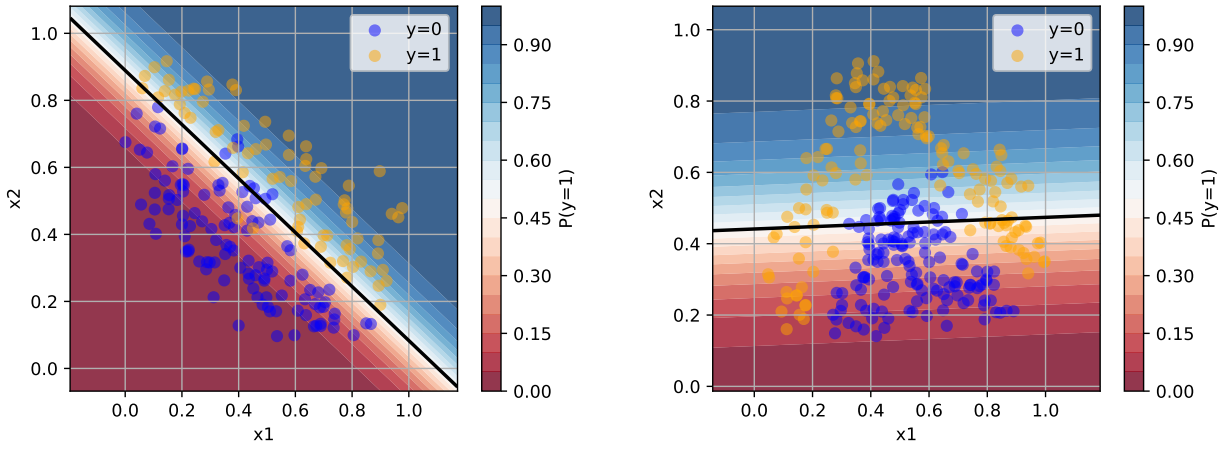
Nevronske mreže in poskusi razlag kompleksnih modelov

Primer posplošenih linearnih modelov, ki smo ga obravnavali v prejšnjem poglavju, je bila logistična regresija. Ta zgradi model, katerega odločitvena meja je linearna. Oglejmo si dva primera podatkov z binarnim razredom (slika 24). Pri prvem primeru logistična regresija uspešno loči oba razreda, pri drugem pa popolnoma odpove. Bi lahko v drugem primeru preoblikovali prostor značilk tako, da bi logistični regresiji uspelo?



Slika 24: Linearno in nelinearno ločljiva množica podatkov ter rezultat modeliranja z logistično regresijo.

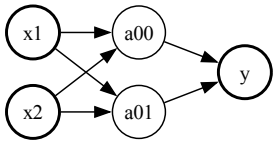
Očitno tudi v drugi množici podatkov s slike 24 obstajajo segmenti, kjer bi razreda lahko ločili z linearno ločnico. Če bi podatke opazoval človek, bi morda lahko začrtal dve premici in območje pod njima razglasil za področje ciljnega razreda. A za to bi morali torej postaviti dve ločeni odločitveni meji, nato pa dodati še nekaj, kar bi oddaljenost od njiju povezalo v končni klasifikator. Lahko torej na vhodu uporabimo dve logistični regresiji (za dve ločitveni meji), nato pa še eno logistično regresijo, ki združi njuna izhoda?

Kombinacije logističnih regresij

Raziščimo to zadnjo idejo kombiniranja logističnih regresij. Zgra-

Opozorilo: bralec naj naše kombiniranje logističnih regresij razume predvsem kot konceptualni korak, ki nam bo omogočil razumeti nevronske mreže. V sodobnih nevronskih mrežah notranje računske enote praviloma uporabljajo enostavnejše aktivacijske funkcije, o katerih bomo govorili nekoliko kasneje.

dimo lahko sistem oziroma arhitekturo z dvema logističnima regresijama v vhodni plasti. Njuna izhoda označimo z a_{00} in a_{01}, kjer prvi indeks označuje številko plasti, drugi pa številko računske enote (logistične regresije) znotraj te plasti. Ker imamo dve plasti, bi lahko aktivacijo v drugi plasti označili z a_{10}, a ker ta že predstavlja naš izhod $P(y = 1)$, jo bomo na diagramu preprosto označili z y. Arhitektura takšnega sistema je prikazana na sliki 25.

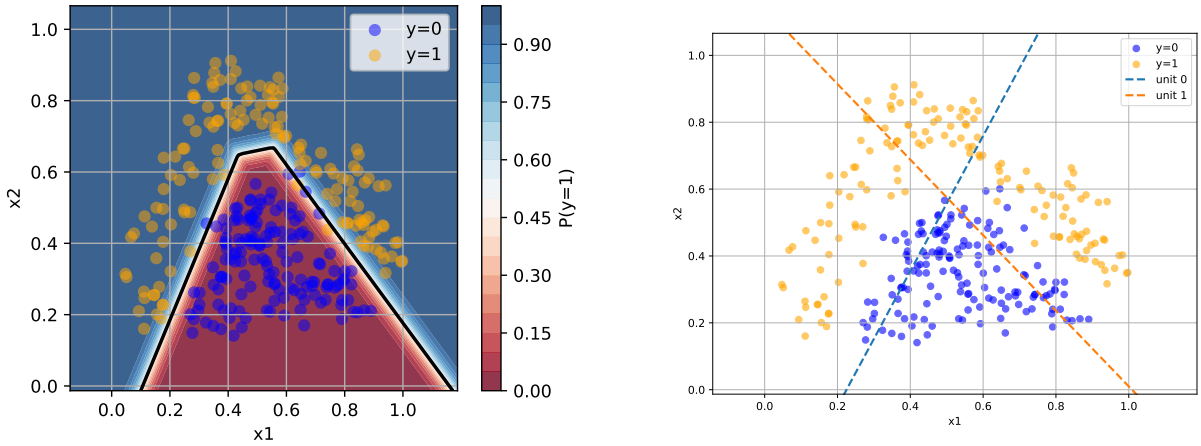


Slika 25: Kombinacija treh logističnih regresij.

Naj samo spomnimo: za izračun aktivacij, to je, izhodnih vrednosti posameznih enot (torej, naših logističnih regresij) najprej izračunamo uteženo vsoto vhodov, $z = w \cdot x + b$, nato pa to vsoto pretvorimo z aktivacijsko funkcijo, to je, pri logistični regresiji z $a = \sigma(z)$. Pri logistični regresiji smo sigmoidno preslikavo uporabili zato, da uteženo vsoto pretvorimo v verjetnost. A ker verjetnosti rabimo le na izhodu našega (kombiniranega) modela, bi lahko v vseh ostalih računskih enotah naše nastajajoče arhitekture uporabili tudi kakšne druge nelinearne transformacijske funkcije.

Transformacijske funkcije morajo biti nelinearne. Če bi bile linearne, njihova uporaba ne bi imela smisla, saj so kombinacije linearnih funkcij še vedno linearne in bi se tako vrnili k enemu samemu računskemu vozlišču z logistično regresijo, ki ne bi moglo tvoriti nelinearnih odločitvenih mej.

Predlagana arhitektura našega modela lepo reši klasifikacijski problem, kot prikazuje slika 26. Ampak, ali ta res zgradi odločitvene meje tako, kot bi pričakovali? Ne povsem. Dve notranji računski vozlišči imata svoji "odločitveni meji" postavljeni v pričakovani smeri (slika 26, desno), a zamaknjeno.



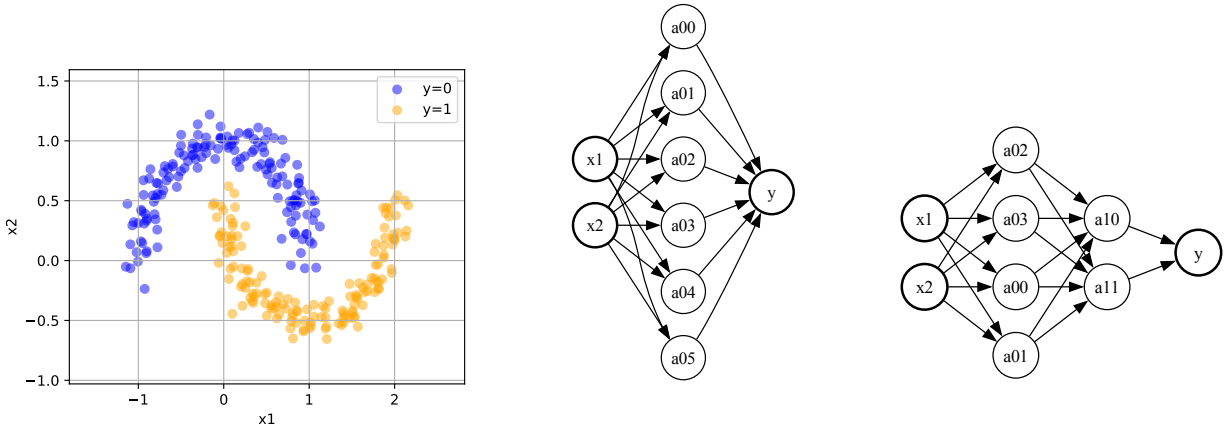
Slika 26: Odločitvena meja in izolinije verjetnosti za celoten model (levo) in za logistični regresiji na vhodu modela (desno).

