következtessünk, prediktív modelleket állítsunk fel.

1.1 A bayesi gondolkodásmód

Kezdésnek vegyünk egy egyszerű példát. Legyen egy érménk, amellyel fejet (jelölje mondjuk 1) vagy írást (0) lehet dobni. A célunk az, hogy sok-sok megfigyelt dobás alapján megpróbáljuk kikövetkeztetni, hogy az érme mekkora valószínűséggel dob egyiket vagy másikat. Jelölje y_i az i-edik megfigyelésünket, θ pedig az említett ismeretlen valószínűséget, amelyre most az érme *paramétereként* fogunk gondolni. Mindezt a valószínűségszámítás nyelvén a következőképpen írhatjuk[1]:

$$p\left(y_i = 1 \mid \theta\right) = \theta,$$

azaz a fej dobásának valószínűsége a θ paraméter ismeretében megegyezik θ-val (nyilván, hiszen éppen így határoztuk meg a paraméter jelentését). Rögtön felírhatjuk az írásra vonatkozó valószínűséget is:

$$p\left(y_i = 0 \mid \theta\right) = 1 - \theta.$$

[1] Ha ez a fajta felírás esetleg nem volna ismerős, javasoljuk a Valószínűségszámítás c. tárgyban tanultak átfutását.

A két mennyiséget egyben is kezelhetjük, így megkapjuk y_i eloszlását[2]:

$$p\left(y_i \mid \theta\right) = \theta^{y_i}(1-\theta)^{1-y_i}, \tag{1.1}$$

[2] Vegyük észre, hogy ha y_i helyére 1-et, vagy 0-t helyettesítünk, éppen az előbbi formulákat kapjuk vissza.

amit *Bernoulli-eloszlásnak* nevezünk. Mivel θ becsléséhez az összes megfigyelést fel szeretnénk használni (jelöljük egyben $\mathbf{y}$-nal), felírjuk ezek együttes eloszlását:

$$\begin{aligned} p\left(\mathbf{y} \mid \theta\right) &= \prod_i p\left(y_i \mid \theta\right) \\ &= \prod_i \theta^{y_i}(1-\theta)^{1-y_i} \\ &= \theta^{\sum_i y_i}(1-\theta)^{\sum_i 1-y_i} \\ &:= \theta^h(1-\theta)^t, \end{aligned} \tag{1.2}$$

ahol feltettük, hogy a megfigyelések függetlenek és azonos eloszlásúak[3] – azaz helyesen jártunk el, amikor az egyes megfigyelésekre vonatkozó eloszlásokat összeszoroztuk. A h-val jelölt mennyiség egyszerűen a dobott fejek száma, t pedig az írásoké.

[3] Angol szakkifejezéssel *IID*, „independent and identically distributed''.

1.1.1 *Maximum likelihood (ML) becslés*

Hogyan állapíthatjuk meg ezek alapján θ értékét? Roppant egyszerűen – olyan θ-t keresünk, ami a legjobban megmagyarázza a megfigyeléseket, más szóval olyat, ami a maximalizálja a $p\left(\mathbf{y} \mid \theta\right)$ valószínűséget[4]. Ezt a függvényt, sőt, általában a

$$p\left(\text{megfigyelések} \mid \text{paraméterek}\right)$$

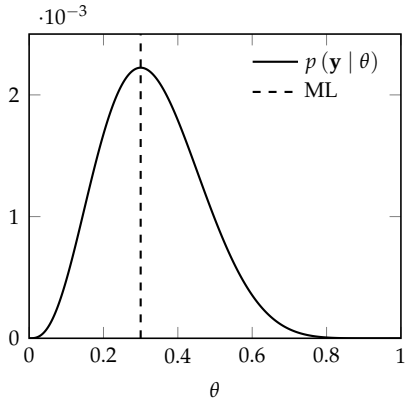
alakú függvényeket *likelihood*-nak nevezzük, ha a paraméter függvényeként gondolunk rájuk. A likelihood-ra és az ML becslésre az 1.1. ábrán láthatunk példát.

[4] Így gondolkodunk: „Ezzel a θ-val valószínűleg ilyesmi megfigyeléseket látnánk, míg egy másikkal ez kevésbé valószínű''.

A likelihood maximalizálásához először negatív logaritmust veszünk, mivel ez gyakran kényelmesebbé teszi a feladat megoldását[5]; a logaritmus szigorúan monoton, úgyhogy nem csaltunk, a szélsőértékhelyek éppen ugyanott lesznek, mint az eredeti függvénynél. A negatív előjel miatt maximalizálás helyett minimalizálási feladatot oldunk meg:

$$L(\theta) := -\ln p\left(\mathbf{y} \mid \theta\right) = -h\ln\theta - t\ln(1-\theta). \tag{1.3}$$

[5] Ez nem csak a számolások egyszerűsödését jelenti, hanem numerikusan is sokkal kedvezőbb – számítógépen sok apró valószínűség összeszorzásakor könnyen beleütközhetünk a számábrázolás korlátaiba, míg logaritmusokkal számolva csupán összegezni kell, ahol ilyesmi nemigen fordul elő.



1.1. ábra. Likelihood függvény és maximum likelihood becslés $h = 3$ és $t = 7$ mellett. A θ paraméter legvalószínűbb értéke $\frac{3}{10}$-nek adódik.

A negatív log-likelihoodra gondolhatunk egyfajta veszteségfüggvényként (*loss*) is, hiszen minél kisebb az értéke, annál „jobb'' a becslésünk θ-ra. Az (1.3) egyenlet jobb oldalán látható veszteségfüggvényt *keresztentrópiának* nevezzük. Minimalizálásához a jól ismert módszert használjuk: deriválunk, majd a deriváltat zérusra állítva megvizsgáljuk a lehetséges szélsőértékhelyeket. Lépésről lépésre

$$\frac{\partial L}{\partial \theta} = -\frac{h}{\theta} + \frac{t}{1-\theta} = 0,$$

amelyet átrendezve a következő – nem túl meglepő – megállapításra jutunk:

$$\theta = \frac{h}{h+t}.$$

A θ-ra vonatkozó maximum likelihood becslés tehát a fejek aránya az összes dobáson belül.

1.1.2 *A Bayes-tétel*

A maximum likelihood becslés jól működik, ha sok megfigyeléssel dolgozunk, kevés adatnál viszont már nem annyira. Meggondolatlanság volna például egyetlen írás dobásából arra következtetni, hogy az érme 1 valószínűséggel írást dob. A valóságban általában van egyfajta előzetes hiedelmünk arról, hogy egy érme milyen valószínűségek mentén „működik'', milyen θ értékeket tartunk hihetőnek.

Bár formálisan nem lesz nagy a változás, valójában itt a valószínűségeknek egy alapvetően más értelmezésére fogunk átváltani (persze a valószínűség továbbra is egy egzakt matematikai definícióval bír, de nem mindegy, hogy hogyan gondolunk rá). Szokatlan lehet például „hiedelmekről'' hallani, szemben az előző szakasz kombinatorikus, $\frac{\text{hasznos eset}}{\text{összes eset}}$-jellegű θ-ja után.

A hétköznapokban a valószínűség tényleg inkább egyfajta szubjektív hiedelmet tükröz. Gondoljunk például arra a mondatra, hogy „holnap 80% eséllyel esni fog az eső''. Ez a kombinatorikus-frekventista értelmezésben valami olyasmit jelentene, hogy Dr. Strange módjára „ötször előre mentünk az időben, és ebből négyszer esett''. A hiedelmek formális kezelésére ad módot a bayesi valószínűségelmélet, amellyel az induktív következtetés, tanulás is megragadható.

Az előzetes hiedelmeinket a *prior* írja le, amely egy eloszlás θ felett, még mielőtt az adatokat láttuk volna:

$$p(\theta).$$

Arra, hogy az előzetes hiedelmeinket (prior) és a megfigyeléseinket (likelihood) szintetizáljuk, a Bayes-tétel ad lehetőséget:

$$p\left(\theta \mid \mathbf{y}\right) = \frac{p\left(\mathbf{y} \mid \theta\right) p\left(\theta\right)}{p\left(\mathbf{y}\right)}.$$

Észrevehetjük, hogy a nevezőben nem szerepel θ, tehát az előbbi egyenletet arányosság erejéig felírva[6]

$$\underbrace{p\left(\theta \mid \mathbf{y}\right)}_{\text{poszterior}} \propto \underbrace{p\left(\mathbf{y} \mid \theta\right)}_{\text{likelihood}} \underbrace{p\left(\theta\right)}_{\text{prior}}. \tag{1.4}$$

[6] A „$\propto$'' szimbólum jelentése: konstans szorzótényezőtől eltekintve egyenlő.

Az egyenlet bal oldalán látható *poszterior* képviseli a „frissített'' hiedelmeinket θ-ra vonatkozóan, amelybe tehát már a megfigyelt adatok is beépülnek.

Egyes kutatások szerint ez játszódik ez az emberi agyban is[7]: a világra vonatkozó előzetes hiedelmeinket jelentő *a priori* eloszlást az idegsejtjeink hálózatával alkalmasan reprezentáljuk, majd az érzékszerveinken át beérkező adatokkal frissítjük; azt pedig a Bayes-tétel mondja meg, hogy ennek hogyan kell történnie.

[7] L. Aitchison és M. Lengyel, ''THE HAMILTONIAN BRAIN: EFFICIENT PROBABILISTIC INFERENCE WITH EXCITATORY-INHIBITORY NEURAL CIRCUIT DYNAMICS'', *PLoS Comput Biol*, 12. évf., 12. sz., e1005186., 2016.

A következő kérdés a prior megválasztása. Ezt a nagyon érdekes és mély filozófiai problémát[8] első körben kikerüljük, és olyan priort választunk, amellyel könnyen tudunk számolni. Az (1.4) egyenletben láttuk, hogy szoroznunk kell, tehát válasszunk olyan mennyiséget, ahol a priorban az (1.2) likelihoodhoz hasonlóan θ és $1-\theta$ hatványai szerepelnek, azaz legyen a priorunk béta-eloszlású α és β *hiperparaméterekkel*[9]:

$$p(\theta \mid \alpha, \beta) = \mathcal{B}eta(\theta \mid \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1},$$

ahol az első, bonyolultnak tűnő tag csupán a normalizációt szolgálja (a görbe alatti terület így lesz 1, amit egy eloszlástól elvárunk).

Most már mindent ismerünk a poszterior kiszámolásához. Az (1.4) egyenlet alapján

$$\begin{aligned} p(\theta \mid \mathbf{y}, \alpha, \beta) &\propto p(\mathbf{y} \mid \theta) \cdot p(\theta \mid \alpha, \beta) \\ &= \theta^h(1-\theta)^t \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1} \cdot \text{const.} \\ &= \theta^{h+\alpha-1}(1-\theta)^{t+\beta-1} \cdot \text{const.} \\ &\propto \mathcal{B}eta(\theta \mid \alpha+h, \beta+t), \end{aligned} \tag{1.5}$$

azaz arra jutottunk, hogy a poszteriorunk is béta-eloszlású[10]! Az efféle prioriokat – nevezetesen, ahol a poszterior is ugyanezen eloszlást követi, csak más paraméterrekkel – *konjugált prioroknak* nevezzük, azaz például mondhatjuk, hogy a béta-eloszlás konjugált prior a Bernoulli-eloszlásra nézve. A frissítés ezekben az esetekben általában egyszerű, nálunk például

$$\mathcal{B}eta(\theta \mid \alpha, \beta) \rightsquigarrow \mathcal{B}eta(\theta \mid \alpha+h, \beta+t),$$

ahol érdekes felfedezést tehetünk: az előzetes hiedelmeinket az α és β „virtuális esetszámok'' kódolják, amelyekhez hozzáadódnak a valóban megfigyelt esetszámok. Ezt az eljárást mutatja be az 1.1 algoritmus.

```
function posterior_Beta_Bernoulli(y; α=1, β=1)
    h = sum(y)
    t = length(y)-h
    return Beta(α+h,β+t)
end
```

1.1. algoritmus. A poszterior eloszlás kiszámolása az **y** megfigyelések és az α, β hiperparaméterek mellett.

[8] S. Rathmanner és M. Hutter, ''A Philosophical Treatise of Universal Induction'', *Entropy*, 13. évf., 6. sz., 1076–1136. old., 2011.

[9] A hiperparaméterek olyan paraméterek, amelyeket mindvégig változatlanul hagyunk; gondolhatunk rá úgy, mint a gépi tanulási módszerünk bemenetére, amelyekkel a tanulást szabályozzuk.

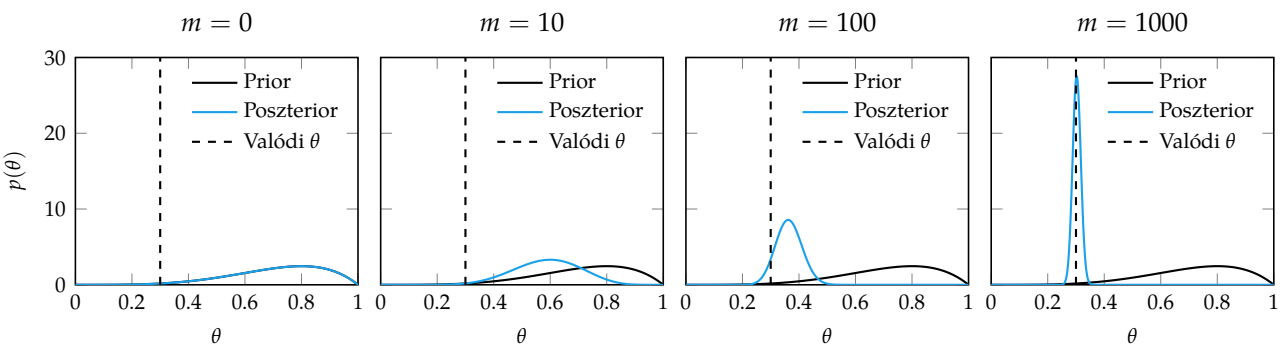
[10] Itt valójában kihasználtuk azt, hogy az ilyen alakú kifejezésekhez tartozó normalizációs konstanst ismerjük. De nem is kell foglalkoznunk vele: az egyenlet elején egy eloszlás szerepel, a végén pedig azt kaptuk, hogy arányosság erejéig egyenlő egy béta-eloszlással; de mivel ezek mind eloszlások, ez csak egyenlősséggel teljesülhet.

Vegyük észre, hogy most sokkal „okosabb'' objektumot kaptunk: θ poszterior eloszlását, ami θ bizonytalanságáról is hordoz információt. Ha ezzel nem szeretnénk foglalkozni, és az ML becsléshez hasonlóan csak egy pontbecslés érdekel minket, könnyen gyárthatunk ilyet, ha a likelihood helyett az (1.5) poszteriort maximalizáljuk (*maximum a posteriori, MAP*); ez már nem csak a megfigyeléseket, hanem az előzetes hiedelmeinket is tartalmazni fogja. A korábbi eljárást megismételve[11]

$$\theta_{\text{MAP}} = \frac{\alpha + h - 1}{\alpha + h + \beta + t - 2}.$$

[11] Próbáljuk meg önállóan levezetni: vegyük az (1.5) kifejezés negatív logaritmusát, a deriváltat állítsuk zérusra és oldjuk meg az egyenletet.

Az 1.2 ábrán a megfigyelések számának hatását láthatjuk a prior és poszterior viszonyára. Kevés adatnál a poszteriort a prior dominálja, a modell sokkal inkább az előzetes hiedelmekre hagyatkozik, mint a megfigyelésekre (éppen ezért fontos a jó prior választása!). Sok adatnál a prior kevés szerephez jut, a poszteriort az adatok dominálják. Látjuk, hogy a bayesi következtetés egyfajta egyensúlyt képvisel a prior és az adat között – de ne feledjük, hogy emögött egzakt matematikai megfontolások álltak; úgy is mondhatjuk, hogy bizonytalan környezetben így „kell'' helyesen következtetni.



1.2. ábra. Prior és poszterior eloszlások m megfigyelés esetén $\alpha = 5$, $\beta = 2$ priorral. A szemléltetés érdekében szándékosan „rossz'' priort választottunk.

Mindenesetre túlzás volna azt állítani, hogy a bayesi valószínűségelmélettel – vagy legalábbis, ahogyan mi használtuk – az egész emberi induktív következtetést egy csapásra formalizáltuk volna. A gépi tanulásban a prior megválasztása rendszerint csak egy technikai eszköz, amellyel a paramétereket a kívánt régió-

ba kényszeríthetjük, nem pedig a háttértudás pontos megragadásának eszköze (emlékezzünk vissza, hogy a könnyű kezelhetőség volt az elsődleges szempont).

1.1.3 *Teljesen bayesi következtetés*

A korábbiakban még mindig nem használtuk ki a bayesi megközelítés teljes erejét. Általában nem is igazán θ érdekel minket, hanem egy $y_{\text{új}}$ értéket szeretnénk jósolni, azaz a kérdésünk

> Az előzetes hiedelmeket és az adatokat figyelembe véve mekkora a valószínűsége annak, hogy a következő alkalommal pl. fejet dobunk?

Az erre vonatkozó *prediktív eloszlást* úgy kapjuk, hogy – a teljes valószínűség tételét felhasználva – θ minden lehetséges értéke szerint összegzünk, megsúlyozva azok poszterior valószínűségével:

$$\begin{aligned} p\left(y_{\text{új}} = 1 \mid \mathbf{y}, \alpha, \beta\right) &= \int p\left(y_{\text{új}} = 1 \mid \theta\right) p\left(\theta \mid \mathbf{y}, \alpha, \beta\right) d\theta \\ &= \int \theta \cdot \mathcal{B}eta\left(\theta \mid \alpha + h, \beta + t\right) d\theta \\ &= \mathbb{E}_{\mathcal{B}eta(\theta \mid \alpha+h, \beta+t)}\left[\theta\right] \\ &= \frac{\alpha + h}{\alpha + h + \beta + t} \end{aligned}$$

Ezt az eljárást nevezzük *bayesi modellátlagolásnak* is; a prediktív eloszlás kiszámításához minden lehetséges modellt figyelembe vettünk, „jókat'' és „rosszakat'' egyaránt, ám a „rossz'' (valószínűtlen) θ-k alacsony súllyal szerepelnek. Az utolsó egyenlethez egyszerűen kikerestük a béta-eloszlás várható értékét korábbi jegyzeteinkből[12].

[12] Vagy Wikipediáról.

1.1.4 *MAP becslés nem konjugált priorokkal*

A maximum a posteriori becslésnél erős feltevéssel éltünk a prior alakját illetően, nevezetesen a könnyű számolhatóságra törekedtünk. Általánosabb esetben a MAP becslés bonyolultabb, mert könnyen olyan egyenletre jutatunk, amelyet nem tudunk zárt formában megoldani. Nézzünk erre egy példát. Legyen a priorunk ún. Kumaraswamy-eloszlású:

$$p\left(\theta \mid \alpha, \beta\right) = \alpha \cdot \beta \cdot \theta^{\alpha - 1} \cdot (1 - \theta^{\alpha})^{\beta - 1},$$

a likelihood-unk változatlanul

$$p(\mathbf{y} \mid \theta) = \theta^h (1-\theta)^t,$$

a poszteriorunk pedig a kettő szorzata:

$$\begin{aligned} p(\theta \mid \mathbf{y}, \alpha, \beta) &\propto p(\mathbf{y} \mid \theta) \cdot p(\theta \mid \alpha, \beta) \\ &= \alpha \cdot \beta \cdot \theta^{\alpha-1} \cdot (1-\theta^\alpha)^{\beta-1} \theta^h (1-\theta)^t \end{aligned}$$

A maximalizáláshoz ismét vesszük a negatív logaritmust, amelyre veszteségfüggvényként gondolunk:

$$L(\theta) = -\ln\alpha - \ln\beta - (\alpha - 1 + h)\ln\theta - t\ln(1-\theta) - (\beta-1)\ln(1-\theta^\alpha)$$

Majd vesszük a deriváltat:

$$\frac{\partial L}{\partial \theta} = -\frac{\alpha - 1 + h}{\theta} + \frac{t}{1-\theta} + (\beta-1)\frac{\alpha\theta^{\alpha-1}}{1-\theta^\alpha}.$$

Sajnos ennek az kifejezésnek a zérushelyét nem tudjuk zárt formában felírni, így a Newton–Raphson módszerhez folyamodunk (A függelék), és numerikusan keressük a megoldást. Ehhez szükség lesz a kifejezés deriváltjára:

$$\frac{\partial^2 L}{\partial \theta^2} = \frac{\alpha - 1 + h}{\theta^2} + \frac{t}{(1-\theta)^2} + (\beta-1)\alpha \cdot \left[\frac{(\alpha-1)\theta^{\alpha-2}}{1-\theta^\alpha} + \frac{\alpha\theta^{2(\alpha-1)}}{(1-\theta^\alpha)^2} \right],$$

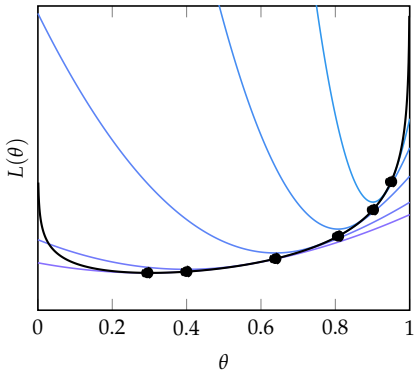
majd a **??**. algoritmussal kiszámoljuk a zérushelyet (1.3. ábra).

1.2 *Naiv bayesi osztályozók*

Készen állunk, hogy az első, gyakorlatban is használt gépi tanulási modellünket összeállítsuk. A naiv bayesi osztályozókat többosztályos feladatokban használjuk, azaz míg a pénzérménél két lehetséges kimenetel – ha úgy tetszik, két osztály – volt, most $y_i \in \{1, 2, \ldots, K\}$ értékű lehet[13]. Ezeket az y_i változókat mostantól *osztálycímkéknek* (*label*) fogjuk nevezni. A korábbiaktól eltérően a modellünknek bemenete is lesz: ezek a minták, amelyeket osztályokba szeretnénk sorolni, és $\mathbf{x}_i \in \mathbb{R}^D$ valós vektorok formáját öltik[14]. Az *ellenőrzött* vagy *felügyelt* gépi tanulási módszerek a következőképpen működnek:

[13] Klasszikus példa a kézzel írott számjegyek felismerése.

[14] Ez a felállás, nevezetesen, hogy a mintáink sokdimenziós valós vektorok, nagyon gyakori a gépi tanulásban. A legtöbb adatot (képek, szöveges adatok, orvosi leletek stb.) ilyen formában kapjuk meg, vagy ilyen alakra hozzuk. Ebben a jegyzetben is szinte kizárólag is vekotros adatokkal fogunk dolgozni.



1.3. ábra. A veszteségfüggvény minimalizálása Newton–Raphson módszerrel. A $\theta_0 = 0.95$ pontból indulva néhány lépésben elérjük a minimumot; az egyes lépésekben a veszteségfüggvény (fekete) másodrendű becslései kék parabolákként ábrázolódnak.

1. Tanítási fázis (*training*): az algoritmusnak ismert $(\mathbf{x}_i, y_i)$ párokat mutatunk, az pedig megpróbálja valamiképpen felismerni az összefüggéseket, mintázatokat keresni az adatokban,

2. Tesztelés fázis (*testing*): az így nyert tudást felhasználjuk egy új, ismeretlen címkéjű $\mathbf{x}$ minta besorolására (megfelelő y jóslására).

Nézzük, hogy történhet ez a gyakorlatban. Annak a valószínűségét, hogy az $\mathbf{x}_i$ mintát a k. osztályba soroljuk, a Bayes-tétel felhasználásával a következőképpen írhatjuk:

$$p\left(y_i = k \mid \mathbf{x}_i\right) = \frac{p\left(\mathbf{x}_i \mid y_i = k\right) p(y_i = k)}{p(\mathbf{x}_i)} \propto p\left(\mathbf{x}_i \mid y_i = k\right) p(y_i = k). \qquad (1.6)$$

Minden $\mathbf{x}_i$ mintára a legvalószínűbb osztályt keressük[15]. Mivel a nevező független k-tól (mindegyik osztályra ugyanaz), ezt akár el is hagyhatjuk, a döntésünket nem fogja befolyásolni. A számlálóban szereplő mennyiségek közül $p(y_i = k)$-t könnyen becsülhetjük a tanító adathalmazban a k. osztályba eső minták arányával. Nehezebb a helyzet a $p\left(\mathbf{x}_i \mid y_i = k\right)$ valószínűséggel, ahol a következő feltételezésekkel élünk:

[15] Azaz itt is egy MAP becslésre fogjuk kihozni a dolgot.

- Az egyes koordináták függetlenek egymástól[16], azaz

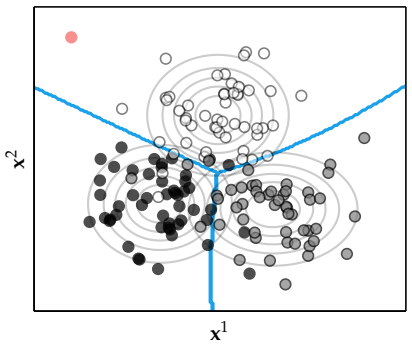
$$p\left(\mathbf{x}_i \mid y_i = k\right) \approx p\left(x_i^1 \;\middle|\; y_i = k\right) \cdot p\left(x_i^2 \;\middle|\; y_i = k\right) \cdot p\left(x_i^3 \;\middle|\; y_i = k\right) \cdots,$$

[16] Innen származik a módszercsalád neve is, hiszen ez egy igen naiv feltevés.

- A koordináták minden osztály esetében normális eloszlást követnek μ_k várható értékkel és σ_k^2 szórással:

$$p\left(x_i^l \,\middle|\, y_i = k\right) = \mathcal{N}\left(x_i^l \,\middle|\, \mu_k, \sigma_k^2\right).$$

A várható értékekre és szórásokra gondolhatunk úgy, mint a k. osztályba tartozó mintahalmaz „középpontjára'' és „kiterjedésére'' az egyes koordinátatengelyek mentén (1.4 ábra). Ezek közvetlenül a tanító adatokból kiszámolhatók, amivel a tanulási fázist le is zártuk; a tesztelés során elég az (1.6) egyenlet alapján kiszámolni a poszteriort és az új mintát a legmagasabb értéket kapott osztályba sorolni (lásd az 1.2 algoritmust).



