Duboke neuronske mreže

Jelić, Nikola

Duboke neuronske mreže

Jelić, Nikola

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Nikola Jelić

DUBOKE NEURONSKE MREŽE

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Saša Singer

Zagreb, srpanj 2020.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Prvo, želim zahvaliti mentoru izv. prof. dr. sc. Saši Singeru na ukazanom povjerenju. Također, svim dosadašnjim mentorima i profesorima na stečenom znanju. Posebno se zahvaljujem prijateljima i mojoj obitelji koji su velika podrška. Za kraj, rekurzivno ću spomenuti i one koji su pomogli njima da pomognu meni. Hvala vam!

Sadržaj

Uvod

Učenje je dio ljudske svakodnevice. To je ciklički proces koji se sastoji se od promišljanja, djelovanja te zaprimanja rezultata. U ovom diplomskom radu ćemo obraditi temu naziva duboke neuronske mreže. To su neuronske mreže koje sadrže tri ili više slojeva. Mrežu je potrebno naučiti koristeći podatke kako bi imala kvalitetnu predikciju.

Neuronske mreže su svoju popularnost stekle izradom snažnijih računala, generiranjem velikih količina podataka nakon dolaska interneta te stvaranjem zajednica okupljenih oko uspješnih softverskih alata. Danas, kada se govori o neuronskim mrežama, uglavnom se misli na duboke neuronske mreže.

Rad je podijeljen u pet poglavlja. U prvom poglavlju opisat ćemo osnovnu jedinicu mreže, raspraviti o nelinearnim problemima te pojasniti pojmove sloja neurona i dubokog učenja.

Kako bismo izradili kvalitetan model, neophodna su nam četiri segmenta: podaci, mjere uspješnosti i funkcije troška, model i algoritmi učenja. Njih opisujemo u drugom poglavlju.

U idućem, pojašnjavamo i uspoređujemo različite optimizacijske algoritme neophodne za učenje. Neki od njih su momentum, Adagrad, RMSprop i Adam.

Četvrto poglavlje opisuje različite arhitekture i njihove primjene. Kao dvije grane istaknule su se konvolucijske neuronske mreže te mreže s povratnim vezama.

U posljednjem poglavlju opisujemo rješenje problema klasifikacije slika. Govorimo o sakupljenim podacima i cilju modeliranja te samoj izradi i učenju modela. Za kraj, zaključujemo o uspješnosti izrade.

Poglavlje 1

Umjetna neuronska mreža

Jedna od najvećih enigmi u medicini je mozak. Mogli bismo reći da je mozak ljudsko računalo, jer izračunava ono što vidimo, čujemo i općenito osjećamo. Također, generiramo misli te odgovaramo na podražaje, npr. govorom ili pokretom. Stoga nam njegovo razumijevanje može pomoći da izradimo bolja računala i strojeve koji bi, na nama sličan način, razumjeli okolinu te nam na taj način pomogli. Vodeći se time znanstvenici su pokušali imitirati neurone kako bi riješili složene probleme.

1.1 TLU-perceptron

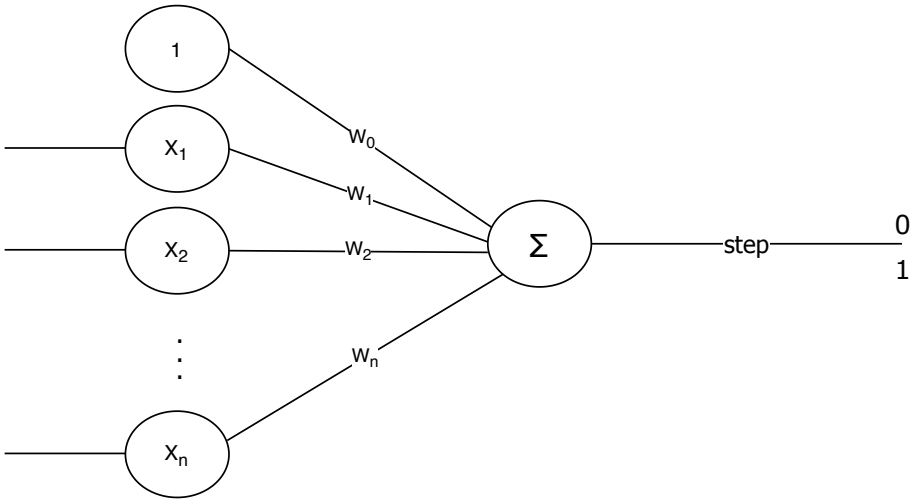
Prvi pokušaj modeliranja umjetnog neurona poznatijeg kao TLU-*perceptron* (engl. *Threshold Logic Unit*) napravili su 1943. godine Warren McCulloch i Walter Pitts [2]. Slika 1.1 nam vizualizira izgled tog neurona u obliku grafa, a sljedeća funkcija ga opisuje:

$$f(\vec{x}) = \begin{cases} 1, & \vec{w} \cdot \vec{x} \geq 0 \\ 0, & \vec{w} \cdot \vec{x} < 0 \end{cases}. \tag{1.1}$$

Vektor $\vec{w}$ sadrži težine. Svaka od težina w_i pridodaje značaj komponenti x_i, pri čemu je dogovorno $x_0 = 1$ fiksan. Primijetimo da ulazni vektor $\vec{x}$ proširimo s početnom vrijednosti 1 te ga pomnožimo s težinama $\vec{w}$. Stoga, funkciju f možemo zapisati i kao:

$$f(\vec{x}) = \begin{cases} 1, & \vec{w} \cdot \vec{x} + b \geq 0 \\ 0, & \vec{w} \cdot \vec{x} + b < 0 \end{cases}. \tag{1.2}$$

Sada vektor $\vec{x}$ ne modificiramo te je $\vec{w}$ istih dimenzija kao i ulazni vektor. Ovim zapisom je smanjen broj operacija za jedno množenje. Vrijednost b se naziva pristranost (engl. *bias*), odnosno prag (engl. *threshold*) po kojem je TLU-*perceptron* nazvan.



Slika 1.1: Prikaz jednog TLU-*perceptrona*

Ulazni podatak je veličine n*, a kao izlaz dobivamo broj koji može biti* 0 *ili* 1*.*

Funkciju f (1.2) možemo zapisati kao kompoziciju $f = step(g)$, pri čemu su $g\colon \mathbb{R}^n \to \mathbb{R}$ i $step\colon \mathbb{R} \to \{0, 1\}$ zadane s:

$$g(\vec{x}\,) = \vec{w} \cdot \vec{x} + b = \sum_{i=1}^{n} w_i \cdot x_i + b, \tag{1.3}$$

$$step(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}. \tag{1.4}$$

Funkcija (1.3) je afina funkcija koja se sastoji od skalarnog produkta s težinama i praga. Time se računa važnost, odnosno značaj pojedinog elementa vektora $\vec{x}$, a pragom se određuje kada će ukupna suma biti veća od 0. U ovom smislu dobivamo značajke naših podataka.

S druge strane, funkciju g možemo gledati kao hiperravninu koja odjeljuje prostor na dva poluprostora, a time poluprostori određuju dvije klase podataka.

Funkcija (1.4) odlučuje hoće li se neuron aktivirati. Stoga, općeniti naziv takvih funkcija je **aktivacijska funkcija**, što ćemo označavati funkcijom a. Sada funkciju (1.2) možemo poopćiti s (gdje je $a = step$):

$$f(\vec{x}\,) = a(\vec{w} \cdot \vec{x} + b). \tag{1.5}$$

Primijetimo da *step* funkcija kao izlaz može dati dvije vrijednosti 0 ili 1, što analogno vrijedi i za funkciju f. Pridružujući svakom ulaznom podatku jednu od

navedenih vrijednosti na taj način klasificira podatke. Dakle, za funkciju f možemo još reći da je binarni klasifikator.

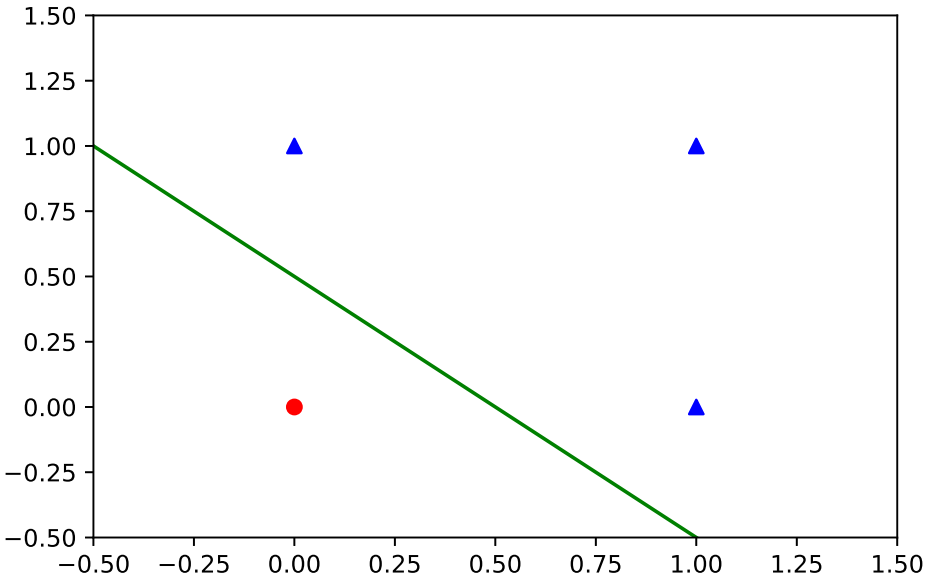
1.2 Nelinearnost

Pogledajmo iduća dva primjera koja će nam predočiti ograničenja linearnih funkcija.

Primjer 1.2.1. Neka je $n = 2$, tj. imamo skup 2-dimenzionalnih podataka $\{(0,0), (0,1), (1,0), (1,1)\} \subseteq \mathbb{R}^2$, kojeg želimo klasificirati u dva skupa:

- $A = \{(0,0)\} \subseteq \mathbb{R}^2$,
- $B = \{(0,1), (1,0), (1,1)\} \subseteq \mathbb{R}^2$.

Ako izradimo TLU-*perceptron* s težinama $w_1 = 1, w_2 = 1, b = -0.5$, dobit ćemo klasifikaciju koja odgovara prikazu na slici 1.2. Možemo primijetiti da smo podatke odijelili pravcem. ◁

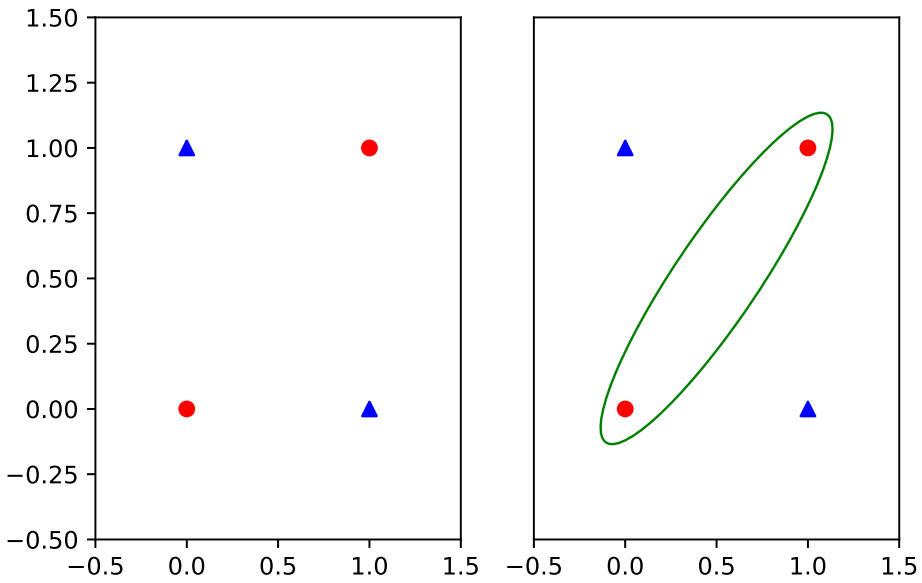


Slika 1.2: Klasifikacija na skupove A i B

Primjer 1.2.2. Jedno od ograničenja TLU-*perceptrona* je njegova linearnost, odnosno što podatke odvajamo hiperravninom. Ako pogledamo lijevi graf na slici 1.3 vidjet ćemo da iste ulazne podatke nije moguće klasificirati pravcem na disjunktne skupove:

- $A = \{(0,0),(1,1)\} \subseteq \mathbb{R}^2$,
- $B = \{(0,1),(1,0)\} \subseteq \mathbb{R}^2$.

Kada bi za skup A funkcija f poprimala vrijednost 0, a za skup B vrijednost 1, tada bi f simulirala logički operator *xor*. Stoga je potrebno priskočiti nelinearnim funkcijama. Mogli bismo konstruirati polinom visokog stupnja ili elipsu, kao na slici 1.3. ◁



Slika 1.3: Korištenje elipse za nelinearnu klasifikaciju

Ne postoji pravac koji bi odijelio kružiće od trokutića. Vidimo da je jednostavno odijeliti primjerke nelinearnom funkcijom, npr. elipsom.

Kao što smo mogli vidjeti iz primjera 1.2.1 i primjera 1.2.2, postoje problemi koji nisu linearno rješivi. Zato su nam potrebne nelinearne funkcije. Kako je kompozicija dvije linearne funkcije ponovno linearna funkcija, barem jedna od komponiranih funkcija treba biti nelinearna. Komponiranjem više linearnih i nelinearnih funkcija dobivamo složenu funkciju koja je najčešće ekspresivnija, odnosno ima veći kapacitet.

1.3 Sloj neurona

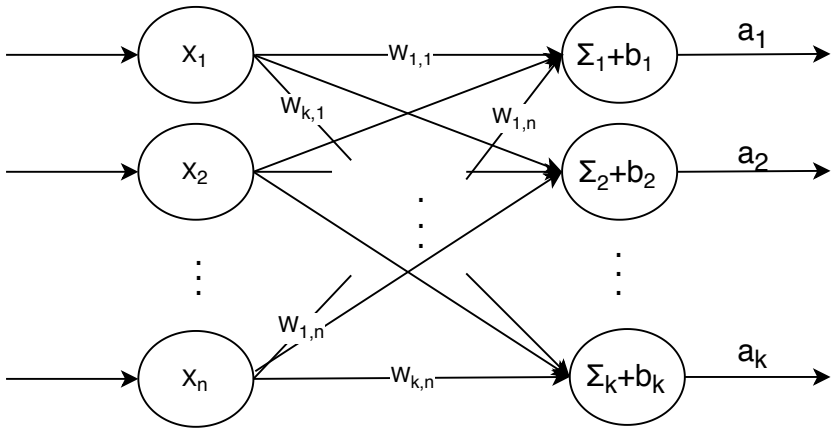
Slika 1.1 prikazuje samo jedan neuron s pripadajućim ulazima i izlazom, koji podatke dijeli na dvije klase (0/1). Što ako želimo klasificirati podatke na tri ili više klasa? Komponirati više neurona ili računati više značajki?

Na početku smo rekli da je ideja bazirana na mozgu, a mozak se sastoji od povezanih (komponiranih) neurona. Primijetimo, ulaz TLU-*perceptrona* predstavlja vektor podataka $\vec{x}$, koji se skalarno množi s vektorom težina $\vec{w}$, dodaje se pristranost b te nakon primjene aktivacijske funkcije dobivamo skalar. To bismo mogli zapisati kao $a(\vec{w}^\top \vec{x} + b)$.