Seveda bi se zmotili, če bi pričakovali, da bosta "odločitveni meji" obeh vhodnih logističnih regresij naše arhitekture lokalno jasno ločevali primere različnih razredov. Izhodna logistična regresija namreč uteži oddaljenosti od teh premic in jih sešteje, ne pa denimo preveri, ali sta obe oddaljenosti pozitivni (kot bi to verjetno storil človek). Zanimivo pa je, da sta ti dve premici – in to povsem pričakovano, če o tem nekoliko premislimo – usmerjeni pravilno in ravno prav zama-

knjeni, da ju lahko uporabi končna logistična regresija. Če bi želeli, bi lahko njune uteži (ne pa tudi njunega položaja) uporabili za nadaljnjo interpretacijo modela.

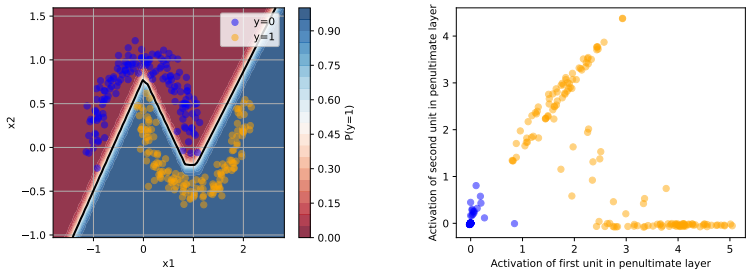
Tu se seveda ne smemo ustaviti. Nadaljujemo lahko s še bolj kompleksnimi in nelinearnimi klasifikacijskimi problemi, kot je denimo ta na sliki 27. Za te nekoliko zahtevnejše podatke smo zasnovali dve alternativni arhitekturi modelov, ki sta v osnovi ponovno le kombinaciji logističnih regresij, a obe uspeta poiskati ustrezne (in pričakovane) odločitvene meje.

Omenjamo "arhitekturo modelov", torej način, kako so računske enote, torej naše logistične regresije, med seboj povezane. Arhitekturo bomo v splošnem določili s številom plasti (nivojev) in številom računskih enot v posamezni plasti ter med enotami sosednjih plasti.



Slika 27: Nekoliko kompleksnejša množica podatkov in dve različici arhitekture modela, ki bi se z njo morda lahko uspešno spopadli.

Opazimo, da smo v drugi varianti arhitekture modela s slike 27 uvedli dodatno plast računskih enot. Namenoma ta plast vsebuje le dve računski enoti, saj lahko njuni aktivaciji vizualiziramo v dvorazsežnem razsevnem diagramu. Ti dve aktivaciji nato pošljemo v končno enoto, logistično regresijo, ki vrača verjetnost ciljnega razreda. Ker imajo logistične regresije linearne odločitvene meje, bi torej moral biti razred točke v prostoru aktivacij predzadnje plasti naše arhitekture linearno ločljiv. In je res (slika 28)!



Slika 28: Odločitvena meja dvoplastnega modela in prostor aktivacij predzadnje plasti, ki (zaradi tega, da lahko aktivacije izrišemo v dvodimenzionalnem razsevnem diagramu) vsebuje le dve računski enoti.

Kombinacije logističnih regresij, ki smo jih raziskovali zgoraj, so se zgodovinsko uveljavile pod imenom *nevronske mreže*. Izraz je star, saj

te arhitekture prvotno niso bile zasnovane kot kombinacije logističnih regresij, kot smo jih predstavili tu, temveč kot računske enote, ki naj bi posnemale nevrone v možganih. O zgodovini morda nekoliko kasneje; zdaj je čas, da ta koncept formaliziramo in zapišemo nekaj kode za njegovo implementacijo.

Nevronske mreže kot postopek preoblikovanja podatkov

Preden se poglobimo v formalizem in implementacijo nevronskih mrež, še morda precej pomembna ugotovitev: zgoraj smo na koncu naših arhitektur modelov uporabili logistično regresijo. Končni model je bil torej eden od posplošenih linearnih modelov. Vse operacije pred to računsko enoto so potem služile pripravi podatkov za ta model, torej preoblikovanju podatkov tako, da jih lahko modeliramo s posplošenim linearnim modelom.

Nevronsko mrežo torej lahko obravnavamo kot transformacijo značilk za končni, posplošeni linearni model, uporabljen na teh novih značilkah. V nasprotju z ročno pripravo podatkov, kjer bi značilke torej oblikovali ročno oziroma za njihov izračun spisali neke funkcije, se te transformacije nevronska mreža nauči hkrati s končnim modelom, v enem samem optimizacijskem postopku. Mreža tako svojo notranjo transformacijo podatkov prilagodi zadnji plasti tako, da postane končna napovedna naloga čim enostavnejša oziroma da model čim bolj ustreže neki optimizacijski funkciji.

Formalni zapis nevronskih mrež

V svoji standardni usmerjeni obliki (t. im. *feedforward neural network*), kjer informacije tečejo od vhodnih spremenljivk do končne plasti nevronske mreže, je nevronska mreža parametrična funkcija $f_\theta : \mathbb{R}^d \to \mathbb{R}^k$, določena kot kompozicija afinih transformacij in po komponentah uporabljenih nelinearnih aktivacijskih funkcij. Mreža je urejena po plasteh in vsebuje vhodno plast (vhodne spremenljivke), eno ali več *skritih plasti* ter izhodno plast. Vsaka skrita plast vsebuje *skrite enote* (ali vozlišča), katerih aktivacije niso neposredno opazovane in služijo kot vmesne predstavitve vhodnih podatkov.

Druge arhitekture nevronskih mrež, med njimi konvolucijske, rekurentne in transformerske mreže, to osnovno obliko razširjajo, a za zdaj zadošča, da začnemo z osnovno, kanonično obliko, ki jo lahko kasneje poljubno nadgradimo.

Za vhod $x \in \mathbb{R}^d$ mreža z L plastmi izračuna zaporedje aktivacij

$$a^{(0)} = x, \qquad z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}, \qquad a^{(\ell)} = \sigma^{(\ell)}\left(z^{(\ell)}\right), \quad \ell = 1, \ldots, L,$$

kjer so plasti $\ell = 1, \ldots, L-1$ skrite plasti, plast $\ell = L$ pa izhodna plast. Pri tem sta $W^{(\ell)}$ in $b^{(\ell)}$ parametra (uteži in odmiki) plasti ℓ, $\sigma^{(\ell)}$ so nelinearne aktivacijske funkcije, parametri celotnega modela pa so

$$\theta = \{W^{(\ell)}, b^{(\ell)}\}_{\ell=1}^{L}.$$

Končni izhod $f_\theta(x) = a^{(L)}$ interpretiramo glede na nalogo, ki jo rešujemo, npr. kot verjetnosti pri klasifikaciji ali kot številčne vrednosti pri regresijskih problemih.

Tako kot pri vseh drugih modelih, ki smo jih doslej obravnavali, bomo tudi tu parametre modela θ določili iz učnih podatkov z minimizacijo izbrane kriterijske funkcije $\mathcal{L}(f_\theta(x), y)$, ki bo skladna z naravo problema. Tipično bomo te funkcije izbirali med temi, ki smo jih spoznali pri posplošenih linearnih modelih, ali pa uporabili njihove razširitve za zahtevnejše in bolj strukturirane tipe podatkov.

Implementacija

Računsko pretvorbo vhodnih podatkov, ki jo določa nevronska mreža, lahko predstavimo kot računski graf in ga nato uporabimo za izračun gradientov parametrov modela. Pri tem velja vse, kar smo uvedli v prejšnjih poglavjih pri bolj enostavnih modelih: v implementaciji bomo za učenje uporabili že razvito funkcijo `train`. Uporabimo lahko tudi regularizacijo, podobno kot smo jo uporabili za glajenje posplošenih linearnih modelov. Z izrazom "implementacija" tu zato mislimo predvsem, ali pa morda celo samo na konstrukcijo računskega grafa, ki ustreza izbrani arhitekturi nevronske mreže.