1.4. ábra. Naiv bayesi osztályozó három osztályra. Az egyes osztályokhoz tartozó mintákat pontok, a tanult normális eloszlások szintvonalait fekete ellipszisek ábrázolják. A kék színnel jelölt döntési felület az egyes osztályokhoz tartozó régiókat határolja; ha a tesztelési fázisban egy új mintát szeretnénk osztályba sorolni, a besorolást az dönti el, hogy ezen határ melyik oldalára esik. A piros mintát így a „fehér'' osztályba sorolnánk.

A naiv bayesi osztályozók könnyen általánosíthatók másféle eloszlásokra is[17]; meglepően jól működnek kategorikus adatoknál, illetve kicsi-közepes adatmennyiség esetén. Bár a függetlenségi feltevés miatt jól skálázhatók, nagy dimenziójú, komplex, nagy mennyiségű adatnál (pl. manapság a mélytanulás alkalmazási területein) prediktív teljesítményük általában elmarad az újabb algoritmusokétól.

[17] Multinomiális eloszlással főleg szövegosztályozási feladatokban használatosak; ekkor a minták nagy dimenziós vektorok, amelyek meghatározott kulcsszavak előfordulásának számát kódolják. A módszer naivitása abban nyilvánul meg, hogy az egyes szavakat függetlennek tekinti, ami összefüggő szövegeknél nyilván nem igaz. Ennek ellenére a megközelítés szép sikereket ért el a spamszűrés területén.

```
struct NaiveBayes
    K    # osztályok száma
    D    # dimenziók száma
    μ    # p(x|y) várható értékek
    σ    # p(x|y) szórások
    py   # p(y)   osztályvalószínűségek

    NaiveBayes(K, D) =
        new(K, D, zeros(K,D), ones(K,D), zeros(K))
end

function train!(m::NaiveBayes,X,y)
    for k in 1:m.K
        class_k   = y.==k
        m.μ[k,:] .= mean(X[class_k,:],dims=1)[:]
        m.σ[k,:] .= std(X[class_k,:],dims=1)[:]
        m.py[k]   = mean(class_k)
    end
end

function predict(m::NaiveBayes,X)
    N     = size(X,1)
    lnpxy = zeros(N,m.K)
    for i in 1:N, k in 1:m.K
        pxy = MultivariateNormal(m.μ[k,:],m.σ[k,:])
        lnpxy[i,k] = logpdf(pxy,X[i,:])
    end
    return argmax.(eachrow(lnpxy .+ log.(m.py')))
end
```

1.2. algoritmus. Naiv bayesi algoritmus többosztályos osztályozásra. Figyeljük meg, hogy a predikció során valószínűségek helyett azok logaritmusával számolunk, így elkerülhetjük a numerikus problémákat (pl. számábrázolás). Az adatvektorokat az $\mathbf{X}$ mátrix tartalmazza (sorokként), $\mathbf{y}$ pedig a címkék vektora.

2. fejezet

Lineáris regresszió

Ebben a fejezetben a gépi tanulás, adatelemzés egyik alapeszközével, a lineáris regresszióval foglalkozunk. Az itt előkerülő ötletekre sok más algoritmusnál, például a neurális hálózatoknál is vissza fogunk utalni. A következő szakaszokban az a célunk, hogy az 1. fejezetben megismert eszköztárat alkalmazva valószínűségelméleti keretbe foglaljuk a módszercsaládot, valamint gyakorlatban is használató algoritmusokat adjunk.

2.1 *Valószínűségi modell*

Az 1.2 szakaszhoz hasonlóan olyan adatokkal fogunk dolgozni, amelyek párokban érkeznek: bemenetek és hozzájuk tartozó kimenetek. A korábbiakkal ellentétben azonban a feladat nem osztályozás, hanem *regresszió*, azaz a kimenetek folytonos értékeket vehetnek fel[1]:

$$\mathbf{x}_i \in \mathbb{R}^D,$$
$$y_i \in \mathbb{R}.$$

Lineáris regressziónál a kettő között lineáris függvénykapcsolatot teszünk fel:

$$y_i \approx \mathbf{w}^\top \mathbf{x}_i + b, \tag{2.1}$$

[1] Ilyen feladat például, ha használt autók árának megbecsülésére szeretnénk gépi tanulási modellt felállítani. Az $\mathbf{x}_i$ bemenet az i. használt autó leírását tartalmazza: súly, gyártási év, kilométeróra állása stb., a hozzá tartozó y_i kimenet pedig az autó ára. A célunk a kettő közötti összefüggés megállapítása, amelyet további, korábban nem látott autók árának megbecsülésére használhatunk fel.

ahol $\mathbf{w}$-t *súlyvektornak*, b-t *eltolásnak* nevezzük; ezeket szeretnénk az adatokból valamiképpen megállapítani[2]. Ha megvagyunk, a (2.1) egyenlet felhasználásával, egyszerű behelyettesítéssel már egy tetszőleges új $\mathbf{x}$ mintához tudunk y-t mondani.

[2] Vegyük észre, hogy ez nem más, mint az $y = ax + b$ egyenes egyenletének többdimenziós analógja; másképpen, itt egy hipersík normálvektorát és eltolását keressük.

A későbbi bonyoldalmak elkerülése érdekében alakítsuk át az $\mathbf{x}_i$ bemeneteket. Vegyünk egy $\phi : \mathbb{R}^D \to \mathbb{R}^{D+1}$ függvényt, amely mindössze annyit tesz, hogy egy $\mathbf{x}_i$-hez hozzáfűz egy fix 1-es értéket; mostantól ezzel a módosított $\phi(\mathbf{x})$ bemenettel fogunk számolni. Ahhoz, hogy a skaláris szorzat értelmes legyen, hasonlóképpen $\mathbf{w}$-t is bővítjük. Az extra elem fogja b szerepét játszani:

$$\begin{bmatrix} \mathbf{w}^0 = b \\ \vdots \\ \mathbf{w} \\ \vdots \end{bmatrix}^\top \underbrace{\begin{bmatrix} 1 \\ \vdots \\ \mathbf{x}_i \\ \vdots \end{bmatrix}}_{\phi(\mathbf{x}_i)} \tag{2.2}$$

Mostantól tehát b-vel nem kell külön foglalkoznunk, és a feladat a következő formát ölti:

$$y_i \approx \mathbf{w}^\top \phi(\mathbf{x}_i). \tag{2.3}$$

Ahhoz, hogy a korábbi eszközeinket használni tudjuk, át kell térnünk a valószínűségek nyelvére. Tegyük fel, hogy a (2.3) egyenlet valamekkora ε zajtól eltekintve teljesül[3], a zaj pedig 0 várható értékű, β^{-1} varianciájú normális eloszlást követ:

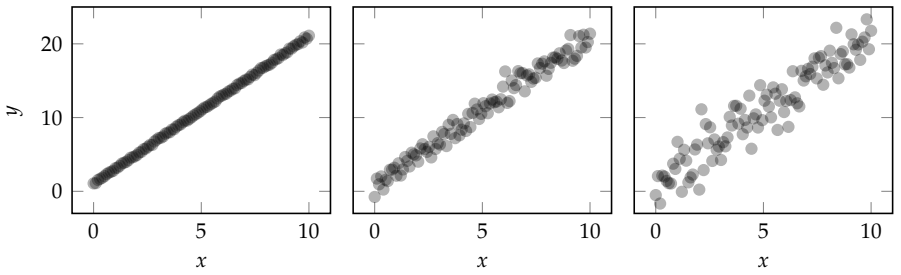
[3] Erre gondolhatunk úgy, hogy y_i értékét nem tudjuk pontosan megismerni, például mérési hiba miatt.

$$\begin{aligned} y_i &= \mathbf{w}^\top \phi(\mathbf{x}_i) + \varepsilon, \\ \varepsilon &\sim \mathcal{N}\left(\varepsilon \,\middle|\, 0, \beta^{-1}\right). \end{aligned}$$

Additív zajról lévén szó, y_i eloszlása is hasonlóan alakul, csupán a várható értéket kell eltolni. Az összefüggés tehát valószínűségi formában így írható[4]:

[4] Emlékezzünk vissza a normális eloszlás képletére.

$$\begin{aligned} p\left(y_i \mid \mathbf{x}_i, \mathbf{w}, \beta\right) &= \mathcal{N}\left(y_i \,\middle|\, \mathbf{w}^\top \phi(\mathbf{x}_i), \beta^{-1}\right) \\ &= \sqrt{\frac{\beta}{2\pi}} e^{-\frac{\beta}{2}\left(y_i - \mathbf{w}^\top \phi(\mathbf{x}_i)\right)^2}. \end{aligned}$$



2.1. ábra. Lineáris függvénykapcsolat egy dimenzióban, különböző mértékű additív zajjal terhelve. A célunk, hogy megtaláljuk az adatokra legjobban illeszkedő egyenest.

Végül, mivel nem egyes mintákat, hanem az összeset egyszerre szeretnénk tekinteni (jelölje $\mathbf{X}$ és $\mathbf{y}$), az előző fejezetben használt függetlenségi feltevésünket megtartva felírhatjuk az együttes eloszlást (likelihoodot):

$$p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \beta\right) = \prod_i \mathcal{N}\left(y_i \;\middle|\; \mathbf{w}^\top \phi(\mathbf{x}_i), \beta\right). \tag{2.4}$$

2.2 *Maximum likelihood megoldás*

Készen állunk arra, hogy maximum likelihood megoldást keressünk, azaz a (2.4) függvényt $\mathbf{w}$ szerint maximalizáljuk. Az előző fejezethez hasonlóan negatív logaritmust véve

$$-\ln p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \beta\right) = \underbrace{\frac{\beta}{2} \sum_i \left(y_i - \mathbf{w}^\top \phi(\mathbf{x}_i)\right)^2}_{L(\mathbf{w})} + \text{const.}, \tag{2.5}$$

ahol a $\mathbf{w}$-t nem tartalmazó tagokat egy konstansba gyűjtöttük, és így megkaptuk az $L(\mathbf{w})$ veszteségfüggvényt, amelynek minimumát keressük. Jobban megnézve ez a veszteségfüggvény nem más, mint a korábbi tanulmányainkból ismert *négyzetes hiba*[5]. A kifejezések egyszerűségének érdekében legyen mostantól

$$\phi_i := \phi(\mathbf{x}_i).$$

Az $L(\mathbf{w})$ veszteség minimalizálásához az ismert módszerhez folyamodunk: gradienst veszünk, ezt zérusra állítva megvizsgáljuk a lehetséges szélsőértékhelyeket:

$$\nabla_{\mathbf{w}} L = -\beta \sum_i \left(y_i - \mathbf{w}^\top \phi_i\right) \phi_i = 0.$$

[5] Szemléletesen, az y_i valóság és a $\mathbf{w}^\top \phi(\mathbf{x}_i)$ becslések közötti eltérést szeretnénk minimalizálni.

Az egyenlet megoldásához szükséges némi mátrix-barkácsolás[6]. Vegyünk egy $\mathbf{\Phi}$ mátrixot, ami soraiban a ϕ_i transzformált mintákat tartalmazza; a mátrixszorzás definíciójára visszaemlékezve észrevehetjük, hogy

$$\underbrace{\begin{bmatrix} \vdots & \vdots & \\ \phi_1 & \phi_2 & \dots \\ \vdots & \vdots & \end{bmatrix}}_{:=\mathbf{\Phi}^\top} \cdot \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \end{bmatrix}}_{\mathbf{y}} = \sum_i y_i \phi_i,$$

amire szükségünk van a gradiens kiszámításához. Hasonlóan adódik, hogy

$$\sum_i \mathbf{w}^\top \phi_i \cdot \phi_i = \mathbf{\Phi}^\top \mathbf{\Phi} \mathbf{w},$$

amivel a gradiens praktikusabb alakra hozható:

$$\nabla_{\mathbf{w}} L = -\beta \left(\mathbf{\Phi}^\top \mathbf{y} - \mathbf{\Phi}^\top \mathbf{\Phi} \mathbf{w} \right) = 0.$$

Végül az egyenletet átrendezve megkapjuk $\mathbf{w}$-t:

$$\mathbf{w} = \left(\mathbf{\Phi}^\top \mathbf{\Phi} \right)^{-1} \mathbf{\Phi}^\top \mathbf{y}, \tag{2.6}$$

amit a predikcióhoz a (2.3) egyenlet mátrixos alakjába helyettesítünk:

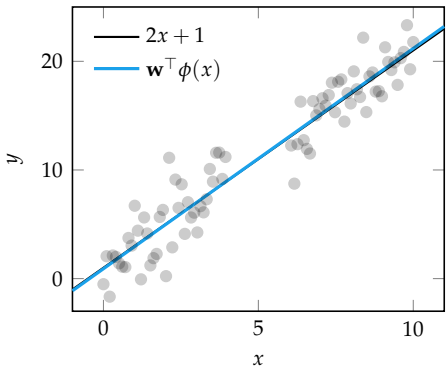
$$\mathbf{y}_{\text{újak}} = \mathbf{\Phi}_{\text{újak}} \mathbf{w}.$$

[6] A legtöbb tudományos kódban, gépi tanulási könyvtárban az effé-le műveletek mátrix-szorzásokként implementálják. Erre a párhuza-mosítás, a számítások hatékonysá-ga miatt van szükség: a grafikus vagy tenzorprocesszorok lényegé-ben ilyen műveletekre vannak op-timalizálva. Óva intünk mindenkit attól, hogy `for` ciklusokat használ-jon ilyen célra, ahogy egyébként az egyenlet sugallaná!

```
struct BasicLinearRegression
    w   # modellsúlyok

    BasicLinearRegression(D) = new(zeros(D))
end
function train!(m::BasicLinearRegression,Φ,y)
    m.w .= (Φ'Φ)\(Φ'y)
end
function predict(m::BasicLinearRegression,Φ_test)
    y_test = Φ_test*m.w
    return y_test
end
```

2.1. algoritmus. Lineáris regresszió. Vegyük észre a „backslash'' operátor használatát a mátrix-invertálás helyett!



2.2. ábra. Lineáris regresszió egy dimenzióban. A valódi függvénykapcsolatot fekete egyenes, a mintákat fekete pontok, az illesztett egyenest kék vonal ábrázolja.

Egy kis kitérő. Gépi tanulásban gyakran találkozunk mátrixok invertálásával, ami $\mathcal{O}(n^3)$ költségű művelet. Az inverzre azonban szinte sosem önmagában van szükségünk, hanem másik mátrixszal vagy vektorral megszorozva. Ezekben az esetekben célszerűbb lineáris egyenletrendszerként gondolni a problémára:

$$\mathbf{A}^{-1}\mathbf{b} = \mathbf{x} \rightsquigarrow \mathbf{A}\mathbf{x} = \mathbf{b},$$

amelyre nagyon hatékony algoritmusaink vannak (minden valamirevaló lineáris algebra csomag tartalmaz ilyen műveleteket). A számításigényes invertálás és az inverz memóriában való tárolása helyett így gyorsabb kódot kapunk.

2.3 *Bázisfüggvények*

Mi a helyzet akkor, ha a függvénykapcsolatunk nem lineáris? Továbbra is egydimenziós mintáknál maradva szeretnénk például polinomokat illeszteni. Egy legfeljebb D-edfokú polinom általánosságban így írható fel:

$$\sum_{d=0}^{D} \mathbf{w}^{(d)} x_i^d = \mathbf{w}^{(0)} + \mathbf{w}^{(1)} x_i + \mathbf{w}^{(2)} x_i^2 + \cdots = \mathbf{w}^\top \phi(x_i),$$

ahol az együtthatókat $\mathbf{w}$ elemei képviselik[7], a ϕ függvény pedig a következőkép-

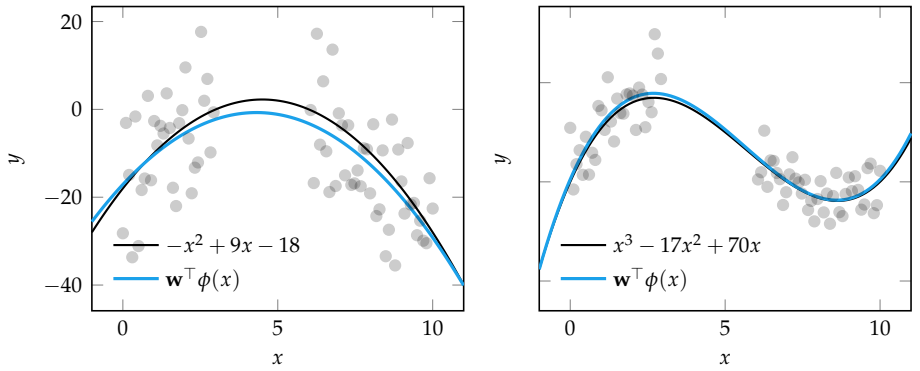
[7] Kivételesen zárójellel jelölve, hogy a hatványozástól meg tudjuk különböztetni.

pen alakítja át a mintáinkat:

$$\phi(x_i) = \begin{bmatrix} 1 \\ x_i \\ x_i^2 \\ \vdots \end{bmatrix}.$$

Ebből kifolyólag a ϕ leképezést, pontosabban ennek koordináta-függvényeit *bázisfüggvényeknek* is szokás nevezni[8]. Ha ebben a formában keressük a maximum likelihood megoldást (azaz továbbra is a négyzetes hibát minimalizáló $\mathbf{w}$ súlyvektort), a korábbi algoritmuson egyáltalán nem kell változtatni, automatikusan a legjobb nemlineáris illesztést kapjuk.

[8] A név adja magát, mivel így y_i-t ezen bázisban írjuk fel, ahol a koordinátákat $\mathbf{w}$ szolgáltatja.



2.3. ábra. Másod- és harmadfokú polinomiális regresszió a ϕ függvény alkalmas választásával. Az algoritmus változatlan, mindössze a minták transzformációját cseréltük le.

2.3.1 *Radiális bázisfüggvények*

Polinomiális regresszió helyett általánosabb nemlineáris illesztést kaphatunk ún. *radiális bázisfüggvények* alkalmazásával. Valójában itt is csupán a ϕ transzformáció ügyes megválasztásáról van szó, de most x_i hatványai helyett b_l „bázispontokat'' tűzünk ki, és minden mintát ezen pontokhoz viszonyított távolságának segítségével reprezentálunk. A leggyakrabban használt radiális bázisfüggvény a Gauss RBF[9], amely egy dimenzióban

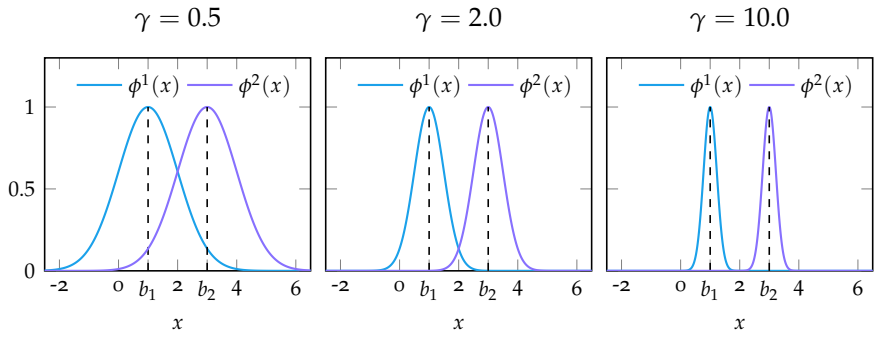
$$\phi^l(x_i) = e^{-\gamma(x_i - b_l)^2},$$

[9] A név onnan ered, hogy a formula nagyon hasonlít a normális eloszlás képletéhez.

ahol b_l az l-edik bázispont. Az x_i mintapont reprezentációja tehát

$$\phi(x_i) = \begin{bmatrix} e^{-\gamma(x_i - b_1)^2} \\ e^{-\gamma(x_i - b_2)^2} \\ \vdots \\ e^{-\gamma(x_i - b_L)^2} \end{bmatrix}.$$

A bázisfüggvény γ paramétere azt befolyásolja, hogy a bázispontok milyen távolra „látnak el'': magas érték esetén x_i reprezentációja csak akkor lesz zérustól eltérő, ha a minta valamelyik bázispont közvetlen közelébe esik, míg alacsony érték esetén a bázisfüggvények átfednek (2.4. ábra).



2.4. ábra. Gauss RBF különböző γ paraméterekkel.

Extrém alacsony értéknél a reprezentációkra

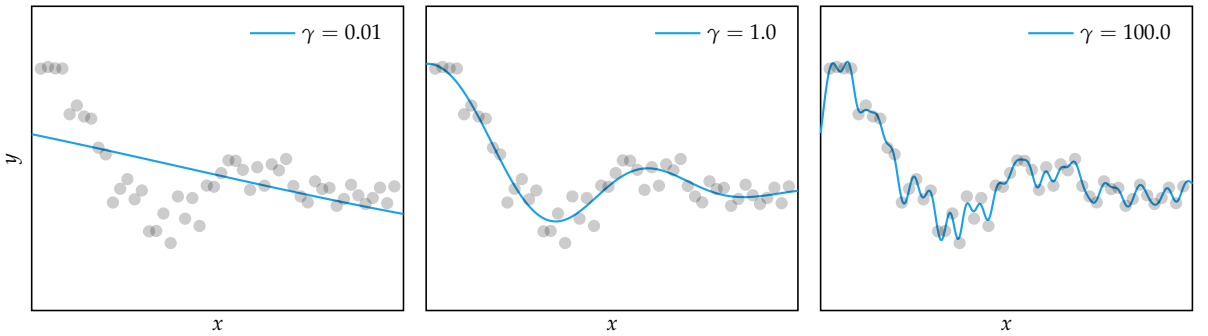
$$\phi(x_i) \approx \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix},$$

azaz a $\mathbf{w}$ együtthatók beállításával tulajdonképpen csak egy konstanst tanulunk[10]. Extrém magas értékeknél, ha például $x_i \approx b_2$, akkor

$$\phi(x_i) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

[10] A szabadsági fokaink száma 1-re redukálódik: bárhogyan is próbálkozunk, az azzal lesz egyenértékű, mintha a $\mathbf{1}^\top \mathbf{w}$ konstanst állítanánk be. A négyzetes hibát minimalizáló konstans nem más, mint az y kimenetek átlaga.

azaz a $\mathbf{w}$ együtthatók felhasználásával minden egyes bázispont „megjegyezhet'' egy-egy y értéket. A 2.5 ábra bemutatja, hogy e két szélsőség között milyen hatással van az illesztésre a γ paraméter.



2.5. ábra. Nemlineáris regresszió Gauss RBF bázisfüggvénnyel. Összesen 100 bázispontot osztottunk el egyenletesen az értelmezési tartományon.

2.4 *MAP becslés és regularizáció*

Az eddigiekben maximum likelihood becslést végeztünk, amely nem vett figyelembe semmiféle előzetes megkötést a $\mathbf{w}$ modellparamétereket illetően. Térjünk vissza a bayesi szemléletmódhoz, és tegyük fel a következő priort[11] $\mathbf{w}$-re:

$$p(\mathbf{w} \mid \alpha) = \mathcal{N}\left(\mathbf{w} \,\middle|\, \mathbf{0}, \alpha^{-1}\boldsymbol{I}\right) = \left(\frac{\alpha}{2\pi}\right)^{\frac{D}{2}} e^{-\frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}}.$$

Ez a prior szemléletesen annyit jelent, hogy a $\mathbf{w}$ súlyvektor elemeit zérus közelében szeretnénk látni. A Bayes-tétel szerint

$$\begin{aligned} p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \beta, \alpha) &\propto p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \beta)\, p(\mathbf{w} \mid \alpha) \\ &\propto e^{-\frac{\beta}{2}\sum_i \left(y_i - \mathbf{w}^\top\phi(\mathbf{x}_i)\right)^2} \cdot e^{-\frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}}, \end{aligned}$$

ahonnan a már megszokott negatív logaritmust véve megismételjük a korábbi számításunkat:

$$-\ln p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \beta, \alpha) = \underbrace{\frac{\beta}{2}\sum_i \left(y_i - \mathbf{w}^\top\phi_i\right)^2 + \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}}_{L(\mathbf{w})} + \text{const.}$$

[11] Ez ismét csak „technikai'' jellegű prior, amellyel nem komplex háttértudásunkat szeretnénk megfogalmazni, hanem – egy később tisztázandó, de jól meghatározott értelemben – arra törekszünk, hogy a modell „egyszerű'' legyen (lásd a 2.5 szakaszt).

A (2.5) egyenletben kiszámolt veszteségfüggvényünk mellé most egy új tag került be. Az efféle tagokat, amelyek tehát arról gondoskodnak, hogy a $\mathbf{w}$ súlyvektor elemei – szebben mondva $\mathbf{w}$ normája – kicsik maradjanak, *regularizációs tagnak*, a súlyvektor normájának csökkentését pedig *regularizációnak* nevezzük. A gradiensünk most

$$-\beta\left(\mathbf{\Phi}^\top\mathbf{y}-\mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{w}\right)+\alpha\mathbf{w}=0,$$

amelyet átrendezve

$$\mathbf{w}=\left(\mathbf{\Phi}^\top\mathbf{\Phi}+\lambda\mathbf{I}\right)^{-1}\mathbf{\Phi}^\top\mathbf{y},$$

ahol $\lambda = \alpha/\beta$. A (2.6) egyenlettel összevetve látjuk, hogy csupán egy apró módosítással kell élnünk, hogy a ML helyett MAP megoldást kapjunk.