Sada, pogledajmo sliku 1.4. Ako umjesto vektora težina $\vec{w}$, koristimo matricu težina $W \in \mathbb{R}^{k \times n}$ i umjesto pristranosti b, koristimo vektor pristranosti $\vec{b} \in \mathbb{R}^k$, dobivamo funkciju:

$$s(\vec{x}) = a(W\vec{x} + \vec{b}), \tag{1.6}$$

gdje je $W\vec{x}$ množenje matrice i vektora, pri čemu je $\vec{x} \in \mathbb{R}^n$. Funkcija a je aktivacijska funkcija. Funkciju s zovemo **sloj neurona**, jer se sastoji od više vektora težina s kojim djelujemo na ulazni podatak.



Slika 1.4: Prikaz sloja neurona

Prilagodimo notaciju slojevima. Neka je $\vec{x}^{(s)} \in \mathbb{R}^n$ izlazni vektor sloja s. Ako je $s = 0$ tada je $\vec{x}^{(0)}$ ulazni podatak. Sada (1.6) možemo zapisati kao:

$$\vec{x}^{(s)} = a^{(s)}(W^{(s)}\vec{x}^{(s-1)} + \vec{b}^{(s)}). \tag{1.7}$$

Primjećujemo da svaki sloj s ima svoju matricu težina $W^{(s)}$, svoj vektor pristranosti $\vec{b}^{(s)}$ te svoju funkciju aktivacije $a^{(s)}$. Aktivacijska funkcija primjenjuje se na

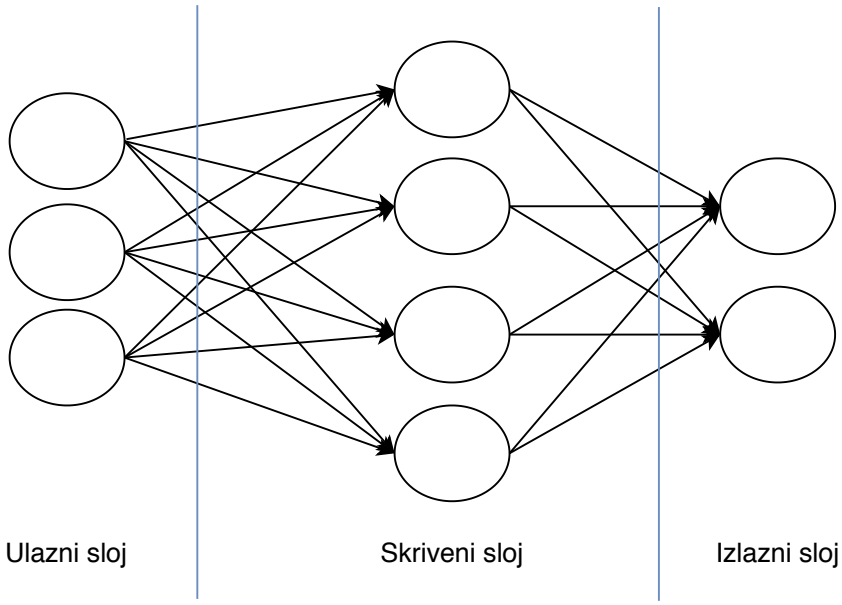
svaki element j vektora zasebno:

$$\vec{x}_j^{(s)} = a^{(s)}(W_{j,:}^{(s)}\vec{x}^{(s-1)} + \vec{b}_j^{(s)}), \quad \forall j \in \{1,2,\ldots,k\}. \tag{1.8}$$

Primijetimo da je sloj zapravo funkcija. U konkretnom slučaju, kompozicija aktivacijske funkcije i afine funkcije. Također, mogli bismo reći da su podaci prvo obrađeni afinim slojem, a zatim obrađeni aktivacijskim slojem.

Na primjeru TLU-*perceptrona* vidimo da se funkcija (1.2) sastoji od sloja (1.3) i sloja (1.4). Također, možemo reći da se funkcija (1.2) sastoji od samo jednog sloja, sebe same. Tu dolazi do problema u terminologiji, zato što se često neki sloj može rastaviti na jednostavnije funkcije, odnosno spajanjem dva ili više slojeva možemo konstruirati novi sloj.

S druge strane, zbog toga što se slojevi programiraju na računalu i podaci prolaze kroz slojeve, često postoji gruba podjela slojeva na **ulazni sloj**, **skriveni sloj** i **izlazni sloj**. Na slici 1.5 možemo primijetiti da ulazni sloj i izlazni sloj nisu funkcije već memorijski blokovi. Kod ulaznog sloja znamo u kojem obliku podaci ulaze, odnosno znamo domenu, a izlazni sloj definira kodomenu. Skriveni sloj, kao što i sam naziv govori, je skriven između ulaznog i izlaznog. Često je moguće da postoji više skrivenih slojeva. Tada, ono što je jednom skrivenom sloju izlaz, to je njegovom sljedbeniku ulaz. Pojam sloja je apstraktan i često ovisi o kontekstu.



Slika 1.5: Prikaz umjetne neuronske mreže i tri osnovna sloja

1.4 Duboko učenje

Umjetna neuronska mreža, ili samo neuronska mreža je funkcija sastavljena od umjetnih neurona. Naziv je prvenstveno inspiriran medicinom, jer se mozak sastoji od mreže neurona. Jedan od razloga popularizacije tog naziva je i marketing. S druge strane, ako pogledamo sliku 1.5, možemo primijetiti graf koji je gusto povezan bridovima, što nalikuje na mrežu. Prikaz neuronske mreže kao grafa će se pokazati bitnim pri opisu optimizacijskih algoritama. Pojam koji nam je od važnosti je i **duboka neuronska mreža**. To je umjetna neuronska mreža koja sadrži više skrivenih slojeva, najčešće tri ili više.

Dvije neuronske mreže možemo uspoređivati po broju skrivenih slojeva. Za onu koja ima više skrivenih slojeva kažemo da je **dublja**. Pri brojanju slojeva postoji konsenzus da se ulazni sloj ne broji. Na primjer, ako neka neuronska mreža ima 10 slojeva, tada ona ima implicitan ulazni sloj (nulti sloj), devet skrivenih i deseti izlazni sloj.

Za kraj ovog dijela napomenut ćemo da znanstveno područje koje proučava duboke neuronske mreže se zove **duboko učenje**, koje je dio strojnog učenja, a strojno učenje je grana šireg područja znanog kao umjetna inteligencija. Strojno učenje proučava algoritme koji su sposobni pronaći pravilnosti u podacima, a duboko učenje kao grana strojnog, iz podataka pronalazi pravilnosti učeći značajke.

Duboko učenje je primjenjivo na različite probleme. S obzirom na vrstu problema koju rješavamo te pripadnih podataka, postoji podjela učenja na **nadzirano**, **nenadzirano** i **učenje s podrškom**.

Kod **nadziranog** učenja podaci su pripremljeni kao uređeni parovi, odnosno primjeri (X_i, y_i), gdje X_i zovemo **ulazni vektor varijabli** (ako je podatak tenzor možemo ga vektorizirati) ili nezavisna varijabla, a y_i zovemo **ciljna varijabla**, odnosno oznaka ili izlazna varijabla i-tog primjera. Primjeri su elementi **prostora primjera**. Ako su ciljne varijable kategoričke tada govorimo o **klasifikaciji**, a ako su numeričke vrijednosti, npr. iz skupa realnih brojeva, onda kažemo da je to **regresijski** problem. Primjer regresije je predviđanje cijene nekretnine na temelju njezine veličine i udaljenosti od centra grada. U nadziranom učenju cilj je da model "što bolje" nauči za određeni ulazni vektor varijabli vratiti traženu oznaku. Pri čemu, kada kažemo "što bolje", zapravo mislimo s obzirom na mjeru.

Suprotno od nadziranog je **nenadzirano** učenje, kod kojeg nemamo zadanu traženu oznaku y_i. Kod takvog učenja se najčešće radi o traženju pravilnosti među primjerima i njihovom grupiranju. Vrsta nenadziranog učenja je **klasteriranje**. Za razliku od svrstavanja podataka u unaprijed poznate klase, tj. klasifikacije, klasteriranje je grupiranje primjera u podgrupe (klastere) na način da su primjeri unutar iste podgrupe međusobno sličniji nego s primjerima iz drugih podgrupa. Najčešće se

kao mjera sličnosti uzima udaljenost pa govorimo da su objekti unutar iste podgrupe međusobno bliži.

Treća vrsta je **podržano** učenje, gdje algoritam s obzirom u kojem se stanju nalazi, djeluje akcijom te dobiva nagradu ili kaznu, odnosno mjeri se koliko je dobro izvedena akcija. S tim podacima algoritam trenira model da bolje predviđa koje je akcije potrebno učiniti u određenom stanju. Kod tog učenja najčešće nema fiksnih podataka, već se podaci dobivaju tijekom izvođenja akcija. Primjeri su igranje šaha ili igre GO, gdje algoritam igra sam protiv sebe te tako generira podatke za učenje.

Poglavlje 2

Četiri segmenta za učenje mreže

U prethodnom poglavlju uvedeni su osnovni pojmovi te je neuronska mreža prikazana kao funkcija. Cilj ovog poglavlja je upoznati se s pojmovima i idejama koje su nam potrebne za konstrukciju neuronskih mreža za specifične probleme te s terminom učenja. Četiri segmenata čine: **podaci**, **mjere uspješnosti i funkcija troška**, **arhitektura modela** te **algoritam učenja**. Svi segmenti su međusobno povezani. Problemu najčešće pristupamo navedenim poretkom. Trebamo znati koji problem rješavamo i kojim podacima raspolažemo. Zatim moramo odrediti za koji model ćemo moći reći da je uspješan, a koji nije. Slijedi konstrukcija te konačno učenje modela.

2.1 Podaci

Prvotno, potrebni su nam podaci. Podatke treba sakupiti i pretprocesirati, što može biti vrlo zahtjevno. Na primjer, želimo napraviti klasifikator madeža na maligne i benigne, koristeći kao ulazni podatak fotografiju. Tada je potrebno prikupiti fotografije madeža. Jedan od problema pri prikupljanju podataka može biti legalnost, jer nam treba pristanak pacijenata. Mogli bismo potražiti baze podataka na internetu za čije slike su pacijenti odobrili korištenje. Ako postoji više različitih baza onda bismo mogli naići na fotografije različitih veličina, fotografirane pod različitim kutom, na duplikate, ... Dakle, podaci mogu dolaziti iz različitih izvora i u različitim formatima te je potrebna **standardizacija**.

Jedan podatak iz skupa podataka zovemo **primjer**. Kako je umjetna neuronska mreža funkcija, svaki primjer je uređen par ulaznog i izlaznog dijela podatka. Moguće su različite kombinacije ulaznih i izlaznih podataka. Neki od mogućih ulaza su slika, tekst, zvuk, podaci o korisniku, itd. Izlazni dio primjera zovemo **oznaka** ili zavisna varijabla. Primjeri su: objekti (na slikama), tekst (prijevod), odluka korisnika

(da/ne), cijena, zvuk, ... Kako računalo radi s brojevima, mogući oblici podataka su skalar, vektor, matrica ili tenzor, ovisno radi li se o regresiji ili klasifikaciji. Kada govorimo o kategorijama kao podacima, tada ih kodiramo kao k–dimenzionalni vektor $\vec{v}$, gdje je k broj kategorija. Pri tome, ako se radi o i–toj kategoriji, tada vektor $\vec{v}$ sadrži nule, osim na i–toj poziciji koja sadrži vrijednost 1. Engleski naziv takvog vektora je *one-hot vector*.

Ulazni se dio primjera sastoji od **značajki**. Tako bi jedna fotografija madeža iz skupa fotografija bila ulazni dio primjera, a piksel značajka. Ako piksel sadrži tri vrijednosti za crvenu, zelenu i plavu boju, tada jedan piksel sadrži tri značajke. Za konkretan primjer, oznaka bi bila maligan ili benigan, što bismo mogli označiti s 0 ili 1.

Podatke možemo podijeliti na **strukturirane** i **nestrukturirane**. Strukturirani podaci su pretraživi, stoga najčešće dolaze u tablicama. Nestrukturirani mogu biti: audio, tekst, slika, ...

Kako je svaki model važno testirati, učestala praksa je podijeliti podatke na podatke učenja (engl. *train*) i testne podatke. Pri tome treba voditi računa da su primjeri nezavisni i jednoliko distribuirani. Kao što sam naziv kaže, podaci za učenje se koriste tijekom faze učenja i izrade modela, a kada mislimo da je model gotov tada ga testiramo na testnim podacima. Ako je model zadovoljio kriterije može ga se koristiti za što je namijenjen, a ako nije zadovoljio kriterije tada ga najčešće odbacujemo. Dakle, podaci za testiranje se ne smiju koristiti u **fazi učenja**, već isključivo u **fazi testiranja**. Za manje skupove od nekoliko tisuća podataka se uzima uzorak od 20%–30% za testne podatke, a ostatak za učenje. Bitno je da, prilikom odjeljivanja podataka, testni podaci budu izabrani na slučajan način kako bismo dobili reprezentativan uzorak.

Kako bismo mjerili uspješnost samog učenja, potrebno je dodatno podijeliti podatke za učenje na dva disjunktna skupa: podatke za učenje i podatke za validaciju. Opće prihvaćen omjer podijele je 80 : 20. Ovakva metoda se zove *Train and test* metoda.

Drugi način bi mogla biti unakrsna validacija, gdje se podaci podijele na k međusobno disjunktnih skupova te se učenje provodi k puta. Svaki puta se za validaciju ostavlja drugi skup, dok se na preostalih $k-1$ skupova provodi učenje. Na kraju se izračuna srednja vrijednost svih validacija.

Za duboko učenje nam je potrebna poveća količina podataka. Potrebna količina ovisi o složenosti problema koji se rješava te varira od nekoliko tisuća primjera do nekoliko milijuna primjera. Većina današnjih implementacija koristi nekoliko stotina tisuća podataka pa na više. Zato se češće uzima 95% podataka za učenje, a za validaciju i test po 2,5%. Ponekad je prihvatljivo, kada su u pitanju ekstremne količine od više desetaka milijuna, odrediti 99% za učenje, a 0,5% za validaciju i za testiranje.

Problem pri učenju stvara tzv. **prokletstvo dimenzionalnosti**. Naime, što je veća dimenzija primjera to nam je potrebno eksponencijalno više podataka ako želimo očuvati gustoću primjera u prostoru. Drugo, s dimenzijom raste i složenost modela koji bi trebao riješiti problem. Stoga se koriste metode selekcije podskupa varijabli ili se konstruiraju nove varijable iz postojećih. Time se reducira dimenzionalnost i olakšava učenje.

2.2 Mjere uspješnosti i funkcija troška

Nakon što smo pripremili podatke i prije nego krenemo s izradom modela, želimo odrediti mjeru uspješnosti kojom ćemo mjeriti kvalitetu modela. Kvalitetu modela mjerimo na skupu podataka (oznaka D), a to može biti testni ili validacijski skup. S y_i označavamo traženu zavisnu varijablu i-tog podatka, a s $\hat{y}_i$ predikciju i-tog primjera. Osim uspješnosti možemo mjeriti i pogrešku modela. S tom mjerom također možemo uspoređivati različite modele. Pri tome tražimo model s minimalnom pogreškom.