Začnimo sicer s krajšo razpravo o aktivacijskih funkcijah. V zgornjih razdelkih smo uporabljali logistično funkcijo, a v notranjih vozliščih nevronskih mrež pogosteje uporabljamo enostavnejše nelinearnosti, ki omogočajo učinkovitejše učenje in hitrejšo konvergenco k optimalnim rešitvam. Tipično se za notranja vozlišča tako namesto logistične uporablja funkcija ReLU (*Rectified Linear Unit*),

$$\mathrm{ReLU}(z) = \max(0, z),$$

ki vrne vhodno vrednost, če je ta pozitivna, sicer pa vrednost nič. Gre za preprosto, a zelo učinkovito nelinearno funkcijo, ki omogoča hitrejše učenje in omili problem izginjajočih gradientov (*vanishing gradient problem*). Do teh pride, ko se gradienti pri povratnem širjenju skozi mrežo postopoma manjšajo, kar je značilno predvsem pri sigmoidnih ali `tanh` aktivacijah. Posledično zgodnejše plasti prejmejo le izjemno šibek učni signal in se njihovi parametri posodabljajo zelo počasi. Funkcija ReLU ta problem omili, saj ima za pozitivne vhode konstanten odvod (enak 1), kar omogoča učinkovitejše širjenje gradientov ter hitrejše in stabilnejše učenje. Za podporo funkciji ReLU zato razširimo našo knjižnico za avtomatsko odvajanje (razred `Value`) z njenim izračunom v smeri naprej in povratnim širjenjem gradientov:

```
def relu(self):
    out = Value(0 if self.data < 0 else self.data, (self,), 'ReLU')
```

```
    def _backward():
        self.grad += (out.data > 0) * out.grad
    out._backward = _backward

    return out
```

Nevronsko mrežo zgradimo z uporabo treh razredov: nevrona (računske enote), plasti nevronske mreže, ki vsebuje nevrone na isti ravni, ter same nevronske mreže, ki te plasti povezuje med seboj. Začnimo z osnovno računsko enoto:

```
class Neuron:

    def __init__(self, nin, activation='relu'):
        self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
        self.b = Value(0)
        self.activation = activation
        self.out = None

    def __call__(self, x):
        act = sum((wi*xi for wi,xi in zip(self.w, x)), self.b)
        out = act.sigmoid() \
           if self.activation == 'sigmoid' else act.relu()
        self.out = out
        return out

    def parameters(self):
        return self.w + [self.b]

    def __repr__(self):
        return f"Neuron({len(self.w)})"
```

Razred `Neuron` implementira nevron, ki sprejme `nin` vhodov, izračuna njihovo uteženo vsoto, prišteje začetno vrednost ter rezultat pošlje skozi aktivacijsko funkcijo, ki je privzeto ReLU. Funkcija `parameters()` vrne seznam parametrov nevrona.

Nevrone posamezne plasti povežemo v razredu `Layer`:

```
class Layer:

    def __init__(self, nin, nout, **kwargs):
        self.neurons = [Neuron(nin, **kwargs) for _ in range(nout)]

    def __call__(self, x):
        out = [n(x) for n in self.neurons]
        return out[0] if len(out) == 1 else out

    def parameters(self):
        return [p for n in self.neurons for p in n.parameters()]
```

```
    def __repr__(self):
        return f"Layer of [{', '.join(str(n) for n in self.neurons)}]"
```

Razred `Layer` predstavlja en sloj mreže, torej plasti nevronov, ki pri polno povezani mreži, kot je ta, ki jo gradimo, vsi prejmejo isti vhod. Izračun v metodi `__call__` pokliče vsak nevron posebej, pri čemer vsak neodvisno izračuna svojo aktivacijo. Če plast vsebuje le en nevron (torej gre za izhodni nevron), klic vrne skalar, sicer pa seznam aktivacij. Parametre plasti dobimo tako, da združimo sezname parametrov vseh njenih nevronov.

Plasti v nevronsko mrežo poveže razred `NeuralNetwork`:

```
class NeuralNetwork:

    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+1], \
            activation="sigmoid" if i == len(nouts)-1 else "relu")
            for i in range(len(nouts))]

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        return [p for layer in self.layers
            for p in layer.parameters()]

    def loss(self, X, ys):
        eps = 1e-8
        yhats = [self(x) for x in X]
        return -sum(y * (yhat + eps).log() + \
           (1-y) * (1-yhat + eps).log() \
           for y, yhat in zip(ys, yhats)) / len(ys)

    def __repr__(self):
        return f"NN of [{', '.join(str(layer)
            for layer in self.layers)}]"
```

Razred `NeuralNetwork` zgradi mrežo tako, da njene plasti zaporedno poveže med seboj. Uporabnik poda število vhodov ter v seznamu število nevronov v posamezni skriti plasti. Vsi nevroni v skritih plasteh uporabljajo funkcijo ReLU, medtem ko zadnja plast uporablja sigmoidno funkcijo za izračun verjetnosti razredov. Funkcija `__call__()` implementira prehod informacij v smeri naprej (angl. *forward pass*). Vsaka plast sprejme aktivacije prejšnje plasti in svoj izhod posreduje naslednji, dokler izhodna plast ne vrne napovedi.

Velja pripomniti, da tu gradimo klasifikacijsko nevronsko mrežo. Našo implementacijo bi bilo mogoče enostavno spremeniti za kakšno drugo, recimo regresijsko nalogo, z uporabo drugačne aktivacijske funkcije v zadnji plasti.

Mreža svoje parametre zbere iz parametrov posameznih plasti, te pa jih zberejo iz svojih nevronov. Spomnimo, da se bodo ti parametri med gradientnim sestopom posodabljali glede na njihove gradiente.

V zgornjo implementacijo smo vključili tudi funkcijo `loss` za binarno klasifikacijo, ki ustreza negativnemu log-verjetju (križni entropiji), uporabljeni pri logistični regresiji. Bralec lahko ta del implementacije brez težav prilagodi za splošnejšo uporabo ali pa določi ustrezen nadrazred, iz katerega nato izpelje modele nevronskih mrež za različne analitske naloge.

Učenje modela z dvema skritima plastema, kot smo ga uporabili za prikaz rezultatov na sliki 28, je nato povsem preprosto:

```
n_hidden = [4, 2]
model = NeuralNetwork(2, n_hidden + [1])
train(model, X, ys, learning_rate=0.2, n_epochs=2000,
      batch_size=50, report_every=100)
```

Konvergenca je pri tej implementaciji in naših podatkih kljub naivni implementaciji računanja gradientov razmeroma hitra. Na tem mestu velja poudariti, da je naša implementacija z uporabo knjižnice za avtomatsko odvajanje iz prejšnjih poglavij namenjena predvsem učenju in razumevanju konceptov, za resnejšo uporabo pa bomo od tu dalje posegli po zmogljivejših Pythonovih knjižnicah, kot je `pytorch`.

Regularizacija

Modeli nevronskih mrež so lahko zelo prilagodljivi in tipično imajo veliko število parametrov, zato so še posebej nagnjeni k preveliki prilagoditvi učni množici. Tako kot pri modelih, ki smo jih obravnavali v prejšnjih poglavjih, lahko tudi tu uporabimo regularizacijske tehnike, s katerimi model omejimo, da se bolje posplošuje na nove, nevidene podatke.

Neposredna razširitev regularizacije iz posplošenih linearnih modelov je uporaba kazenskih členov nad parametri modela. Najpogostejša je kazen ℓ_2, kjer, kot že sicer vemo, kriterijski funkciji dodamo člen, sorazmeren vsoti kvadratov uteži,

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_{\ell} \|W^{(\ell)}\|_2^2,$$

pri čemer parameter $\lambda > 0$ določa moč regularizacije. Takšna regularizacija spodbuja mrežo, da ohranja majhne uteži, kar vodi do bolj gladkih preslikav in manjše občutljivosti na šum v podatkih. Kazen ℓ_1, ki se nam je sicer zdela zelo uporabna pri posplošenih linearnih

modelih zaradi izbora značilk,

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_{\ell} \|W^{(\ell)}\|_1,$$

spodbuja redkost (angl. *sparsity*), a se v nevronskih mrežah uporablja precej redkeje oziroma skoraj nikoli. Nevronske mreže se namreč običajno zanašajo na porazdeljene predstavitve, kjer se veliko majhnih prispevkov združuje v uporabne značilke, poleg tega pa lahko negladka narava kazni ℓ_1 oteži optimizacijo.

V praksi kazen ℓ_2 pogosto implementiramo neposredno v optimizacijskem postopku kot t. im. *weight decay*. Namesto da bi regularizacijski člen eksplicitno dodali kriterijski funkciji, prilagodimo pravilo za posodobitev parametra w:

$$w \leftarrow (1 - \eta\lambda)\, w - \eta \nabla_w \mathcal{L},$$

kjer je η hitrost učenja, z λ pa uravnavamo stopnjo regularizacije. Tu pri vsakem koraku gradientnega sestopa uteži nekoliko skrčimo proti nič. Prav zato je *weight decay* standardni način implementacije regularizacije ℓ_2 v sodobnih knjižnicah za nevronske mreže, oba izraza pa se v praksi pogosto uporabljata skoraj kot sopomenki.