A regularizáció hatását legegyszerűbben egy példán keresztül érthetjük meg. Először is módosítsuk az algoritmusunkat:

```
struct LinearRegression
    w   # modellsúlyok
    λ   # regularizáció

    LinearRegression(D; λ=0) = new(zeros(D),λ)
end
function train!(m::LinearRegression,Φ,y)
    m.w .= (Φ'Φ + m.λ*I)\(Φ'y)
end
function predict(m::LinearRegression,Φ_test)
    return Φ_test*m.w
end
```

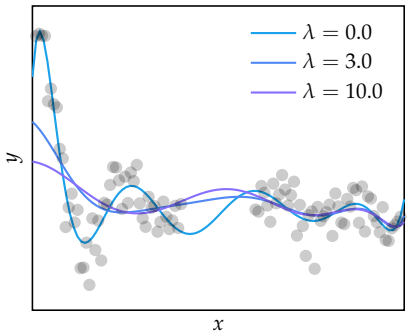
2.2. algoritmus. Lineáris regresszió regularizációval.

A 2.6 ábrán látjuk az illesztések eredményét különböző regularizációs együtthatók mellett. A regularizációval az illesztett függvény oszcillációját tudjuk mérsékelni, amely – mint látni fogjuk – a modell általánosító képességére lesz jótékony hatással[12]. A következő szakaszban ezt vizsgáljuk meg részletesebben.

[12] További kellemes mellékhatás, hogy az egységmátrix λ-szorosának hozzáadásával a problémát numerikusan is kezelhetőbbé tesszük (a sajátértékek eltolásával a $\mathbf{\Phi}^\top\mathbf{\Phi}$ mátrix kondíciószámát csökkentjük).

2.5 *A bias-variancia dilemma*

Egy gépi tanulási modelltől – sőt, valójában bármilyen tudományos modelltől – két dolgot várunk: egyrészt illeszkedjen a megfigyeléseinkhez, másrészt pedig szolgáltasson pontos jóslatokat a további kísérleteket illetően. Kevés hasznát vesszük



2.6. ábra. Lineáris regresszió különböző erősségű regularizációval. A λ paraméter szemléletesen azt befolyásolja, hogy az illesztett függvény mennyire oszcilláljon.

egy olyan modellnek, amely „mindent megmagyaráz, de semmit sem jósol meg''[13]. A bázisfüggvényes megközelítéssel megtehetjük például, hogy minden tanítómintát felveszünk bázispontnak, γ-t pedig kellően nagynak választjuk. Ekkor a modellünk tökéletesen memorizál, reprodukál minden tanító be- és kimenetet, ám a tesztmintákra – hacsak nem esnek nagyon-nagyon közel egy bázisponthoz – nem ad értelmes jóslatot, nem képes általánosítani.

[13] Ahogyan erre Laplace is rámutatott Napóleonnal folytatott – állítólagos – párbeszédében.

Az általánosítóképesség vizsgálatához tekintsük át ismét a modellünket. Egy f függvénykapcsolatot szeretnénk véges sok adatból megbecsülni, amelyeket zérus várható értékű, normális eloszlású zaj terhel:

$$y = f(\mathbf{x}) + \varepsilon,$$
$$\varepsilon \sim \mathcal{N}\left(\varepsilon \,\middle|\, 0, \beta^{-1}\right).$$

Láttuk, hogy lineáris regressziónál az $\hat{f}$ becslés a következő formát ölti:

$$\hat{f}(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}),$$

ahol a ϕ bázisfüggvények gondoskodnak a nemlineáris esetekről. Sajnos a mintahalmaz végessége nehézzé teszi mind a becslést, mind a modell minősítését. A valódi $p(\mathbf{X}, \mathbf{y})$ eloszlást nem ismerjük, csupán egy ebből származó mintahalmaz áll rendelkezésünkre; ennek ellenére általánosságban, a konkrét mintahalmaztól függetlenül kellene minősíteni a modellünket.

Ennek érdekében az elméleti vizsgálódások során az összes lehetséges mintahalmaz feletti várható értékkel fogunk dolgozni. A modell várható négyzetes hibája a következőképpen dekomponálható (a várható érték tehát most az összes

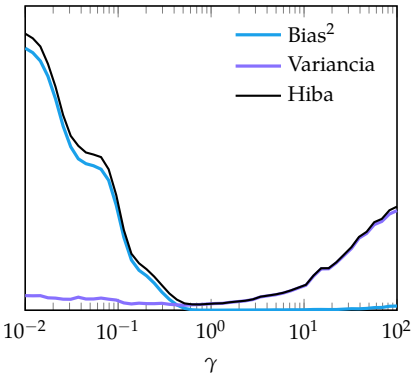
lehetséges mintahalmaz felett értendő):

$$\begin{aligned}
\mathbb{E}\left[(y-\hat{f}(\mathbf{x}))^2\right] &= \mathbb{E}\left[\left(f(\mathbf{x})+\varepsilon-\hat{f}(\mathbf{x})\right)^2\right] \\
&= \mathbb{E}\left[\left(f(\mathbf{x})-\mathbb{E}\left[\hat{f}(\mathbf{x})\right]+\mathbb{E}\left[\hat{f}(\mathbf{x})\right]-\hat{f}(\mathbf{x})+\varepsilon\right)^2\right] \\
&= \left(f(\mathbf{x})-\mathbb{E}\left[\hat{f}(\mathbf{x})\right]\right)^2+\mathbb{E}\left[\left(\mathbb{E}\left[\hat{f}(\mathbf{x})\right]-\hat{f}(\mathbf{x})\right)^2\right]+\mathbb{E}\left[\varepsilon^2\right] \\
&\quad +2\left(f(\mathbf{x})-\mathbb{E}\left[\hat{f}(\mathbf{x})\right]\right)\mathbb{E}\left[\left(\mathbb{E}\left[\hat{f}(\mathbf{x})\right]-\hat{f}(\mathbf{x})\right)\right] \\
&\quad +2\left(f(\mathbf{x})-\mathbb{E}\left[\hat{f}(\mathbf{x})\right]\right)\mathbb{E}\left[\varepsilon\right] \\
&\quad +2\mathbb{E}\left[\varepsilon\left(\mathbb{E}\left[\hat{f}(\mathbf{x})\right]-\hat{f}(\mathbf{x})\right)\right] \\
&= \underbrace{\left(f(\mathbf{x})-\mathbb{E}\left[\hat{f}(\mathbf{x})\right]\right)^2}_{\text{bias}^2}+\underbrace{\mathbb{V}\left[\hat{f}(\mathbf{x})\right]}_{\text{variancia}}+\underbrace{\mathbb{V}\left[\varepsilon\right]}_{\text{zaj}},
\end{aligned}$$

ahol a várható érték tulajdonságai mellett az ε zaj függetlenségét használtuk ki. A tagok értelmezése a következő:

- A bias (torzítás) megmutatja, hogy mekkora az eltérés a valódi függvény és a becslés várható értéke között, azaz mennyire „alkalmas'' az alkalmazott modell az összefüggés megragadására. Nagy bias esetén *alulilleszkedésről* (*underfitting*) beszélünk, a módszerünk túl egyszerű ahhoz, hogy az összefüggést megtanulja (pl. a 2.5. ábra első diagramja).
- A variancia megmutatja, hogy a modell becslései mennyire térnek el egymástól különböző, zajos mintahalmazokon tanítva. Nagy variancia arra enged következtetni, hogy a modell a zajra is ráilleszkedik, azaz *túlilleszkedik* (*túltanul*, *overfitting*), és ebből fakadóan az általánosító képessége gyenge lesz. Erre úgy is gondolhatunk, hogy a modellünk túl komplex (pl. a 2.5. ábra utolsó diagramja).
- A megfigyelési zajjal semmit nem tudunk kezdeni; ez a modellünktől független, mindenképpen jelen van.

A modell komplexitásának befolyásolására általában több eszköz közül is választhatunk. Regressziónál a túlilleszkedés elkerülésére próbálkozhatunk például a regularizációs együttható növelésével (amely a varianciacsökkentés irányába hat) vagy a bázispontok számának csökkentésével.



2.7. ábra. Négyzetes hiba, bias és variancia alakulása egy görbeillesztési feladatban az RBF különböző γ paraméterei mellett. A γ paraméter optimális értéke 0.5 körül található; ez alatt a modell alulilleszkedik (magas bias, alacsony varianca), felette pedig túlilleszkedik (alacsony bias, magas variancia).

2.6 *Bayesi lineáris regresszió*

Végül tekintsük a lineáris regresszió teljesen bayesi változatát. Az előző fejezethez hasonlóan bayesi modellátlagolást kell végezni, azaz a

$$p\left(y_{\text{új}} \mid x_{\text{új}}, \mathbf{X}, \mathbf{y}, \beta, \alpha\right) = \int p\left(y_{\text{új}} \mid x_{\text{új}}, \mathbf{w}, \beta\right) p\left(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \beta, \alpha\right) d\mathbf{w}$$

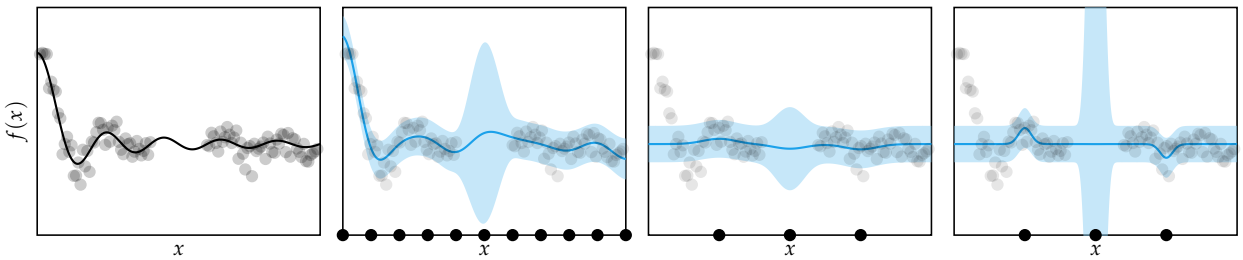
integrált kell megoldani. Ettől most eltekintünk – kiszámolható zárt formában, de sok barkácsolást igényel – és csak a végeredményt közöljük. A $\mathbf{w}$ modellparaméterek poszterior eloszlása normálisnak adódik:

$$\begin{aligned} p\left(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \beta, \alpha\right) &= \mathcal{N}\left(\mathbf{w} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}\right), \\ \boldsymbol{\mu} &= \boldsymbol{\Sigma}^{-1}\left(\beta \boldsymbol{\Phi}^{\top} \mathbf{y}\right), \\ \boldsymbol{\Sigma} &= \beta \boldsymbol{\Phi}^{\top} \boldsymbol{\Phi} + \alpha \mathbf{I}. \end{aligned}$$

Ezek után a normális eloszlás tulajdonságait kihasználva megmutatható, hogy a keresett eloszlás is normálisnak adódik, mégpedig

$$\begin{aligned} p\left(y_{\text{új}} \mid x_{\text{új}}, \mathbf{X}, \mathbf{y}, \beta, \alpha\right) &= \mathcal{N}\left(y_{\text{új}} \,\middle|\, \mu, \sigma^2\right), \\ \mu &= \phi\left(x_{\text{új}}\right)^{\top} \boldsymbol{\mu}, \\ \sigma^2 &= \frac{1}{\beta} + \phi\left(x_{\text{új}}\right)^{\top} \boldsymbol{\Sigma}^{-1} \phi\left(x_{\text{új}}\right) \end{aligned}$$

paraméterekkel. A módszer egy lehetséges megvalósítását mutatja a 2.3 algoritmus. Az eredmények a 2.8 ábrán láthatók.



2.8. ábra. Bayesi lineáris regresszió várható értékkel és szórással. A bázispontok helyét fekete körök jelölik. A bal oldalon a valódi függvény látható, a további ábrákon illesztések más-más γ paraméterrel.

```
struct BayesianLinearRegression
    μ   # várható érték
    Σ   # kovarianciamátrix
    α   # súly prior (hiperparaméter)
    β   # pontosság (hiperparaméter)

    BayesianLinearRegression(D;α=0.1,β=100.0) =
        new(zeros(D),zeros(D,D),α,β)
end
function train!(m::BayesianLinearRegression,Φ,y)
    α,β = m.α,m.β

    m.Σ .= β*Φ'Φ + α*I
    m.μ .= m.Σ\(β*Φ'y)
end
function predict(m::BayesianLinearRegression,Φ_test)
    μ,Σ,β = m.μ,m.Σ,m.β

    pμ  = Φ_test*μ
    pσ² = 1/β .+ sum(Φ_test'.*(Σ\Φ_test'),dims=1)
    return pμ, pσ²
end
```

2.3. algoritmus. Bayesi lineáris regresszió.

3. fejezet

Lineáris klasszifikáció

Az 1.2. szakaszban megismerkedtünk egy egyszerű osztályozóval, a 2. fejezet pedig a lineáris modellek egy családját tárgyalta. Ebben a fejezetben a két ötlet ötvözésével új modellt állítunk össze, amit a gyakorlatban is sokszor használunk, és a neurális hálózatok alapegységeként is gondolhatunk rá.

3.1 *Valószínűségi modell*

A felállás egészen hasonló a regressziós esethez: a bemenetek D-dimenziós vektorok, a kimenetek viszont két osztályt reprezentálnak[1].

$$\mathbf{x}_i \in \mathbb{R}^D,$$
$$y_i \in \{0, 1\}.$$

[1] Egyelőre tehát bináris osztályozásnál maradunk; a többosztályos eseteket nagyobb általánosságban a következő fejezetben tárgyaljuk.

Gondolhatunk például egy orvosi diagnosztikai problémára: az $\mathbf{x}_i$ minta az i. pácienst írja le (például kora, neme, laborleletei stb.), y_i pedig a „egészséges'' vagy „beteg'' állapotot. A célunk, hogy a modellt ismert esetekkel tanítva később egy $\mathbf{x}_{\text{új}}$ páciensről is meg tudjuk állapítani, hogy egészséges vagy beteg. Mi lehet tehát annak a valószínűsége, hogy a páciensünk beteg? A Bayes-tételt felhasználva[2]

[2] A Bayes-tételnek ez az alakja mindössze annyiban különbözik a korábbitól, hogy nevezőben használtuk a teljes valószínűség tételét.

$$p(y_i = 1 \mid \mathbf{x}_i) = \frac{p(\mathbf{x}_i \mid y_i = 1)\, p(y_i = 1)}{p(\mathbf{x}_i \mid y_i = 1)\, p(y_i = 1) + p(\mathbf{x}_i \mid y_i = 0)\, p(y_i = 0)} \tag{3.1}$$
$$= \frac{1}{1 + \frac{p(\mathbf{x}_i \mid y_i = 0) p(y_i = 0)}{p(\mathbf{x}_i \mid y_i = 1) p(y_i = 1)}}. \tag{3.2}$$

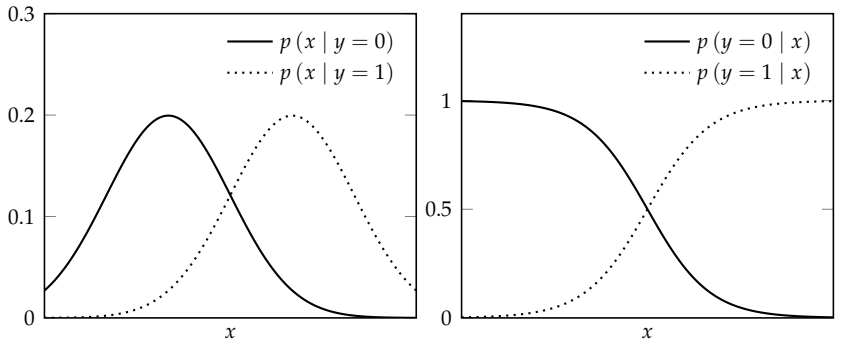
A $p\left(\mathbf{x}_i \mid y_i = 1\right)$-vel jelölt mennyiség a „beteg'' osztályba tartozó minták feltételes eloszlását jelöli (például milyen laborleletekre számítunk egy beteg ember esetében), míg a $p(y_i = 1)$ a betegség *a priori* valószínűsége (például milyen gyakran fordul elő az adott populációban).

A feltételes eloszlások megbecsülése nehéz feladat. Ebben a fejezetben nem is igen vállalkozunk rá, hanem teszünk néhány egyszerűsítő feltevést[3]:

$$p\left(\mathbf{x}_i \mid y_i = 1\right) = \mathcal{N}\left(\mathbf{x}_i \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}\right) = \frac{1}{Z} \cdot e^{-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_1)},$$
$$p\left(\mathbf{x}_i \mid y_i = 0\right) = \mathcal{N}\left(\mathbf{x}_i \mid \boldsymbol{\mu}_0, \boldsymbol{\Sigma}\right) = \frac{1}{Z} \cdot e^{-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_0)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0)},$$

azaz feltesszük, hogy a két osztályhoz tartozó minták normális eloszlást követnek, ráadásul a kovarianciamátrixuk is megegyezik. A Z konstans a normalizációt jelöli, ami a két eloszlásnál ugyanaz. E feltevésünk mellett a (3.1) egyenlet bal és jobb oldalán látható eloszlásokról a 3.1. ábra tájékoztat.

[3] Ez tényleg durva egyszerűsítés. Semmi okunk nincs azt hinni, hogy a feltételes eloszlások valóban ilyen egyszerűek volnának; csupán a könnyű számolhatóságra törekszünk. Később, az 5. fejezetben megvizsgáljuk, hogy hogyan lehet ezeket is az adatokból megbecsülni.



3.1. ábra. Az egyes osztályokhoz tartozó minták feltételes eloszlása és a poszterior valószínűségek egy dimenzióban. A két grafikon között a Bayes-tétel biztosítja az átjárást. Az x minta alacsony értéke esetén valószínűbb, hogy a páciensünk egészséges ($y = 0$); magas értéknél fordított a helyzet.

Végezzük el a (3.2) egyenlet nevezőjében látható osztást. Az $\mathbf{x}_i$-t tartalmazó tagokra

$$\frac{p\left(\mathbf{x}_i \mid y_i = 1\right)}{p\left(\mathbf{x}_i \mid y_i = 0\right)} = e^{-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_0)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0)},$$
$$= e^{\left(\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)\right)^\top \mathbf{x}_i - \frac{1}{2}\left(\boldsymbol{\mu}_1^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_1 - \boldsymbol{\mu}_0^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_0\right)},$$

azaz a négyzetes tagok kiestek, a hatványkitevőben csak $\mathbf{x}_i$-ben lineáris és konstans tagok maradtak. Egy huszárvágással gyűjtsük most össze a lineáris tagok

együtthatóit $\mathbf{w}$-be, az összes $\mathbf{x}_i$-t nem tartalmazó tagot pedig b-be. A keresett valószínűség a következőképpen egyszerűsödik[4]:

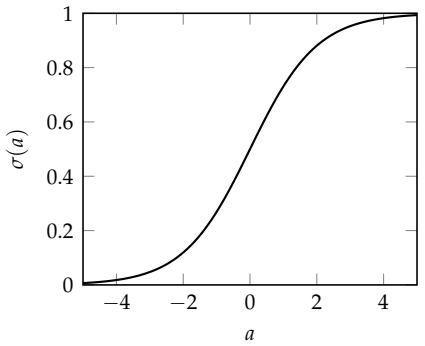
$$p\left(y_i = 1 \mid \mathbf{x}_i\right) = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x}_i + b)}} = \sigma\left(\mathbf{w}^\top \mathbf{x}_i + b\right),$$

ahol a

$$\sigma(a) := \frac{1}{1 + e^{-a}}$$

ún. *szigmoid* függvényt vezettük be.

[4] Úgy is mondhatjuk, hogy eszünk ágában sincs a lineáris és konstans tagokat a fenti bonylult formában számolgatni, sem pedig a kovarianciamátrixok felírásával vacakolni; megelégszünk azzal, hogy *valamilyen* $\mathbf{w}$-re és b-re igaz az állítás. Célravezetőbb $\mathbf{w}$-re és b-re ismeretlen paraméterként gondolni, és például maximum likelihood módon tanulni őket.



3.2. ábra. Szigmoid függvény. Vegyük észre, hogy a függvény a $(0, 1)$ intervallumba képez; szemléletes jelentése az $y = 1$ osztályba tartozás valószínűsége.

Ez már lényegében egy lineáris modell. Ha b-t a lineáris regresszióhoz hasonlóan, a (2.2) egyenletben látott módon hozzáfűzzük a $\mathbf{w}$ súlyvektorhoz, csupán abban térünk el a regressziós esettől, hogy most a skaláris szorzat eredményét egy további σ függvény a $(0, 1)$ intervallumba képezi. Összefoglalva tehát a keresett valószínűségek lineáris/szigmoid összefüggéssel becsülhetők:

$$p\left(y_i = 1 \mid \mathbf{x}_i, \mathbf{w}\right) = \sigma\left(\mathbf{w}^\top \phi(\mathbf{x}_i)\right) := \sigma_i,$$
$$p\left(y_i = 0 \mid \mathbf{x}_i, \mathbf{w}\right) = 1 - \sigma_i.$$

A modell befejezéséhez kövessük az (1.1) egyenletnél alkalmazott stratégiát. Az iménti két esetet egyesítve az y_i kimenet eloszlása Bernoulli-eloszlást követ:

$$p\left(y_i \mid \mathbf{x}_i, \mathbf{w}\right) = \sigma_i^{y_i}(1 - \sigma_i)^{1 - y_i} = \mathcal{B}ern\left(y_i \,\middle|\, \sigma\left(\mathbf{w}^\top \phi(\mathbf{x}_i)\right)\right),$$

a minták között függetlenséget feltéve pedig felírhatjuk a likelihoodot:

$$p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}\right) = \prod_i \sigma_i^{y_i}(1-\sigma_i)^{1-y_i} = \prod_i \mathcal{B}ern\left(y_i \;\middle|\; \sigma\left(\mathbf{w}^\top \phi(\mathbf{x}_i)\right)\right). \tag{3.3}$$

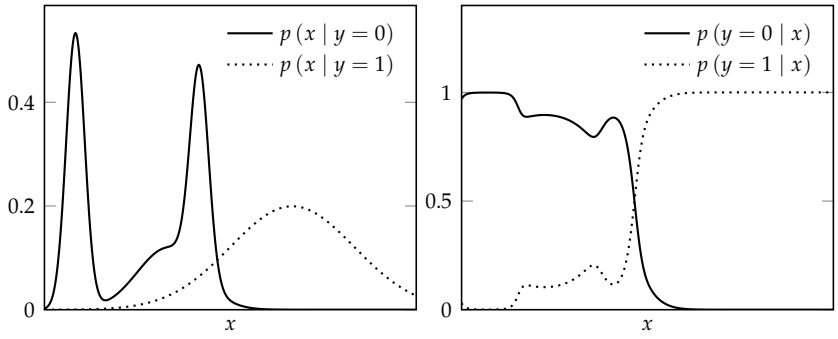
A célunk ismét a likelihood maximalizálása a $\mathbf{w}$ súlyvektor szerint; ennek birtokában könnyedén adhatunk becslést egy új mintára is, csupán a

$$y_{\text{új}} = \begin{cases} 0, & \text{ha } \sigma(\mathbf{w}^\top \phi(\mathbf{x}_{\text{új}})) \leq 0.5 \\ 1, & \text{ha } \sigma(\mathbf{w}^\top \phi(\mathbf{x}_{\text{új}})) > 0.5 \end{cases}$$

formulát kell alkalmaznunk.

Ezt a modellt – kissé félrevezető módon – *logisztikus regressziónak* is szokták hívni, és klasszikus példája az ún. *diszkriminatív modelleknek*. Azzal, hogy a $\mathbf{w}$ súlyvektort közvetlenül az adatból tanuljuk, elveszítettük a lehetőséget, hogy az adatok feltételes eloszlásának várható értékéről, kovarianciájáról bármit megtudjunk, a modell csupán az $p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}\right)$ valószínűségeket próbálja becsülni.

A generatív modellek ezzel szemben a teljes $p(\mathbf{X}, \mathbf{y})$ együttes eloszlást próbálják megtanulni, amivel gyakorlatilag mindent megkapunk: outlier detekciót, osztályozást, mintagenerálást stb., ez azonban sok mintát és számítási erőforrást követel meg. Ha csupán osztályozni szeretnénk, ez pazarló lehet (lásd a 3.3. ábrát).



3.3. ábra. Ha az osztályokba tartozó minták bonyolultabb eloszlást követnek, a poszterior valószínűségekre még mindig „elég jó'' lehet a szigmoid becslés; nem szükséges mintákat és erőforrásokat pazarolni $\mathbf{x}$ feltételes eloszlásainak megtanulására, ha az osztályozást nem befolyásolják. A bal oldali grafikonon lévő fekete „huplik'' például nemigen vannak hatással az osztályozásra, jobb oldalt még mindig „majdnem'' szigmoidokat látunk.

3.2 *Maximum likelihood megoldás*

A maximum likelihood megoldás kiszámolásához a már jól ismert receptet követjük. A veszteségfüggvény a likelihood negatív logaritmusaként adódik:

$$L(\mathbf{w}) = -\ln p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}\right) = -\sum_i y_i \ln \sigma_i + (1 - y_i) \ln(1 - \sigma_i),$$

amelyet *keresztentrópiának* nevezünk.