Mjera uspješnosti (odnosno pogreške) se definira ovisno o cilju i vrsti problema. Ako se radi o regresijskom problemu tada se najčešće koriste mjere pogreške. Jedna je srednja apsolutna greška – **MAE** (engl. *Mean Absolute Error*):

$$MAE_D = \frac{1}{|D|} \sum_{y_i \in D} |y_i - \hat{y}_i|. \tag{2.1}$$

Druga, učestalija mjera greške je srednje kvadratna greška – **MSE** (engl. *Mean Squared Error*):

$$MSE_D = \frac{1}{|D|} \sum_{y_i \in D} \|y_i - \hat{y}_i\|_2^2. \tag{2.2}$$

S druge strane, ako se radi o klasifikaciji, često ćemo koristiti **točnost** (engl. *accuracy*). Točnost je omjer dobro klasificiranih primjera i ukupnog broja primjera, definirana s (2.3). Ako imamo klase koje su jednakih važnosti, na primjer pri klasifikaciji fotografija životinja na mačke i pse, kao mjeru uspješnosti možemo koristiti točnost, jer želimo da mreža što više slika klasificira tamo gdje pripadaju.

U primjeru klasifikacije madeža na benigne i maligne, ne želimo točnost kao mjeru uspješnosti. Pretpostavljamo da će model, zbog toga što je većina madeža benigna, dobivati većinom benigne madeže kao ulaz pri učenju i pri klasifikaciji.

Predočimo to brojevima. Neka je omjer benignih i malignih madeža jednak $995 : 5$. Ako model sve madeže svrstava kao benigne, tada će točnost iznositi 99,5%. Taj model nije dobar, iako ima visoku točnost, jer ne prepoznaje bolesti, što bi trebala biti primarna svrha modela.

Kod binarne klasifikacije postoji podjela testnog skupa podataka prema predviđanju modela na:

- **stvarno pozitivne**, oznaka TP (engl. *true positives*) — skup primjera koji su predviđeni kao pozitivni i to jesu,
- **lažno pozitivne**, oznaka FP (engl. *false positives*) — skup primjera koji su predviđeni kao pozitivni, a to nisu,
- **stvarno negativne**, oznaka TN (engl. *true negatives*) — skup primjera koji su predviđeni kao negativni i uistinu jesu,
- **lažno negativne**, oznaka FN (engl. *false negatives*) — skup primjera koji su predviđeni kao negativni, a zapravo nisu.

Sada, pomoću navedenih oznaka možemo definirati mjere **točnost**, **preciznost** i **osjetljivost** kao:

$$točnost = \frac{|TP| + |TN|}{|TP| + |FP| + |TN| + |FN|}, \tag{2.3}$$

$$preciznost = \frac{|TP|}{|TP| + |FP|}, \tag{2.4}$$

$$osjetljivost = \frac{|TP|}{|TP| + |FN|}. \tag{2.5}$$

Stoga postoje mjere koje kvalitetno rješavaju prethodno navedeni problem. Mjera koja povezuje osjetljivost i preciznost je F_1-mjera definirana s:

$$F_1 = \frac{2}{osjetljivost^{-1} + preciznost^{-1}}. \tag{2.6}$$

Navedena mjera je korisna za problem klasifikacije madeža na benigne i maligne, jer balansira preciznost i osjetljivost, koji zapravo mjere udjele *lažno pozitivnih* i *lažno negativnih* primjera. Ako je preciznost visoka, tada je osjetljivost najčešće niska i obratno. Poželjno je da vrijednost F_1-mjere bude što bliža vrijednosti 1, jer će onda i *osjetljivost* i *preciznost* također biti bliže vrijednosti 1.

Kod učenja podrškom i nenadziranog učenja najčešće je potrebno koristiti nestandardne mjere koje sami konstruiramo. Za klasteriranje, što je vrsta nenadziranog učenja, često se koristi euklidska udaljenost.

Izbor mjere uspješnosti se može činiti trivijalnim, ali najčešće nije jednostavno izabrati mjeru koja će se ponašati očekivano. Ako se modelira ranije proučavan problem, tada možemo pronaći rezultate drugih znanstvenika u znanstvenim radovima

i vidjeti mjere koje su oni koristili. Time bismo mogli usporediti naše rezultate s njihovim. S druge strane, možemo si zadati neku razinu uspješnosti s kojom bismo bili zadovoljni, odnosno koja bi značila da je model upotrebljiv.

Funkcija troška (engl. *cost function*) je također mjera, koja je ujedno i funkcija cilja. Njezina svrha je optimizacija modela, odnosno učenje modela. Neke literature funkciju troška odvajaju od **funkcije gubitka** (engl. *loss function*) koju još nazivaju **funkcijom greške** (engl. *error function*). Tada funkciju troška definiraju kao prosjek vrijednosti funkcije gubitka svih primjera. Odnosno, neka je D skup primjera uređenih parova (X_i, y_i), f model neuronske mreže sa svim pripadnim parametrima $\vec{\theta}$ te L funkcija gubitka. Tada definiramo funkciju troška J kao:

$$J(D;\, \vec{\theta}) = \frac{1}{|D|} \sum_{(X_i, y_i) \in D} L(f(X_i;\, \vec{\theta}), y_i). \tag{2.7}$$

Sada vidimo da zadavanjem funkcije gubitka L definiramo i funkciju troška J, koja ovisi o parametrima $\vec{\theta}$. Ideja je da funkcija gubitka vrati visoke vrijednosti za pogrešnu predikciju, a nulu za točnu. Tada će i funkcija troška poprimati veće vrijednosti za pogrešne predikcije. Zato je **učenje** traženje parametara $\vec{\theta}$ za koje funkcija troška daje **minimalnu** vrijednost nad trening podacima.

Dva primjera funkcije troška koja smo obradili su **MAE** i **MSE**. Sada ćemo navesti neke funkcije gubitka, iz čega će slijediti pripadne funkcije troška. Prva je tzv. *Huber loss* definirana s:

$$L_\delta(y_i, \hat{y}_i) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & \text{za } |y_i - \hat{y}_i| \leq \delta \\ \delta |y_i - \hat{y}_i| - \frac{1}{2}\delta^2, & \text{inače} \end{cases}. \tag{2.8}$$

Ova funkcija je kvadratna za male reziduale, a linearna za velike. Time je robusna na tzv. *outliere*.

Prethodno navedene funkcije gubitka su namijenjene regresijskim problemima. Funkcija od značaja za optimizaciju kategoričkih varijabli je tzv. *(categorical) cross–entropy loss function*. Postoji i binarna verzija, ali ćemo definirati općenitiju verziju. Neka imamo k–dimenzionalne *one-hot* vektore, tj. zavisnu varijablu i–tog primjera $\vec{y}$ te pripadnu predikciju $\hat{\vec{y}}$. Tada spomenutu funkciju gubitka definiramo kao:

$$L(\vec{y}, \hat{\vec{y}}) = -\sum_{j}^{k} \vec{y}_j \log(\hat{\vec{y}_j}). \tag{2.9}$$

Kako su za $\vec{y}_j$ moguće vrijednosti pojedinih elemenata 0 i 1 te za $\hat{\vec{y}_j}$ vrijednosti iz $[0, 1]$, a u praksi zapravo iz $\langle 0, 1 \rangle$, tada možemo primijetiti kako ova funkcija za

točnu predikciju poprima vrijednost blizu 0, a za pogrešnu predikciju vrijednost veću od 0. Znači da je funkcija dobro definirana u smislu da minimizacijom vrijednosti optimiziramo model.

2.3 Arhitektura modela

Kao što smo ranije rekli, model je funkcija kojom želimo procijeniti neke vrijednosti od interesa na temelju podataka. Pri tome, nezavisne varijable definiraju oblik ulaznog sloja, a zavisne definiraju izlazni sloj te ovise o vrsti podataka. Najjednostavnija neuronska mreža ne sadrži skriveni sloj, tj. sastoji se od samo jednog sloja. Primjer je TLU-*perceptron*.

U prethodnom smo poglavlju pomoću primjera 1.2.2 pokazali kako su nam potrebni nelinearni modeli za simuliranje funkcije *xor*. Kao jednu mogućnost spomenuli smo elipsu. Sada ćemo pokazati kako se taj problem može riješiti pomoću višeslojnog *perceptrona* (**MLP** – engl. *Multilayer perceptron*).

Primjer 2.3.1. Neka imamo skup kao u primjeru 1.2.1, tj. imamo skup 2–dimenzionalnih podataka $\{(0,0), (0,1), (1,0), (1,1)\} \subseteq \mathbb{R}^2$. Želimo ga klasificirati na dva disjunktna skupa kao u primjeru 1.2.2:

- $A = \{(0,0), (1,1)\} \subseteq \mathbb{R}^2$,
- $B = \{(0,1), (1,0)\} \subseteq \mathbb{R}^2$.

Dakle, želimo simulirati logički operator *xor*. Za skup A želimo da naš model poprima vrijednost 0, a za skup B vrijednost 1. Znamo da vrijedi

$$xor(x_1, x_2) = and(or(x_1, x_2), not(and(x_1, x_2))), \quad \forall x_1, x_2 \in \{0, 1\}.$$

Sada ćemo pokazati da je moguće zapisati funkcije *and*, *or* i $not \circ and$ pomoću TLU-*perceptrona*, tako da pronađemo odgovarajuće težine vektora $\vec{w}$ i pristranosti b:

$$or(x_1, x_2) = step(1 \cdot x_1 + 1 \cdot x_2 - 1), \tag{2.10}$$
$$and(x_1, x_2) = step(1 \cdot x_1 + 1 \cdot x_2 - 2), \tag{2.11}$$
$$not \circ and(x_1, x_2) = step(-1 \cdot x_1 - 1 \cdot x_2 + 1). \tag{2.12}$$

Nakon što smo izveli logičke operatore kao funkcije, možemo ih povezati u cjelinu kao višeslojnu neuronsku mrežu, tako što ćemo zadati matrice težina $W^{(1)}$ i $W^{(2)}$ te

vektore pristranosti $\vec{b}^{(1)}$ i $\vec{b}^{(2)}$, a jedna od mogućnosti je:

$$\vec{x}^{(1)} = step\left(\underbrace{\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}}_{W^{(1)}} \begin{bmatrix} \vec{x}_1^{(0)} \\ \vec{x}_2^{(0)} \end{bmatrix} + \underbrace{\begin{bmatrix} -1 \\ 1 \end{bmatrix}}_{\vec{b}^{(1)}}\right), \tag{2.13}$$

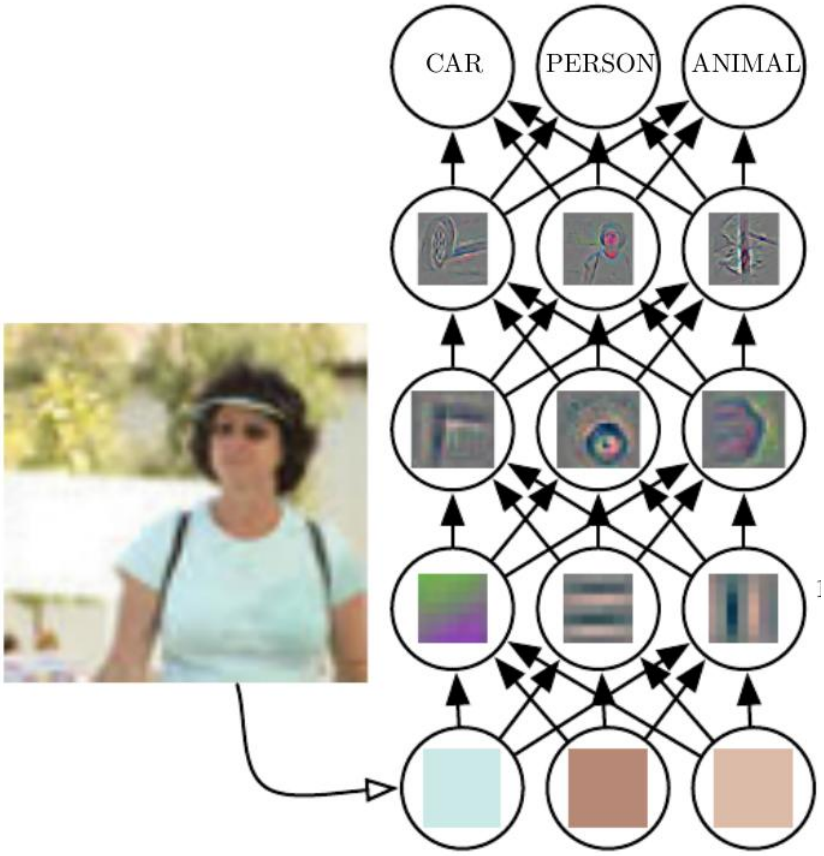
$$\vec{x}^{(2)} = step\left(\underbrace{\begin{bmatrix} 1 & 1 \end{bmatrix}}_{W^{(2)}} \begin{bmatrix} \vec{x}_1^{(1)} \\ \vec{x}_2^{(1)} \end{bmatrix} + \underbrace{[-2]}_{\vec{b}^{(2)}}\right). \tag{2.14}$$

Sada je $\vec{x}^{(2)} = xor(\vec{x}^{(0)})$. Time smo pokazali da je moguće pomoću dva sloja simulirati funkciju *xor*, odnosno da već dvoslojna neuronska mreža ne mora biti linearna. Ako promotrimo (2.13) i (2.14), vidjet ćemo da se obje funkcije sastoje od kompozicija *step* funkcije i afine funkcije, što znači da *step* funkcija omogućuje nelinearnost. ◁

Prethodni primjer 2.3.1 nam je predočio dvije stvari. Prvo, već sa samo dvije vrste funkcija, *step* i afinom, možemo kompozicijama dobiti nelienarnost koja nam je potrebna kako bismo rješavali složenije probleme. Time smo uočili ulogu aktivacijskih funkcija u prepoznavanju nelinearnosti.

Drugo, dubina neuronske mreže je bitan faktor. Iako TLU-*perceptron* također sadrži *step* i afinu funkciju, on ne može opisati nelinearan problem. S druge strane, važnost dubine dolazi do izražaja pri **učenju reprezentacija**, odnosno značajki. Pogledajmo sliku 2.1. Možemo vidjeti da ulazni sloj prihvaća piksele. Nakon toga se obrađuju te, prolaskom kroz skrivene slojeve, model prvo primjećuje rubove i kontrast, zatim zaobljenja na slici i teksture pa manje objekte, kao što su pjege, oči, nos, ... Na kraju se prepoznaju cjeline, kao osoba, auto, životinja, itd.

Za uspješnost modela, osim količine skrivenih slojeva, važan je i broj neurona u pojedinom sloju. Što je veći broj neurona, to će biti veći broj težina, a time i kompleksnost samog modela.



Slika 2.1: Prepoznavanje reprezentacija

Uočavamo značaj dubine, tako što se postepeno, prolaskom kroz slojeve, prepoznaju na početku sitniji detalji kao što su rubovi, a kasnije semantičke cjeline poput nosa, naočala, kose, itd. Slika je preuzeta iz [5, str. 6].

Aktivacijske funkcije

Pokazali smo da aktivacijske funkcije imaju važnu ulogu za nelinearno ponašanje modela. U ovom odjeljku ćemo opisati neke od najraširenijih funkcija. Predstavit ćemo: **linearnu**, **step funkciju**, **sigmoidalnu**, **tangens hiperbolni**, **ReLU** te **LeakyReLU**. Njihov prikaz se nalazi na slikama 2.2 i 2.3.