Tretja pogosto uporabljena regularizacijska tehnika pa je *dropout*. Med učenjem vsako aktivacijo $a_i^{(\ell)}$ neodvisno nastavimo na nič z verjetnostjo p, z verjetnostjo $1 - p$ pa jo ohranimo. To lahko zapišemo kot

$$\tilde{a}^{(\ell)} = m^{(\ell)} \odot a^{(\ell)}, \qquad m_i^{(\ell)} \sim \text{Bernoulli}(1-p),$$

kjer $\odot$ označuje množenje po komponentah. S tem preprečimo, da bi se mreža preveč zanašala na posamezne računske enote, in jo prisilimo, da razvije redundantne in bolj robustne notranje predstavitve. Pristop izpuščanja povezav lahko razumemo tudi kot obliko ansambelskega učenja: pri vsaki iteraciji dejansko učimo nekoliko drugačno podmrežo, dobljeno z odstranitvijo naključne podmnožice enot, pri čemer vse te podmreže delijo iste parametre. Končni model lahko zato razumemo kot približek povprečenja napovedi velike množice takšnih podmrež. Med uporabo modela potem uporabimo vse enote. V sodobnih implementacijah se ustrezno skaliranje običajno izvede že med učenjem (t. i. "inverted dropout"), zato med uporabo modela niso potrebne dodatne prilagoditve.

Dropout so leta 2014 predlagali Srivastava, Hinton, Krizhevsky, Sutskever in Salakhutdinov v članku, objavljenem v reviji *Journal of Machine Learning Research*.

Dropout se je veliko uporabljal v zgodnjem obdobju učenja nevronskih mrež (približno med letoma 2012 in 2016), danes pa je v sodobnih arhitekturah manj pogost. Še vedno je uporaben pri manjših modelih ali kadar imamo malo podatkov, medtem ko sta regularizacija z *weight decay* in bogatenje podatkov (*data augmentation*) danes običajnejša izbira.
Večina sodobnih knjižnic za globoko učenje, kot je PyTorch, privzeto implementira "inverted dropout".

Posebej preprosta, a učinkovita oblika regularizacije je *zgodnje ustavljanje* (*early stopping*). Namesto da kriterijsko funkcijo na učni množici minimiziramo neomejeno dolgo, spremljamo uspešnost na validacijski množici in učenje ustavimo pri iteraciji t^*, kjer je izguba na validacijski množici $\mathcal{L}_{\text{val}}^{(t)}$ najmanjša. Po tej točki nadaljnja optimizacija običajno še zmanjšuje napako na učni množici, a povečuje napako na validacijski množici, kar kaže na prenaučenost. Zgodnje

ustavljanje lahko razumemo tudi kot implicitno omejevanje norme parametrov: gradientni sestop najprej zajame grobo strukturo podatkov, šele kasneje pa začne prilagajati tudi šum. Če učenje prekinemo dovolj zgodaj, preprečimo, da bi se model preveč prilagodil učni množici in postal pretirano kompleksen. V praksi je zgodnje ustavljanje zelo enostavno implementirati in pogosto pomembno izboljša sposobnost posploševanja.

Regularizacijo lahko dosežemo tudi na ravni podatkov z *bogatenjem podatkov* (*data augmentation*). Namesto da spreminjamo model, razširimo učno množico tako, da iz obstoječih primerov ustvarimo dodatne primere, pri čemer ohranimo njihove razrede. V dvodimenzionalni množici podatkov lahko na primer točko $x = (2, 1)$ rahlo pošumimo in ustvarimo bližnje točke, kot sta $(2.1, 1.0)$ ali $(1.9, 1.2)$, ki obdržijo isti razred. S tem predpostavimo, da majhne spremembe vhoda ne bi smele vplivati na izhod modela. Takšen postopek poveča raznolikost učnih podatkov in zmanjša težnjo modela, da bi si zapomnil posamezne primere, kar poveča robustnost in splošnost.

Inicializacija in normalizacija

Učenje nevronskih mrež z gradientnimi metodami je zelo občutljivo na velikost vhodov, izbor aktivacij in parametrov. Če teh količin ustrezno ne nadzorujemo, lahko optimizacija postane zelo počasna ali celo povsem odpove.

Preprost, a zelo učinkovit korak je normalizacija vhodnih značilk. V praksi posamezno značilko običajno standardiziramo (povprečje nič in enotska varianca). S tem zagotovimo, da so vsi vhodi na primerljivi skali, kar vodi do stabilnejših gradientov in hitrejše konvergence gradientnega sestopa.

Enako pomembna je inicializacija parametrov mreže. Če uteži inicializiramo s prevelikimi vrednostmi, lahko aktivacije med širjenjem skozi plasti hitro narastejo, kar vodi do nestabilnega obnašanja in eksplodirajočih gradientov. Če pa so uteži premajhne, se aktivacije krčijo proti nič in gradienti lahko izginejo, zaradi česar se mreža ne more učinkovito učiti. Sodobne inicializacijske sheme ta problem rešujejo tako, da začetne uteži izberejo na način, ki ohranja varianco aktivacij približno konstantno skozi vse plasti. Pogosti izbiri sta inicializacija Xavier (primerna za sigmoidne ali `tanh` aktivacije) ter inicializacija He (namenjena mrežam z ReLU).

Inicializacija Xavier (Glorot) iz leta 2010 postavi $\mathrm{Var}(w) = \frac{2}{n_{\text{in}}+n_{\text{out}}}$ in je primerna za sigmoidne oziroma `tanh` aktivacije. Inicializacija He (2015) uporablja $\mathrm{Var}(w) = \frac{2}{n_{\text{in}}}$, kar bolje ustreza mrežam z ReLU, kjer je približno polovica aktivacij enaka nič. Obe metodi poskušata ohraniti aktivacije in gradiente na stabilni skali skozi vse plasti ter tako izboljšati učenje.

V zahtevnejših primerih lahko med učenjem normaliziramo tudi vmesne aktivacije, na primer z uporabo paketne normalizacije (*batch normalization*). Takšne tehnike dodatno stabilizirajo optimizacijo, vendar za potrebe tega poglavja že skrbna priprava vhodnih podatkov in smiselna inicializacija zagotovita večino praktičnih koristi.

Globina, širina in kapaciteta modela

Arhitekturo nevronske mreže določata njena *globina* (število plasti) in *širina* (število enot v posamezni plasti). Povečevanje ene ali druge omogoča mreži predstavitev kompleksnejših funkcij. Pravzaprav lahko že mreža z eno samo skrito plastjo aproksimira poljubno zvezno funkcijo na omejenem območju, če vsebuje dovolj veliko število enot. To opažanje, pogosto imenovano *lastnost univerzalne aproksimacije*, nakazuje, da globina strogo gledano ni nujna za izrazno moč modela. Vendar pa lahko takšne plitve mreže za predstavitev funkcij, ki jih globlje arhitekture opišejo precej bolj kompaktno, zahtevajo nepraktično veliko število enot.

Lastnost univerzalne aproksimacije sta dokazala Cybenko (Approximation by superpositions of a sigmoidal function, *Mathematics of Control, Signals, and Systems*, 1989) in Hornik (Approximation capabilities of multilayer feedforward networks, *Neural Networks*, 1991). Dostopnejšo razlago podajajo Goodfellow in sod. v knjigi *Deep Learning* (2016), poglavje 6.

Globoka nevronska mreža je nevronska mreža z več kot eno skrito plastjo. Izraz "globoka" se nanaša na nalaganje plasti, ki modelu omogoča učenje vedno abstraktnejših predstavitev. Izraz se je uveljavil v 2000-ih letih ob ponovnem vzponu večplastnih nevronskih mrež, predvsem po zaslugi dela Geoffreyja Hintona in sodelavcev.

Globina omogoča nevronskim mrežam gradnjo *kompozicijskih predstavitev*. Vsaka plast transformira izhod prejšnje plasti in iz vhodov postopoma gradi vedno abstraktnejše značilke. Takšna hierarhična struktura pogosto vodi do učinkovitejših predstavitev: funkcije, ki bi v plitvi mreži zahtevale veliko število enot, lahko globoka mreža implementira z bistveno manj parametri. Širina pa določa bogastvo predstavitev znotraj posamezne plasti, saj mreži omogoča učenje več različnih značilk na isti ravni abstrakcije. V praksi tako k izrazni moči modela prispevata tako globina kot širina, njuno ustrezno razmerje pa je odvisno od konkretnega problema.