Ebben a fejezetben az egyszerűség kedvéért a maximum likelihood megoldással foglalkozunk. A teljesen bayesi verziót későbbre halasztjuk, mert bonyoltabb technikákat igényel; a maximum a posteriori becsléshez a lineáris regresszióhoz hasonlóan a

$$p\left(\mathbf{w} \mid \alpha\right) = \mathcal{N}\left(\mathbf{w} \;\middle|\; \mathbf{0}, \alpha^{-1}\mathbf{I}\right)$$

priort tesszük fel, majd a Bayes-tétel alapján

$$p\left(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \alpha\right) \propto p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}\right) p\left(\mathbf{w} \mid \alpha\right),$$

így a veszteségfüggvénybe ismét csak egy regularizációs tag kerül be:

$$L(\mathbf{w}) = -\ln p\left(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \alpha\right) = -\sum_i y_i \ln \sigma_i + (1 - y_i) \ln(1 - \sigma_i) + \frac{\alpha}{2}\mathbf{w}^\top \mathbf{w}.$$

A keresztentrópia minimalizálásához a szükségünk lesz a szigmoid függvény deriváltjára:

$$\sigma'(a) = \frac{-1 \cdot -e^{-a}}{(1 + e^{-a})^2} = \frac{1}{1 + e^{-a}} \cdot \frac{e^{-a}}{1 + e^{-a}} = \sigma(a) \cdot (1 - \sigma(a)),$$

így a gradiens a következő alakot ölti[5]:

$$\begin{aligned}
\nabla_{\mathbf{w}} L &= -\sum_i y_i \frac{1}{\sigma_i} \sigma_i (1 - \sigma_i) \phi_i - (1 - y_i) \frac{1}{1 - \sigma_i} \sigma_i (1 - \sigma_i) \phi_i \\
&= -\sum_i \left(y_i (1 - \sigma_i) - (1 - y_i) \sigma_i\right) \phi_i \\
&= -\sum_i (y_i - \sigma_i) \phi_i.
\end{aligned}$$

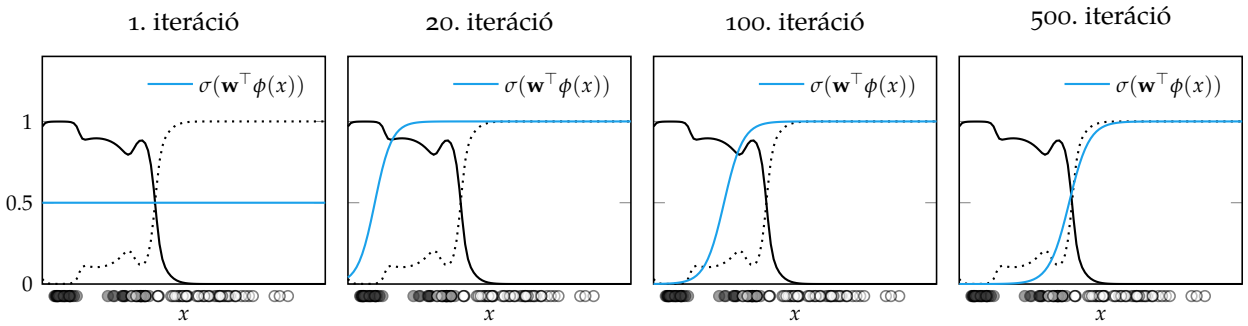
[5] Figyeljük meg, hogy ha a σ_i becslés „egyetért'' az y_i kimenettel (pl. egy 1 címkével rendelkező mintára 0.99-es valószínűséget mond), akkor a gradiens is kicsinek adódik.

Sajnos most nem tehetjük meg azt, amit a regressziónál: hiába állítjuk zérusra a gradienst, nem jutunk olyan egyenlethez, amelyet könnyen meg tudnánk oldani. Ehelyett a veszteségfüggvény minimalizálásához az 1.1.4. szakaszhoz hasonlóan numerikusan próbálkozunk, azaz $\mathbf{w}$-t iteratív módon mindaddig módosítjuk, amíg minimumba nem jutunk. Erre a legegyszerűbb séma, ha mindig a gradienssel ellentétes irányba lépünk, azaz

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L, \tag{3.4}$$

ahol az η a *tanulási tényező* (*learning rate*), amely a lépések nagyságát szabályozza[6].

[6] Ennek belövésére a következő fejezetben látunk stratégiákat.

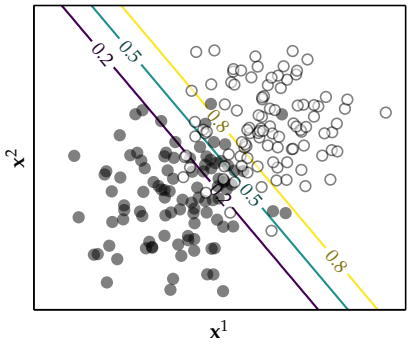


3.4. ábra. Logisztikus regresszió egy dimenzióban. Az x tengelyen a tanítóminták láthatók. A fekete vonalak a két osztály valódi poszterior valószínűségét ábrázolják; a kék vonal a modellből származó szigmoid becslés.

Az egydimenziós esetet a 3.4. ábra, a kétdimenziós esetet a 3.5. ábra, a teljes eljárást a 3.1. algoritmus szemlélteti; a hatékony implementációhoz ismét a műveletek mátrixos alakját használjuk:

$$\begin{aligned} \sum_i (y_i - \sigma_i)\phi_i \quad &\rightsquigarrow \quad \mathbf{\Phi}\,(\mathbf{y} - \boldsymbol{\sigma})\,, \\ \begin{bmatrix} \mathbf{w}^\top \phi(\mathbf{x}_1) \\ \mathbf{w}^\top \phi(\mathbf{x}_2) \\ \vdots \\ \mathbf{w}^\top \phi(\mathbf{x}_N) \end{bmatrix} \quad &\rightsquigarrow \quad \mathbf{\Phi}\mathbf{w}, \end{aligned} \tag{3.5}$$

ahol a $\mathbf{\Phi}$ mátrix soraiban a transzformált $\phi(\mathbf{x}_i)$ adatokat tartalmazza.



3.5. ábra. Logisztikus regresszió két dimenzióban. Az adatokat osztályuktól függően fekete illetve fehér pontok ábrázolják. A szintvonalak a tanult modell kimenetét, azaz a fehér osztály becsült valószínűségét ábrázolják.

```
struct LogisticRegression
    w   # modellsúlyok

    LogisticRegression(D) = new(zeros(D))
end
function train!(m::LogisticRegression, Φ, y; η=0.01, iters=100)
    for i in 1:iters
        s = σ.(Φ * m.w)
        d = y .- s

        m.w .+= η * Φ'd
    end
end
function predict(m::LogisticRegression, Φ_test)
    return σ.(Φ_test * m.w)
end
```

3.1. algoritmus. Logisztikus regresszió gradiens-módszerrel.

3.3 *Bázisfüggvények több dimenzióban*

A 2.3. szakaszban láttuk, hogyan lehet egy lineáris regressziós modellt nemlineárissá tenni a ϕ leképezés módosításával. Az ötlet klasszifikációs modelleknél ugyanúgy működik; ezt nézzük meg általános, többdimenziós esetben. Használjuk a Gauss RBF többváltozós verzióját és az egyszerűség kedvéért válasszuk bázispontoknak magukat a mintapontokat[7]. Ekkor egy minta reprezentációja

[7] Ha így járunk el, a γ paraméter megválasztásánál ügyelnünk kell, hogy ne illeszük alul vagy túl a modellt.

$$\phi(\mathbf{x}_i) = \begin{bmatrix} k(\mathbf{x}_i, \mathbf{x}_1) \\ k(\mathbf{x}_i, \mathbf{x}_2) \\ \vdots \\ k(\mathbf{x}_i, \mathbf{x}_N) \end{bmatrix},$$

a mintákat a sorokban tartalmazó $\mathbf{\Phi}$ mátrix pedig

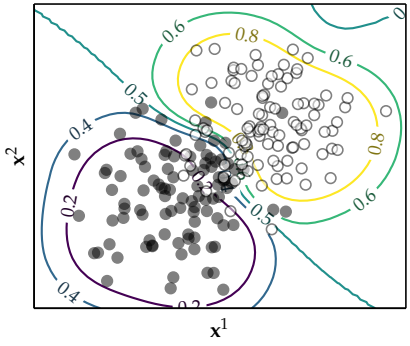
$$\mathbf{\Phi} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & k(\mathbf{x}_N, \mathbf{x}_2) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix},$$

ahol k-t a következőképpen definiáltuk[8]:

[8] $k(\cdot, \cdot)$-ra úgy is gondolhatunk, mint egy függvényre, ami a minták „hasonlóságát'' méri. A hasonlóságokon alapuló tanulás ötletét részletesebben is körbejárjuk a 8. fejezetben.

$$k(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \left\| \mathbf{x}_i - \mathbf{x}_j \right\|^2}.$$

A tanult nemlineáris osztályozót a 3.6. ábra mutatja.



3.6. ábra. Nemlineáris osztályozás logisztikus regresszióval és Gauss RBF bázisfüggvénnyel. A szintvonalak a „fehér'' osztályba tartozás becsült valószínűségét ábrázolják.

3.4 *Az implementáció részletei*

Végül megnézünk néhány gyakorlati szempontot is. Gépi tanulási számításoknál gyakran előfordul, hogy ha csak a tankönyvben olvasott formulákat implementáljuk, a számábrázolás korlátai vagy más numerikus anomáliák miatt az algoritmus rosszul, vagy egyáltalán nem működik[9]. Optimalizációs algoritmusból rengeteget ismerünk; részletesebben a következő fejezetben fogunk foglalkozni velük, most csupán – illusztrációképpen – a Newton–Raphson módszert nézzük meg.

[9] Ez sokszor úgy jelentkezik, hogy az eddig helyesen működő algoritmusunk egyszer csak elkezd `NaN`-okat dobálni.

3.4.1 *Numerikus stabilitás*

Különösen hatványozás esetén kell ügyelnünk a számábrázolás kérdésére. A szigmoid függvényben például találkozunk e-addal is; a túlcsordulás elkerülése érdekében célszerű ennek argumentumát úgy alakítani, hogy a kitevőbe negatív szám kerüljön. A 3.1. táblázat azt mutatja, hogy a szigmoidot, illetve ennek származékait milyen formában érdemes felírni az a argumentumtól függően.

	$\sigma(a)$	$1-\sigma(a)$	$\ln\sigma(a)$	$\ln(1-\sigma(a))$
$a>0$	$\frac{e^{-a}}{1+e^{-a}}$	$\frac{1}{1+e^{-a}}$	$-a-\ln(1+e^{-a})$	$-\ln(1+e^{-a})$
$a\leq 0$	$\frac{1}{1+e^{a}}$	$\frac{e^{a}}{1+e^{a}}$	$-\ln(1+e^{a})$	$a-\ln(1+e^{a})$

3.1. táblázat. Numerikus stabilitás érdekében végzett átalakítások.

A táblázat felhasználásával a keresztentrópiát is „biztonságosabb'' formára hozhatjuk[10]:

$$-y\ln(\sigma(a))-(1-y)\ln(1-\sigma(a))=\begin{cases}a(1-y)+\ln(1+e^{-a}), & \text{ha } a\geq 0\\ -ay+\ln(1+e^{a}) & \text{különben.}\end{cases}$$

[10] A keresztentrópia gépi tanulási könyvtárakban, pl. TensorFlow-ban vagy PyTorch-ban is így van megvalósítva.

```
function σ(a)
    u = exp(-abs(a))
    return a>0 ? 1/(1+u) : u/(1+u)
end
function BCE(a,y)
    u = log1p(exp(-abs(a)))
    return a>0 ? a*(1-y)+u : -a*y+u
end
```

3.2. algoritmus. Numerikusan stabil szigmoid függvény és keresztentrópia.

3.4.2 *Optimalizáció a Newton–Raphson módszerrel*

Másodrendű módszerként a Newton–Raphson algoritmustól[11] gyorsabb konvergenciát remélünk. Egy-egy lépés kiszámolásához a gradiens mellett a Hesse-mátrixra van szükségünk[12]:

$$\nabla_{\mathbf{w}} L = -\sum_i \underbrace{(y_i - \sigma_i)}_{:=d_i} \phi_i = -\mathbf{\Phi}^\top \mathbf{d},$$

$$\nabla^2_{\mathbf{w}} L = \sum_i \underbrace{\sigma_i(1-\sigma_i)}_{:=\sigma'_i} \phi_i \phi_i^\top = \mathbf{\Phi}^\top \mathbf{A} \mathbf{\Phi},$$

ahol $\mathbf{A}$ diagonális mátrix, ami a $\boldsymbol{\sigma}'$ deriváltakat tartalmazza a főátlón:

$$\mathbf{A} = \begin{bmatrix} \sigma'_1 & & 0 \\ & \ddots & \\ 0 & & \sigma'_N \end{bmatrix}$$

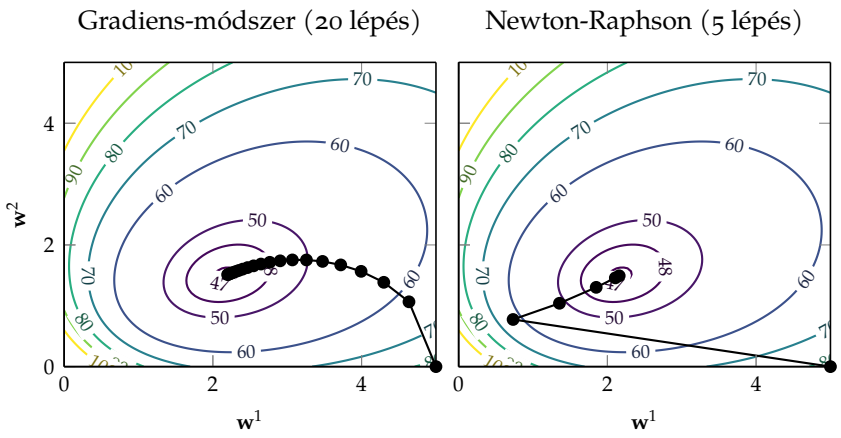
Az iteráció során megtett lépések

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot (\mathbf{\Phi}^\top \mathbf{A} \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{d},$$

ahol az α lépéshosszt többféleképpen is megválaszthatjuk (a 3.3. algoritmus egy egyszerű iránymenti keresési sémát használ).

[11] Lásd az A függeléket.

[12] A Hesse-mátrix elemszáma négyzetesen nő a minták számával, ennélfogva a módszer csak kis-közepes adatmennyiségnél alkalmazható.



3.7. ábra. A gradiens-módszer és a Newton–Raphson eljárás konvergenciája egy kétdimenziós problémában. A veszteséget, mint $\mathbf{w}$ függvényét szintvonalakkal, az algoritmusok lépéseit fekete vonallal jelöltük. Az utóbbi módszer lépései költségesebbek, de az $\begin{bmatrix}5\\0\end{bmatrix}$ pontból indítva jóval kevesebb lépés alatt konvergál az optimális $\mathbf{w}$-hez.

```
function train_newton!(m::LogisticRegression, Φ, y; iters=5)
    for i in 1:iters
        l = Φ * m.w                  # Logit
        s = σ.(l)                    # Szigmoid aktiváció
        d = y .- s

        g = -Φ'd                     # Gradiens
        H = Φ'*( s .* (1 .- s) .* Φ) # Hesse-mátrix
        p = -H\g                     # Új irány

        # Iránymenti keresés visszalépéssel
        α    = 1.
        loss = sum(BCE.(l,y))
        while sum(BCE.(Φ*(m.w .+ α*p),y)) > loss + α*dot(g,p)*1e-4
            α *= 0.5
        end
        m.w .+= α*p
    end
end
```

3.3. algoritmus. Newton–Raphson módszer logisztikus regresszióra iránymenti kereséssel. Az α lépéshossz beállításához először kiszámítjuk a lépés irányát, majd ezen irány mentén – megközelítőleg – minimumot keresünk.

3.5 *Klasszifikációs modellek kiértékelése*

A gépi tanulási munkafolyamatok fontos eleme a modellek prediktív teljesítményének kiértékelése, amelyet a – szándékosan erre a célra fenntartott – teszt adathalmazon végzünk. Gyakran előfordul azonban, hogy nem áll rendelkezésre dedikált teszthalmaz, így ennek előállításáról is magunknak kell gondoskodnunk. Kézenfekvő stratégia a *keresztkiértékelés*, ahol a tesztmintákat a teljes adathalmazról választjuk le, például:

1. Az adathalmazt véletlenszerűen 5 megközelítőleg egyenlő részre vágjuk,
2. Az előbbiek közül 4 rész képezi a tanítóhalmazt, a fennmaradó pedig a teszthalmazt, amelyen a prediktív teljesítményt mérjük[13],
3. A kiértékelést elvégezzük többször oly módon, hogy mindig másik rész játssza a teszthalmaz szerepét (tehát összesen 5×),
4. Az 1-3. lépéseket megismételjük 10×, és összegyűjtjük (vagy átlagoljuk) a kapott teljesítménymetrikákat.

[13] Fontos, hogy az így előállított teszthalmazhoz semmilyen formában nem nyúlhatunk hozzá, az előfeldolgozás során sem! Gyakori hiba például, hogy az adatok normalizálása a szétvágás előtt, a teljes adathalmazt figyelembe véve történik. Ez „csalás'', hiszen így a mintaátlagba a tesztminták is beleszólnak, azaz rejtett módon a teszt adathalmaz tulajdonságait is felhasználjuk a tanítás során.

A következő kérdés a teljesítménymetrikáké. Klasszifikációnál szokás felírni a *konfúziós mátrixot*, amely két osztály esetén a következőképpen fest:

	Jósolt +	Jósolt −
Valódi +	TP	FN
Valódi −	FP	TN

3.2. táblázat. Konfúziós mátrix két osztályra. A TP, FP, TN, FN jelölések rendre a valós pozitív, fals pozitív, valós negatív, fals negatív találatok számát jelentik.

A prediktív teljesítmény kiértékeléséhez használatos metrikákat az előbbi esetszámokból származtatjuk:

$$\text{recall vagy true positive rate (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$
$$\text{false positive rate (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$
$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

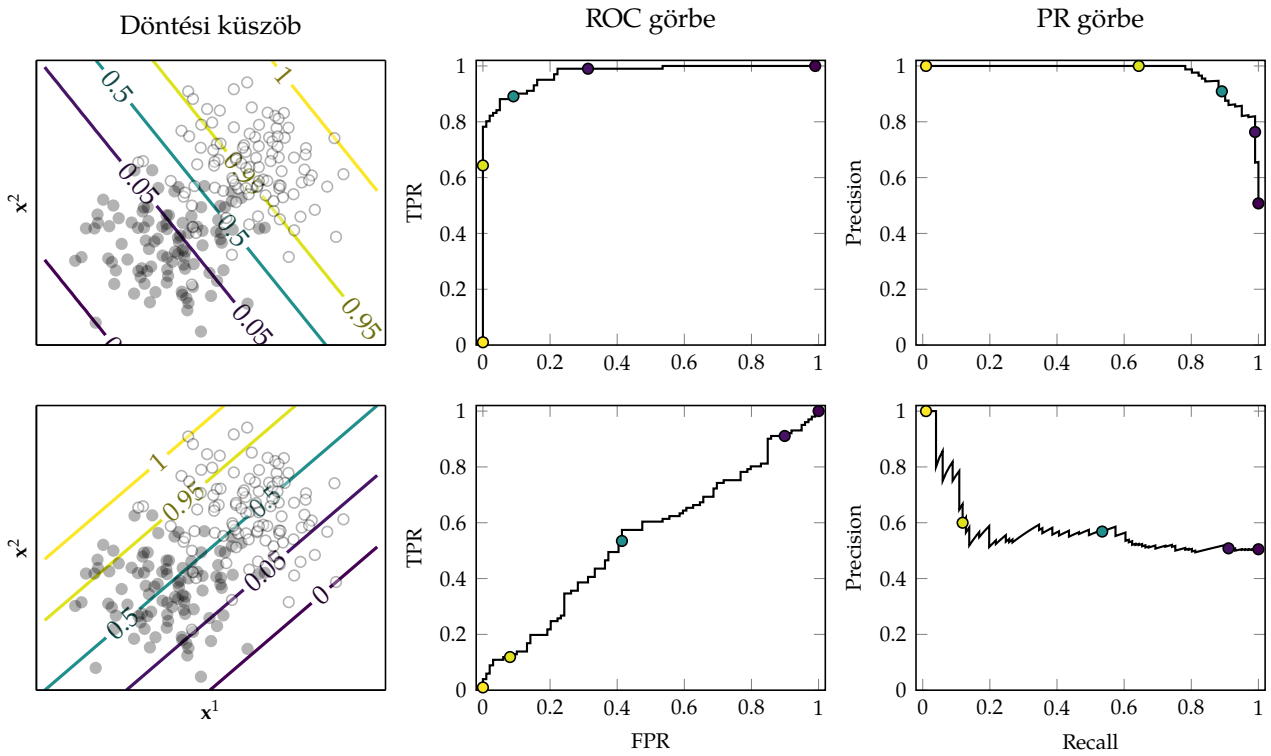
Az így kapott értékek függnek a *döntési küszöbtől*, azaz attól, hogy a szigmoid kimenet mely értékétől kezdve jósoljuk az adott tesztmintát pozitívnak. Ha például a döntési küszöböt zérusnak választjuk, akkor minden tesztmintát automatikusan pozitívnak jósolunk (azaz FN és TN értéke zérus lesz). Ebben az esetben a TPR és az FPR értékek is triviálisan 1-nek adódnak. Hasonlóképpen, ha a döntési küszöböt 1-re állítjuk, akkor mindent negatívnak jósolunk, azaz TPR és FPR is zérus lesz[14]. A szélsőségek között „tologatva'' a döntési küszöböt más-más TPR és FPR értékeket kapunk. Az FPR-TPR közötti összefüggést írja le az ún. *receiver operating characteristic* (*ROC*) görbe; hasonlóképpen származtathatjuk a *precision-recall* (*PR*) görbét is (3.8. ábra.).

[14] A biztonság kedvéért ellenőrizzük le ezeket az állításokat a fenti formulákba való behelyettesítéssel.

A görbék alakjára vonatkozóan – némi gondolkodás után – a következő megfigyeléseket tehetjük:

- Az ROC görbe mindenképpen a $(0, 0)$ pontból indul és az $(1, 1)$ pontba tart,
- Egy tökéletes osztályozó ROC görbéje érinti a $(0, 1)$ pontot, egy „jó'' osztályozóé pedig megközelíti azt[15],
- Egy teljesen véletlenszerűen működő osztályozó ROC görbéje az $y = x$ egyenes mentén, „átlósan'' halad,
- A PR görbe mindenképpen a $(0, 1)$ pontból indul és az $(1, r)$ pontba tart, ahol r elárulja, hogy a teszthalmazban mekkora a pozitívok aránya.

[15] Szokás kiszámolni az ROC görbe alatti területet (AUC), amely egy tökéletes osztályozónál 1-nek adódik, véletlenszerű osztályozónál pedig 0.5-nek; kisebb értékeknél az osztályozó „fordítva'' működik.



3.8. ábra. ROC és PR görbék „jó'' és „rossz'' lineáris klasszifikációs modellekre. A különböző színekkel jelölt döntési küszöbökhöz tartozó pontokat megfelelő színű körök jelzik az ROC és PR görbéken. Látjuk, hogy egy jól működő osztályozó ROC görbéje megközelíti a $(0, 1)$ pontot, míg egy gyakorlatilag véletlenszerűen működő osztályozó ROC görbéje az $x = y$ egyenes közelében marad.

4. fejezet

Neurális hálózatok

Ebben a fejezetben általánosítjuk az előző fejezetben megismert fogalmakat, és megérkezünk az egyik legelterjedtebb gépi tanulási módszercsaládhoz, a neurális hálózatokhoz.

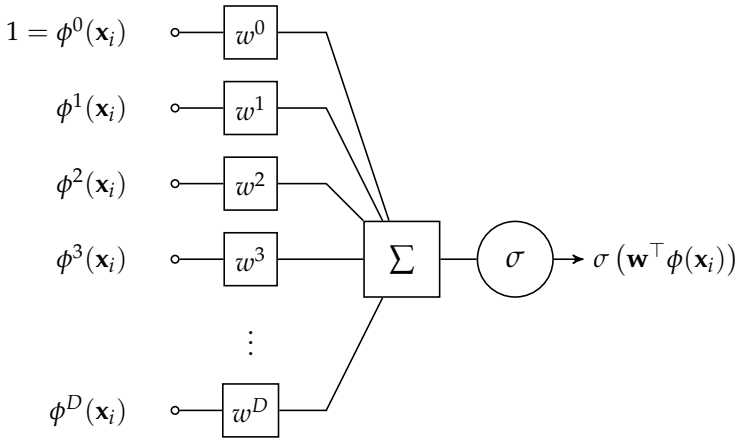
4.1 Logisztikus regresszió és a perceptron

Idézzük fel az előző fejezet osztályozó modelljét, amely egy $\mathbf{x}_i$ bemenetet és a hozzá tartozó y_i kimenetet feltéve a következőképpen festett:

$$p\left(y_i = 1 \mid \mathbf{x}_i, \mathbf{w}\right) = \sigma\left(\mathbf{w}^\top \phi(\mathbf{x}_i)\right).$$

Vizsgáljuk meg, milyen lépéseket végzünk el az egyenlet jobb oldalán látható leképezés során!

1. Az $\mathbf{x}_i$ bemenő mintát valamilyen ϕ függvénnyel transzformáljuk,
2. A kapott vektort megszorozzuk a $\mathbf{w}$ súlyvektorral, azaz $\phi(\mathbf{x}_i)$ és $\mathbf{w}$ elemeit páronként összeszorozzuk, majd a szorzatokat összeadjuk; erre úgy is gondolhatunk, hogy a transzformált bemenet minden egyes eleméhez egy-egy – a későbbiekben beállítható, finomhangolható – súlyt rendelünk,
3. A kapott skalárra alkalmazzuk a σ függvényt, amely a megsúlyozott és összegzett bemenet alapján a végső kimenetet dönti el.



4.1. ábra. Perceptron, mint az agyi információfeldolgozás valószínűségi modellje.

A 4.1. ábra az előbbi lépéseket szemlélteti diagramos formában. A modellt eredetileg az agyi információfeldolgozás formalizálására javasolták, és a hangzatos *perceptron* nevet kapta[1].

Ha jobban megnézzük, a diagram valóban „virtuális idegsejtre'' emlékeztet; a súlyozott bemenő ágak megfelelnek az idegsejt ingerület felvételéért felelős rövid nyúlványainak (*dendritek*), az összegző a *sejttestnek*, ahol az is eldől, hogy az idegsejt milyen választ ad (*szigmoid aktiváció*), a kimenő ág pedig a válasz továbbadásáért felelős hosszú nyúlvány (*axon*) analógja. Az idegsejtműködést vezérlő differenciálegyenletek leírása Hodgkin és Huxley nevéhez fűződik, és mintegy öt évvel megelőzte a perceptront; a párhuzamok mélyebben is megjelennek, például a válasz (*akciós potenciál*) kialakulása során az ioncsatornák nyílása valóban szigmoid karakterisztikát követ. Fontos azonban megjegyezni, hogy ezeknek a neurális modelleknek kevés köze van ahhoz, ahogy az agyi „osztályozás'', például objektumok felismerése történik; célszerű a perceptronra továbbra is statisztikai eszközként tekinteni.