Linearnu funkciju spominjemo kao osnovnu funkciju i ona zapravo ne pridonosi nelinearnom učenju. Može se koristiti kako bi povećala ili smanjila vrijednosti prethodnog sloja. Na primjer, ako je u prethodnom sloju bila aktivacijska *step* funkcija, a želimo umjesto mogućih vrijednosti $\{0, 1\}$, vrijednosti $\{0, 30\}$, tada je potrebno *step*

funkciju pomnožiti s 30. Opći oblik linearne funkcije je:

$$a(\vec{x}) = \alpha\vec{x}, \tag{2.15}$$

gdje je α parametar funkcije.

Prva koja pridonosi nelinearnosti modela i koju smo više puta spominjali je *step* funkcija. Tu ćemo ponoviti njezin zapis, kako bismo je mogli usporediti s drugim funkcijama (slika 2.2):

$$a(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}. \tag{2.16}$$

No, ona ima mane. Nije glatka na cijeloj domeni te su njezine derivacije jednake nuli, što će pri učenju modela zadavati probleme.

Funkcija koja je zaobišla navedene probleme, s time da je kodomena zadržana u intervalu $\langle 0, 1\rangle$, je **sigmoidalna** funkcija. Najčešće se koristi u izlaznom sloju za binarnu klasifikaciju. Njezina formula glasi:

$$a(x) = \frac{1}{1 + e^{-x}}. \tag{2.17}$$

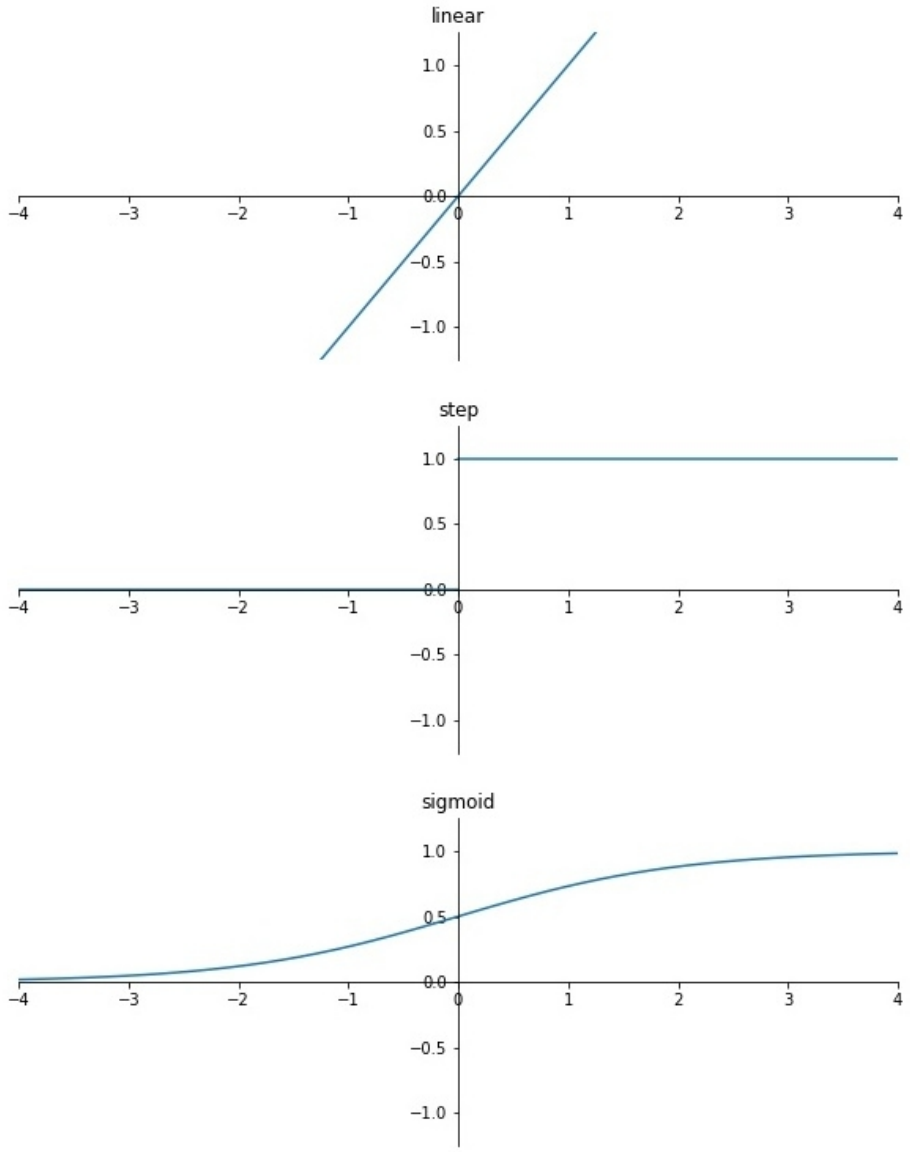
Analogno, za klasifikaciju tri ili više klasa koristimo funkciju **softmax**. Pretpostavimo da imamo k klasa. Tada funkciju definiramo kao $a\colon \mathbb{R}^k \to \mathbb{R}^k$:

$$a(\vec{x})_i = \frac{e^{\vec{x}_i}}{\sum_{j=1}^{k} e^{\vec{x}_j}}, \qquad \text{za } i \in \{1, 2, \ldots, k\}. \tag{2.18}$$

Za skrivene slojeve češće se koristi **tangens hiperbolni**, koji je gladak na cijeloj domeni. Njegova slika funkcije je malo drugačija od sigmoidalne funkcije te je jednaka $\langle -1, 1\rangle$. Graf funkcije možemo vidjeti na slici 2.3, a definiramo ju s:

$$a(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.19}$$

Problem koji se javlja kod *sigmoidalne* funkcije i *tangensa hiperbolnog* su derivacije povećih pozitivnih i negativnih brojeva. Naime, pogledom na grafove tih funkcija na slikama 2.2 i 2.3, vidimo da derivacije postaju jako male, približne nuli, već za vrijednosti -3 i 3. Odnosno, konvergiraju prema nuli kako težimo prema beskonačnostima, neovisno o predznaku. To je problem, jer ćemo vidjeti da se modeli uče iterativno gradijentima, pa za jako male gradijente, blizu nule, model se jako sporo uči.



Slika 2.2: Grafovi linearne, step i sigmoidalne funkcije

Linearna funkcija ne doprinosi nelinearnosti, a step funkcija doprinosi. Problem kod nje je što nije derivabilna u nuli, a inače je derivacija jednaka 0. *Sigmoidalna funkcija rješava taj problem, ali za većinu vrijednosti je derivacija blizu nule pa se model sporo uči.*

Posljednje dvije aktivacijske funkcije koje ćemo spomenuti su **ReLU** (engl. *rectified linear unit*) i **LeakyReLU**. Potrebna je mala modifikacija da bi se dobio *LeakyReLU*. Najčešće se koriste u skrivenim slojevima, a njihovu popularnost omogućilo je nekoliko svojstava koja olakšavaju učenje. Prvo, pri uniformno inicijaliziranim težinama, oko nule, otprilike pola neurona će biti aktivirano. Drugo, pri učenju s gradijentima, više gradijenata će ostati sačuvano, odnosno različito od nule. Stoga model uči mnogo brže ako ih koristimo u skrivenim slojevima, umjesto npr. sigmoidalne funkcije. Treće, jednostavno se računa, što možemo vidjeti iz formula (2.20) i (2.21). Definirajmo *ReLU* kao:

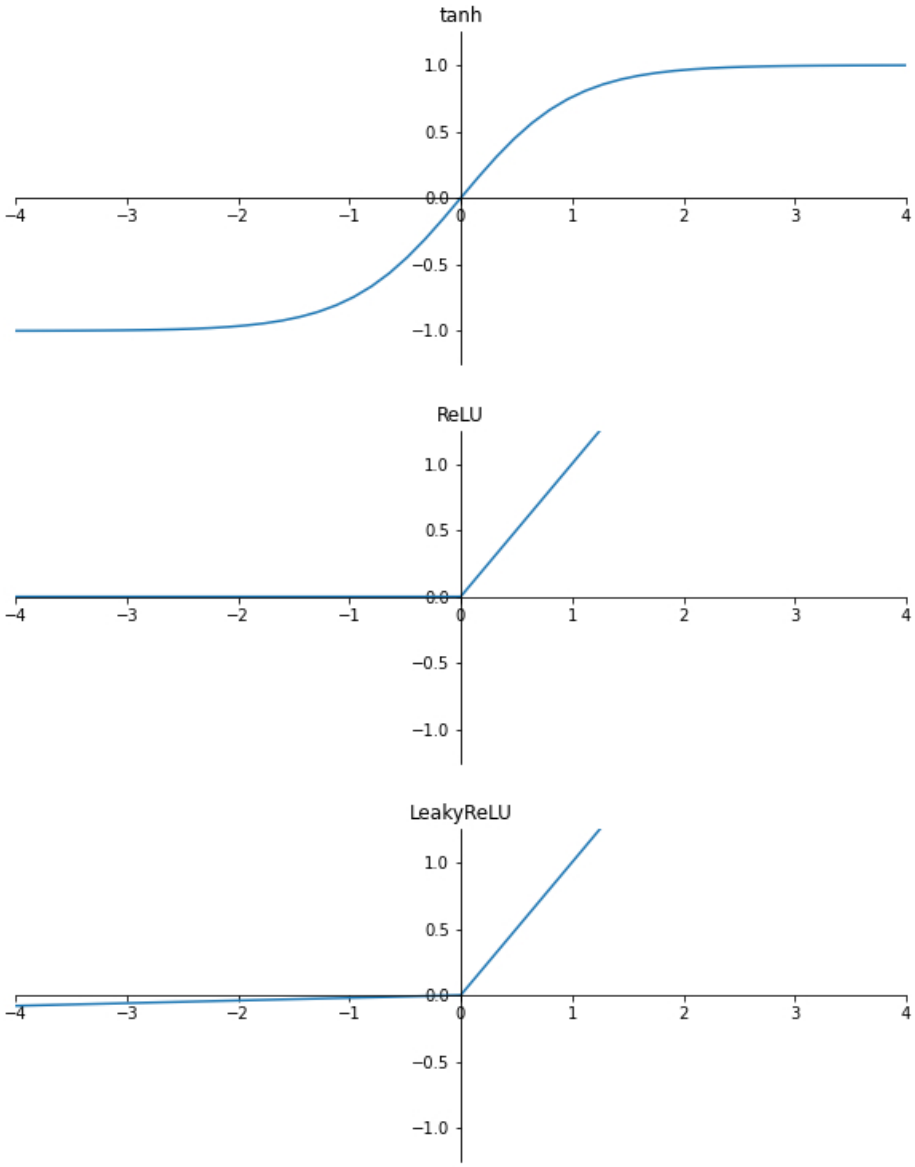
$$a(x) = \max\{0, x\}. \tag{2.20}$$

Vidimo da je funkcija prilično jednostavna što doprinosi popularnosti. Definirajmo i *LeakyReLU* s:

$$a(x) = \max\{\alpha x, x\}, \tag{2.21}$$

gdje je α jako mali broj, najčešće 0,01. Problem kod ovih aktivacijskih funkcija je derivacija u nuli. Taj problem se rješava tako da dogovorno stavimo 1, iako može i 0. Obje funkcije su podjednako dobre.

Kao što smo vidjeli, svaka od navedenih aktivacijskih funkcija ima svoje mane i prednosti. Neke su bolje za skrivene slojeve, a neke je najbolje koristiti isključivo u zadnjem sloju. Kako bilo, važan su faktor za učenje nelinearnosti u podacima.



Slika 2.3: Prikaz grafova tangensa hiperbolnog, ReLU te LeakyReLU funkcije

Tangens hiperbolni se, za razliku od sigmoidalne, pokazao kao bolja aktivacijska funkcija za skrivene slojeve, no zbog svoje jednostavnosti i povoljnih osobina, za gradijente su se ReLU i LeakyReLU pokazali još boljima.

Slojevi kao operacije

Prethodno smo spomenuli samo jednu operaciju koja transformira podatke, odnosno prethodne slojeve. To je bila afina funkcija. Drugi nazivi za taj sloj neurona, koji su uvriježeni u dubokom učenju su **potpuno povezani sloj** (engl. *fully connected layer*) te se još može naići na engleski naziv *dense layer*. Nazivi su opravdani, jer svaki neuron ovisi o, i povezuje se sa svim neuronima iz prethodnog sloja. Ako je izlaz prethodnog sloja n–dimenzionalan i imamo sloj s k neurona, tada će dimenzija matrice težina W biti $k \times n$ i dimenzija vektora pristranosti $\vec{b}$ biti k, što znači da je ukupan broj parametara $(n+1)k = n \cdot k + k$.

Iako je moguće slike vektorizirati te koristiti potpuno povezani sloj, iduća operacija se pokazala značajnom za njihovu obradu. Naziv operacije je **konvolucija** i njezino računanje možemo vidjeti na prikazima 2.4 i 2.5, koji su preuzeti iz [11]. Za osnovno korištenje nam je potrebna dvodimenzionalna slika visine n_H i širine n_W. Također, potrebna je matrica težina, dimenzija $f \times f$, koju nazivamo **filter**, odnosno jezgra (engl. *kernel*). Njime djelujemo iterativno na sliku tako da ga pomičemo po cijeloj slici, računajući skalarni produkt filtera i dijela slike, u tom trenutku prekrivenog filterom. Neke implementacije skalarnom produktu pridodaju i pristranost b. Tu vrijednost spremamo u **matricu rezultata** na poziciju koju ima gornji lijevi vrh promatranog dijela slike. Parametri koje možemo mijenjati su:

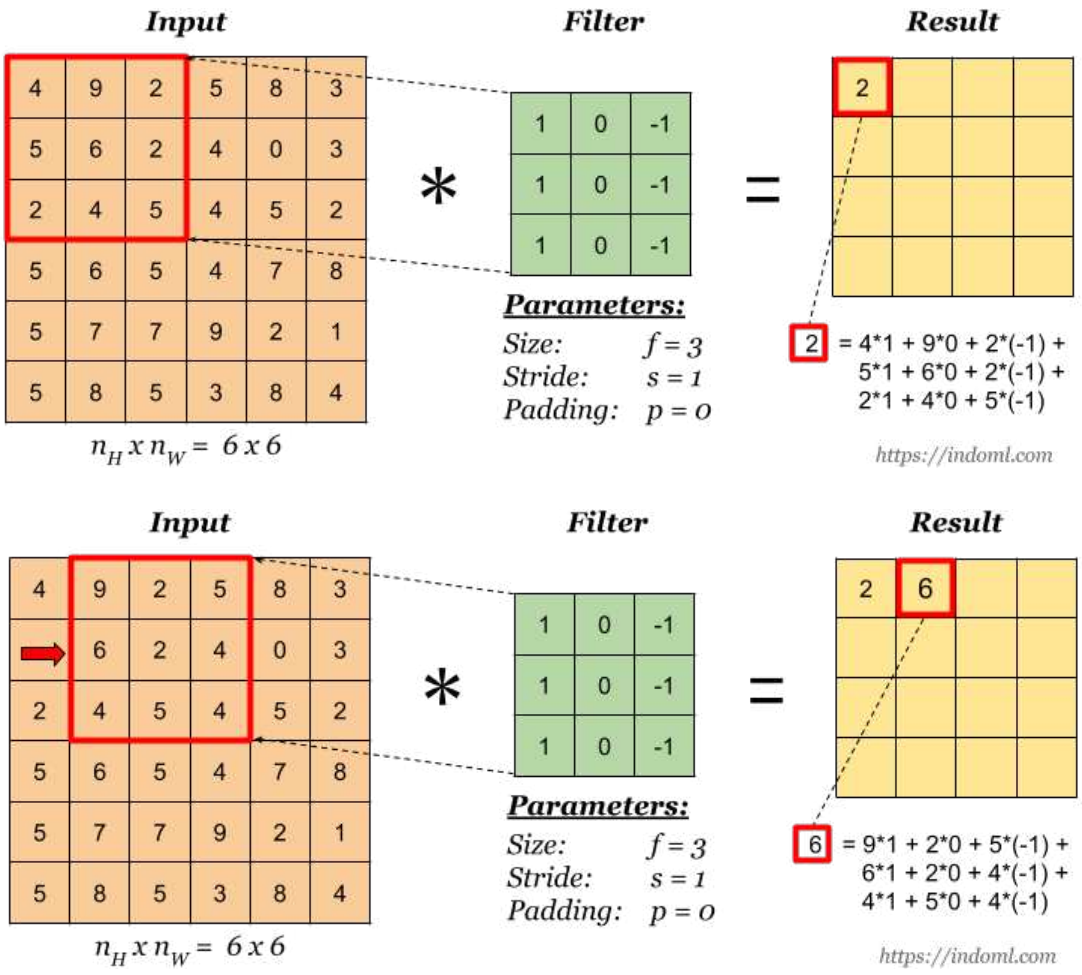
- veličina **filtera** – f,
- **korak** (engl. *stride*) za koji pomičemo filter – s,
- **dopunjenje** (engl. *padding*) je širina (visina) kojom sliku, prije konvolucije, obrubimo nulama – p.