Skupna *kapaciteta* nevronske mreže, to je, njena sposobnost aproksimacije širokega razreda funkcij, je zato določena z arhitekturo in številom parametrov. Modeli s premajhno kapaciteto bodo preenostavni (angl. *underfitting*) in ne bodo zajeli pomembnih vzorcev, medtem ko se lahko preveliki modeli preveč prilagodijo učni množici (angl. *overfitting*). Tako kot pri drugih modelih je tudi pri nevronskih mrežah nadzor kapacitete s premišljeno izbiro arhitekture in regularizacijo ključen za dobro uspešnost. V praksi izbiro primerne globine in širine pogosto vodijo eksperimentiranje, domensko znanje in uspešnost na validacijski množici, ne pa stroga teoretična pravila.

Hitrost učenja in njeno prilagajanje

Medtem ko oblika modela določa, katere funkcije lahko predstavimo, hitrost učenja v veliki meri določa, ali bomo te funkcije v praksi sploh uspeli poiskati. Tudi ob pravilno izračunanih gradientih in dobro zasnovani arhitekturi lahko neustrezno veliki koraki povsem preprečijo konvergenco. Hitrost učenja je zato eden ključnih praktičnih parametrov: tudi s pravilnimi gradienti lahko neustrezna velikost koraka prepreči uspešno učenje modela.

Hitrost učenja η določa velikost posodobitev parametrov. Če je prevelika, lahko optimizacija postane nestabilna in ne konvergira; če je

premajhna, učenje napreduje zelo počasi ali celo obstane. Pogosta strategija je zato, da začnemo z razmeroma veliko hitrostjo učenja in jo med učenjem postopoma zmanjšujemo. Takšni postopki spreminjanja hitrosti učenja (angl. *learning rate schedules*), na primer stopničasto ali eksponentno zmanjševanje, omogočajo hitro začetno napredovanje in nato natančnejše prilagajanje v bližini rešitve. V praksi običajno zadostujejo že preprosti urniki v kombinaciji s spremljanjem uspešnosti na učni in validacijski množici, skrbna nastavitev hitrosti učenja pa ima pogosto večji vpliv kot bolj zapletene spremembe same arhitekture modela.

Nevronske mreže s knjižnico PyTorch

Podobno implementacijo nevronske mreže, kot smo jo "ročno" zgradili v tem poglavju, gradi tudi knjižnica PyTorch. Seveda na veliko bolj premišljen način, saj pri tem uporablja vektorje in večrazsežne matrike, ki jim pravimo tudi tenzorji. Za podatke iz slike 26 lahko v PyTorchu zgradimo model tako, da sestavimo zaporedje plasti:

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(2, 2),
    nn.ReLU(),
    nn.Linear(2, 1),
    nn.Sigmoid(),
)
```

Prva plast `nn.Linear(2, 2)` sprejme dve vhodni značilki in izračuna dve linearni kombinaciji vhodov, torej aktivaciji dveh skritih enot. Nato funkcija `nn.ReLU()` nad njima uporabi nelinearno transformacijo. Druga linearna plast `nn.Linear(2, 1)` ti dve aktivaciji združi v eno samo vrednost, funkcija `nn.Sigmoid()` pa jo pretvori v verjetnost ciljnega razreda.

Vhodne podatke za učenje lahko preberemo iz vhodne datoteke in jih pretvorimo v tenzorje.

```
import pandas as pd
import torch

df = pd.read_excel(path)
y = torch.tensor(df.iloc[:, 0].astype(float).values, dtype=torch.float32).view(-1, 1)
X = torch.tensor(df.iloc[:, 1:3].astype(float).values, dtype=torch.float32)
```

Prvi stolpec podatkovne tabele predstavlja razred y, preostala dva stolpca pa vhodni značilki X. Ker PyTorch pričakuje numerične tenzorje določene oblike, podatke pretvorimo v tip `float32`, pri ciljni

spremenljivki pa z `view(-1, 1)` zagotovimo obliko stolpčnega vektorja.

Za učenje moramo določiti cenovno funkcijo in optimizacijski postopek, ki bo posodabljal parametre modela glede na izračunane gradiente.

```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.2)
```

Kriterijska funkcija `BCELoss` implementira nam že znano križno entropijo za binarno klasifikacijo, optimizator `Adam` pa hitrost posodabljanja parametrov prilagaja posameznim utežem in zato pogosto omogoča hitrejše ter stabilnejše učenje.

Namesto optimizatorja `Adam` bi lahko uporabili tudi stohastični gradientni sestop `SGD`, ki smo ga uporabljali vse do sedaj in je konceptualno preprostejši, a bomo ostali pri postopku Adam, ki zahteva manj ročnega prilagajanja hitrosti učenja.

Učenje implementiramo s spodnjo kodo:

```
n_epochs = 10000

for epoch in range(n_epochs):
    y_pred = model(X)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print(f"epoch {epoch:4d} | loss {loss.item():.6f}")