[1] F. Rosenblatt, ''THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN.'', *Psychological Review*, 65. évf., 6. sz., 386–408. old., 1958.

4.2 *Többrétegű neurális hálózatok*

A ma használatos neurális hálózatokhoz a perceptron általánosításain keresztül jutunk el. Először is, a továbbiakban nem fogunk az osztályozáshoz ragaszkodni, és a valószínűségi értelmezést is jórészt elengedjük, így arra sincs különösebb okunk, hogy a szigmoid függvényhez ragaszkodjunk[2]. Jelöljük ehelyett az *aktivációs függvényt* általánosan g-vel, azaz egy neuron a következő számítást fogja megvalósítani:

$$g(\mathbf{w}^\top \phi(\mathbf{x}_i)).$$

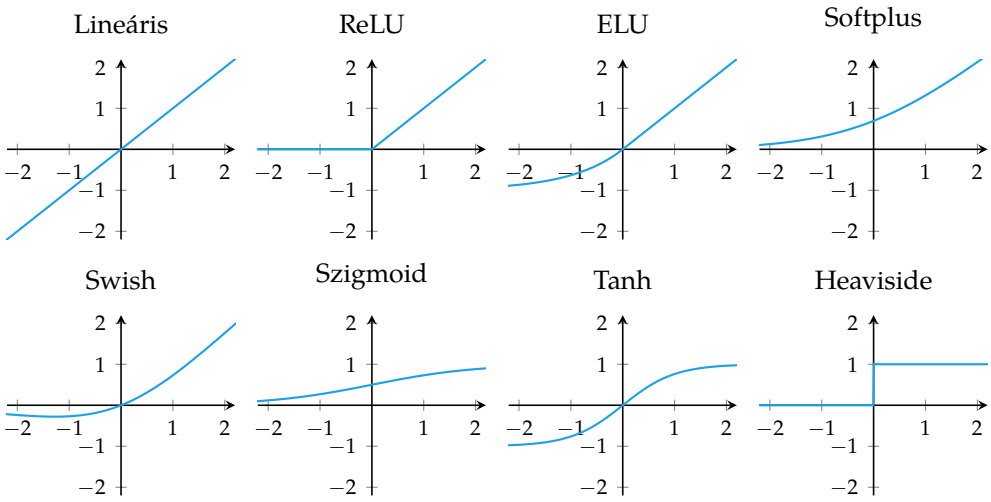
[2] Az aktivációs és veszteségfüggvények alkalmas megválasztását a következő szakaszban tárgyaljuk.

Néhány lehetőség g-re:

```
linear(a)    = a
ReLU(a)      = max(0,a)
ELU(a)       = ifelse(a<0,exp(a)-1,a)
softplus(a)  = log1p(exp(a))
swish(a)     = a/(1+exp(-a))
Heaviside(a) = ifelse(a<0,0,1)
```

4.1. algoritmus. Néhány aktivációs függvény (nem feltétlenül numerikusan stabil implementáció!).

Nézzük meg őket grafikonon is:



4.2. ábra. Gyakran használt aktivációs függvények.

A neuronra (perceptronra) mostantól úgy fogunk gondolni, mint elemi egységre, és elkezdünk bonyolultabb architektúrákat összeépíteni. Azzal kezdjük, hogy több neuront egy *rétegbe* (*layer*) szervezünk, ahol minden egyes neuron ugyanazokat a bemeneteket kapja meg. Írjuk fel az összes bemenő mintát mátrixos alakban soronként, ahogy megszoktuk:

$$\mathbf{\Phi} = \begin{bmatrix} \cdots & \phi(\mathbf{x}_1)^\top & \cdots \\ \cdots & \phi(\mathbf{x}_2)^\top & \cdots \\ & \vdots & \\ \cdots & \phi(\mathbf{x}_N)^\top & \cdots \end{bmatrix},$$

majd a neuronok súlyvektorait is rendezzük mátrixba, de oszloponként:

$$\mathbf{W} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_n \\ \vdots & \vdots & & \vdots \end{bmatrix}.$$

Több neuron kimenetét a (3.5) egyenlethez hasonlóan kapjuk, de ezúttal a réteghez tartozó súlymátrixszal kell szoroznunk:

$$g(\mathbf{\Phi}\mathbf{w}) \rightsquigarrow g(\mathbf{\Phi}\mathbf{W}),$$

ahol g alkalmazása elemenként történik. Az eredmény egy olyan mátrix lesz, ahol a sorok száma megegyezik a bemenő minták számával, az oszlopok száma pedig a neuronok számával; az (i, j)-edik elem az j. neuron válasza az i. bemenetre[3]:

$$\begin{bmatrix} g(\phi(\mathbf{x}_1)^\top \mathbf{w}_1) & g(\phi(\mathbf{x}_1)^\top \mathbf{w}_2) & \cdots & g(\phi(\mathbf{x}_1)^\top \mathbf{w}_L) \\ g(\phi(\mathbf{x}_2)^\top \mathbf{w}_1) & g(\phi(\mathbf{x}_2)^\top \mathbf{w}_2) & \cdots & g(\phi(\mathbf{x}_2)^\top \mathbf{w}_L) \\ \vdots & \vdots & \ddots & \vdots \\ g(\phi(\mathbf{x}_N)^\top \mathbf{w}_1) & g(\phi(\mathbf{x}_N)^\top \mathbf{w}_2) & \cdots & g(\phi(\mathbf{x}_N)^\top \mathbf{w}_L) \end{bmatrix}.$$

[3] Erre úgy is gondolhatunk, hogy a mátrix soraiban továbbra is transzformált mintákat tartalmaz, de a transzformáció most abban áll, hogy az adatot átküldtük még egy rétegnyi neuronon is. A így transzformált minta dimenzionalitása nyilvánvalóan megegyezik a réteg neuronszámával.

Végül kapcsoljunk egymás után több réteget:

$$g^{(3)}(g^{(2)}(g^{(1)}(\mathbf{\Phi}\mathbf{W}^{(1)})\mathbf{W}^{(2)})\mathbf{W}^{(3)}).$$

Itt ügyelnünk kell a dimenziók egyeztetésére a rétegek között: egy réteg kimenetében az oszlopok számának meg kell egyeznie a következő réteg súlymátrixában a sorok számával[4]. Az első (tehát a fenti képletben a legbelső) réteget szokás bemeneti, a legutolsót kimeneti, a közteseket pedig rejtett rétegeknek nevezni.

[4] Másképpen, a következő réteg neuronjainak fogadnia kell az előző rétegből származó „továbbtranszformált'' mintákat.

4.3 *Tanítás hibavisszaterjesztéssel*

A neurális hálózatunk tanításánál a már megszokott módon járunk el:

1. A hálózatnak ismert bemenet-kimenet párokat mutatunk (tanító halmaz),
2. A hálózat kimeneteit összevetjük az elvárt kimenetekkel, ahol az eltérést a veszteségfüggvény méri,
3. A hálózat súlyait gradiens-alapon addig-addig hangoljuk, amíg el nem jutunk a veszteségfüggvény egy minimumához,
4. A tanítás után reménykedünk benne, hogy a hálózat új, ismeretlen mintákra is helyes válaszokat fog adni.

Az eljárás szemléltetéséhez vegyünk egy 100 mintából álló adathalmazt, ahol a minták 10 dimenziósak, valamint három neurális réteget, amelyek rendre 5, 3, 2 neuronból állnak. Jelölje a rétegek sorszámát (l), az egyes rétegek kimenetét $\mathbf{o}^{(l)}$, $\mathbf{o}^{(0)} := \mathbf{\Phi}$ pedig jelölje a bemenetet. Összefoglalva:

l	$\dim(\mathbf{W}^{(l)})$	$\mathbf{o}^{(l)}$	$\dim(\mathbf{o}^{(l)})$
(0)	–	$\mathbf{\Phi}$	100×10
(1)	10×5	$g^{(1)}(\mathbf{o}^{(0)}\mathbf{W}^{(1)})$	100×5
(2)	5×3	$g^{(2)}(\mathbf{o}^{(1)}\mathbf{W}^{(2)})$	100×3
(3)	3×2	$g^{(3)}(\mathbf{o}^{(2)}\mathbf{W}^{(3)})$	100×2

4.1. táblázat. Egy egyszerű neurális architektúra. Vegyük észre, hogy az utolsó réteg két neuront tartalmaz, tehát a megszokott példáinkkal ellentétben a neurális hálózatunknak most több kimenete van.

Vegyük most az L veszteségfüggvényt, azaz[5]

$$L(\mathbf{o}^{(3)}) = L(g^{(3)}(\mathbf{o}^{(2)}\mathbf{W}^{(3)}))$$

[5] Az egyszerűség kedvéért az elvárt kimeneteket kihagytuk a notációból.

a hálózat prediktív teljesítményét minősíti. A veszteségfüggvény gradiense az utolsó réteg súlyai szerint a láncszabály felhasználásával számolható. Vegyük észre, hogy a $\mathbf{W}^{(3)}$ súlymátrix az $\mathbf{o}^{(3)}$ kimenetben van elrejtve, azaz „át kell differenciálnunk'' az L veszteségfüggvényen, a $g^{(3)}$ aktivációs függvényen és az $\mathbf{o}^{(2)}\mathbf{W}^{(3)}$ szorzaton[6]:

$$\underbrace{\nabla_{\mathbf{W}^{(3)}} L}_{3\times 2} = \underbrace{{\mathbf{o}^{(2)}}^{\top}}_{3\times 100} (\underbrace{\nabla_{\mathbf{o}^{(3)}} L}_{100\times 2} \circ \underbrace{{g^{(3)}}'(\mathbf{o}^{(2)}\mathbf{W}^{(3)})}_{100\times 2}),$$

[6] Ellenőrzésképpen megjelenítettük az egyes részeredmények méretét is. Látjuk, hogy a gradiens mérete megegyezik a súlymátrixéval, úgyhogy valószínűleg nem rontottuk el a számolást.

ahol $\circ$ elemenkénti szorzást jelent. A többi súlymátrix szerinti gradienshez csupán a láncszabállyal kell egyre beljebb és beljebb lépkednünk. A $\mathbf{W}^{(2)}$ súlymátrixhoz például $\mathbf{o}^{(2)}$-n keresztül jutunk el, a korábbi deriváltak pedig a láncszabály értelmében szorzótényezőként fognak megjelenni. Sokat egyszerűsít, ha egyes részeredményeket elnevezünk:

$$\delta^{(3)} := \nabla_{\mathbf{o}^{(3)}} L \circ {g^{(3)}}'(\mathbf{o}^{(2)}\mathbf{W}^{(3)}).$$

A $\mathbf{W}^{(2)}$ szerinti gradiens így

$$\underbrace{\nabla_{\mathbf{W}^{(2)}} L}_{5\times 3} = \underbrace{{\mathbf{o}^{(1)}}^{\top}}_{5\times 100} (\underbrace{\delta^{(3)}}_{100\times 2} \underbrace{{\mathbf{W}^{(3)}}^{\top}}_{2\times 3} \circ \underbrace{{g^{(2)}}'(\mathbf{o}^{(1)}\mathbf{W}^{(2)})}_{100\times 3}).$$

Ismét elnevezve

$$\delta^{(2)} := \delta^{(3)} {\mathbf{W}^{(3)}}^{\top} \circ {g^{(2)}}'(\mathbf{o}^{(1)}\mathbf{W}^{(2)})$$

az első réteg súlymátrixa szerinti gradiens[7]

$$\underbrace{\nabla_{\mathbf{W}^{(1)}} L}_{10\times 5} = \underbrace{{\mathbf{o}^{(0)}}^{\top}}_{10\times 100} (\underbrace{\delta^{(2)}}_{100\times 3} \underbrace{{\mathbf{W}^{(2)}}^{\top}}_{3\times 5} \circ \underbrace{{g^{(1)}}'(\mathbf{o}^{(0)}\mathbf{W}^{(1)})}_{100\times 5}).$$

Készen vagyunk az összes réteggel. A súlymátrixok frissítését a legegyszerűbb esetben a (3.4) egyenletben megismert módon végezhetjük, azaz

$$\mathbf{W}^{(n)} \leftarrow \mathbf{W}^{(n)} - \eta \nabla_{\mathbf{W}^{(n)}} L,$$

ahol η továbbra is a tanulási tényezőt jelöli.

[7] Látjuk, hogy a gradienseket a hálózat kimenete felől, a bemenet felé haladva tudjuk kiszámolni a hiba visszafelé „terjesztésével'' (*backpropagation*), ahonnan az algoritmus neve is származik. A 4.8. szakaszban látni fogjuk, hogy a gondolat ennél sokkal általánosabb, valójában visszafelé-módú automatikus differenciálásról van szó.

```
struct MLP
    net
end
MLP(
    Chain(
        Dense(10,5,relu),
        Dense(5,3,relu),
        Dense(3,2)
    )
)
```

4.2. algoritmus. A példában szereplő többrétegű neurális hálózat (multilayer perceptron) megvalósítása.

4.4 *Aktivációs függvények és veszteségfüggvények*

Tegyük fel, hogy regressziót szeretnénk végezni neurális hálózatokkal. Az előző szakaszban tárgyalt tanítási eljáráshoz szükségünk van konkrét aktivációs függvényekre és veszteségfüggvényre. Emlékezzünk vissza a 2. fejezet regressziós modelljeire, ahol valamilyen $\mathbf{x}_i \in \mathbb{R}^D$ bemenetekre és $y_i \in \mathbb{R}$ kimenetekre olyan f függvényt kerestünk, ami bizonyos értelemben jól megragadja a kettő közötti kapcsolatot, azaz

$$y_i \approx f(\mathbf{x}_i).$$

Az f függvény lehetett például lineáris ($\mathbf{w}^\top \mathbf{x}_i + b$), polinomiális ($\mathbf{w}^\top \phi_{poly}(\mathbf{x}_i)$), vagy akár valami sokkal bonyolultabb is. Esetünkben – a későbbi tárgyalásmód megkönnyítése érdekében – gondoljunk f-re úgy, mint egy tetszőlegesen bonyolult neurális hálózatra, ám az utolsó réteg aktivációs függvénye nélkül.

A regressziónál feltettük, hogy az $y_i \approx f(\mathbf{x}_i)$ kapcsolat valamilyen normális eloszlású zajtól eltekintve egyenlőséggel teljesül; majd felírtuk a likelihood-ot, amelyből a veszteségfüggvényt is megkaptuk, nevezetesen

$$p\left(y_i \mid \mathbf{x}_i\right) = \mathcal{N}\left(y_i \,\middle|\, f(\mathbf{x}_i), \beta^{-1}\right) \xrightarrow{-\ln} \frac{\beta}{2}\left(y_i - f(\mathbf{x}_i)\right)^2.$$

A neurális hálózatok nyelvére lefordítva ez annyit tesz, hogy regressziós esetben a veszteségfüggvényt négyzetes hibának választjuk, az utolsó réteg kimenetével pedig nem is kell semmi továbbit kezdenünk[8]. A rejtett rétegek aktivációs függvényeiről még nem szóltunk; itt kevesebb megkötés van, manapság leginkább a *ReLU* függvényt „szokás'' választani.

Nézzük most az osztályozást, ahol az $y_i \in \{0, 1\}$ címkékkel némi valószínűségelméleti barkácsolást követően[9] a σ függvényre jutottunk:

$$p\left(y_i = 1 \mid \mathbf{x}_i\right) = \frac{e^{f(\mathbf{x}_i)}}{1 + e^{f(\mathbf{x}_i)}} = \sigma(f(\mathbf{x}_i)),$$

azaz az utolsó rétegben még egy szigmoid aktivációs függvényt kell használnunk; az így felépített hálózat az $y_i = 1$ osztályba tartozás valószínűségét jósolja. Az $y_i = 1$ és $y_i = 0$ eseteket egybeépítve a likelihood Bernoulli-nak, a veszteségfüggény pedig *bináris keresztentrópiának* adódott:

$$\begin{aligned} p\left(y_i \mid \mathbf{x}_i\right) &= \mathcal{B}ern\left(y_i \mid \sigma(f(\mathbf{x}_i))\right) = \sigma(f(\mathbf{x}_i))^{y_i} + \left(1 - \sigma(f(\mathbf{x}_i))\right)^{1-y_i} \\ &\xrightarrow{-\ln} -y_i \ln \sigma(f(\mathbf{x}_i)) - (1 - y_i) \ln(1 - \sigma(f(\mathbf{x}_i))). \end{aligned}$$

[8] Ez egyenértékű azzal, hogy az utolsó réteg aktivációs függvénye lineáris. A 4.2. ábráról nem is tudnánk mást választani, hiszen f-nek az egész $\mathbb{R}$-be kell képeznie; más esetben előfordulhatna olyan y_i, amit a hálózat soha nem lesz képes „eltalálni''.

[9] Lásd a 3.1. szakaszt.

Hogyan válasszuk meg az aktivációs függvényt, ha többosztályos osztályozást szeretnénk végezni? Legyen $y_i \in \{1, 2, \ldots, K\}$. A korábbiakhoz hasonlóan itt is egy valószínűségi modelltől érdemes elindulnunk:

$$p\left(y_i \mid \mathbf{x}_i\right) = ?$$

Ehhez a kétosztályos gondolatmenetet általánosítjuk K osztályra, és a hálózat kimenetét K méretűre választjuk. Jelölje $f^k(\mathbf{x}_i)$ a kimenet k. komponensét. Ekkor a k. osztályra a 3.1. szakaszban megismert egyszerűsítéssel

$$\begin{aligned} p\left(y_i = k \mid \mathbf{x}_i\right) &= \frac{p\left(y_i = k \mid \mathbf{x}_i\right) p\left(y_i = k\right)}{p\left(y_i = 1 \mid \mathbf{x}_i\right) p\left(y_i = 1\right) + p\left(y_i = 2 \mid \mathbf{x}_i\right) p\left(y_i = 2\right) + \cdots + p\left(y_i = K \mid \mathbf{x}_i\right) p\left(y_i = K\right)} \\ &\text{„}=\text{''} \frac{e^{f^k(\mathbf{x}_i)}}{e^{f^1(\mathbf{x}_i)} + e^{f^2(\mathbf{x}_i)} + \cdots + e^{f^K(\mathbf{x}_i)}} := softmax^k(f(\mathbf{x}_i)), \end{aligned}$$

ahol az ún. *softmax* függvényt definiáltuk:

$$softmax(f(\mathbf{x}_i)) = \begin{bmatrix} \frac{e^{f^1(\mathbf{x}_i)}}{\sum_{j=1}^K e^{f^j(\mathbf{x}_i)}} \\ \frac{e^{f^2(\mathbf{x}_i)}}{\sum_{j=1}^K e^{f^j(\mathbf{x}_i)}} \\ \vdots \\ \frac{e^{f^K(\mathbf{x}_i)}}{\sum_{j=1}^K e^{f^j(\mathbf{x}_i)}} \end{bmatrix}.$$

A softmax kimenetének k. komponense tehát a k. osztályba tartozás valószínűségét jelenti. Természetesen a numerikus stabilitást itt is érdemes szem előtt tartani.

```
function softmax(a)
    m  = maximum(a)
    ea = exp.(a .- m)
    return ea ./ sum(ea)
end
function logsoftmax(a)
    m  = maximum(a)
    ea = exp.(a .- m)
    return a .- log(sum(ea)) .- m
end
```

4.3. algoritmus. Softmax aktivációs függvény és logaritmusának numerikusan stabil implementációja. A túlcsordulás elkerülése érdekében levonjuk a bemenet maximumát, amely a softmax függvény értékén nem változtat (bizonyítsuk be!). Hasonló trükköt alkalmazunk a logaritmus kiszámításakor is.

A kétosztályos esethez hasonlóan a $p\left(y_i = k \mid \mathbf{x}_i\right)$ valószínűségeket egybeépíthetjük, ha Bernoulli helyett a kategorikus eloszlást[10] használjuk:

$$p\left(y_i \mid \mathbf{x}_i\right) = \mathcal{C}at\left(y_i \mid softmax(f(\mathbf{x}_i))\right) = \prod_k softmax^k(f(\mathbf{x}_i))^{y_i^k}.$$

[10] Amit ezen okból kifolyólag tréfásan Multinoulli-nak is szoktak nevezni.

Az y változóra – talán kissé zavart keltő módon – itt úgy gondolunk, mint egy vektorértékű indikátor-változóra, amely csupa zérust és egyetlen 1-est tartalmaz a „megfelelő'' helyen[11], azaz például

$$y_i = 2 \quad \rightsquigarrow \quad y_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

[11] Ezt nevezik *one-hot* kódolásnak.

A veszteségfüggvényt a megszokott módon, a likelihood negatív logaritmusaként kapjuk:

$$-\ln p\left(y_i \mid \mathbf{x}_i\right) = -\sum_k y_i^k \cdot \ln\Big(softmax^k(f(\mathbf{x}_i))\Big) = -\sum_k y_i^k \cdot \left(f^k(\mathbf{x}_i) - \ln \sum_j e^{f^j(\mathbf{x}_i)}\right),$$

és *kategorikus keresztentrópiának* nevezzük.

Érdekes megfigyelni, hogy valójában mit is büntet ez a veszteségfüggvény. A jobb oldalon szereplő `logsumexp` mennyiségről például tudjuk[12], hogy

$$\ln \sum_j e^{f^j(\mathbf{x}_i)} \approx \max_j f^j(\mathbf{x}_i),$$

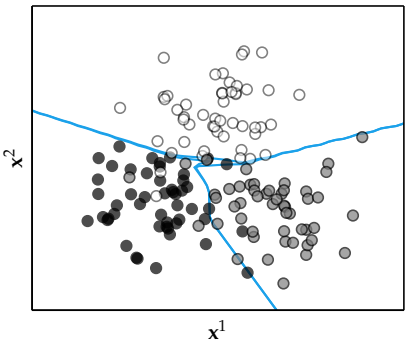
[12] Gondoljunk arra, hogy a szummát az exponenciális függvény miatt a legnagyobb komponens dominálja.

és a

$$\sum_k y_i^k \cdot \Big(\dots\Big)$$

tag pedig „kiválasztja'' a valódi osztályt; minden más osztály a szummában zérus együtthatóval szerepel[13]. A kettőt összevetve arra jutunk, hogy ha a k. az $\mathbf{x}_i$ minta valódi osztálya, és $f^k(\mathbf{x}_i)$ értéke eltér a $\max_j f^j(\mathbf{x}_i)$ kifejezés értékétől, zérusnál nagyobb veszteséget kapunk. Más szóval, a veszteségfüggvény azt bünteti, ha a neurális hálózat kimenetének legnagyobb komponense nem a k-adik.

[13] Továbbra is one-hot kódolás mellett.



4.3. ábra. Háromosztályos osztályozás többrétegű neurális hálózazzal és softmax aktivációs függvénnyel.

4.5 *Regularizáció neurális hálózatokban*

A neurális hálózatok különösen hajlamosak a túlilleszkedésre. A problémát többféleképpen is orvosolhatjuk; a teljesség igénye nélkül néhány technika[14]:

- A neurális architektúra megszorítása (például rétegek számának, méretének csökkentése),
- Korai leállás (a tanítás megállítása, amikor a prediktív teljesítmény egy dedikált *validációs halmazon* mérve már nem csökken),
- Zaj hozzákeverése az általánosítóképesség növelése érdekében,
- A neuronok véletlenszerűen kiválaszott részhalmazainak ki-be kapcsolása (*dropout*),
- A neurális hálózat súlyainak korlátozása (*weight decay*).

[14] Ennek a jegyzetnek nem célja egy általános neurális hálózat/deep learning kurzus anyagának felölelése, úgyhogy a technikákat nem részletezzük.

A súlyok megszorítása a lineáris regresszió regularizált változatához hasonlóan történik. A szemléltetéshez használjunk egyetlen neuront, amelynek súlyvektorára priort teszünk:

$$p\left(y \mid \mathbf{x}, \mathbf{w}\right) = \mathcal{P}\left(y \,\middle|\, f(\mathbf{w}^\top \phi(\mathbf{x}))\right),$$
$$p\left(\mathbf{w} \mid \alpha\right) = \mathcal{N}\left(\mathbf{w} \,\middle|\, \mathbf{0}, \alpha^{-1}\mathbf{I}\right),$$

a poszterior így

$$p\left(\mathbf{w} \mid \mathbf{x}, y, \alpha\right) \propto p\left(y \mid \mathbf{x}, \mathbf{w}\right) \cdot p\left(\mathbf{w} \mid \alpha\right).$$

A veszteségfüggvény a poszterior negatív logaritmusaként adódik[15]:

$$
\begin{aligned}
-\ln p\left(\mathbf{w} \mid \mathbf{x}, y, \alpha\right) &= -\ln p\left(y \mid \mathbf{x}, \mathbf{w}\right) - \ln p\left(\mathbf{w} \mid \alpha\right) \\
&= \underbrace{-\ln p\left(y \mid \mathbf{x}, \mathbf{w}\right)}_{\text{korábbi loss}} + \underbrace{\frac{\alpha}{2}\left\|\mathbf{w}\right\|_2^2}_{\text{regularizáció}},
\end{aligned}
$$

azaz a korábbi veszteségfüggvényünk egy extra regularizációs taggal bővül[16].

Attól függően, hogy milyen priort használunk (a fenti példában normális eloszlásút), a regularizációs tag is más-más formát ölthet. Néhány gyakrabban használt példány:

- $\left\|\cdot\right\|_2$ (L_2-norma, normális eloszlású prior). A leggyakrabban használt regularizációs tag, amely a súlyokat egyenletesen csökkenti.
- $\left\|\cdot\right\|_1$ (L_1-norma, Laplace-eloszlású prior). Akkor érdemes használni, ha ún. *ritka* megoldást keresünk; ez a fajta regularizáció úgy csökkenti a súlyokat, hogy minél több legyen zérus.
- $\left\|\cdot\right\|_\infty$ (L_∞-norma vagy max-norma). Egzotikusabb, kevésbé agresszív regularizáció, amely csak a súlyok legnagyobbikát csökkenti („clipping'').
- Elastic net. Az L_1 és L_2 normák lineáris kombinációja, amely a 2010-es évek közepéig volt igazán népszerű.