Napomena 2.3.1. Neke implementacije pružaju mogućnost zadavanja parametara po širini i visini. Tako bismo imali f_H, f_W, s_H, s_W, p_H i p_W.

Sada, ako znamo dimenzije ulazne slike, veličinu filtera, korak te dopunjenje, možemo izračunati dimenzije matrice rezultata $k_H \times k_W$:

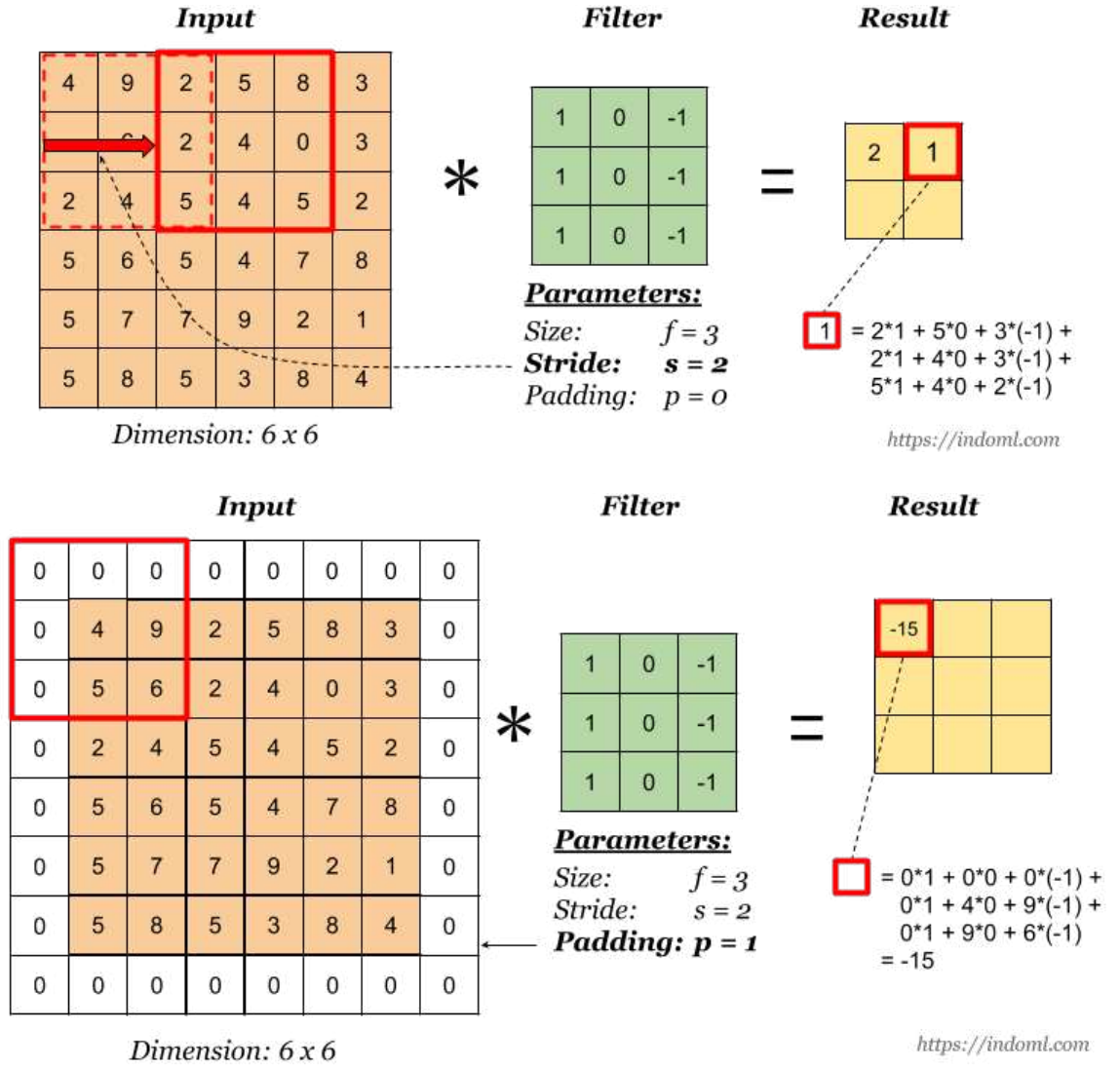
$$k_H = \left\lfloor \frac{n_H - f_H + 2p_H}{s_H} \right\rfloor + 1, \tag{2.22}$$

$$k_W = \left\lfloor \frac{n_W - f_W + 2p_W}{s_W} \right\rfloor + 1. \tag{2.23}$$



Slika 2.4: Prikaz računanja konvolucije

Vidimo matricu kao ulaz dimenzija 6×6*, bez dopunjenja nulama, te filter s težinama, veličine* 3×3*. U ovom slučaju nema pristranosti* b*. Vidimo kako filter djeluje na dio slike te se iterativno kreće po jedan korak udesno. Kada filter dođe do desnog ruba, spušta se za veličinu koraka prema dolje (ovdje je korak jednak* 1*) te nastavlja slijeva nadesno dopunjivati matricu rezultata.* Preuzeto iz [11].



Slika 2.5: Gornji prikaz pojašnjava korake, a donja slika prikazuje dopunjenje nulama

(gore) *U ovom slučaju veličina koraka je* 2. *Filter se iterativno kreće za* 2 *udesno te ne smije prijeći desni rub ulazne matrice. Kada dođe do kraja ruba tada se spušta za veličinu koraka* (2) *prema dolje te, ponovno polazeći slijeva, nastavlja dopunjivati matricu rezultata.* ***(dolje)*** *Donja slika prikazuje dopunjenje nulama. Zbog parametara dobivamo matricu rezultata dimenzija* 3×3. Preuzeto iz [11].

Možemo se upitati zašto nam treba još jedna operacija. Koja je razlika između potpuno povezanog sloja i konvolucijskog sloja? Konvolucija se pokazala jako dobra za slike, jer se isti filter koristi na različitim dijelovima slike. Stoga je konvolucija **invarijantna na translacije** objekta na slici.

Postoje dvije osnovne vrste konvolucije. **Validna** konvolucija, koja ne uključuje dopunjenje nulama te **istovjetna** (engl. *same*) konvolucija, gdje se želi očuvati dimenzija ulazne slike. Da bi se očuvala dimenzija, potrebno je koristiti dopunjenje.

Primijetimo da smo konvoluciju opisali kao operaciju nad matricama, odnosno slikama gdje je pojedini piksel broj. No, to nije uvijek slučaj. Slike uglavnom imaju piksele s tri boje: crvenom, zelenom i plavom. Tada svaki piksel sadrži trodimenzionalan vektor s pripadnim količinama boja te na sliku gledamo kao na trodimenzionalan tenzor, odnosno kvadar i možemo selektirati matrice po bojama. Svaku od tih selektiranih matrica nazivamo **kanal** (engl. *channel*). Broj kanala, odnosno dubinu označavamo s n_C. Da bi filter bio kompatibilan sa slikom, potrebno mu je dodati kanale, tj. treću dimenziju. Stoga, dimenzija filtera je $f_H \times f_W \times n_C$.

Razna softverska okruženja često pružaju mogućnost poretka dimenzija, tako da se nude dvije opcije. Prva, koju smo naveli, jest da kanali dolaze na kraju (engl. *channels last*), a u drugoj, gdje su dimenzije poredane kao $n_C \times f_W \times f_H$, kanali dolaze na početku (engl. *channels first*).

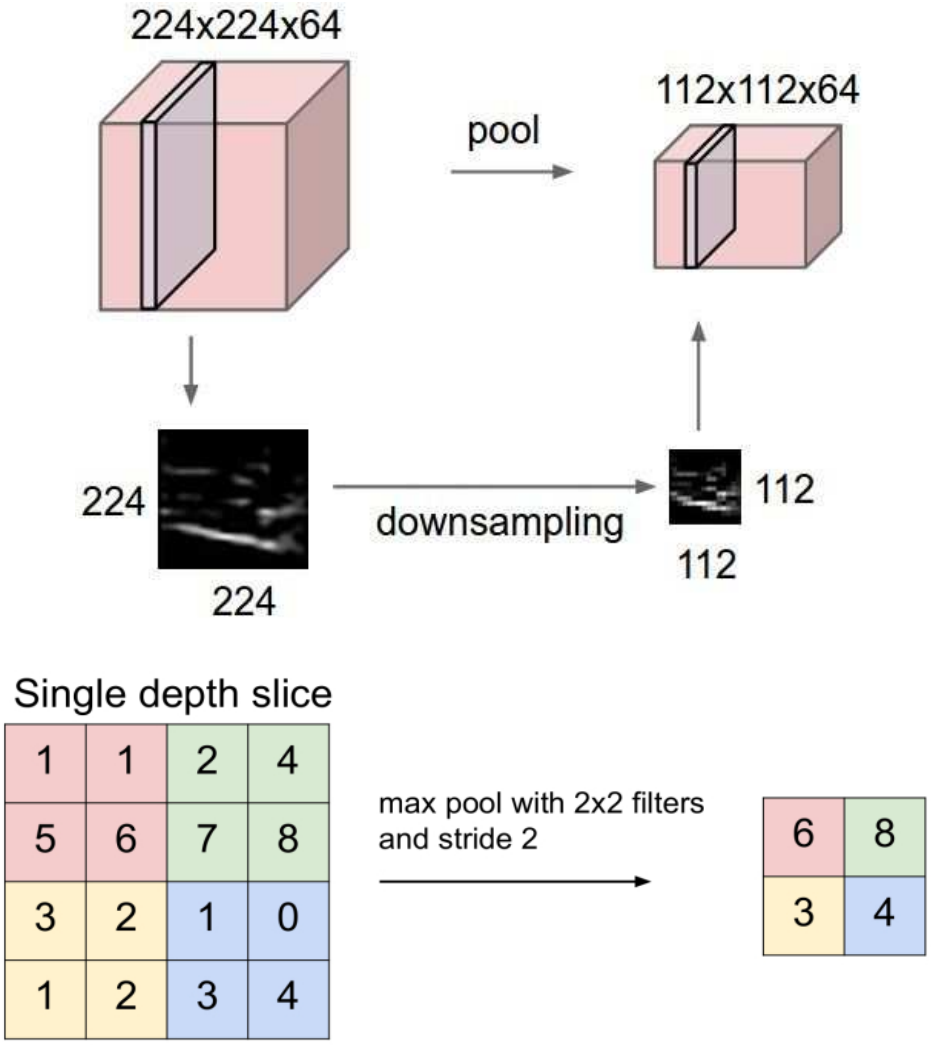
Jedan filter uočava samo jedan uzorak, odnosno značajku. Stoga nam je potrebno više filtera po konvolucijskom sloju, kako bismo detektirali više značajki. Tada su matrice rezultata svih filtera jednakih dimenzija ($k_H \times k_W$) te ih slažemo kao kanale, u nizu jedne iza drugih. Ako broj filtera označimo s n_F, tada je dimenzija rezultata jednaka $k_H \times k_W \times n_F$. Na taj je način u idućem sloju omogućena ponovna uporaba konvolucijskog sloja.

Kao i u potpuno povezanom sloju, na rezultat možemo djelovati nekom od aktivacijskih funkcija. Najčešće je to $ReLU$ funkcija.

Napomena 2.3.2. Ovdje smo obradili dvodimenzionalnu konvoluciju. Analogno, možemo poopćiti ideju za jednu, tri ili više dimenzija. Na primjer, za jednodimenzionalne podatke bismo kao drugu dimenziju imali kanale. Jednodimenzionalnu konvoluciju možemo primijeniti nad sekvencijalnim podacima.

Sloj koji nam je također od interesa, jer se koristi neposredno nakon konvolucijskog sloja, zove se **sažimajući sloj** (engl. *pooling layer*). Naziv je dobio po tome što sakuplja izražene značajke. Osim toga, djeluje tako da reducira dimenzionalnost visine i širine. Najpoznatiji slojevi su **sažimanje maksimuma** (engl. *max pooling*) i **sažimanje prosjeka** (engl. *average poling*). Oba djeluju kao n_C filtera dimenzija $2 \times 2 \times 1$ s korakom 2. Na slici 2.6 možemo vidjeti da *max pooling* uzima maksimume svih trenutno promatranih vrijednosti. Kako se kao aktivacijska funkcija u skrivenim

slojevima uglavnom koristi *ReLU* funkcija, tada veća vrijednost ima veći značaj pa ima smisla uzimati maksimum. Analogno, *average pooling* bi uzimao njihov prosjek.



Slika 2.6: Prikaz sažimajućeg sloja

(gore) *Vidimo kako sažimajući sloj prepolavlja dimenzije visine i širine. Tako se smanjuje broj potrebnih težina za idući sloj, a time i broj izračuna. Na fotografijama vidimo da su važne značajke preživjele.* ***(dolje)*** *Vidimo na koji način se izračunava sažimanje. Uzima se maksimum od trenutnih ćelija te se jezgra pomiče korakom veličine* 2. Primjeri su preuzeti iz [11].

Za sada smo govorili o slojevima koji nam omogućuju razvoj **usmjerenih neuronskih mreža** (engl. *Feedforward Neural Network*). One ne sadrže cikluse, već se informacija prenosi od ulaznog sloja prema izlaznom sloju. Također, postoje tzv. **neuronske mreže s povratnim vezama** (engl. *Recurrent Neural Networks* — **RNN**). One sadrže slojeve s ciklusima. Dolaze do izričaja kod sekvencijalnih podataka, kao što su vremenski nizovi, tekst ili zvuk.

Kod nizova, element niza ovisi o prethodnom ili o više prethodnih elemenata niza. Zato je potrebno omogućiti da se informacija o elementu niza prosljeđuje ne samo prema izlaznom sloju, već i prema obradi idućeg elementa. Ideja ovog sloja, a time i ovakvih tipova neuronskih mreža je da se prolaskom kroz niz jednog primjera obradi element po element, na način da se informacija prema izlaznom sloju množi s jednom matricom težina W_{hy}, a prema obradi idućeg elementa niza s drugom matricom težina W_{hh}. Tako možemo izgraditi smislenu neuronsku mrežu koja će niz preslikavati u niz. Ovakvi slojevi su pogodini za modeliranje tekstovnog prevođenja. Ilustracija povratnih veza 2.7 preuzeta je iz [11].