print(f"Final loss: {loss.item():.6f}")
```

Koda je sicer silno podobna tej, ki smo jo sestavili v prejšnjih poglavjih in uporabljali z našo knjižnico za strojno računanje gradientov. V vsaki iteraciji zgradimo računski graf, ga s prehodom skozi mrežo z učnimi podatki opremimo z vrednostmi izračunanih tenzorjev (spremenljivk v računskem grafu), gradiente nastavimo na nič ter s klicem `backward()` izračunamo gradiente. Nato z `optimizer.step()` posodobimo parametre modela.

Onkraj polno povezanih mrež

Nevronske mreže, ki smo jih obravnavali doslej, so bile *polno povezane* oziroma goste, saj je bila vsaka enota v posamezni plasti povezana z vsemi enotami prejšnje plasti. Takšne arhitekture so zelo splošne in lahko načeloma aproksimirajo zelo širok razred funkcij. Vendar pa imajo lahko vhodni podatki v številnih aplikacijah dodatno strukturo, ki jo lahko izkoristimo za gradnjo učinkovitejših in uspešnejših modelov. Če to strukturo vključimo v arhitekturo, v model vnesemo

določeno induktivno pristranost (angl. *inductive bias*), ki lahko pomembno izboljša učenje.

Odličen primer so *konvolucijske nevronske mreže* (*convolutional neural networks*, CNN), ki so zasnovane za podatke s prostorsko strukturo, kot so slike, ali pa za zaporedne podatke. Namesto da bi vsak vhod povezali z vsako enoto, CNN uporabljajo lokalne povezave in deljene uteži: isti majhen filter uporabijo na različnih delih vhoda. S tem zmanjšajo število parametrov in omogočijo mreži, da zaznava lokalne vzorce (na primer robove ali teksture) ne glede na njihov položaj v sliki.

Konvolucijske nevronske mreže je v poznih osemdesetih letih predstavil Yann LeCun s sodelavci. Ena prvih uspešnih aplikacij je bila mreža LeNet-5 (1998), razvita v laboratorijih AT&T Bell Labs in objavljena v *Proceedings of the IEEE*, kjer so CNN uporabili za prepoznavanje ročno pisanih števk.

Drug pomemben razred so *rekurentne nevronske mreže* (*recurrent neural networks*, RNN), ki so prilagojene zaporednim podatkom. Pri teh modelih izračun poteka korak za korakom, mreža pa ohranja skrito stanje, ki povzema pretekle vhode, zato so primerne za naloge, povezane s časovnimi vrstami ali besedili. V zadnjem času so pri številnih nalogah modeliranja zaporedij prevlado prevzele *transformerske* arhitekture ki temeljijo na mehanizmih pozornosti (*attention*), s katerimi lahko model neposredno povezuje različne dele vhodnega zaporedja med seboj in tako učinkovito modelira dolgoročne odvisnosti.

RNN so bile prav tako predstavljene zelo zgodaj, v osemdesetih letih, med drugim v delih Johna Hopfielda (1982) ter Davida Rumelharta, Geoffreyja Hintona in Ronalda Williamsa (1986).

Transformerske arhitekture so bile predlagane nedavno, uvedli so jih Vaswani in sod. leta 2017. Zaradi sposobnosti učinkovitega modeliranja dolgoročnih odvisnosti s pomočjo mehanizmov pozornosti (*attention*) in vzporednega računanja so v številnih aplikacijah, zlasti pri obdelavi naravnega jezika, večinoma nadomestile rekurentne modele.

Nevronske mreže pa niso omejene zgolj na napovedovanje izhodov iz vhodov, temveč jih lahko uporabimo tudi za *generiranje* podatkov. Pri *generativnem strojnem učenju* želimo modelirati osnovno porazdelitev podatkov tako, da lahko iz nje vzorčimo nove primere. Primeri vključujejo modele, ki generirajo realistične slike, besedila ali zvok. V to skupino sodijo arhitekture, kot so avtokodirniki (*autoencoders*), ki se učijo stisnjenih predstavitev s pomočjo rekonstrukcije vhodov, generativne nasprotniške mreže (*generative adversarial networks*, GAN) ter jezikovni modeli, temelječi na transformerskih arhitekturah. Čeprav vsi ti modeli gradijo na istih principih, ki smo jih obravnavali v tem poglavju – kompozicijah odvedljivih funkcij, učenih z gradientno optimizacijo – so njihovi cilji in uporabe precej širši ter danes predstavljajo samo jedro sodobne umetne inteligence.

Razložljivost nevronskih mrež

Razložljivost je danes zelo aktivno raziskovalno področje. Metode, kot so pripisovanje pomembnosti značilkam (*feature attribution*), zemljevidi občutljivosti (*saliency maps*) in analiza mehanizmov pozornosti (*attention analysis*), poskušajo nevronske mreže narediti bolj razumljive, zlasti v aplikacijah z visokimi tveganji.

Očitno vprašanje je, ali lahko nevronske mreže interpretiramo podobno kot preprostejše modele. Pri linearnih modelih lahko razlaga modelov izhaja neposredno iz modela samega: koeficienti linearne regresije opisujejo vpliv posamezne vhodne spremenljivke na izhod. Parametri nevronske mreže pa so porazdeljeni skozi več plasti in med seboj delujejo na izrazito nelinearen način. Posamezne uteži ali enote zato praviloma nimajo preproste, samostojne interpretacije. Model moramo razumeti predvsem kot sistem, ki se uči zaporedja transformacij, s katerimi vhodne podatke preslika v predstavitev,

uporabno za končno nalogo.

Vsaj delni vpogled v delovanje nevronskih mrež pa lahko dobimo z analizo njihovih napovedi, opazovanjem, kako se izhodi spreminjajo glede na vhode, ali z vizualizacijo vmesnih predstavitev, kot smo to naredili že v preprostem primeru v tem poglavju. Takšni pristopi omogočajo bolj globalno razumevanje tega, česa se je model naučil, četudi preprosta interpretacija na ravni posameznih parametrov ni na voljo.

V nadaljevanju se zato lotimo dveh pristopov. Prvi bo neposreden: z računskim grafom lahko enostavno ugotovimo, kako vhodne spremenljivke vplivajo na vrednosti izhodnih. Gradiente, izračunane iz računskega grafa, smo do sedaj uporabljali pri osveževanju vrednosti parametrov, a prav tako jih lahko izračunamo za vhodne spremenljivke in s tem dobimo informacijo, kako spremembe vhodnih spremenljivk vplivajo na izhod mreže. Drugi pristop bo namesto gradientov raje spreminjal vhode, jih "zamajal", in pogledal, kako se spremeni izhod. Oba pristopa bomo uporabili za razlago vpliva atributov pri izbranem primeru, nekako tako, kot bi uporabili nomogram, le da ga, prvotno, zgradimo za en sam primer. Pregled čez celoten spekter primerov lahko seveda dobimo tako, da tovrstno razlago uporabimo za vse primere v (na primer) testni množici, in tovrstne "razlage" grafično prikažemo v vizualizaciji, ki bo za take kompleksnejše modele nadomestila nomogram.

Shapleyjeve razlage s seštevanjem prispevkov

Pričnimo z drugo omenjeno tehniko, kjer (za zdaj) predpostavimo, da lahko uporabimo modele tako, da iz njih izključimo izbran nabor spremenljivk in naročimo modelu, da upošteva samo vse ostale spremenljivke. Začnimo s primerom. Recimo, da opazujemo ceno enosobnih stanovanj v nekem malem mestecu, kjer uporabimo model, ki pravi, da je cena odvisna od oddaljenosti stanovanja od centra, starosti stanovanja in lege (osončenost). Model smo zgradili iz podatkov, zdaj pa za novo sončno stanovanje blizu centra, za katerega je model napovedal ceno 250,000 EUR, zanima, katera vhodna spremenljivka je k tej napovedi najbolj prispevala.

Da bi pravično ocenili prispevek vrednosti vsakega vhodnega atributa, moramo njegov prispevek opazovati v kontekstu vseh ostalih spremenljivk, to je v kontekstu koalicij. Na primer, ko nekaj o stanovanju že vemo, recimo poznamo njegovo starost, nas zanima, koliko pri oblikovanju cene stanovanja prispeva vedenje o oddaljenosti stanovanja od centra? Ali pa, če o stanovanju ne vemo še ničesar (napoved modela bi takrat morala biti enaka povprečni ceni stanovanj v naši bazi podatkov), kako se napovedana cena spremeni, če zvemo za

njegovo starost? In, na primer, če že vemo, da imamo opravka z novim stanovanjem v centru mesta, kakšna je sprememba napovedane cene, če zvemo, da je lega stanovanja sončna?

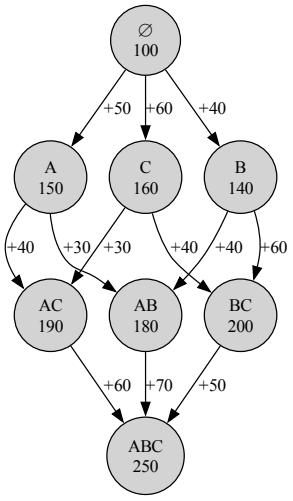
Za poenostavitev notacije naše atributne vrednosti poimenujmo:

- A: oddaljenost (blizu centra)
- B: starost (novo stanovanje)
- C: lega (sončna lega)

Model sedaj povprašajmo po napovedi vrednosti stanovanja pri vseh možnih kombinacijah atributov. Vrednosti teh, in spremembe pri dodajanju ene od atributnih vrednosti na vhod modela, lahko predstavimo z grafom s slike 29, ki ga lahko imenujemo mreža podmnožic (angl. *subset lattice*) ali pa kar mreža koalicij atributov (angl. *Shapley coalition lattice*). V tem grafu vsako vozlišče predstavlja možno kombinacijo, oziroma koalicijo atributov na vhodu modela, prehodi v grafu pa spremembe koalicij, ko zvemo za eno dodatno vrednost atributa.

Lloyd Shapley je leta 1953 predstavil Shapleyjeve vrednosti v članku *A Value for n-Person Games*, objavljenem v zborniku *Contributions to the Theory of Games II*.

Če ne vemo ničesar o stanovanju ($\varnothing$), je napovedana vrednost 100,000 € (recimo, malo je nizka, a za primer bo zadoščalo). Če zvemo, da je stanovanje blizu centra mesta (A), se vrednost poveča na 150,000 €, razlika, ki nam jo prinese vedenje o tem atributu, pa je +50,000 €. Če po drugi strani vemo, da gre za novo stanovanje v centru mesta (AB), za katerega nam model napove, da stane 180,000 €, nam dodatno vedenje o njegovi sončni legi (C) prispeva k ceni 70,000 €.



Slika 29: Mreža koalicij za model napovedovanja vrednosti nepremičnine (A: stanovanje je blizu centra, B: je novo, C: na sončni legi.)

Shapleyjeve vrednosti razdelijo razliko med napovedjo, ko poznamo vrednosti vseh atributov, in napovedjo, ko vrednosti teh atributov ne poznamo. Zanima nas sedaj, od kod razlika pri novem sončnem stanovanju v centru oziroma kateri od atributov prispeva največ k tej razliki. Ta je po Shapleyju enaka povprečni dodani vrednosti cene stanovanja v kateri koli situaciji, torej pri vseh možnih koalicijah ostalih atributnih vrednosti.

Da bi jih opazovali, poglejmo, na koliko možnih načinov se skozi mrežo koalicij sprehodimo od popolne nevednosti (ne poznamo vrednosti nobenega atributa) do situacije, ko poznamo vrednosti vseh (ABC). Vseh možnih takšnih poti je šest, kar je enako tudi številu permutacij A, B in C ($3! = 3 \times 2 = 6$):

1. A-B-C
2. A-C-B
3. B-A-C
4. B-C-A

5. C-A-B
6. C-B-A

Pri vsaki permutaciji lahko zdaj izračunamo robni (marginalni) doprinos posameznega atributa, torej kolikšen je skok v ceni, ko to značilnost dodamo že obstoječi "koaliciji" prejšnjih.

Permutacija 1: $A \to B \to C$

- $\varnothing \to A$ (blizu): 100,000 → 150,000 → +50,000
- $A \to B$ (novo): 150,000 → 180,000 → +30,000
- $A, B \to C$ (sonce): 180,000 → 250,000 → +70,000

Permutacija 2: $A \to C \to B$

- $\varnothing \to A$ (blizu): 100,000 → 150,000 → +50,000
- $A \to C$ (sonce): 150,000 → 190,000 → +40,000
- $A, C \to B$ (novo): 190,000 → 250,000 → +60,000

Permutacija 3: $B \to A \to C$

- $\varnothing \to B$ (novo): 100,000 → 140,000 → +40,000
- $B \to A$ (blizu): 140,000 → 180,000 → +40,000
- $A, B \to C$ (sonce): 180,000 → 250,000 → +70,000

Permutacija 4: $B \to C \to A$

- $\varnothing \to B$ (novo): 100,000 → 140,000 → +40,000
- $B \to C$ (sonce): 140,000 → 200,000 → +60,000
- $B, C \to A$ (blizu): 200,000 → 250,000 → +50,000

Permutacija 5: $C \to A \to B$

- $\varnothing \to C$ (sonce): 100,000 → 160,000 → +60,000
- $C \to A$ (blizu): 160,000 → 190,000 → +30,000
- $A, C \to B$ (novo): 190,000 → 250,000 → +60,000

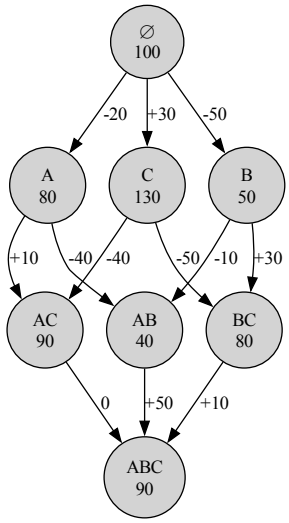
Značilnost	Prispevki po permutacijah	Povprečni prispevek
A	50, 50, 40, 50, 30, 50 → 270	45,000 €
B	30, 60, 40, 40, 60, 40 → 270	45,000 €
C	70, 40, 70, 60, 60, 60 → 360	60,000 €

Permutacija 6: $C \to B \to A$

- $\varnothing \to C$ (sonce): 100,000 → 160,000 → +60,000
- $C \to B$ (novo): 160,000 → 200,000 → +40,000
- $B, C \to A$ (blizu): 200,000 → 250,000 → +50,000

Za vsako značilnost izračunajmo povprečni doprinos: Največ je torej prispevala sončna lega (C), enako pa oddaljenost od centra in starost. Prispevki se tudi (priročno) seštejejo v napovedano vrednost, kjer je ta 100,000 €, ko o stanovanju ne vemo ničesar, in $100{,}000 + 45{,}000 + 45{,}000 + 60{,}000 = 250{,}000$ €, ko so nam znane vse tri vrednosti vhodnih atributov.

Za vajo še en primer, kjer tokrat upoštevamo, da je stanovanje daleč od centra, na vasi (A), staro (B) in s sončno lego (C). Mrežo koalicij kaže slika 30. Ker nam je postopek zdaj znan, lahko tokrat na hitro upoštevamo vse možne poti skozi graf in ugotovimo, da so spremembe takrat, ko zvemo za A (na vasi), enake -20, -20, -40, +10, -10, +10, oziroma je povprečna sprememba enaka -11,667 €. Za starost dobimo -31,667 €, za osončeno lego pa -33,333. Od tod razlika -10,000 € v primerjavi z napovedjo modela, ko ne vemo ničesar. Stanovanje je torej cenejše kot povprečno stanovanje, najbolj pa mu ceno zmanjšuje starost.



Slika 30: Mreža koalicij za model napovedovanja vrednosti starega (B), a lepo sončnega (C) stanovanja na vasi (A).

Shapleyjeve razlage za modele strojnega učenja

Kako zgornjo idejo Shapleyjevih razlag uporabimo v strojnem učenju? Problem je, da vsi modeli, ki smo jih gradili do sedaj, potrebujejo vrednosti vseh vhodnih spremenljivk za njihovo delovanje. Pri Shapleyjevih vrednostih pa moramo model večkrat vprašati po napovedi tudi takrat, ko poznamo le podmnožico atributov. V praksi spremenljivke "odstranimo" tako, da jih nadomestimo z njihovimi pričakovanimi vrednostmi oziroma opazujemo napoved modela pri različnih vrednostih, katerih porazdelitev dobimo iz učne množice. Postopek je sicer računsko zahteven, saj moramo za dano koalicijo model uporabiti mnogokrat, da lahko ocenimo povprečno vrednost modela pri dani koaliciji in različnih "manjkajočih" vrednostih atributov. Dodaten problem predstavlja tudi eksponentna rast števila koalicij s številom atributov. V praksi zato uporabljamo različne približke in optimizacije, kot so vzorčenje permutacij, aproksimacije z

lokalnimi modeli ali pa posebni algoritmi za posamezne vrste modelov (na primer drevesa odločanja), kjer lahko Shapleyjeve vrednosti izračunamo bistveno hitreje.

Oglejmo si primer uporabe te tehnike pri napovedovanju deleža telesne maščobe moških. Podatke bomo prebrali, jih preoblikovali za uporabo s knjižnico PyTorch, si zapomnili imena atributov in razdelili na učno in testno množico:

Podatki so na voljo .