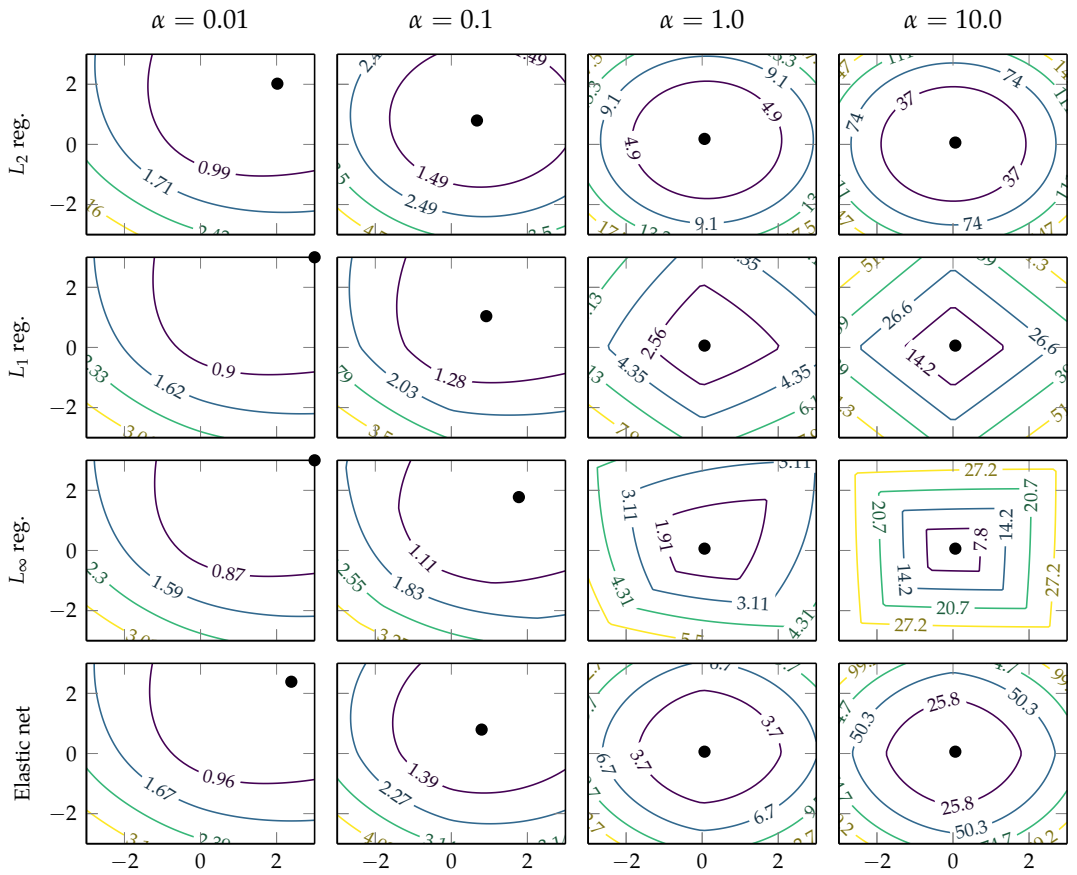
A regularizációs tagok hatását a 4.4. ábra szemlélteti.

[15] Tehát most is MAP megoldást keresünk.

[16] Figyeljük meg, hogy a megoldásnak két némiképp ellentétes célnak kell megfelelnie: egyrészt a veszteséget kell minimalizálni, másrészt a súlyokat kell bizonyos értelemben kicsin tartani. A kettő közötti egyensúlyt az α regularizációs együttható szabályozza.

4.6 Optimalizáció neurális hálózatokban

A korábbi fejezetekben a modelljeink viszonylag egyszerű veszteségfüggvénnyel bírtak; vagy volt analitikus megoldásunk, vagy legalábbis paramétereiben lineáris modellekkel foglalkoztunk. Ha a modell paramétereiben lineáris, a veszteség pedig konvex függvénye a kimenetnek, akkor a veszteségfüggvény, mint a paraméterek függvénye is konvex lesz, amit könnyű minimalizálni. Általában viszont – a több rétegnek köszönhetően – a neurális modellek már a súlyokban sem lineárisak, a veszteségfüggvényük nagyon bonyolult lehet, ami igencsak megnehezíti az optimalizációt.



4.4. ábra. Kétdimenziós súlyvektorral rendelkező regularizált modellek veszteségfüggvénye különböző típusú regularizációs tagok és regularizációs együtthatók mellett. A minimumot fekete pont jelöli. Érdemes megfigyelni, hogy mindegyik tag igyekszik az origó felé „terelni'' súlyvektort, ám mindegyik másképpen.

4.6.1 Nehézségek

A tipikus anomáliák sorba szedéséhez célszerű a veszteségfüggvényt egy pont körül Taylor-sorba fejteni[17]. Jelölje θ a neurális hálózat összes súlyát, ekkor

$$L(\theta) \approx L(\theta_0) + \nabla_\theta L\Big|_{\theta_0} (\theta - \theta_0) + \frac{1}{2} (\theta - \theta_0)^\top \mathbf{H}\Big|_{\theta_0} (\theta - \theta_0) .$$

[17] Feltesszük, hogy a veszteségfüggvény kétszer differenciálható; a sorba fejtés lényegében azt árulja el nekünk, hogy „lokálisan hogyan néz ki'' a veszteségfüggvény.

A felümerülő problémákhoz vizsáljuk meg a $\mathbf{H}$ Hesse-mátrix sajátértékeit.

Lokális minimumok. Ha $\nabla_\theta L = 0$ és $\mathbf{H}$ összes sajátértéke pozitív, lokális minimumban vagyunk. Ez általában nem okoz problémát; a lokális minimumok a gyakorlatban ritkák, és még mindig „elég jók'' ahhoz, hogy kielégítő prediktív teljesítményt kapjunk.

Nyeregpontok. Ha $\nabla_\theta L = 0$ és a $\mathbf{H}$ mátrixnak vannak pozitív és negatív sajátértékei is, ún. nyeregpontban vagyunk. Ezek gyakoribbak a lokális minimumoknál, és bizonyos algoritmusoknak gondot okozhatnak[18]. Ritka azonban az az eset, hogy pontosan egy nyeregpontra érkezve ragadna be az algoritmus (leginkább a súlyok helytelen inicializálásánál fordul elő).

[18] A klasszikus másodrendű módszerek például előszeretettel ragadnak be nyeregpontokba.

Platók. Ha $\nabla_\theta L \approx 0$ és $\mathbf{H}$ sajátértékei is zérus közelében vannak, a veszteségfüggvény lokálisan lapos. Az efféle kiterjedt platókról az optimalizációs algoritmus csak nagyon lassan, vagy egyáltalán nem képes kijutni. Ilyen jelenség léphet fel például, ha mindenütt szigmoid aktivációs függvényt használunk, ám történetesen az összes aktiváció a szigmoid függvény lapos részére kerül[19]. Éppen ezért újabban a rejtett rétegekben ReLU aktivációt szokás használni, amelynek képe zérus felett lineáris, tehát nem lapul el és erős gradiensünk lesz. Az optimalizáció azonban még így is beragadhat, ha túl sok neuron kerül kikapcsolt állapotba (ReLU zérus alatti része).

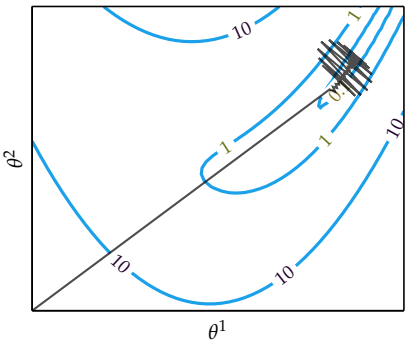
[19] A tankönyvek rendszerint ezt jelölik meg az „eltűnő gradiens'' (*vanishing gradient*) probléma klasszikus okaként, különösen, ha ráadásul négyzetes hibát használunk veszteségfüggvénynek.

Keskeny völgyek, szakadékok. A leggyakrabban felmerülő probléma az, ha $\mathbf{H}$ sajátértékei között sok nagyságrend eltérés van[20]. A mátrix kondíciószáma a

$$\kappa(\mathbf{H}) = \frac{|\lambda_{max}(\mathbf{H})|}{|\lambda_{min}(\mathbf{H})|}$$

[20] Intuitíve arról van szó, hogy bizonyos irányokban a függvény nagyon gyorsan változik, míg más irányokban szinte egyáltalán nem.

hányados; a magas kondíciószámmal rendelkező mátrixot – és a kapcsolódó optimalizálási problémát – rosszul kondicionáltnak nevezzük. A gradiens-alapú módszerek ilyenkor csak rendkívül lassan, a völgy szélei között ide-oda pattogva, oszcillálva képesek haladni.



4.5. ábra. Gradiens-alapú optimalizáció rosszul kondicionált Hesse-mátrix esetén. Az algoritmus oszcilláló mozgásra kényszerül.

A jelenség magyarázata az, hogy a gradiens mindig merőleges a szintvonalakra; ha a völgy túl keskennyé válik, az algoritmus minden lépésben „túlszalad'', a következő lépésben pedig ellenkező irányú korrekcióra kényszerül. A probléma elkerüléséhez választhatunk alacsonyabb tanulási tényezőt, ám ez szintén a tanulást lassítja. Másik megoldás az ún. *batch normalizáció* használata, ami a neurális hálózat összes rétegének kimenetét zérus átlagra és egységnyi varianciára normalizálja, a tanulás során pedig ezeket az extra műveleteket a 4.3. szakaszban látottakhoz hasonlóan visszaterjeszti; a völgyek ekkor jobban navigálhatóvá válnak.

4.6.2 Algoritmusok

Az előző szakaszban tárgyalt problémák elkerüléséhez valamivel bonyolultabb algoritmusokra van szükség. Kézenfekvőnek tűnhet másodrendű módszerek, például a Newton–Raphson módszer többdimenziós analógjainak alkalmazása. A neurális hálózatok azonban kifejezetten akkor hasznosak, ha nagy mennyiségű tanító adat áll rendelkezésre, ez pedig jórészt kizárja a másodrendű módszereket – a Hesse-mátrix tárolása például a minták számában négyzetes költségű[21].

Valójában éles adaton a gradiens-módszer sem használható eddig tárgyalt formájában, hacsak nincs nagyon sok memóriával rendelkező szuperszámítógépünk.

[21] Természetsen léteznek approximatív megoldások (pl. az L-BFGS algoritmus), amelyek időnként neurális hálózatok kapcsán is előkerülnek.

A gyakorlatban a tanító adatokat inkább nem egyszerre, hanem kisebb adagokban (ún. *minibatch-ekben*) adjuk az algoritmusnak, így a tárkomplexitás a töredékére csökken[22]. Mivel a minibatch-ek kiválasztása általában véletlenszerűen történik, a gradiens-módszer ezen variánsát *Stochastic Gradient Descent* (*SGD*) névvel illetik. A minták függetlenségét kihasználva

$$\nabla_\theta L(f(\mathbf{X},\theta),\mathbf{y})) = \nabla_\theta \sum_i L(f(\mathbf{x}_i,\theta),y_i) = \sum_i \nabla_\theta L(f(\mathbf{x}_i,\theta),y_i),$$

azaz a gradiens elméletileg előállítható a minibatch-eken vett gradiensek összegeként. A gyakorlatban minden egyes minibatch-et súlyfrissítés követ; az algoritmus sztochasztikus természetének köszönhetően immár nem mindig a tényleges – teljes adaton számolt – gradiens irányába mozdulunk el, csupán „átlagosan'' haladunk egy minimum felé.

[22] Ez gyakorlatilag elkerülhetetlen, ha például GPU-n szeretnénk tanulni; általában 10-1000 méretű minibatch-ekkel dolgozunk. A minibatch-méret eldöntése általában kísérletezés kérdése, érdemes a hardver optimális kihasználására törekedni

```
struct GradientDescent
    η   # tanulási tényező

    GradientDescent(θ;η=0.001) = new(η)
end
function step!(opt::GradientDescent,θ,∇f)
    return @. θ - opt.η * ∇f
end
```

4.4. algoritmus. Gradiens-módszer egyszerű megvalósítása.

Mind a lokális minimumokból, nyeregpontokból, platókból való kiszabadulásban, mind az oszcillációk visszafogásában segít a *momentum-módszer*, amely „lendületet'' ad az algoritmusnak:

```
struct Momentum
        η   # tanulási tényező
        α   # momentum csillapítás
        v   # momentum vektor

        Momentum(θ;η=0.001,α=0.9) = new(η,α,zeros(length(θ)))
    end
    function step!(opt::Momentum,θ,∇f)
        @. opt.v = opt.α*opt.v + opt.η*∇f
        return @. θ - opt.v
    end
```

4.5. algoritmus. Momentum-módszer. Egy lépés az aktuális gradiens és az előző lépés lineáris kombinációjaként adódik, amely túllendítheti az algoritmust a lokális minimumokon, valamint az oszcillációkat is mérsékli.

A tanulási tényező megválasztása nehéz kérdés[23]: túl kis érték esetén a hálózat nagyon lassan vagy egyáltalán nem tanul, túl nagy érték esetén pedig össze-vissza ugrál a hibafelületen és nem konvergál (különösen a kis minibatch-méretből adódó „zajos'' lépések esetén). A megfelelő érték attól is függ, hogy a tanulás melyik fázisában járunk: az elején még távol vagyunk az optimumtól és megengedhetünk nagyobb ugrásokat, a vége felé pedig remélhetőleg valamilyen optimum közelébe kerültünk, tehát apró lépésekkel kell finomhangolnunk.

Ezt az ötletet használja a tanulási tényező adaptív beállítására az *AdaGrad* algoritmus:

[23] A tanulási tényező beállításának bevált módszere a GSD (Grad Student Descent): bízzuk a munkát egy doktoranduszra, aki addig csavargatja, amíg egyszer csak működik.

```
struct AdaGrad
    η   # tanulási tényező
    G   # akkumulált gradiens

    AdaGrad(θ;η=0.1) = new(η,zeros(length(θ)))
end
function step!(opt::AdaGrad,θ,∇f)
    @. opt.G += ∇f^2
    return @. θ - opt.η/sqrt(opt.G) * ∇f
end
```

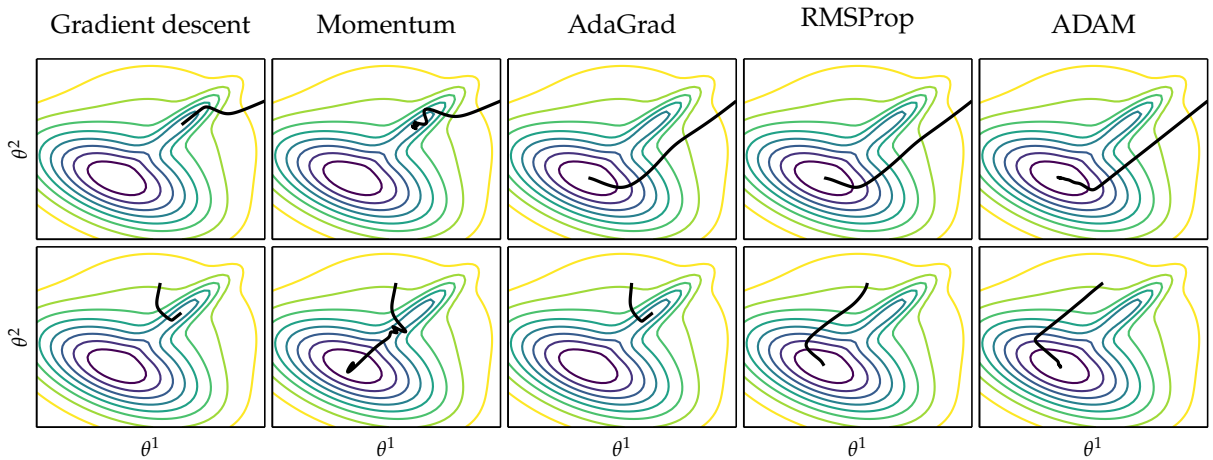
4.6. algoritmus. AdaGrad algoritmus a tanulási tényező adaptív beállításával. Minden iterációnál leosztunk az akkumulálódó gradienssel, a lépések így egyre kisebbek lesznek.

Az AdaGrad algoritmus módosított változata az *RMSProp*, amely valamivel bonyolultabb sémát alkalmaz a tanulási tényező adaptív változtatására:

```
struct RMSProp
    η   # tanulási tényező
    α   # csillapítás
    G   # akkumulált gradiens

    RMSProp(θ;η=0.01,α=0.9) = new(η,α,zeros(length(θ)))
end
function step!(opt::RMSProp,θ,∇f)
    @. opt.G = opt.α*opt.G + (1-opt.α)*∇f^2
    return @. θ - opt.η/sqrt(opt.G) * ∇f
end
```

4.7. algoritmus. Az RMSProp algoritmus exponenciális lecsengetést alkalmaz a gradiens-akkumulációnál.



4.6. ábra. Optimalizációs algoritmusok konvergenciája a veszteségfüggvény szintvonalaival, különböző helyekről indítva.

A korábbi ötleteket ötvözi az *ADAM* algoritmus, amely adaptív tanulási tényezővel, első- és másodrendű momentumbecslésekkel dolgozik:

4.8. algoritmus. Az ADAM algoritmus.

```
mutable struct ADAM
    η   # tanulási tényező
    α   # csillapítás
    β   # csillapítás
    v   # első momentum
    G   # második momentum
    i   # iteráció

    ADAM(θ;η=0.01,α=0.9,β=0.9) =
        new(η,α,β,zeros(length(θ)),zeros(length(θ)),0)
end
function step!(opt::ADAM,θ,∇f)
    @. opt.v = opt.β*opt.v + (1-opt.β)*∇f
    @. opt.G = opt.α*opt.G + (1-opt.α)*∇f^2
    opt.i   += 1
    v        = opt.v ./ (1-opt.α^opt.i)
    G        = opt.G ./ (1-opt.β^opt.i)
    return @. θ - opt.η/sqrt(G) * v
end
```

Végül a 4.9. ábra összefoglalja a tanítás menetét.

```
function train!(mlp::MLP, x, y; epochs=10, batchsize=512)
    ps   = params(mlp.net)
    opt  = ADAM()
    data = DataLoader((x,y),batchsize=batchsize,shuffle=true)

    for epoch in 1:epochs, batch in data
        x_,y_ = batch
        grad = gradient(ps) do
            logitcrossentropy(mlp.net(x_),y_)
        end
        update!(opt, ps, grad)
    end
end
```

4.9. algoritmus. Többosztályos osztályozó neurális hálózat tanítása a `Flux` könyvtár felhasználásával. Az adatokat 512 méretű minibatchekben adagoljuk, a tanításhoz az ADAM algoritmust használjuk. Vegyük észre, hogy a gradienst nem nekünk kell megadni; a 4.8. szakaszban látni fogjuk, hogyan lehet automatikusan kiszámolni. A tanítás így néhány sornyi kóddal megoldható.

4.7 *Konvolúciós neurális hálózatok*

A valós adatelemzési, gépi tanulási problémák egyik legjellemzőbb vonása az, hogy az adat több szabadsági fokkal rendelkezik, mint maga az adatot generáló „valóság''. Egy orvosi adatelemzési feladatban például egy-egy páciens sokdimenziós leírása nem vehet fel akármilyen értékeket; gondoljunk csak a testsúly, koleszterinszint és vérnyomás összefüggéseire. Úgy is mondhatjuk, hogy az adatokat tartalmazó vektortérben az adatok egy alacsonyabb dimenziójú sokaságon helyezkednek el[24], a látott értékek például egy közös okra, a súlyos elhízásra vezethetők vissza.

Hasonló jelenséget látunk képfeldolgozási feladatokban, ahol a szomszédos vagy egymás közelében lévő képpontok színe függ össze, például mert egy kutyát vagy arcot ábrázolnak. Minderről az eddig megismert, *teljesen összekötött*[25] neurális hálózatok mit sem tudnak, hiszen minden neuron részesül a teljes bemenetből. A lokalitás megőrzéséhez tehát olyan neurális hálózatot kell építenünk, ahol egy-egy neuron a képnek csak egy kisebb foltjából kap bemenetet[26]. Ezt a foltot a neuron *receptív mezejének* is nevezik.

A képi adatok további jellemzője a *térbeli invariancia*, azaz az objektumok többékevésbé ugyanúgy festenek, bárhol is tűnnek fel a képen. Ha egy neuron megtanult valamilyen alakzatot felismerni a maga receptív mezejében (például a bal felső sarokban), a jobb alsó sarokért felelős neuron nagyjából ugyanazzal a súlyvektorral ismerheti fel az alakzatot. Más szóval nemcsak, hogy a súlyok nagy része zérus, hanem súlyokon több neuron is osztozhat. Egy-egy réteg súlymátrixa tehát

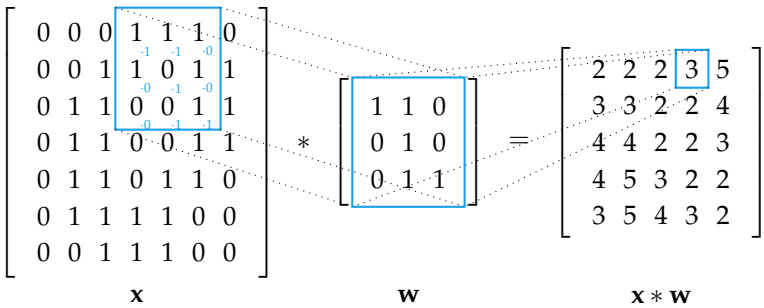
[24] Az efféle látens struktúra felderítésére több módszert is látunk a 10. fejezetben.

[25] Szakkifejezéssel *dense* vagy *fully connected*, ami alatt azt értjük, hogy a következő réteg egy neuronja a megelőző réteg összes neuronjából kap bemenetet

[26] Ennek kellemes következménye, hogy a paraméterszám is radikálisan csökken; a réteghez tartozó súlymátrix ritka, elemeinek többsége zérus.

speciális szerkezettel rendelkezik: egyrészt ritka, másrészt sok ismétlődő elemet tartalmaz[27].

A *konvolúciós neurális hálózat* (*CNN*) ezeket az ötleteket a következőképpen valósítja meg. Az ismétlődő súlyok $K \times K$ méretű mátrix formáját öltik, amelyet *kernelnek* vagy *filternek* nevezünk. Jelöljük $\mathbf{w}$-vel (K tipikusan 3-10 méretű). Ezt a kernelt „toljuk végig'' az $\mathbf{x}$ bemeneten, páronként összeszorozva a kép és a kernel megfelelő pozícióján található értékeket, majd összegzünk, ahogy a neurális hálózatoknál már megszoktuk. A műveletet konvolúciónak nevezzük, és a következő ábra szemlélteti:



4.7. ábra. Konvolúció 3×3 kernelmérettel. Figyeljük meg, hogy alapesetben a kimenet mérete csökken a bemenethez képest.

Formulával[28]

$$(\mathbf{x} * \mathbf{w})^{ij} = \sum_{m=-K}^{K} \sum_{n=-K}^{K} \mathbf{x}^{i+m,j+n} \cdot \mathbf{w}^{mn}$$

Korábbi tanulmányaink során láttunk már hasonlót. Vízszintes irányú élek kiemeléséhez például használható a

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Sobel-kernel. A konvolúciós neurális hálózatok hasonló kerneleket alkalmaznak, egy képi régióra jellemzően több különbözőt is (*csatornák*). A trükk az, hogy a kernelek most egy neurális hálózat súlyrendszerének felelnek meg, tehát gradiens-alapon tanulhatók. A tanítás során a súlyok úgy állnak be, hogy a hálózat megtanulja felismerni az aktuális feladat megoldásához – például képek osztályozásához – fontos képi jegyeket. Az egymás után csatolt konvolúciós

[27] A gyakorlatban ezért a réteg kimenetének számolását nem is a szokásos mátrix-szorzással valósítjuk meg, hiszen az túl pazarló volna.

[28] A kép széleinek kezelése többféleképpen történhet. Ha ugyanakkora képet szeretnénk kapni, mint a bemenet, *padding*-et kell végeznünk, azaz a bemenet széleit virtuálisan szegélyezzük zérussal (*zero padding*), vagy az értékek ismétlésével (*same padding*). Ha kifejezetten csökkenteni akarjuk a képméretet (például kis paraméterszámra törekszünk), a kernelt használhatjuk nagyobb lépésközzel (*stride*).

rétegek gyakran megtanulnak egyre bonyolultabb alakzatokat felismerni; míg a korai rétegek elemi formákat, a későbbiek egyre komplexebb és absztraktabb alakzatokat detektálnak.

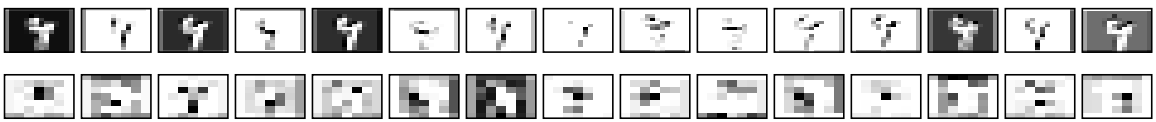
A terminológia itt főleg élettani eredetű. A „receptív mező'' kifejezést eredetileg a retina és a látókéreg idegsejtjeinél használták. A konvolúciós hálók emlékeztetnek arra, ahogy a látórendszer működik; a retina egyfajta előfeldolgozást végez, helyi kontrasztokat érzékel. A látókéreg első rétege egyszerű alakzatokat – különböző orientációjú éleket – ismer fel, további rétegei pedig a korábbiakból építkezve egyre bonyolultabb alakzatokat detektál.