Neka je s $x_1, x_2, \ldots, x_t$ označen niz elemenata jednog primjera te s $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_t$ niz elemenata izračunatog izlaza. Tada se jedna iteracija standardnog **RNN** modela računa na sljedeći način:

$$a_t = a^1(W_{hh}a_{t-1} + W_{hx}x_t + b_h), \tag{2.24}$$

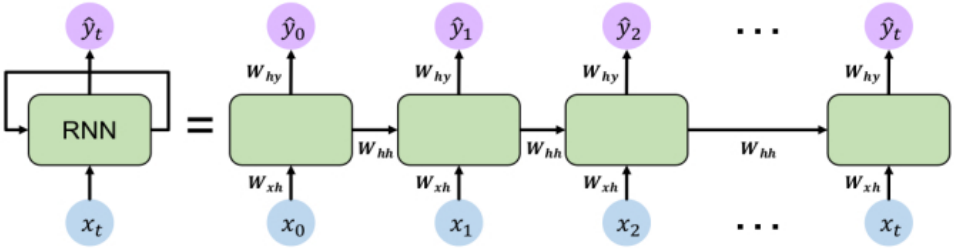
$$\hat{y}_t = a^2(W_{hy}a_t + b_y), \tag{2.25}$$

gdje su a^1 i a^2 aktivacijske funkcije, pri čemu je a^1 najčešće *tanh*, a a^2 *softmax*, zbog računanja predikcija.

Nadogradnja ove ideje su dvosmjerne veze. Naime, ponekad se ključna informacija može nalaziti pri kraju niza te je moguće da sadašnjost niza ovisi o budućnosti. Za takve probleme osmišljeni su slojevi s **dvosmjernim povratnim vezama**. Mana takvog modela je potreba da unaprijed imamo čitav niz spreman za obradu. Kod neuronskih mreža s jednosmjernim povratnim vezama moguće je obrađivati podatke u hodu (engl. *online*).

Ovim potpoglavljem smo stekli uvid u arhitekturu modela neuronskih mreža. Prepoznali smo važnost dubine za učenje reprezentacija i prepoznavanje nelinearnosti među podacima. Također, značajne su i aktivacijske funkcije, jer su one temelj nelinearnosti. Bez njih bi kompozicija linearnih slojeva bila opetovano linearna.

Obradili smo nekoliko tipova slojeva za razne primjene. Potpuno povezan sloj je predstavljen kao osnova. Njime možemo rješavati problem klasifikacije i regresije. Također ga je moguće koristiti za obradu računalnog vida, no bolje rješenje daju konvolucijski slojevi sa slojevima sažimanja. Za kraj je predstavljena ideja povratnih veza koja olakšava obradu nizova.



Slika 2.7: Prikaz sloja s povratnim vezama

Ilustracija prikazuje niz (x_t) *kao ulaze koji generiraju niz* $(\hat{y}_t)$*. Svaki element* x_i *se, nakon osnovne obrade, množi matricom* W_{hy} *te rezultat prosljeđuje prema izlaznom sloju. Također se, nakon osnovne obrade, množi matricom* W_{hh} *te prosljeđuje obradi idućeg elementa.* Primjeri su preuzeti iz [11].

2.4 Algoritam učenja

Nakon što za određeni problem prikupimo i pretprocesiramo podatke, odredimo mjere uspješnosti i funkciju troška J te konstruiramo neuronsku mrežu kao funkciju $f(x;\, \vec{\theta})$ s pripadnim parametrima $\vec{\theta}$, model je potrebno **trenirati**, odnosno **naučiti**. Nazivi asociraju na proces učenja iskustvom, odnosno iz podataka. Težine W i pristranosti b su najčešći parametri, no, ovisno o slojevima mreže, mogući su i neki drugi. Zapravo tražimo parametre $\vec{\theta}$, takve da trošak na podacima za učenje D bude minimalan:

$$\arg\min_{\vec{\theta}} J(D;\, \vec{\theta}). \tag{2.26}$$

Time dobivamo težine $\vec{\theta}$ koje definiraju traženu funkciju. Prostor koji pretražujemo je uglavnom nekonveksan, zbog čega postoji mnogo lokalnih minimuma. Stoga ne tražimo egzaktan globalni minimum, već onaj koji će zadovoljiti naše kriterije. Ovdje se nameće pojam **generalizacije**. To je sposobnost dobre predikcije na novim primjerima. Ako mreža ima mali trošak nad podacima za učenje, a veliki nad testnim podacima, kažemo da je model **prenaučen** (engl. *overfitting*), što znači da loše generalizira. Moguća je i **podnaučenost** (engl. *underfitting*) modela. Tada je trošak velik već nad podacima za učenje.

Kako dimenzija parametara $\vec{\theta}$ varira od nekoliko tisuća do nekoliko milijuna, moguće i stotina milijuna, prostor kojeg pretražujemo je značajan. Za pretraživanje bismo mogli koristiti neku od stohastičkih metoda, no zbog ovisnosti težina dubljih slojeva o prethodnim to nije opcija. Druga opcija bi mogla biti Newtonova metoda. Ona koristi derivacije drugog reda, zbog čega u praksi također nije moguća, jer ima prevelik broj parametara.

Opće prihvaćena metoda učenja je **gradijentni spust**. To je način minimizacije funkcije troška J tako što se iterativno korigiraju parametri $\vec{\theta}$ sa suprotnom orijentacijom gradijenta $\nabla_{\vec{\theta}} J(D;\, \vec{\theta})$. Kako je gradijent vektor smjera rasta funkcije, ovim se načinom spuštamo niz liticu. Pri tome postoji **faktor učenja** ϵ (engl. *learning rate*), kojim množimo gradijent kako bismo regulirali njegovu veličinu. Ako je gradijent prevelik, lako može doći do oscilacija u korigiranju težina, odnosno divergencije. S druge strane, ako je ϵ premali, konvergencija k minimumu može biti spora.

Postoje tri podvrste gradijentnog spusta s obzirom koju količinu podataka koristimo pri jednom korigiranju parametara. **Osnovni**, koji koristi sve podatke za učenje odjednom, pri računanju derivacija i korigiranju težina. Engleski naziv je *batch gradient descent*, u smislu da se koriste svi podaci pri svakom korigiranju. Mana ovog algoritma je što za veliku količinu podataka troši puno memorije, jer računa gradijente od svih primjera odjednom.

Idući je **stohastični** spust (engl. *stochastic gradient descent* – **SGD**). On korigira težine za svaki podatak posebno. Na neki način je suprotnost osnovnog algoritma. Njegova mana je što svaki primjer može imati jako različit gradijent u odnosu na druge primjere pa se događa velika oscilacija, kako učenje iterira kroz podatke. Prednost je što lagano možemo dodavati nove primjere tijekom učenja, tzv. *online* učenje.

Posljednji je tzv. *mini-batch gradient descent*. Taj algoritam dijeli podatke na slučajan način u manje skupove, tzv. **grupe**, te iterativno računa gradijente i korigira težine za svaku grupu posebno. Njegova prednost je što koristi dovoljno primjera pri korigiranju da se ne događaju velike oscilacije među gradijentima različitih grupa. Također, koristi broj primjera koji stanu u memoriju, tj. moguća je matrična optimizacija. Zato je najkorišteniji. Zbog spremanja u memoriju, veličina grupa (engl. *mini-batch size*) je uglavnom potencija broja 2. Najčešće je to: $32, 64, 128, \ldots, 1024$.

Jedno iteriranje kroz čitav skup podataka za učenje naziva se **epoha**. Učenje se provodi kroz više desetina ili stotina epoha. Često, svakih nekoliko epoha validiramo učenje na prikladnim podacima, kako bismo se uvjerili da će model dobro generalizirati, odnosno da ne dolazi do pretreniranja ili podtreniranja.

Algorithm 1: Algoritam učenja pomoću gradijetnog spusta s grupama

for $i \leftarrow 1$ *to epohe* **do**
  $permutiraj(D)$;
  for $j \leftarrow 1$ *to* $broj\_grupa$ **do**
    $\vec{g} \leftarrow \nabla_{\vec{\theta}} J(D_j;\, \vec{\theta})$;
    $\vec{\theta} \leftarrow \vec{\theta} - \epsilon \cdot \vec{g}$;
  end
end

Svoju popularnost algoritam gradijentnog spusta je stekao otkrićem algoritma **propagiranja greške unazad** (engl. *error backpropagation*). Algoritam koristi ideju derivacija komponiranih funkcija. Za primjer, neka je $x \in \mathbb{R}$ i neka su $f\colon \mathbb{R} \to \mathbb{R}$ i $g\colon \mathbb{R} \to \mathbb{R}$ funkcije i neka je $y = f(x)$ i $z = g(y) = g(f(x))$. Tada, ako su funkcije f i g derivabilne u x i y, za derivaciju kompozicije vrijedi:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.27}$$

Ovu vezu možemo poopćiti za funkcije više varijabli. Neka su $\vec{x} \in \mathbb{R}^n$ i $\vec{y} \in \mathbb{R}^m$ te $f\colon \mathbb{R}^n \to \mathbb{R}^m$ i $g\colon \mathbb{R}^m \to \mathbb{R}$ diferencijabilne funkcije više varijabli u $\vec{x}$ i $\vec{y}$. Neka vrijedi $\vec{y} = f(\vec{x})$ i $z = g(\vec{y}) = g(f(\vec{x}))$. Tada vrijedi:

$$\frac{\partial z}{\partial \vec{x}_i} = \sum_j \frac{\partial z}{\partial \vec{y}_j}\frac{\partial \vec{y}_j}{\partial \vec{x}_i}, \tag{2.28}$$

ili, ekvivalentno:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z. \tag{2.29}$$

Kao što možemo vidjeti, svrha propagiranja greške unazad je **računanje gradijenata**. Prvo se, redom kroz slojeve, računaju funkcije te se propagiraju izračunate vrijednosti unaprijed (engl. *forward propagation*). Izračunate vrijednosti se spremaju. Na kraju dobijemo izlaznu vrijednost neuronske mreže. Tada se izračuna funkcija troška te se ulančano, propagiranjem unazad pomoću spremljenih vrijednosti funkcija dobivenih propagiranjem unaprijed, izračunavaju diferencijali pojedinih težina. Za svaki primjer iz grupe ponovimo proces dviju propagacija te akumuliramo diferencijale pripadnih težina. Pri tome, računanje propagiranja grupe se implementira tako da se cijela grupa odjednom propagira, najčešće kao matrica, gdje su primjeri poredani po stupcima. Nakon toga se težine korigiraju pomoću gradijentnog spusta ili nekog drugog algoritma baziranog na gradijentima.

Na kraju, napomenimo da **inicijalizacija parametara** treba biti slučajna. U suprotnom, ako dva neurona istog sloja imaju jednake težine, oni će učiti na jednak način. Također, matrica težina W ne smije biti nula, dok pristranosti b smiju, jer ne djeluju direktno na podatke. Pokazalo se poželjnim da su težine inicijalizirane oko nule, zbog aktivacijskih funkcija.

Poglavlje 3

Optimizacijski algoritmi

Ranije smo opisali gradijentni spust. Cilj ovog poglavlja je pojasniti njegove **nadogradnje**. Naime, učenje, ovisno o problemu, podacima, veličini modela, vrsti slojeva, može trajati od nekoliko minuta do nekoliko sati, dana ili tjedana. Zato želimo algoritme koji su stabilni i koji konvergiraju brže.

S druge strane, do sada navedeni algoritmi koriste faktor učenja ϵ kojeg je potrebno ugađati. Često se koriste metode **simuliranog kaljenja** ili **unaprijed definiranog rasporeda** smanjivanja faktora učenja. Također, pri nekonveksnoj optimizaciji, unatoč ustaljenom vjerovanju da algoritam zapinje u lokalnom minimumu, zapravo je najčešće riječ o **sedlastim točkama**. Naime, s porastom broja parametara, smanjuje se vjerojatnost za lokalne minimume, a neuronska mreža ih ima od nekoliko tisuća do nekoliko milijuna.

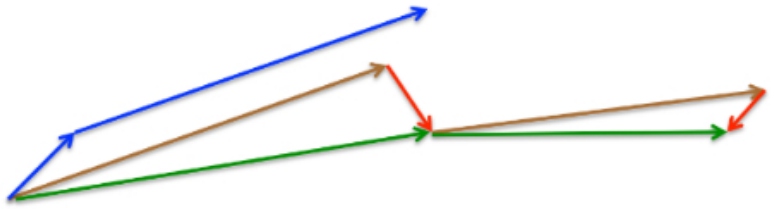
Osim navedenih, problem stohastičnog gradijentnog spusta je taj što, pri traženju minimuma, su mogući dijelovi prostora koji imaju neke dimenzije strmije od drugih, što možemo zamisliti kao konture u eliptičnom obliku. To dovodi do oscilacija i moguće divergencije. Kao rješenje tog problema osmišljen je **momentum** (moment).

Algorithm 2: Gradijentni spust s momentumom, s grupama

$\vec{v} \leftarrow 0$;
for $i \leftarrow 1$ *to epohe* **do**
  permutiraj(D);
  for $j \leftarrow 1$ *to broj_grupa* **do**
    $\vec{v} \leftarrow \gamma\vec{v} + \epsilon \cdot \nabla_{\vec{\theta}} J(D_j; \vec{\theta})$;
    $\vec{\theta} \leftarrow \vec{\theta} - \vec{v}$;
  end
end

Ideja momentuma je da koristimo **eksponencijalni pomični prosjek**. U momentu $\vec{v}$ akumuliramo gradijente tako da posljednji gradijent najviše utječe, prethodni malo manje, itd. Osim toga, moment se ubrzava u smjeru dimenzija čiji uzastopni diferencijali pokazuju u istom smjeru, a usporava ako osciliraju. Faktor γ određuje doprinos prethodnih gradijenata. Kada bi γ bio 0, gledao bi se samo trenutni gradijent, a kada bi γ bio 1, gledali bi se svi gradijenti jednako. Za standard je uzeta vrijednost 0,9.

Momentumom smo ispravili prethodno naveden problem, jer se smanjila mogućnost naglog zakretanja smjera kojim korigiramo parametre. Na žalost, niti ovo rješenje nije idealno. Pogledajmo sliku 3.1. Ako prvo računamo trenutni gradijent (mali plavi vektor) te ga pridodamo momentu (veliki plavi vektor), dobit ćemo drugačiji rezultat u odnosu da smo se prvo pomaknuli za moment (lijevi smeđi vektor) te u toj točki izračunali novi gradijent (lijevi crveni vektor). Vidimo da je poredak računanja trenutnog gradijenta i momenta bitan. Drugi način **sprečava prebrzo kretanje** momenta, odnosno ako moment naiđe na uzbrdicu, lakše se može usporiti i usmjeriti prema minimumu. U protivnom bi mogao prijeći preko uzbrdice te ne bi konvergirao prema minimumu. Stoga želimo da se trenutni gradijent računa nakon momenta. Ovaj algoritam se naziva **Nesterovljev ubrzani gradijent**.



Slika 3.1: Moment

Plavi vektori prikazuju prvo računanje trenutnog gradijenta pa dodavanje momenta. Vidimo da njihov rezultantni vektor nije jednak lijevom zelenom, kojeg bismo dobili da se prvo pomaknemo za moment, a zatim računamo trenutni gradijent. Primjer sa slike je preuzet iz [15].