```
import pandas as pd

d = pd.read_excel("body-fat-brozek.xlsx")
y = torch.tensor(d.iloc[:, 0].values, dtype=torch.float32).view(-1, 1)
X = torch.tensor(d.iloc[:, 1:].values, dtype=torch.float32)
feature_names = d.columns[1:]

n = int(0.7 * len(X))
perm = torch.randperm(len(X))
tr = perm[:n]
te = perm[n:]
```

Sledi standardizacija učnih podatkov,

```
m = X[tr].mean(0)
s = X[tr].std(0)
X = (X - m) / s
```

Bralec bo prav gotovo opazil, da smo standardizacijske parametre pridobili samo na učni množici, uporabili pa smo jih potem na celotni množici podatkov, torej ustrezno prilagodili tudi vhodne podatke testne množice. Gradnjo modela implementira spodnja koda:

```
net = nn.Sequential(
    nn.Linear(X.shape[1], 2),
    nn.ReLU(),
    nn.Linear(2, 1),
)

opt = torch.optim.Adam(net.parameters(), lr=0.1, weight_decay=1e-3)

for epoch in range(2000):
    pred = net(X[tr])
    loss = nn.functional.mse_loss(pred, y[tr])
    opt.zero_grad()
    loss.backward()
    opt.step()
    if epoch % 200 == 0:
        print(f"{epoch:5d} {loss.item():7.4f}")
```

Uspešnost učenja lahko ocenimo na testni množici:

```
net.eval()
with torch.no_grad():
    p = net(X[te])

r2 = 1 - (((y[te] - p) ** 2).sum() / ((y[te] - y[te].mean()) ** 2).sum())
print("test R2:", r2.item())
```

Metoda `net.eval()` model preklopi v način evalvacije, kjer se nekatere plasti, če te obstajajo (na primer `dropout` ali `batch normalization`), obnašajo drugače kot med učenjem. Blok `with torch.no_grad():` izklopi računanje gradientov, saj pri napovedovanju na testni množici teh ne potrebujemo. S tem zmanjšamo porabo pomnilnika in pospešimo izvajanje.

Vrednotenje računa statistiko R^2 na testni množici. Tu se je dobro prepričati, da je dobljena vrednost smiselna, saj je naša učna množica majhna in bi se nekoliko bolj kompleksni modeli lahko hitro preveč prilagodili učnim podatkom (*overfitting*). Primerno bi bilo rezultate primerjati tudi z navadno linearno regresijo, to pa lahko storimo (tudi) z uporabo zelo enostavnega nevronskega modela, kot je na primer:

```
net = nn.Sequential(
    nn.Linear(X.shape[1], 1),
)
```

Razred `nn.Sequential` smo tu skoraj nekako zlorabili in ga uporabili kot nadomestek za linearno regresijo: model vsebuje le eno linearno plast brez aktivacijskih funkcij, zato je njegovo delovanje enakovredno navadnemu linearnemu modelu. Knjižnica `pytorch` ima seveda tudi primernejše klice za gradnjo linearnih modelov, kot je `torch.nn.Linear`, lahko pa bi za klasično linearno regresijo uporabili knjižnico `scikit-learn` in razred `LinearRegression`. Rezultat bi moral biti enak, a je na bralcu, da to preveri.

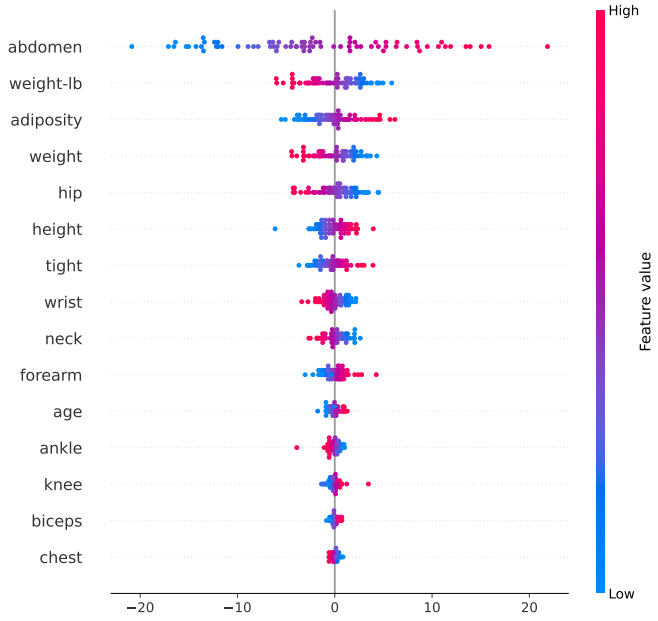
Ostane nam še izračun SHAPovih vrednosti. Te bomo izračunali za vse primere v testni množici in pri tem uporabili knjižnico `shap` ter funkcijo `DeepExplainer`, ki uporabi primere učne množice za oceno porazdelitev vrednosti atributov, s katerimi oceni pričakovano obnašanje modela pri "manjkajočih" atributih.