Végül vizsgáljuk meg, hogyan végezhetünk osztályozást vagy regressziót konvolúciós hálózattal. Az ötlet csupán annyi, hogy a konvolúciós rétegek a képek adaptív előfeldolgozását végzik, fontos jegyeket kiemelve; a hálózat utolsó rétegei pedig teljesen összekötöttek és működésükben nem különböznek a korábban megismert neurális hálózatoktól. Erre az architektúrára mutat példát a 4.10. algoritmus, amelyet kézzel írott számjegyek felismerésére használunk.

```
MLP(
    Chain(
        Conv((4,4),1=>15,relu,stride=2,pad=1),
        Conv((4,4),15=>15,relu,stride=2,pad=1),
        flatten,
        Dense(735,10)
    )
)
```

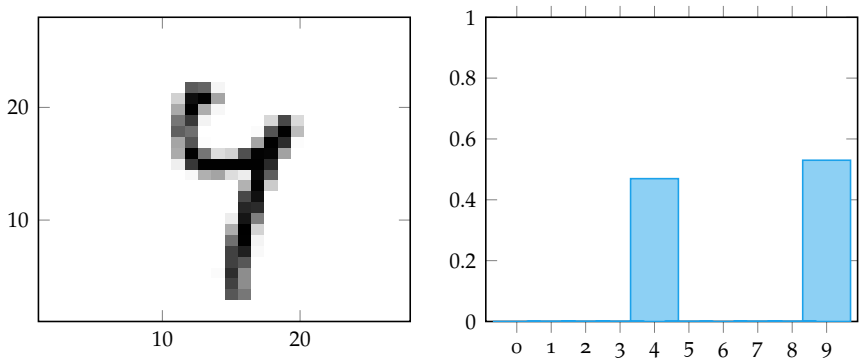
4.10. algoritmus. Képfelismerésre használható neurális hálózat 4×4 konvolúciós kernelekkel és 15 csatornával két rétegben. A bemenetet 28×28 méretű képek alkotják; a kimenet a lehetséges `0`-`9` számjegyeknek megfelelően 10 méretű.

A hálózatot a 4.9. algoritmussal tanítjuk. Érdemes megfigyelni a tanítás során kialakuló átmeneti reprezentációkat, azaz az egyes konvolúciós rétegek kimenetét, amelyek tehát a fontosnak ítélt jegyeket tartalmazzák:



4.8. ábra. Konvolúciós rétegek kimenete a 4.10. neurális hálózatban.

Az átmeneti reprezentációk értelmezése általában nehézkes. Tapasztalatok szerint a neurális hálózatok nem olyan reprezentációkkal dolgoznak, mint az emberek, így nemigen lehet megmondani, hogy mi alapján jutott a hálózat az adott következtetésre. A példában szereplő hálózat mindazonáltal kb. 99% pontosságot ér el a számjegyek felismerésében. A 4.9. ábra egy kevésbé egyértelmű esetet mutat:



4.9. ábra. A bal oldalon a bemenetet látjuk, a jobb oldalon pedig a softmax kimenetet, azaz a számjegy osztályának poszterior eloszlását.

4.8 Automatikus differenciálás

Végül ejtsünk pár szót a gradiens-alapú tanítás gyakorlati megvalósításáról. A gradiens előállítása esetenként bonyolult számolást kíván meg (lásd 4.3. szakasz), ám maga az elv egyszerű: csupán a deriválási szabályokat kell teljesen mechanikusan alkalmazni. Az efféle „favágást'' jobb a számítógépre bízni, ahogy a 4.9. algoritmusnál is láttuk. Az ötlet ráadásul nem korlátozódik a neurális hálózatok tanítására; bármilyen (differenciálható) numerikus program gradiense előállítható a szabályok ismételt alkalmazásával[29]. A 4.6.2. szakasz optimalizációs módszereit hozzávéve rögtön megkapjuk a következtető algoritmusokat anélkül, hogy akár egy sornyi kódot leírnánk (persze magát a modellt azért még nekünk kell összeállítani). A `PyTorch`, `TensorFlow`, `Flux` stb. könyvtárak ezt az eszköztárat adják a kezünkbe, kiegészítve a GPU, TPU-n történő futtatással.

Nézzünk erre egy példát. Vegyünk egy

$$F\colon \mathbb{R}^3 \to \mathbb{R}$$

[29] Az automatikus differenciálás és differenciálható programozás (∂P) újabban nagy népszerűségre tett szert nem csak a gépi tanulásban, hanem fizikai rendszereknél, például a meteorológiában vagy a robotikában is.

függvényt, amely három lépésben számolható:

$$F(\mathbf{x}) = h(g(f(\mathbf{x}))),$$

ahol

$$\begin{aligned} f &\colon \mathbb{R}^3 \to \mathbb{R}^3, \\ g &\colon \mathbb{R}^3 \to \mathbb{R}^6, \\ h &\colon \mathbb{R}^6 \to \mathbb{R}, \end{aligned}$$

amelyek akár egy neurális hálózat rétegei is lehetnek[30], vagy bármilyen más differenciálható leképezések. Általánosságban az F függvény l-edik komponensének[31] deriváltja az $\mathbf{x}^i$. komponens szerint a láncszabály felhasználásával írható:

$$\frac{\partial F^l}{\partial \mathbf{x}^i} = \sum_k \sum_j \frac{\partial h^l}{\partial g^k} \cdot \frac{\partial g^k}{\partial f^j} \cdot \frac{\partial f^j}{\partial \mathbf{x}^i}.$$

[30] Ebben az esetben az $\mathbf{x}$ változó a súlyokat jelenti.

[31] Természetesen most csak egy komponensünk van, tehát $l = 1$.

A jobb oldalon szereplő számításban felfedezhetjük a deriváltakat tartalmazó mátrixok (Jacobi-mátrixok) szorzatát. Emlékeztetőül az F függvény Jacobi-mátrixa

$$\nabla F = \left[\frac{\partial F^l}{\partial \mathbf{x}^i}\right] = \begin{bmatrix} \frac{\partial F^1}{\partial \mathbf{x}^1} & \frac{\partial F^1}{\partial \mathbf{x}^2} & \cdots & \frac{\partial F^1}{\partial \mathbf{x}^I} \\ \frac{\partial F^2}{\partial \mathbf{x}^1} & \frac{\partial F^2}{\partial \mathbf{x}^2} & \cdots & \frac{\partial F^2}{\partial \mathbf{x}^I} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F^L}{\partial \mathbf{x}^1} & \frac{\partial F^L}{\partial \mathbf{x}^2} & \cdots & \frac{\partial F^L}{\partial \mathbf{x}^I} \end{bmatrix},$$

amely tehát a h, g és f függvények Jacobi-mátrixának szorzataként áll elő:

$$\nabla F = \nabla h \cdot \nabla g \cdot \nabla f.$$

Ha tehát a fontosabb elemi függvények, „primitívek'' – összeadás, kivonás, szorzás, exponenciális stb. – deriváltjait elő tudjuk állítani, ezekből a bonyultabb programok gradiense is kiszámolható, csupán az egyes lépésekben adódó Jacobi-mátrixokat kell összeszorozni. Az előbbi formulát képszerűen is ábrázolhatjuk:

$$\underbrace{\begin{bmatrix} * & * & * \end{bmatrix}}_{\nabla F} = \underbrace{\begin{bmatrix} * & * & * & * & * & * \end{bmatrix}}_{\nabla h} \cdot \underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{\nabla g} \cdot \underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{\nabla f}.$$

Hogyan érdemes nekiállni ∇F kiszámításának? Elindulhatunk például jobbról, a ∇g és ∇f mátrixokat összeszorozva marad még

$$\underbrace{\begin{bmatrix} * & * & * \end{bmatrix}}_{\nabla F} = \underbrace{\begin{bmatrix} * & * & * & * & * & * \end{bmatrix}}_{\nabla h} \cdot \underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{\nabla g \cdot \nabla f},$$

a végső gradienst így egy 1×6 és egy 6×3 mátrix szorzata adja. A teljes számolás során összesen 72 darab szorzást és 51 darab összeadást kell elvégeznünk. Másrészről viszont ha ∇h és ∇g szorzatával kezdünk, akkor

$$\underbrace{\begin{bmatrix} * & * & * \end{bmatrix}}_{\nabla F} = \underbrace{\begin{bmatrix} * & * & * \end{bmatrix}}_{\nabla h \cdot \nabla g} \cdot \underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{\nabla f},$$

azaz ugyanaz az eredmény egy 1×3 és egy 3×3 mátrix szorzataként áll elő. A teljes műveletigény 27 szorzás és 21 összeadás. Általánosságban sok bemenettel és kevés kimenettel rendelkező programok esetében – mint például a neurális hálózatok – célszerű az utóbbit választani.

A konkrét megvalósítások előtt még azt kell észrevennünk, hogy sok esetben pazarló lehet a teljes Jacobi-mátrix kiszámítása. Ha például

$$g(\mathbf{x}) = \begin{bmatrix} 2\mathbf{x}^1 + 3 \\ 2\mathbf{x}^2 - 1 \\ 2\mathbf{x}^3 \\ 3 \\ 4 \\ 5 \end{bmatrix},$$

akkor a Jacobi-mátrix sok zérust tartalmaz:

$$\nabla g = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Vegyük azt az esetet, amikor jobbról balra haladunk. Figyelembe véve, hogy ez a ∇g mátrix nem önmagában kell, csupán jobbról meg szeretnénk szorozni valamivel, nem érdemes konkrétan mátrixként reprezentálni; jobb, ha közvetlenül a neki megfelelő leképezést valósítjuk meg. Például:

```
# Mátrixszal:
∇g_times(v) = [2 0 0
               0 2 0
               0 0 2
               0 0 0
               0 0 0
               0 0 0] * v
# JVP-vel:
∇g_times(v) = vcat(2v,zeros(3,size(v,2)))
```

4.11. algoritmus. A ∇g Jacobi-mátrix szorzása jobbról egy $\mathbf{v}$ vektorral vagy mátrixszal. A mátrixszorzás elvégzése helyett annak eredményét egyszerűbben is kiszámolhatjuk; esetünkben ez egy lineáris operátor, amely egy vektort kétszeresére nyújt, majd hozzácsap három 0-t (mátrix-értékű v esetén ugyanezt alkalmazzuk v összes oszlopára).

A szakirodalom a műveletet *jacobian vector product* (*JVP*) névvel illeti. A jobbról balra haladó kiértékelés nem más, mint JVP-k kompozíciója; a primitív JVP-k lényegében a deriválási szabályokkal egyeznek meg, amelyek könnyen implementálhatók. Ugyanígy járunk el, ha balról jobbra haladva értékelünk ki. Ebben az esetben a Jacobi-mátrixok mindig balról szorzódnak; az ennek megfelelő művelet a *vector jacobian product* (*VJP*).

Legyen most

$$f(\mathbf{x}) = \begin{bmatrix} \mathbf{x}^1 + \mathbf{x}^2 \\ \mathbf{x}^1 \cdot \mathbf{x}^3 \\ 3\mathbf{x}^2 \end{bmatrix}, \qquad \nabla f = \begin{bmatrix} 1 & 1 & 0 \\ \mathbf{x}^3 & 0 & \mathbf{x}^1 \\ 0 & 3 & 0 \end{bmatrix},$$

és tegyük fel, hogy birtokunkban van a ∇f-fel való szorzás JVP formájában. Ez lehetővé teszi, hogy f összes komponensének $\mathbf{x}^1$ szerinti deriváltját kiszámítsuk.

$$\frac{\partial f}{\partial \mathbf{x}^1} = \begin{bmatrix} 1 & 1 & 0 \\ \mathbf{x}^3 & 0 & \mathbf{x}^1 \\ 0 & 3 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = JVP_f \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ \mathbf{x}^3 \\ 0 \end{bmatrix}.$$

Hasonlóképpen járhatunk el az $\mathbf{x}^2$ szerinti deriváltakra[32]:

$$\frac{\partial f}{\partial \mathbf{x}^2} = \begin{bmatrix} 1 & 1 & 0 \\ \mathbf{x}^3 & 0 & \mathbf{x}^1 \\ 0 & 3 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = JVP_f \left(\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix}.$$

[32] Az $\mathbf{x}^3$ szerinti esetet ellenőrizni házi feladat.

A JVP-k tehát arra jók, hogy az összes kimenet deriváltját megkapjuk egy-egy bemenetre nézve (magyarán a Jacobi-mátrix egy-egy oszlopát).

VJP-kkel dolgozva fordított a helyzet: ekkor egy-egy kimenet deriváltját kaphatjuk meg az összes bemenetre nézve, azaz a Jacobi-mátrix egy-egy sorát[33]:

$$\nabla_{\mathbf{x}} f^1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ \mathbf{x}^3 & 0 & \mathbf{x}^1 \\ 0 & 3 & 0 \end{bmatrix} = VJP_f \left(\begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \right) = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$$

$$\nabla_{\mathbf{x}} f^2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ \mathbf{x}^3 & 0 & \mathbf{x}^1 \\ 0 & 3 & 0 \end{bmatrix} = VJP_f \left(\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right) = \begin{bmatrix} \mathbf{x}^3 & 0 & \mathbf{x}^1 \end{bmatrix}.$$

[33] Látjuk, hogy egyetlen kimenet és sok bemenet esetén ez a legpraktikusabb, hiszen egy csapásra az összes deriváltat megkapjuk. Neurális hálózatoknál az egyetlen kimenet lehet a veszteség, a VJP-k kompozíciójával pedig egy menetben a súlyokra vonatkozó összes derivált (gradiens) a birtokunkba kerül. Ez nem más, mint a hibavisszaterjesztéses algoritmus álruhában.

A jobbról balra haladó stratégiát *előre-módú* (*forward-mode*) automatikus differenciálásnak nevezik, mivel a deriváltakat a számítással párhuzamosan számoljuk: ahogy az f, g, h műveletek egymást követik, ugyanúgy rögtön a $\nabla f \cdot \mathbf{v}$, $\nabla g \cdot \nabla f \cdot \mathbf{v}$, $\nabla h \cdot \nabla g \cdot \nabla f \cdot \mathbf{v}$ szorzatokat is kiszámoljuk. Ennek előnye, hogy a gradiensek számolásához szükséges információt nem kell a memóriában tárolni. Formálisan egy f művelet a következő formát ölti:

$$\mathbf{x} \mapsto \left(f(\mathbf{x}), \Delta \mapsto \nabla f \Big|_{\mathbf{x}} \cdot \Delta \right),$$

ahol az első elem a függvény kimenete, a másik pedig a JVP-t megvalósító eljárás, amit azon nyomban ki is értékelünk.

A balról jobbra haladó stratégiát *visszafelé-módú* (*reverse mode*) automatikus differenciálásnak nevezik, mivel a deriváltakat fordított sorrendben számoljuk. Formálisan az f művelet a következőképpen néz ki:

$$\mathbf{x} \mapsto \left(f(\mathbf{x}), \tilde{\Delta} \mapsto \tilde{\Delta}^\top \cdot \nabla f \Big|_{\mathbf{x}} \right),$$

ahol a második elem a VJP-t megvalósító eljárás. Ennek kiértékeléséhez először végig kell futtatni a teljes számítást (*forward pass*, tehát f, g, majd h), „megjegyezni'' az deriváltak számolásához szükséges információt (VJP-ket), majd a végén fordított sorrendben komponálni a VJP-ket ($\tilde{\mathbf{v}}^\top \cdot \nabla h$, $\tilde{\mathbf{v}}^\top \cdot \nabla h \cdot \nabla g$, majd $\tilde{\mathbf{v}}^\top \cdot \nabla h \cdot \nabla g \cdot \nabla f$).

Végül pár szó az implementációról. Látjuk, hogy a műveleteket ki kell egészíteni a deriváltak számításához szükséges információval; az automatikus differenciálást támogató programcsomagok erre meglehetősen eltérő filozófiát követnek. Az *operator overloading* technikán alapuló könyvtárak (pl. `PyTorch, JAX, TensorFlow`) speciális wrapper adattípusokat használnak, amelyekre nézve a primitív operátorok túlterheltek, és rögtön a JVP/VJP-ket is kiszámolják. A *source code transformation* eszközök (pl. `Zygote`) a forráskód transzformációjával (saját compiler-rel) állítják elő a gradienseket számító kódot. Újabb megközelítés az `Enzyme` könyvtár, amely az LLVM alacsony szintű reprezentációját használja, így sokkal inkább nyelvfüggetlen, és képes a már optimalizált kódhoz is deriváltakat előállítani.

5. fejezet

Variációs közelítés

Ebben a fejezetben áttérünk a generatív modellezésre. A korábbiaknál jóval nehezebb feladatra vállalkozunk: míg eddig egy $p(y \mid \mathbf{x})$ mennyiségre voltunk kíváncsiak (azaz címkéket jósoltunk az adatpontokhoz), most az együttes $p(\mathbf{x}, y)$ eloszlást keressük[1]. Az együttes eloszlás birtokában akár új adatokat is tudunk generálni.

[1] Ebben a fejezetben általánosabbak leszünk, az esetleges címkéket is beleértjük az adatba, és csupán $p(\mathbf{x})$-et írunk.

5.1 *Evidence lower bound (ELBO)*

Egy valószínűségi modellben szereplő változókat két csoportra oszthatjuk: megfigyelt és rejtett változókra. A megfigyelt változók alatt rendszerint olyasmit értünk, ami a következtető algoritmusunk bemeneteként szolgálhat; jelölje mindezen adatok összességét $\mathbf{x}$. Bayesi következtetésnél a célunk a $\mathbf{z}$-vel jelölt rejtett változók eloszlását megbecsülni; ez tehát azon változók összessége, amelyekre következtetni szeretnénk[2]. Ezen eloszlások birtokában készen is vagyunk a generatív modellel; új adatok generálásához csupán mintavételezni kell, ami általában könnyen kivitelezhető[3] (lásd a 7. fejezetet).

[2] Lineáris regressziónál például az ismeretlen súlyvektor játssza a rejtett változók szerepét, de lehetnek sokkal bonyolultabb, hierarchikus modelljeink is sokféle rejtett változóval.

[3] Ebben az értelemben tehát minden teljesen bayesi modell generatív.

A feladat tehát a rejtett változók eloszlásának megbecsülése az adatok ismeretében; formálisan a $p(\mathbf{z} \mid \mathbf{x})$ mennyiségre vagyunk kíváncsiak. Szokásunkhoz híven próbálkozzunk meg a Bayes-tétellel:

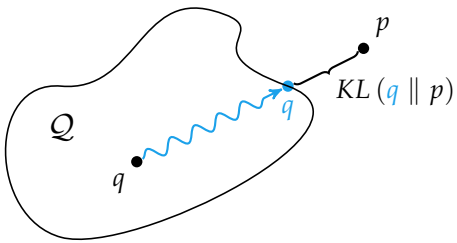
$$p(\mathbf{z} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})}{p(\mathbf{x})} = \frac{p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})}{\int p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z}) d\mathbf{z}},$$

azonban itt kellemetlen meglepetéssel szembesülünk: a nevezőt általában nem tudjuk kiszámolni[4]! A korábban használt másik trükkünk, miszerint nagyvonalúan eltekintünk a nevezőtől – hiszen $\mathbf{z}$-ben konstans – és csak a számlálót maximalizáljuk $\mathbf{z}$ szerint, megint csak nem működik, mivel most nem egy optimális értéket szeretnénk kapni $\mathbf{z}$-re, hanem egy eloszlást.

A megoldáshoz vezessünk egy $q(\mathbf{z})$ közelítő (ún. *variációs*) eloszlást; ezt fogjuk addig-addig javítgatni, amíg „közel nem kerülünk'' a valódi $p(\mathbf{z} \mid \mathbf{x})$ eloszláshoz. A kettő közötti eltérést a *Kullback–Leibler divergenciával*[5] mérjük:

$$KL(q(\mathbf{z}) \parallel p(\mathbf{z} \mid \mathbf{x})) := -\int q(\mathbf{z}) \ln \frac{p(\mathbf{z} \mid \mathbf{x})}{q(\mathbf{z})} d\mathbf{z}.$$

A $q(\mathbf{z})$ eloszlást mi mondhatjuk meg. Ha egy rögzített $\mathcal{Q}$ függvénycsaládból választjuk, a feladatunk ilyesféleképpen alakul:



5.1. ábra. Következtetés a variációs elvvel. A $q \in \mathcal{Q}$ eloszlást úgy módosítjuk, hogy a lehető „legközelebb'' kerüljön a valódi p eloszláshoz.

Ez egyben azt is jelenti, hogy a variációs módszerekkel általában nem a valódi eloszlást találjuk meg[6]. A függvénycsalád megválasztása éppen ezért kulcsfontosságú. Általában olyasmire törekszünk, ami kellően flexibilis, tehát közel kerülhetünk a valódi eloszláshoz, másrészről viszont könnyen kezelhető[7]. A következő felbontás képezi a variációs módszerek alapját:

$$\underbrace{\int q(\mathbf{z}) \ln \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})} d\mathbf{z}}_{:=\mathcal{L}[q(\mathbf{z})]} \underbrace{- \int q(\mathbf{z}) \ln \frac{p(\mathbf{z} \mid \mathbf{x})}{q(\mathbf{z})} d\mathbf{z}}_{KL(q(\mathbf{z}) \parallel p(\mathbf{z} \mid \mathbf{x}))} = \int q(\mathbf{z}) \ln \frac{p(\mathbf{x},\mathbf{z}) q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x}) q(\mathbf{z})} d\mathbf{z}$$

$$= \int q(\mathbf{z}) \ln p(\mathbf{x}) d\mathbf{z}$$

$$= \ln p(\mathbf{x}) \int q(\mathbf{z}) d\mathbf{z}$$

$$= \ln p(\mathbf{x}),$$

[4] Az integrálra általában konjugált modelleknél van zárt formulánk; egy példa erre a bayesi lineáris regresszió, ahol az integrálban szereplő mindkét eloszlás normális.

[5] Ez bizonyos értelemben a kanonikus választás, de léteznek másféle divergenciák is. Vegyük észre, hogy a feladat ezt a mennyiséget minimalizálni a q függvény (!) szerint. Az efféle problémák fizikában nagyon gyakoriak, *variációszámítással* oldhatók meg, innen a módszercsalád neve.

[6] Ellentétben a 7. fejezet módszereivel, amelyek egzaktak.

[7] Ennek érdekében $q(\mathbf{z})$-t gyakran neurális hálózatokkal paraméterezzük fel. A későbbi szakaszokban látni fogunk egy egyszerű és egy valamivel bonyolultabb példát.

ahol az $\mathcal{L}[q(\mathbf{z})]$ mennyiség neve *evidence lower bound* (*ELBO*)[8], a KL-divergenciát pedig már ismerjük. A lényeget kiírva kaptunk egy

$$\ln p(\mathbf{x}) = \mathcal{L}[q(\mathbf{z})] + KL(q(\mathbf{z}) \parallel p(\mathbf{z} \mid \mathbf{x}))$$

alakú felbontást. Az egyenlet jobb oldalán szereplő két mennyiség közül a KL-divergenciát szeretnénk $q(\mathbf{z})$ szerint minimalizálni. A variációs módszerek trükkösen járnak el: a közvetlen minimalizálás helyett inkább $\mathcal{L}$-t maximalizálják $q(\mathbf{z})$ szerint. Mivel egyenlet bal oldala $q(\mathbf{z})$-ben konstans, a KL-divergenciának kötelessége lesz csökkenni!

[8] Mivel $KL(\cdot \parallel \cdot) \geq 0$, rögtön látjuk, hogy $\ln p(\mathbf{x}) \geq \mathcal{L}[q(\mathbf{z})]$, azaz jogos az „alsó korlát'' kifejezés. Az „evidencia'' kifejezés pedig arra utal, hogy az ELBO-t mindig egy megfigyelt adathalmaz alapján számoljuk.

Nézzük, hogyan valósulhat meg $\mathcal{L}$ maximalizálása a gyakorlatban[9]. Vegyünk negatív előjelet (tehát mostantól minimalizálunk), valamint használjuk fel, hogy $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})$ és szedjük kétfelé az ELBO-t:

$$\begin{aligned} -\mathcal{L}[q(\mathbf{z})] &= -\int q(\mathbf{z}) \ln \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} d\mathbf{z} \\ &= -\int q(\mathbf{z}) \ln p(\mathbf{x} \mid \mathbf{z})\, d\mathbf{z} - \int q(\mathbf{z}) \ln \frac{p(\mathbf{z})}{q(\mathbf{z})} d\mathbf{z} \\ &= \underbrace{\mathbb{E}_{q(\mathbf{z})}[-\ln p(\mathbf{x} \mid \mathbf{z})]}_{\text{várható veszteség}} + \underbrace{KL(q(\mathbf{z}) \parallel p(\mathbf{z}))}_{\text{„regularizáció''}}. \end{aligned} \tag{5.1}$$

[9] A most mutatott módszer csupán a sok közül, vannak más stratégiák is.

Az első tagban felfedezhetünk egy negatív log-likelihood-jellegű mennyiséget[10], a második tag pedig arról gondoskodik, hogy a $q(\mathbf{z})$ eloszlás ne távolodjon el túlságosan a $p(\mathbf{z})$ priortól, azaz a korábbi fejezetekben látott regularizációs tagokhoz hasonlóan viselkedik[11].

[10] Aha, felbukkant egy veszteségfüggvény! Kár, hogy el van dugva egy várható érték belsejébe.

[11] Ez a megközelítés akkor működik, amikor a KL-divergenciát zárt formában tudjuk számolni, például ha mind a priort, mind a variációs eloszlást normális eloszlásúnak tesszük fel. Ha a KL-divergencia nem számolható zárt alakban, akkor az 5.4. szakasz nyújt segítséget.

Nézzük meg, hogy a várható veszteséggel mit tudunk kezdeni. A kiszámolásához több verzió lehetséges:

1. A várható érték zárt formában számolható. Ez ritka; éppen azért használjuk a variációs közelítést, mert nem bírunk ezekkel az integrálokkal.

2. Közelítjük az integrált. A Laplace-approximáció[12] például osztályozási problémáknál jól működik; jónéhány más esetben kevésbé.

3. Kvadratúrát használunk. Ha a variációs eloszlás normális, a Gauss–Hermite kvadratúra[13] nagyon gyorsan és pontosan tudja közelíteni a várható értéket.