Postavlja se pitanje kako zapravo izračunati gradijent, jer se gradijenti računaju s određenim podacima i određenim težinama. Stoga bismo trebali spremati težine prije računanja gradijenta, zatim oduzeti moment pa ih vratiti pri korigiranju težina. Takve komplikacije nisu potrebne. Neka su $\vec{v}_t$ i $\vec{\theta}_t$ moment i parametri u pripadnim

Algorithm 3: Nesterovljev ubrzani gradijent, s grupama

```
v⃗ ← 0;
for i ← 1 to epohe do
    permutiraj(D);
    for j ← 1 to broj_grupa do
        v⃗ ← γv⃗ + ϵ · ∇_θ⃗ J(D_j; θ⃗ − γv⃗);
        θ⃗ ← θ⃗ − v⃗;
    end
end
```

trenucima t te D skup podataka za izračune. Tada vrijedi:

$$\vec{v}_t = \gamma\vec{v}_{t-1} + \epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\theta}_{t-1} - \gamma\vec{v}_{t-1}), \tag{3.1}$$

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \vec{v}_t. \tag{3.2}$$

Sada uvrstimo $\vec{v}_t$:

$$\vec{v}_t = \gamma\vec{v}_{t-1} + \epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\theta}_{t-1} - \gamma\vec{v}_{t-1}), \tag{3.3}$$

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \left[\gamma\vec{v}_{t-1} + \epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\theta}_{t-1} - \gamma\vec{v}_{t-1})\right]. \tag{3.4}$$

Za kraj supstituirajmo $\vec{\Theta}_{t-1} = \vec{\theta}_{t-1} - \gamma\vec{v}_{t-1}$:

$$\vec{v}_t = \gamma\vec{v}_{t-1} + \epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\Theta}_{t-1}), \tag{3.5}$$

$$\begin{aligned}\vec{\Theta}_t &= -\gamma\vec{v}_t + \vec{\Theta}_{t-1} + \gamma\vec{v}_{t-1} - \left[\gamma\vec{v}_{t-1} + \epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\Theta}_{t-1})\right] \\ &= \vec{\Theta}_{t-1} - \gamma\vec{v}_t - \epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\Theta}_{t-1}) \\ &= \vec{\Theta}_{t-1} - \gamma^2\vec{v}_{t-1} - (\gamma+1)\epsilon \cdot \nabla_{\vec{\theta}} J(D;\, \vec{\Theta}_{t-1}).\end{aligned} \tag{3.6}$$

Gornje jednadžbe podsjećaju na računanje gradijenta s momentom. Sada, ako zamijenimo poredak izračuna, pri čemu je $\vec{v}_0 = 0$, dobivamo izračun Nesterovljevog ubrzanog gradijenta koji je dan s algoritmom 3. Napomenimo kako se pokazao dobrim za učenje neuronskih mreža s povratnim vezama.

Do sada su parametri $\vec{\theta}_i$, za sve i, koristili jednaki faktor učenja ϵ. Algoritam koji se pokazao dobar za duboke modele i koji skalira faktor učenja za svaki parametar posebno, ovisno o svim prethodnim gradijentima, naziva se **Adagrad**. Zapravo se za svaki parametar θ_i posebno računa suma svih dotadašnjih kvadrata diferencijala $\vec{g}_i$, koji se spremaju na dijagonali matrice, G_{ii}. Tada se korijenom sume dijeli faktor učenja ϵ te se na taj način kroz iteracije smanjuje korak gradijenta.

Algorithm 4: Adagrad s grupama

for $i \leftarrow 1$ *to epohe* **do**
  $permutiraj(D)$;
  for $j \leftarrow 1$ *to broj_grupa* **do**
    $\vec{g} \leftarrow \nabla_{\vec{\theta}} J(D_j; \vec{\theta})$;
    $\vec{\theta} \leftarrow \vec{\theta} - \frac{\epsilon}{\sqrt{G+\delta}} * \vec{g}$;
  end
end

U pseudokodu algoritma 4, $*$ označava množenje dijagonalne matrice G i vektora $\vec{g}$ po elementima. Parametar δ služi izbjegavanju dijeljenja s nulom, stoga se uzima kao 10^{-8}, a za ϵ se uzima 10^{-2} te ga nije potrebno ugađati tijekom učenja. Nazivnik bez korijena se u praksi pokazao lošim. S iteracijama, matrica G je rastuća, stoga ukupan faktor učenja konvergira u nulu.

Sljedeće bi bilo poželjno da faktor učenja ϵ nije potreban. To nam u nekom smislu omogućuje **RMSprop**. Krenimo prvo od toga da s iteracijama Adagrad sve slabije uči, zbog prethodno navedene konvergencije. Kako bismo riješili taj problem, opet koristimo eksponencijalni pomični prosjek. Ovaj puta za kvadrate gradijenata.

Algorithm 5: RMSprop s grupama

$\vec{w} \leftarrow 0$;
for $i \leftarrow 1$ *to epohe* **do**
  $permutiraj(D)$;
  for $j \leftarrow 1$ *to broj_grupa* **do**
    $\vec{g} \leftarrow \nabla_{\vec{\theta}} J(D_j; \vec{\theta})$;
    $\vec{w} \leftarrow \gamma\vec{w} + (1-\gamma)\vec{g}^2$;
    $\vec{\theta} \leftarrow \vec{\theta} - \frac{\epsilon}{\sqrt{\vec{w}+\delta}} * \vec{g}$;
  end
end

Algoritam je dobio naziv zbog korijena prosječnih kvadrata gradijenata (engl. *root mean squared* – **RMS**). Za γ je preporučena vrijednost 0,9, a za ϵ 0,001. Zbog toga što nije potrebno ugađanje faktora učenja, ne smatramo ga bitnim parametrom algoritma.

Za kraj ćemo predstaviti **Adam** (engl. *Adaptive Moment Estimation*) algoritam koji skalira faktor učenja. To je kombinacija RMSprop algoritma i momentuma. Za svaki parametar računaju se eksponencijalni pomični prosjeci za diferencijale i za

kvadrate diferencijala. Kako su prvi prosjeci blizu nuli, potrebno ih je korigirati pa ih dijelimo s $(1 - \beta^t)$, gdje je t ukupni broj iteracija od početka algoritma.

Algorithm 6: Adam s grupama

$\vec{v} \leftarrow 0$;
$\vec{w} \leftarrow 0$;
$t \leftarrow 1$;
for $i \leftarrow 1$ *to epohe* **do**
  $permutiraj(D)$;
  for $j \leftarrow 1$ *to broj_grupa* **do**
    $\vec{v} \leftarrow \beta_1 \vec{v} + (1 - \beta_1) \nabla_{\vec{\theta}} J(D_j;\, \vec{\theta})$;
    $\vec{w} \leftarrow \beta_2 \vec{w} + (1 - \beta_2) \nabla_{\vec{\theta}} J(D_j;\, \vec{\theta})^2$;
    $\vec{v}_k \leftarrow \frac{\vec{v}}{1-\beta_1^t}$;
    $\vec{w}_k \leftarrow \frac{\vec{w}}{1-\beta_2^t}$;
    $\vec{\theta} \leftarrow \vec{\theta} - \frac{\epsilon}{\sqrt{\vec{w}_k} + \delta} * \vec{v}_k$;
    $t \leftarrow t + 1$;
  end
end

Za moment smo koristili, kao i ranije, oznaku $\vec{v}$. Za drugi moment oznaka je $\vec{w}$, a korigirane vrijednosti su $\vec{v}_k$ i $\vec{w}_k$. Preporučene vrijednosti su 10^{-8} za δ, 0,9 za β_1 te 0,999 za β_2. Za ovaj algoritam također nije potrebno modificirati δ tijekom učenja. Ovaj algoritam se preporuča ako korisnik nema iskustva s učenjem modela.

U prethodnim algoritmima smo koristili grupe. Primijetimo, ako je grupa veličine jednaka 1, tada se radi o stohastičkom obliku algoritma. Ako je cijeli skup podataka za učenje jedna grupa, onda se radi o općenitoj vrsti gradijentnog spusta.

Navest ćemo još neke optimizacijske algoritme. **Adadelta** je sličan RMSprop algoritmu te je razvijen u isto vrijeme. Oba algoritma su popravljala nedostatke Adagrada. Ako povežemo ideje Adama s Nesterovljevim, dobiti ćemo **Nadama**. Ako, umjesto računanja drugog momenta, u algoritmu Adam koristimo maksimum, dobiti ćemo **Adamax**.

Kod problema sedlastih točki, za razliku SDG-a i SDG-a s momentom, algoritmi RMSprop, Adagrad, Adadelta i Nesterovljev algoritam ga rješavaju uspješno.

Danas postoje različite paralelizacije računanja i softverska okruženja koja to nude. Neka od njih namijenjena za duboko učenje jesu caffe [7], keras [3], pyTorch [13] i tensorflow [1]. U sebi imaju ugrađene optimizacijske algoritme te implementaciju

propagiranja unazad. Paralelizacije su omogućene na CPU–ima, GPU–ima te TPU–ima. Time je omogućeno višestruko ubrzanje učenja, od nekoliko desetina, stotina pa i do tisuću puta.

Kod učenja dubokih modela, zbog učestalog množenja diferencijala pri propagiranju greške unazad, često može doći do **eksplodirajućih** ili **nestajućih** gradijenata. Može se dogoditi da gradijenti postanu jako veliki te trošak počne divergirati ili naglo naraste. Moguće je da dobijemo vrijednost NaN, što bi značilo da moramo učenje započeti ponovno. Suprotno od toga su nestajući gradijenti. Također, zbog množenja jako malih vrijednosti, može se dogoditi da model prestane učiti ili uči jako sporo.

Rješenje za eksplodirajuće gradijente je postavljanje praga. Ako je veći od praga tada stavimo da je jednak pragu. Tako smjer gradijenta ostane približno isti, a ne događaju se veliki skokovi. S druge strane, nestajuće gradijente je teže riješiti. Najčešće je potrebno modificirati model ili učiti ispočetka nekim drugim optimizacijskim algoritmom.

Neke metode koje pomažu učenju, osim već spomenutog permutiranja podataka, simuliranog kaljenja i rasporeda faktora učenja, su:

- **normalizacija grupe** (engl. *batch normalization*) — kako se propagiranjem unaprijed prosljeđuje cijela grupa sloj po sloj, tada je moguće napraviti sloj koji će normalizirati ulazne podatke te ih proslijediti idućem sloju,
- **rano zaustavljanje učenja** (engl. *early stopping*) — nakon svake epohe se radi validacija nad podacima na kojima se ne uči pa ako vidimo da se učenje pogoršava stanemo,
- tzv. **dropout sloj** — određeni postotak neurona prethodnog sloja na slučajan način množimo s nulom i na taj način ga deaktiviramo. Time preživjeli neuroni moraju naučiti znanje koje su imali deaktivirani,
- **šumiranje gradijenta** — gradijentu pridodamo vrijednosti iz normalne distribucije $\mathcal{N}(0, \sigma^2)$ pri svakom korigiranju težina,
- **kurikularno učenje** — umjesto permutacije podataka, smisleno poredamo primjere kako bismo olakšali učenje,
- **augmentacija podataka** — smisleno kreiranje novih podataka. Na primjer, sliku možemo rotirati i zrcaliti. Time ne utječemo na semantiku, a generiramo novi podatak.

Poglavlje 4

Mrežne arhitekture i primjene

4.1 Konvolucijske neuronske mreže

Konvolucijska neuronska mreža (engl. *convolutional neural network* — **CNN**) je mreža koja sadrži konvolucijski sloj ili više njih. Konvolucijski sloj smo obradili ranije. Sada ćemo obraditi neke od najpoznatijih konvolucijskih mreža čija je svrha rješavanje problema u području **računalnog vida**. Dakle, radi se o dvodimenzionalnim slojevima. Rjeđe se koriste jednodimenzionalna konvolucija u obradi teksta, zvuka ili nizova, te trodimenzionalna za npr. obradu medicinskih podataka, kao što su trodimenzionalna slika magnetne rezonance.

Napomenimo da postoji konvencija, kada se govori o broju slojeva koje sadrži neka mreža, da se broje slojevi koji sadrže težine. Na primjer, ako imamo konvolucijski sloj te nakon njega sloj sažimanja, ta dva sloja bismo gledali kao jedan, jer sažimanje nema težina.

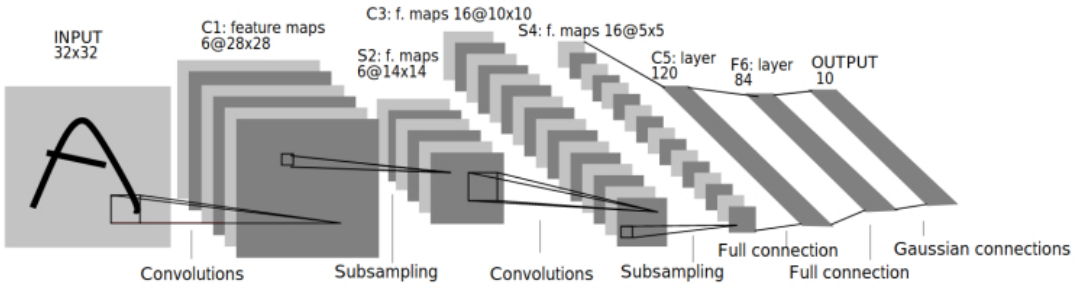
Problemi računalnog vida kojeg rješavaju mreže jesu:

- **klasifikacija** ili **prepoznavanje** (engl. *classification*) — želimo prepoznati koji je objekt na slici, a pri tome znamo moguće klase,
- **lokalizacija objekta** (engl. *object localization*) — želimo pronaći objekt na slici i uokviriti ga,
- **klasifikacija s lokalizacijom** (engl. *classification and localization*) — pitamo se koji je objekt i gdje se nalazi,
- **detekcija objekata** (engl. *object detection*) — klasifikacija s lokalizacijom i prepoznavanjem više objekata,
- **semantička segmentacija** (engl. *semantic segmentation*) — cilj je za svaki piksel odrediti klasu obzirom na kontekst, a svaki piksel pripada nekoj klasi,

- **segmentacija instanci** (engl. *instance segmentation*) — za svaki piksel želimo odrediti klasu kojoj pripada te, dodatno, instancu objekta na slici.

LeNet-5

Jedna od najpoznatijih prvih mreža je **LeNet-5** [9]. Konstruirana je 1998. godine za prepoznavanje rukom pisanih znamenki. Primijenjena je u bankarskom sektoru za čitanje čekova. U članku su obradili starije metode za prepoznavanje slova, za koje su bile potrebne heuristike i ručno ugrađene značajke. Prednost LeNet-5 je što pomoću **gradijenta uči značajke**.



Slika 4.1: CNN LeNet

Slika prikazuje arhitekturu LeNet-5 neuronske mreže. Preuzeto iz [9].

Kao što možemo vidjeti na slici 4.1, dimenzija ulaza je $32 \times 32 \times 1$ te nije korišteno dopunjenje nulama. Zatim, korišten je konvolucijski sloj s filterom veličine 5×5 i korakom 1. Također je korišteno prosječno sažimanje. Prva konvolucija C1 sadrži 6, a zatim C3 sadrži 16 filtera. Korištena je sigmoidalna funkcija kao **aktivacijska nakon sažimanja**, što je za današnje modele neuobičajeno. Kao zadnji konvolucijski sloj C5, korišteno je 120 filtera dimenzija 5×5. Kako je dimenzija ulaza u taj sloj također 5×5, dobije se izlaz dimenzija $1 \times 1 \times 120$, što je **ekvivalentno** korištenju **potpuno povezanog sloja**. Zatim je korišten potpuno povezan sloj te se kao izlaz mreže dobiva vektor veličine 10, jer predviđamo jednu od 10 znamenki. Ukupan broj parametara je otprilike 60 tisuća.