```
import shap

background = X[tr]
X_test = X[te]
explainer = shap.DeepExplainer(net, background)
shap_values = explainer.shap_values(X_test)

plt.figure()
shap.summary_plot(
    shap_values.squeeze(),
    X_test.numpy(),
    feature_names=feature_names,
    show=False,
)
plt.savefig("shap-summary.pdf", bbox_inches="tight")
plt.close()
```

Rezultat je podan na grafu na sliki 31, ki (pričakovano, in kot smo v prejšnjih analizah z linearno regresijo že pokazali) pokaže, da je za napoved deleža telesne maščobe daleč najpomembnejši atribut obseg trebuha (`abdomen`). Večje vrednosti tega atributa praviloma povečajo napoved modela, manjše pa jo zmanjšajo. Pomembni so tudi telesna teža, zamaščenost (adipioznost) in obseg bokov, medtem ko imajo na primer starost, obseg prsnega koša in bicepsa bistveno manjši vpliv na napoved modela.



Slika 31: Povzetek SHAPovih vrednosti za nevronski model napovedovanja deleža telesne maščobe. Vsaka točka predstavlja primer iz testne množice, njen položaj na osi x pa pove, koliko je posamezna značilnost prispevala k napovedi modela. Barva predstavlja vrednost atributa (modra: nizka vrednost, roza: visoka vrednost). Značilnosti so razvrščene po povprečnem absolutnem vplivu na napoved modela.

Gradientni prispevki

Podobno analizo kot z vrednostmi SHAPa lahko izvedemo z gradientnim pristopom. Tu, kot smo že pisali, gradientov ne uporabimo za posodabljanje parametrov modela, temveč jih izračunamo glede na vhodne spremenljivke. Gradient $\partial f(x)/\partial x_j$ pove, kako občutljiva je napoved modela na majhno spremembo j-te značilke pri izbranem primeru. Če je gradient velik, bi majhna sprememba te značilke močno spremenila napoved; če je blizu nič, je napoved v tej točki na značilko skoraj neobčutljiva.

Sam gradient nam poroča le o lokalni občutljivosti, ne poda pa nam prispevka dejanske vrednosti značilke pri danem primeru. Zato pogosto uporabimo produkt gradienta in vrednosti vhoda, torej

$$x_j \frac{\partial f(x)}{\partial x_j}.$$

Zgornje je približek k napovedi, kako vrednost atributa prispeva k odmiku značilke od referenčne vrednosti. Ker smo podatke standardizirali, je referenčna vrednost naravno enaka nič, zato produkt gradienta in vhoda smiselno meri prispevek posamezne značilke k napovedi za dani primer.

```
X_test_grad = X[te].clone().detach().requires_grad_(True)
preds = net(X_test_grad)
net.zero_grad()
preds.sum().backward()
grads_all = X_test_grad.grad.detach().numpy()
X_test_np = X_test_grad.detach().numpy()

grad_times_input = grads_all * X_test_np
```

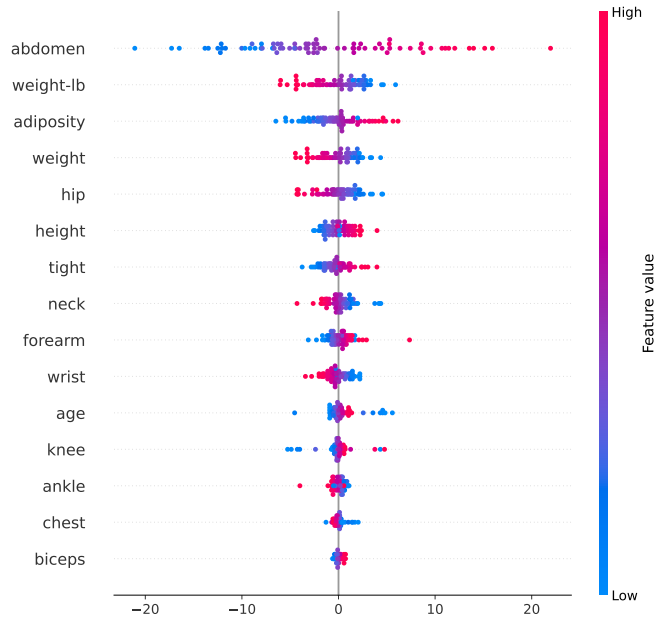
Za prikaz rezultatov lahko tu uporabimo kar implementacijo iz knjižnice `shap`:

```
plt.figure()
shap.summary_plot(
    grad_times_input,
    X_test_np,
    feature_names=feature_names,
    plot_type="dot",
    show=False,
)
plt.savefig("gradient-times-input.pdf", bbox_inches="tight")
plt.close()
```

Pri našem izjemno enostavnem modelu so rezultati takšne analize zelo podobni rezultatom SHAPa (slika 32). To ni presenetljivo. Uporabljena mreža je majhna, ima le eno ozko skrito plast, vhodni podatki so standardizirani, funkcija, ki se je model nauči, pa je zato razmeroma gladka in blizu linearni. V takih primerih lokalni gradienti pogosto dobro opišejo vpliv posameznih značilk.

Prednost gradientnih prispevkov je predvsem računska učinkovitost. Za vse primere jih dobimo z enim samim povratnim prehodom skozi mrežo, zato so posebej priročni pri velikih nevronskih mrežah in velikih podatkovnih množicah. Njihova slabost pa je, da so izrazito lokalni: povedo, kaj se zgodi pri majhni spremembi vhoda v okolici opazovanega primera, ne pa nujno, kako bi se napoved spreminjala pri večjih spremembah ali pri (rahlo) drugačnih kombinacijah značilk.

SHAPove vrednosti so po drugi strani običajno interpretacijsko bolj privlačne, saj prispevke značilk opredelijo glede na razliko med osnovno napovedjo in napovedjo za konkretni primer ter pri tem upoštevajo različne koalicije značilk. Zaradi tega so pogosto stabil-



Slika 32: Analiza modela za napovedovanje deleža telesne maščobe z gradientno izračunanimi prispevki vrednosti atributov k napovedani vrednosti modela.

nejše in bližje intuiciji o "prispevkih" posameznih atributov, vendar so računsko veliko zahtevnejše in pri nevronskih mrežah pogosto temeljijo na približkih. V praksi zato izbira metode je odvisna od namena: za hitro diagnostiko nevronskih modelov so gradientni pristopi zelo uporabni, za poročanje in razlago posameznih napovedi uporabnikom pa se pogosto uporabljajo SHAPove ali sorodne perturbacijske metode.

Od nevronskih mrež do generativne umetne inteligence

Ideje, razvite v tem poglavju, vodijo tudi do mnogo večjih generativnih modelov, ki danes prevladujejo v sodobni umetni inteligenci. Čeprav so današnji sistemi, temelječi na nevronskih mrežah, po arhitekturi in obsegu bistveno kompleksnejši, so še vedno zgrajeni iz istih osnovnih gradnikov: kompozicij odvedljivih transformacij, učenih iz podatkov z gradientno optimizacijo, pri čemer gradiente izračunamo z avtomatskim odvajanjem. Pri sodobnih generativnih modelih se spreminjajo predvsem arhitektura, obseg računanja in cilj učenja. Kompleksne globoke mreže, kot so jezikovni modeli, na primer generirajo besedilo z napovedovanjem verjetnih nadaljevanj zaporedja, medtem ko modeli, temelječi na difuziji (*diffusion models*), generirajo slike, zvok ali video tako, da se naučijo obračati postopen proces dodajanja šuma. A kljub temu so vsi ti sistemi še vedno nevronske mreže, učene po istih osnovnih principih, ki smo jih obravnavali v tem poglavju.