[12] Lásd a C. függeléket.

[13] Lásd a C. függeléket.

4. Ha semmi más nem működik, de a variációs eloszlásból legalább mintavételezni tudunk, a várható értéket közelíthetjük S darab $\mathbf{z}_s \sim q(\mathbf{z})$ minta átlagával is, azaz

$$\mathbb{E}_{q(\mathbf{z})}\left[-\ln p\left(\mathbf{x} \mid \mathbf{z}\right)\right] \approx \frac{1}{S}\sum_{s=1}^{S} -\ln p\left(\mathbf{x} \mid \mathbf{z}_s\right).$$

   A gyakorlatban sokszor egyetlen minta is elég, de előfordul, hogy gradiens-alapú optimalizációnál túlságosan sztochasztikus gradienseket kapunk, ami jelentősen lassítja a tanulást.

A következő szakaszokban megvizsgáljuk $q(\mathbf{z})$ megválasztását különféle modellek esetében, valamint bevetjük a 4. fejezet módszereit a hatékony tanuláshoz.

5.2 *Bayesi logisztikus regresszió*

A 3. fejezet logisztikus regressziós modelljeinél csak ML vagy MAP megoldásokat tudtunk keresni. Variációs módszerrel azonban teljesen bayesi megoldást is kaphatunk, ha az előző szakaszban $\mathbf{z}$ szerepét a rejtett $\mathbf{w}$ súlyokra, $\mathbf{x}$ szerepét pedig az $(\mathbf{X}, \mathbf{y})$ megfigyelt be- és kimenetekre osztjuk. Magukat az adatpontokat most békén hagyjuk: egyelőre nem cél a teljes adat eloszlásának modellezése, csupán a logisztikus regressziót szeretnénk „bayesiesíteni''[14].

A logisztikus regresszió valószínűségi modellje a (3.3) egyenlet alapján

$$p\left(\mathbf{y} \mid \mathbf{X}, \mathbf{w}\right) = \prod_i \mathcal{B}ern\left(y_i \;\middle|\; \sigma\left(\mathbf{w}^\top \phi(\mathbf{x}_i)\right)\right),$$
$$p(\mathbf{w}) = \mathcal{N}\left(\mathbf{w} \mid \mathbf{0}, \mathbf{I}\right).$$

A $p\left(\mathbf{w} \mid \mathbf{X}, \mathbf{y}\right)$ poszteriorra vagyunk kíváncsiak. Vezessük be a közelítő variációs eloszlást a következőképpen:

$$q(\mathbf{w}) := \mathcal{N}\left(\mathbf{w} \;\middle|\; \boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2)\right),$$

azaz az eloszlást normálisnak választottuk az egyszerű számolás érdekében. A $\boldsymbol{\mu}$ és $\boldsymbol{\sigma}^2$ paraméterek a *variációs paraméterek*, amelyeket gradiens-alapon szeretnénk optimalizálni. Az optimalizációnál a negatív ELBO játssza a célfüggvény szerepét, ami a következőképpen alakul[15]:

[14] A teljes generatív modellezésre a következő szakaszban látunk majd példát.

[15] Vesd össze az (5.1) egyenlettel.

$$
\begin{aligned}
-\mathcal{L}\left[q(\mathbf{w})\right] &= \mathbb{E}_{q(\mathbf{w})}\left[-\ln p\left(\mathbf{X},\mathbf{y} \mid \mathbf{w}\right)\right] + KL\left(q(\mathbf{w}) \parallel p(\mathbf{w})\right) \\
&= \mathbb{E}_{q(\mathbf{w})}\left[-\ln p\left(\mathbf{y} \mid \mathbf{X},\mathbf{w}\right)\right] + KL\left(q(\mathbf{w}) \parallel p(\mathbf{w})\right) + \text{const.} \\
&= \mathbb{E}_{q(\mathbf{w})}\left[\sum_i BCE(\mathbf{y}_i, \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}_i))\right] + KL\left(q(\mathbf{w}) \parallel p(\mathbf{w})\right) + \text{const},
\end{aligned}
$$

ahol kihasználtuk, hogy $\mathbf{w}$ független $\mathbf{X}$-től, tehát

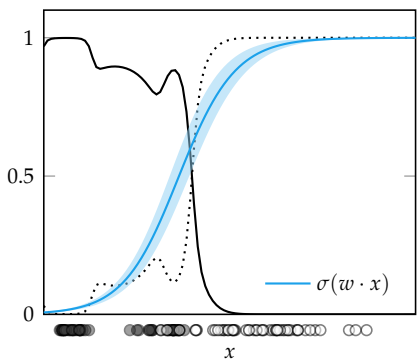
$$
p\left(\mathbf{X},\mathbf{y} \mid \mathbf{w}\right) = p\left(\mathbf{y} \mid \mathbf{X},\mathbf{w}\right) p\left(\mathbf{X} \mid \mathbf{w}\right) = p\left(\mathbf{y} \mid \mathbf{X},\mathbf{w}\right) p(\mathbf{X}) = p\left(\mathbf{y} \mid \mathbf{X},\mathbf{w}\right) \cdot \text{const.}
$$

A várható veszteségben a korábbi logisztikus regresszióval teljesen analóg módon bináris keresztentrópia kerül elő, a két normális eloszlás közötti KL-divergenciát pedig zárt alakban számolhatjuk[16]:

[16] Miután a formulát lenéztük Wikipediáról.

$$
KL\left(q(\mathbf{w}) \parallel p(\mathbf{w})\right) = \frac{1}{2}\sum_n \left((\boldsymbol{\mu}^n)^2 + (\boldsymbol{\sigma}^2)^n - \ln(\boldsymbol{\sigma}^2)^n - 1\right).
$$

Most már minden adott gradiens-módszer alkalmazásához; ki tudjuk számolni a várható veszteséget (például a mintavételezős módszerrel) és a KL-divergenciát is. Az eredményeket egy dimenzióban az 5.2. ábra, két dimenzióban az 5.3 szemlélteti.



5.2. ábra. Bayesi logisztikus regresszió egy dimenzióban. Az x tengelyen a minták láthatók, a kék függvény a $q(w)$ variációs eloszlás alapján számolt szigmoid, ahol most már a bizonytalanságot is látjuk.

Az 5.1. algoritmus egy egyszerű implementációt mutat be.

```julia
struct BayesianLogisticRegression
    μ       # q(w) várható érték
    logσ²   # q(w) log variancia

    BayesianLogisticRegression(dim) =
        new(zeros(dim),zeros(dim))
end
function train!(m::BayesianLogisticRegression, x, y; η=0.01, iters=100)
    ps = Params([m.μ,m.logσ²])
    for i in 1:iters
        grad = gradient(ps) do
            σ²   = exp.(m.logσ²)
            kl   = 0.5*sum(@. m.μ^2 + σ² - m.logσ² - 1)
            w    = m.μ .+ randn(size(σ²)) .* σ²
            bce  = sum(BCE.(x*w,y))
            elbo = -bce + kl
        end

        m.μ     .+= η*grad[m.μ]
        m.logσ² .+= η*grad[m.logσ²]
    end
end
```

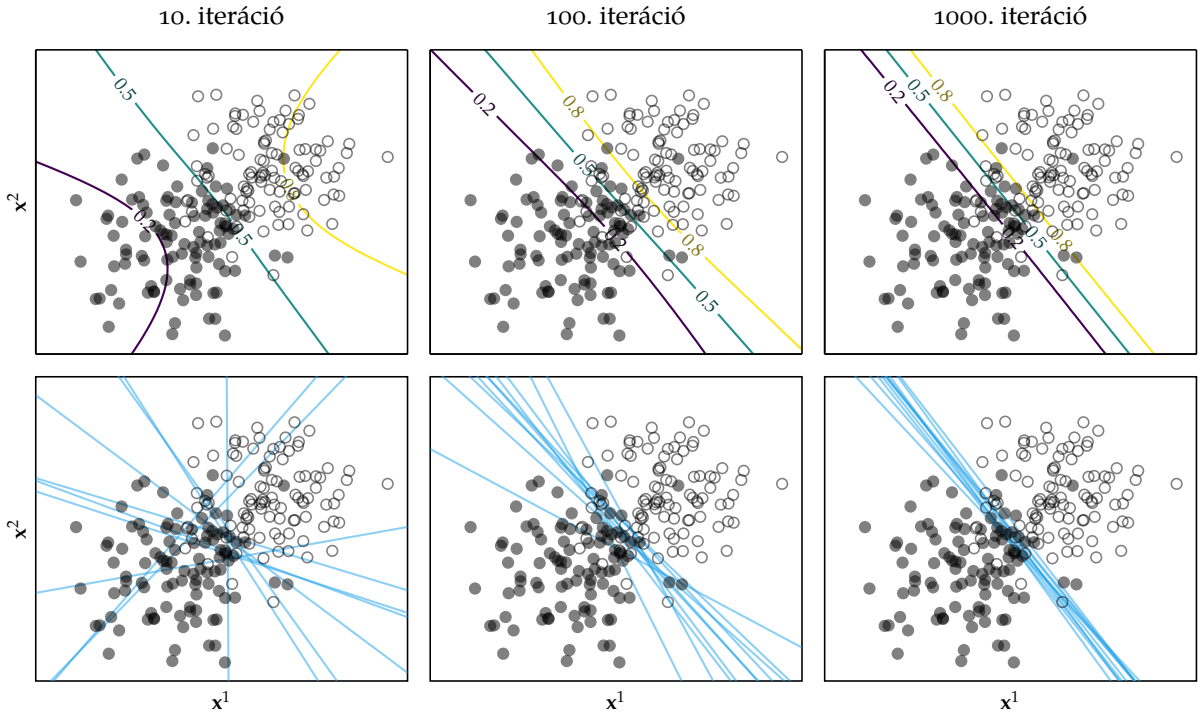
5.1. algoritmus. Variációs bayesi logisztikus regresszió. A variációs eloszlás várható értékét és varianciájának logaritmusát tanuljuk (így a variancia nem csúszhat zérus alá az optimalizáció során). A tanuláshoz a legegyszerűbb gradiensmódszert használjuk.

5.3 *Variációs autoenkóder*

Variációs autoenkóderrel a teljes adat eloszlását – $p(\mathbf{x})$-et – modellezhetjük. Ehhez speciális valószínűségi modellt teszünk fel, azaz $p(\mathbf{x} \mid \mathbf{z})$ alakját megkötjük. Szemléletesen mondva megszabjuk, hogy egy bizonyos, alacsony dimenziójú $\mathbf{z}$ rejtett változó ismeretében hogyan lehet mintákat generálni. Ez durva megkötésnek tűnhet, hiszen így jelentős megkötéseket teszünk $p(\mathbf{x})$-re is; éppen ezért a generatív folyamatot „adaptívan'', neurális hálózatokkal valósítjuk meg. Sőt, ha már itt tartunk, hasonlóan járunk el a $q(\mathbf{z})$ variációs eloszlással is, tehát mindkét eloszlást neurális hálózatokkal paraméterezzük, amelyeknek ψ és θ súlyrendszerei tanulhatók. Formálisan[17]

$$p(\mathbf{x} \mid \mathbf{z}) \rightsquigarrow p_\psi(\mathbf{x} \mid \mathbf{z}) := \mathcal{N}\left(\mathbf{x} \mid g_\psi(\mathbf{z}), \mathbf{I}\right),$$
$$q(\mathbf{z}) \rightsquigarrow q_\theta(\mathbf{z} \mid \mathbf{x}) := \mathcal{N}\left(\mathbf{z} \,\middle|\, \mu_\theta(\mathbf{x}), \sigma^2_\theta(\mathbf{x})\right).$$

[17] Természetesen nem kötelező a normális eloszlásokhoz ragaszkodni, de ez a leggyakoribb választás.



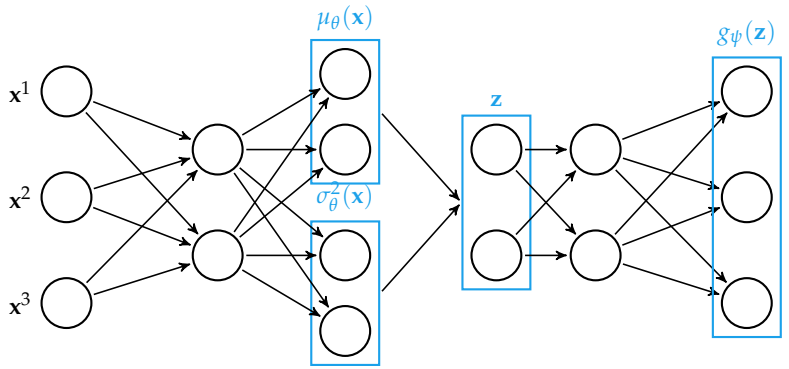
5.3. ábra. Bayesi lineáris regresszió két dimenzióban. A felső sor a poszterior várható értéket, az alsó sor pedig a poszteriorból származó mintákat mutatja a tanítás során.

A variációs eloszlás paramétereit tanuló hálózatok – nevezetesen μ_θ és σ^2_θ – bemenetét $\mathbf{x}$ szolgáltatja[18]. Ettől az architektúra tényleg egyfajta autoenkóder struktúrát vesz fel, hiszen $\mathbf{x}$ egyfajta bemenetként és célváltozóként is megjelenik: bemenetként $q_\theta(\mathbf{z} \mid \mathbf{x})$-ben, célváltozóként $p_\psi(\mathbf{x} \mid \mathbf{z})$-ben. Ezekkel a választásokkal

$$-\mathcal{L}\left[q_\theta(\mathbf{z} \mid \mathbf{x})\right] = \underbrace{\mathbb{E}_{q_\theta(\mathbf{z} \mid \mathbf{x})}\left[\frac{1}{2}\sum_n \left(\mathbf{x}^n - g_\psi(\mathbf{z})^n\right)^2\right]}_{\text{„rekonstrukciós loss''}} + \underbrace{KL\left(q_\theta(\mathbf{z} \mid \mathbf{x}) \parallel p(\mathbf{z})\right)}_{\text{„regularizáció''}},$$

ahol $p(\mathbf{z})$-t rendszerint standard normális eloszlásnak választjuk. Diagramon:

[18] Ezt minden további nélkül megtehetjük; lényegében csak egy speciális alakot választottunk $q(\mathbf{z})$-nek. Más generatív architektúrák (pl. GAN) nem élnek ezzel a lehetőséggel; valójában el is tekinthetnénk tőle és tanulhatnánk $q(\mathbf{z})$ paramétereit közvetlenül, nem pedig egy $\mathbf{x}$ bemenetre kötött neurális hálózattal.



5.4. ábra. Variációs autoenkóder sematikus rajza. Tényleg egyfajta autoenkódert látunk: a bemeneti $\mathbf{x}$ mintát próbáljuk a hálózat másik végén rekonstruálni.

A célfüggvény első tagja az $\mathbf{x}$ minta és a $g_\psi(\mathbf{z})$ rekonstrukció közötti várható veszteséget méri, ahol a veszteségfüggvény $-\ln p_\psi(\mathbf{x} \mid \mathbf{z})$. A második tag arról gondoskodik, hogy a $q(\mathbf{z} \mid \mathbf{x})$ variációs eloszlás ne távolodjon el a $p(\mathbf{z})$ priortól. A variációs autoenkóderre tehát úgy is gondolhatunk, hogy egy enkóderből és egy dekóderből áll; előbbi a rejtett $q(\mathbf{z} \mid \mathbf{x})$ eloszlás paramétereit tanulja, míg utóbbi a $q(\mathbf{z})$-ből vett minta alapján rekonstruálja az eredeti bemenetet. Úgy is mondhatjuk, hogy $\mathbf{x}$-et átküldjük egy „információs útszűkületen''; mivel $\mathbf{z}$ rendszerint sokkal alacsonyabb dimenzionalitású, mint $\mathbf{x}$, a hálózat kénytelen robusztus módon a lehető legtöbb információt eltárolni $\mathbf{x}$-ből az optimális rekonstrukcióhoz.

Nézzük a variációs autoenkóder „alkatrészeit'':

```
struct VAE
    enc      # enkóder
    q_μ      # q(z)
    q_logσ²  # q(z)
    dec      # dekóder

    VAE(enc,q_μ,q_logσ²,dec) =
        new(enc,q_μ,q_logσ²,dec)
end
```

5.2. algoritmus. Variációs autoenkóder megvalósítása. Az előző szakaszhoz hasonlóan a variancia logaritmusát tanuljuk.

Az enkódert és a dekódert neurális hálózatokként valósítjuk meg.

```
VAE(
    Chain(                      # enkóder
        flatten,
        Dense(28*28,500,relu),
        Dense(500,500)),
    Dense(500,2),               # q(z)
    Dense(500,2),               # q(z)
    Chain(                      # dekóder
        Dense(2,500,relu),
        Dense(500,500,relu),
        Dense(500,28*28,tanh),
        x->reshape(x,28,28,1,:))
)
```

5.3. algoritmus. Variációs autoenkóder létrehozása. A rejtett változó dimenzionalitását 2-re állítottuk.

A tanulás menetét az 5.4. algoritmus mutatja.

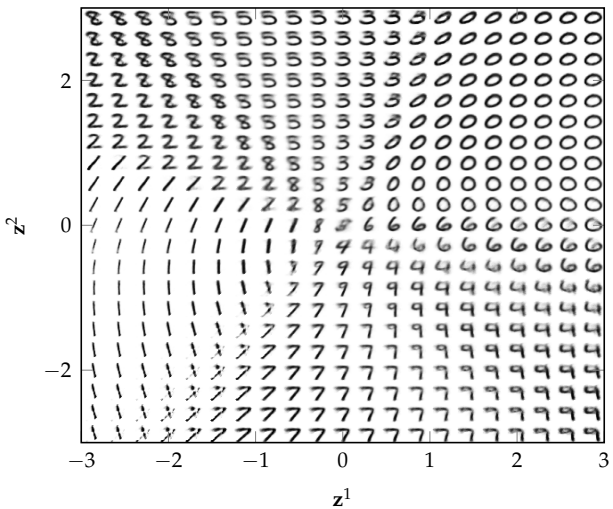
5.5. ábra. Variációs autoenkóder tanítása után rekonstruált minták az 5.3. architektúrával.

```
function train!(vae::VAE, x; epochs=10, batchsize=512)
    ps   = params(vae.enc,vae.q_μ,vae.q_logσ²,vae.dec)
    opt  = ADAM()
    data = DataLoader(x,batchsize=batchsize,shuffle=true)

    for epoch in 1:epochs, batch in data
        grad = gradient(ps) do
            a       = vae.enc(batch)
            μ,logσ² = vae.q_μ(a), vae.q_logσ²(a)
            σ²      = exp.(logσ²)
            kl      = 0.5*sum(@. μ^2 + σ² - 1 - logσ²)
            z       = μ .+ randn(size(σ²)) .* σ²
            sq      = -sum((batch .- vae.dec(z)).^2)
            elbo    = -sq + kl
        end
        update!(opt, ps, grad)
    end
end
```

5.4. algoritmus. Variációs autoenkóder tanítása a `Flux` könyvtár felhasználásával.

Mintageneráláshoz mindössze a dekódert kell használnunk; ez éppen arra lett „kiképezve'', hogy egy ismert eloszlásból[19] származó mintákat **x**-ekké alakítson. Érdekes megfigyelni, mit látunk, ha a dekódert különböző **z**-kkel hajtjuk meg:



[19] Ez úgy értendő, hogy a variációs eloszlásból, de mivel az remélhetőleg közel van a $p(\mathbf{z})$ priorhoz, utóbbit használhatjuk a generáláshoz.

5.6. ábra. A **z** rejtett változó hatása a generált mintákra. Látjuk, hogy a rejtett változók kétdimenziós terében más-más számjegyek máshonnan szeretnek generálódni.

5.4 *Versengő modellek*

Mind a bayesi lineáris regressziónál, mind a variációs autoenkódernél feltettük, hogy az ELBO-ban a KL-divergenciát zárt formában tudjuk számolni. Ez csak akkor igaz, ha „egyszerű'' variációs eloszlást és priort választunk. Tekintsünk most el ettől. A negatív ELBO továbbra is[20]:

[20] Lásd az (5.1) egyenletet.

$$\begin{aligned}-\mathcal{L}\left[q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)\right] &= \mathbb{E}_{q_\theta(\mathbf{z} \mid \mathbf{x})}\left[-\ln p_\psi\left(\mathbf{x} \mid \mathbf{z}\right)\right] + KL\left(q_\theta\left(\mathbf{z} \mid \mathbf{x}\right) \parallel p(\mathbf{z})\right) \\ &= \mathbb{E}_{q_\theta(\mathbf{z} \mid \mathbf{x})}\left[-\ln p_\psi\left(\mathbf{x} \mid \mathbf{z}\right)\right] - \mathbb{E}_{q_\theta(\mathbf{z} \mid \mathbf{x})}\left[\ln \frac{p(\mathbf{z})}{q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)}\right].\end{aligned}$$

Figyeljük meg, hogy a második várható értékben szereplő hányados értéke egy adott $\mathbf{z}$ esetén

$$\ln \frac{p(\mathbf{z})}{q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)} = \begin{cases} \ll 0, & \text{ha } \mathbf{z} \sim q_\theta\left(\mathbf{z} \mid \mathbf{x}\right), \\ \gg 0, & \text{ha } \mathbf{z} \sim p(\mathbf{z}). \end{cases}$$

Alkalmazzuk erre a szigmoid függvényünket:

$$\sigma\left(\ln \frac{p(\mathbf{z})}{q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)}\right) = \begin{cases} \sim 0, & \text{ha } \mathbf{z} \sim q_\theta\left(\mathbf{z} \mid \mathbf{x}\right), \\ \sim 1, & \text{ha } \mathbf{z} \sim p(\mathbf{z}). \end{cases}$$

Ez egészen olyan, mintha valamiféle osztályozási feladattal állnánk szemben: egy-egy $\mathbf{z}$-ről szeretnénk eldönteni, hogy a $p(\mathbf{z})$ priorból, vagy a $q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)$ variációs eloszlásból származik. Erre az osztályozási feladatra vezessünk be egy $D(\mathbf{x}, \mathbf{z})$ neurális hálózatot, amelyet *diszkriminátornak* fogunk nevezni. Az erre vonatkozó veszteségfüggvény legyen

$$\mathbb{E}_{p(\mathbf{z})}\left[\ln \sigma(D(\mathbf{x}, \mathbf{z}))\right] + \mathbb{E}_{q_\theta(\mathbf{z} \mid \mathbf{x})}\left[\ln\left(1 - \sigma(D(\mathbf{x}, \mathbf{z}))\right)\right],$$

ami valójában a bináris keresztentrópiának a „folytonos változata''; másképp írva

$$\int p(\mathbf{z}) \cdot \ln \sigma(D(\mathbf{x}, \mathbf{z})) + q_\theta\left(\mathbf{z} \mid \mathbf{x}\right) \cdot \ln\left(1 - \sigma(D(\mathbf{x}, \mathbf{z}))\right) d\mathbf{z}.$$

Ha ez nem győzött meg minket, másképp is igazolhatjuk, hogy a diszkriminátor veszteségfüggvényét célszerű így választani. Korábban láttuk, hogy a[21]

[21] Ez ugyanaz, mint az előbbi egyenlet, csak egyszerűsítettünk a notáción.

$$p \ln \sigma + q \ln(1 - \sigma),$$

típusú mennyiségek szélsőértékhelye egyszerű deriválással és zérusra rendezéssel adódik:

$$\frac{p}{\sigma} - \frac{q}{1-\sigma} = 0 \quad \Rightarrow \quad \sigma = \frac{p}{p+q}.$$

Az optimális σ-t tisztességesen kiírva

$$\begin{aligned} \sigma = \frac{1}{1+e^{-D(\mathbf{x},\mathbf{z})}} &= \frac{p(\mathbf{z})}{p(\mathbf{z}) + q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)} \\ e^{-D(\mathbf{x},\mathbf{z})} &= \frac{q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)}{p(\mathbf{z})} \\ D(\mathbf{x},\mathbf{z}) &= \ln \frac{p(\mathbf{z})}{q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)}, \end{aligned}$$

azaz a diszkriminátor kimenete a fenti veszteségfüggvénnyel tényleg a keresett mennyiséget közelíti.

A tanulásnál tehát kettős feladatunk van: egyrészt tanítjuk az eredeti modellünket az ELBO maximalizálásával, másrészt a diszkriminátort is a fenti veszteség minimalizálásával. Úgy is mondhatnánk, hogy a modell generatív része megpróbálja minél jobban „utánozni'' a $p(\mathbf{z})$ priort a $q_\theta\left(\mathbf{z} \mid \mathbf{x}\right)$ variációs poszteriorral, míg a diszkriminátor arra törekszik, hogy a két eloszlásból származó mintákat megkülönböztesse egymástól. Az így összeállított a modellt *versengő autoenkódernek*[22] nevezzük.

[22] Angolul *adversarial autoencoder*, *AAE*.

6. fejezet

Expectation-Maximization

7. fejezet

Markov Chain Monte Carlo

8. fejezet

Szupportvektor-gépek

9. fejezet

Főkomponens-analízis

10. fejezet

Megerősítéses tanulás

11. fejezet

Federált tanulás

A. függelék

Newton–Raphson módszer

B. függelék

Feltételes optimalizáció Lagrange-multiplikátorokkal

C. függelék

Közelítő módszerek integrálok kiszámításához

Irodalomjegyzék

1. L. Aitchison és M. Lengyel, ''THE HAMILTONIAN BRAIN: EFFICIENT PROBABILISTIC INFERENCE WITH EXCITATORY-INHIBITORY NEURAL CIRCUIT DYNAMICS'', *PLoS Comput Biol*, 12. évf., 12. sz., e1005186., 2016 (hiv. old. 4).
2. S. Rathmanner és M. Hutter, ''A Philosophical Treatise of Universal Induction'', *Entropy*, 13. évf., 6. sz., 1076–1136. old., 2011 (hiv. old. 5).
3. F. Rosenblatt, ''THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN.'', *Psychological Review*, 65. évf., 6. sz., 386–408. old., 1958 (hiv. old. 42).