AlexNet

Iduća mreža, **AlexNet** [8] je dobila naziv po tvorcu. Produkt je ImageNet-ovog [4] natjecanja u klasifikaciji 1,2 milijuna slika visoke rezolucije u 1000 kategorija.

Mreža se sastoji od 5 konvolucijskih slojeva te 3 potpuno povezana. Izlaz mreže je dimenzije 1000, sa *softmax* aktivacijskom funkcijom. Prva dva sloja sadrže tzv. *local response-normalization* — ***LRN***. Ideja je da se normalizira vrijednosti po kanalima. U kasnijim arhitekturama to se nije pokazalo korisnim [12]. Zanimljivo je da prvi konvolucijski sloj sadrži filtere dimenzija $11 \times 11 \times 3$ s korakom 4. Kasniji slojevi koriste filtere 5×5 te 3×3. Korišten je *max pooling* te **ReLU** aktivacijska funkcija. **Max pooling** je dimenzija 3×3 s korakom 2. Za bolju generalizaciju koristili su tada novu **dropout** metodu.

Za razliku od LeNet-5 mreže, AlexNet sadrži 60 milijuna parametara. Trenirana je s dva GPU-a. Pokazali su, da ako izostave neki konvolucijski sloj, izgubili bi oko 2% za TOP-1 testnu grešku, gdje je **TOP-n** mjera kojom mjerimo nalazi li se tražena oznaka među najboljih n predikcija modela. Drugim riječima, dubina je važna za kvalitetu mreže. Ovaj model je uvjerio zajednicu da duboko učenje funkcionira.

VGG16

Kao i AlexNet, **VGG16** [16] je nastala kao rezultat ImageNet-ovog natjecanja 2014. godine. Osvojili su prvo mjesto za lokalizaciju i drugo za problem klasifikacije. Mreža se sastoji od 16 slojeva, iako su implementirali više mreža raznih dubina. Time je još jednom potvrđena uloga dubine.

Kao što možemo vidjeti na slici 4.2, mreža se sastoji od bloka konvolucija nakon kojih slijedi *max pooling*. Dimenzije konvolucijskih filtera su 3×3 s korakom 1. Ovdje je korištena *istovjetna* konvolucija, čime je omogućeno očuvanje ulaznih dimenzija. Zato je za reduciranje dimenzionalnosti bilo potrebno sažimanje 2×2 s korakom 2. Dakle, s dubinom se smanjuju širina i visina, a broj kanala raste eksponencijalno s faktorom 2. Pravilo rasta broja kanala te pada širine i visine smo mogli primijetiti kod LeNet-5 mreže. Za aktivaciju je korištena ReLU funkcija te su pokazali da **LRN ne pospješuje uspješnost**, već vodi do povećane memorije i duljeg računanja. Za kraj, mreža sadrži **138 milijuna** parametra, što je ekstremno i danas.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Slika 4.2: VGG slojevi

Možemo vidjeti arhitekturu mreže VGG16 pod stupcem D) te VGG19 pod stupcem E). Preuzeto iz [16].

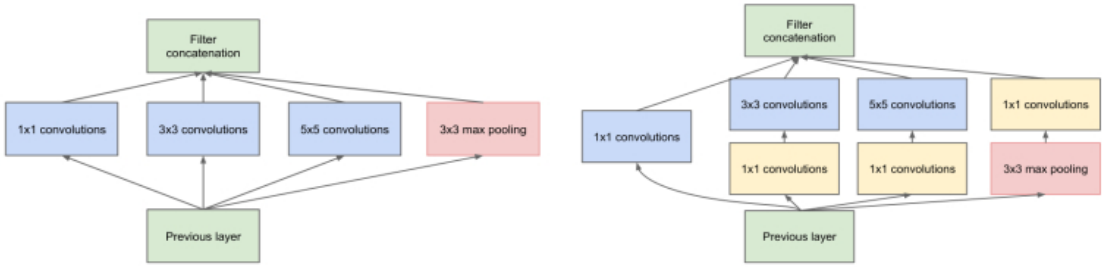
GoogLeNet

Ova mreža je vrste **Inception** [17]. GoogLeNet je specifična mreža od 22 sloja s kojom su se natjecali u ImageNet-ovom natjecanju 2014. godine. U klasifikaciji TOP-5 su pobijedili sa 6,67% greške, a VGG16 je bio drugi sa 7,32%. Model je učen stohastičkim gradijentom s momentom 0,9 i raspoređivanjem smanjenja faktora učenja za 4% svakih 8 epoha.

Nepisano pravilo je da s više podataka modeliramo jednostavnije, a s manje kompleksnije modele. Kako bi kreirali modele bolje kvalitete znanstvenici posežu za složenijim modelima, uključujući dubinu i broj neurona u svakom sloju. Iz toga slijedi veći broj parametara, što znači lakše pretreniranje.

Cilj Google-ovog tima je bio bolje iskoristiti korištenje resursa potrebnih za računanje. Inspiraciju su dobili od tzv. **Network in network** [10] arhitekture, gdje je osmišljen način za **ponavljanje manjih blokova** neuronskih mreža.

U tom radu je korištena konvolucija s dimenzijom filtera 1×1. Takav sloj služi sažimanju po kanalima. Naime, širina i visina će ostati ista, a broj kanala će se promijeniti ovisno o broju filtera. Koliko se stavi takvih filtera, toliko će biti kanala. Time možemo reducirati broj kanala. Npr. neka imamo podatke dimenzija $20 \times 20 \times 200$. Ako stavimo 30 filtera dimenzija 1×1, dobiveni izlaz će biti dimenzija $20 \times 20 \times 30$. S ovakvom konvolucijom treba biti oprezan, jer može biti usko grlo modela.



Slika 4.3: Usporedba sa i bez konvolucije s 1×1 filterima

(lijevo) *Vidimo prikaz bloka koji sadrži* 4 *sloja. Tri s istovjetnom konvolucijom (plavi) i jedan s istovjetnim sažimanjem i korakom* 1 *(crveni). Stoga, širina i visina ostaju jednake te ih možemo konkatenirati po kanalima. Tako dobivamo rezultat s velikim brojem kanala.* ***(desno)*** *S konvolucijom* 1×1 *reduciramo broj kanala te time i računarsku složenost.* Preuzeto iz [17].

Slika 4.3 prikazuje kako praktično reducirati složenost računanja. Uzmimo za primjer dva ulančana konvolucijska sloja. Ako bilo kojem povećamo broj filtera, složenost računanja će se povećati značajno. Tako GoogLeNet ima 12 puta manje parametara nego AlexNet.

ResNet

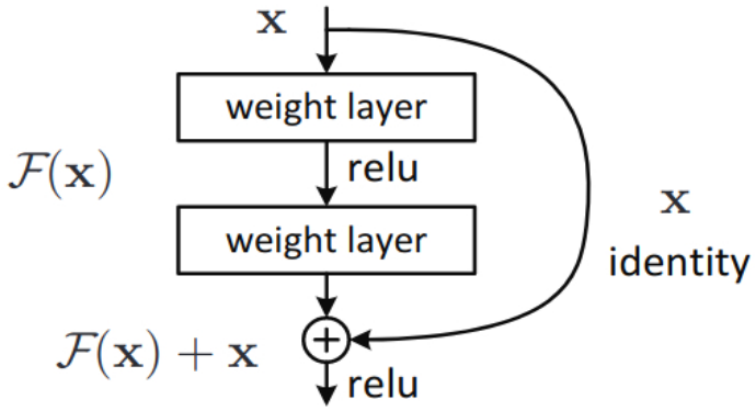
Ranije smo spomenuli problem eksplodirajućih i nestajućih gradijenata. Uzrok tome je dubina mreže. Genijalno rješenje tog problema predstavljeno je s **ResNet**

arhitekturom [6], predstavljenom na ImageNet natjecanju 2015. godine. Osvojila je prva mjesta za detekciju, lokalizaciju i segmentaciju na ovom i drugim natjecanjima.

Rješenje navedenog problema se sastoji od rezidualnog bloka, predstavljenog na slici 4.4. Ideja je podatke provući kroz sloj istovjetne konvolucije s ReLU funkcijom, kako bi se očuvala dimenzija. Zatim kroz još sloj jedan istovjetne konvolucije te, prije ReLU aktivacije, pridodati ulaz bloka. Time dobivamo:

$$relu(x + F(x)), \tag{4.1}$$

gdje su s $F(x)$ označena dva sloja istovjetnih konvolucija s ReLU aktivacijom između njih. Prednost ovog bloka je što učeći težine, zapravo uči težine funkcije F, koja predstavlja reziduale po kojima je mreža dobila naziv. Stoga su, u najgorem slučaju, težine jednake 0, što znači da se kroz blok propagira identiteta.



Slika 4.4: Rezidualni blok

Vidimo dva sloja težina koji su za ResNet slojevi istovjetne konvolucije. Učeći težine, učimo funkciju F. Preuzeto iz [6].

Pogledajmo sada primjer korištenja ovog bloka 4.5. Običnu konvolucijsku mrežu s 34 sloja bi bilo jako teško za naučiti iz prethodno navedenih razloga. Dodavanjem veza preskoka dobivamo ResNet s 34 sloja. Ona se lako može učiti pomoću SGD-a te se pokazala dobrom za generalizaciju. Ovakav tip mreža je dobar i za druge probleme osim računalnog vida.

Mreža koja se koristila za natjecanje imala je stotinjak slojeva, a testirali su i mreže s više od 1000. Rezultat koji su postigli na podacima ImageNet-a s ResNet-152, za TOP-1 grešku je 21,43%, a za TOP-5 5,71%. Kako se radi o mjerenju greške, tada je cilj imati što manji postotak pogrešno klasificiranih. S druge strane, VGG16 je postigao 28,07% te 9,33% za TOP-5.

Slika 4.5: Usporedba ResNet-a s VGG-om

Lijevo se nalazi prikaz VGG19, a u sredini neuronska mreža s 34 *sloja. Kako bismo omogućili učenje te mreže, potrebno ju je podijeliti na blokove. Desna neuronska mreža je ResNet s* 34 *sloja te vidimo veze preskoka, odnosno podjelu na blokove.* Preuzeto iz [6].

Ostale konvolucijske mreže i primjene

Za detekciju objekata, među najpoznatijima su dvije mreže, R-CNN i YOLO. **R-CNN** za klasificiranje i lokalizaciju koristi pomalo naivan pristup. Sastoji se od toga da prolazi po slici i traži 2000 ključnih regija. Birane segmente slike provlači kroz neuronsku mrežu i na taj način klasificira sliku. Time se određuje klasa i dio slike. Nadogradnje tog algoritma su *Fast* R-CNN i *Faster* R-CNN.

YOLO je skraćenica od *you only look once*. Preneseno, ta mreža gleda samo jedanput. Sliku podijeli na $n \times n$ pravokutnika i u svakom se pravokutniku može ili ne mora nalaziti objekt ili više njih. Dimenzija izlaza je također $n \times n$. Ovisno koliko objekata želimo da je moguće detektirati u svakom od pravokutnika, toliko više puta ćemo imati kanala u izlazu. Ako tražimo samo jedan objekt po pravokutniku, tada će dimenzija izlaza biti $n \times n \times (5 + k)$, gdje je k broj mogućih klasa. Naime, 5 je kanala potrebno za 5 varijabli. To su: vjerojatnost p da se nalazi objekt, x-koordinata centra objekta, y-koordinata centra objekta, visina objekta h te širina objekta w. Ako je vraćena visoka vjerojatnost p, tada zadnjih k vrijednosti određuje vjerojatnosti da je detektirana pojedina klasa.

Neki od poznatih primjera koje je moguće riješiti konvolucijskom mrežom su prepoznavanje lica i verifikacija. Verifikacijom želimo pomoću fotografije i neke identifikacijske iskaznice potvrditi da su dvije fotografije od iste osobe. Prepoznavanjem lica želimo pomoću jedne fotografije odrediti postoji li ta osoba u bazi podataka. Drugi problem je teži, jer se radi o $1 : n$ problemu, dok se u prvom radi o $1 : 1$.

Oponašanje stila (engl. *style transfer*) je još jedan primjer koji je započeo kao dio umjetnosti. Ako želimo našu fotografiju obogatiti na način da izgleda kao da ju je naslikao Vincent van Gogh, Pablo Picasso ili možda Rembrandt, tada je to moguće uz duboke konvolucijske mreže. Prvo, uzmemo upotpunosti zašumiranu sliku, a za stil želimo da zašumirana slika daje podjednake vrijednosti u dubokim slojevima kao i Picassova slika. Da bi zapravo bila kao naša slika, potrebno je da u ranijim slojevima poprima vrijednosti kao izvorna fotografija. Sada iteriranjima pomoću gradijenata mijenjamo piksele na zašumiranoj slici, tako da budu zadovoljeni prethodni kriteriji. S vremenom bismo trebali dobiti našu fotografiju s Picassovim stilom.

Za kraj, prethodno navedeni modeli su od važnosti zbog mogućnosti učenja prijenosom (engl. **transfer learning**). Ako imamo neku neuronsku mrežu koja jako dobro radi nad vrstom podataka koja je na neki način slična našim potrebama, tada možemo zamrznuti prethodne slojeve te učiti zadnjih nekoliko slojeva s našim primjerima. Ti slojevi su najčešće potpuno povezani slojevi. Npr. možemo uzeti ResNet-100, koji je prethodno naučen na ImageNet podacima i učiti ga prepoznavanju malignosti madeža.

4.2 Mreže s povratnim vezama

Nizove bismo mogli obrađivati pomoću konvolucija ili potpuno povezanim slojevima. Tada bi značajke bile dijeljene sa susjednim elementima, odnosno kod potpuno povezanih slojeva ne bi bile dijeljene. Zato je mreža s povratnim vezama puno pogodnija za njihovu obradu.

Ranije smo spomenuli neuronske mreže s povratnim vezama. Tada smo opisali jednostavnu arhitekturu modela RNN, koji koristi *tanh* aktivacijsku funkciju. Ona je **usko grlo**, jer kod nizova veće duljine raste dubina propagiranja informacija. Tada dolazi do nestajućih gradijenata. Dvije arhitekture koje dobro rješavaju taj problem su **GRU** te **LSTM**, koja je osmišljena ranije. Koristimo ih za:

- obradu prirodnog jezika (engl. *natural language processing* — **NLP**),
- digitalnu obradu govora (engl. *speech recognition*),
- DNA analizu,
- burzovne predikcije,
- generiranje teksta,
- generiranje glazbe,
- prepoznavanje kretnji iz videa (engl. *video motion recognition*).

Kada govorimo o obradi niza, implementacije neuronskih mreža možemo podijeliti s obzirom na oblik ulaznih i izlaznih podataka. Tada imamo:

- **mnogo**–naprama–**mnogo** — dvije su opcije $n : n$ i $n : m$. Prva je jednostavnija, jer je za drugu najčešće potrebno kodiranje. Tada imamo enkoder i dekoder niza.
- **mnogo**–naprama–**jedan** — ovakve arhitekture su učestale. Primjeri su burzovne predikcije, predikcije ocjene korisnika na temelju teksta, predikcije iduće riječi, itd.
- **jedan**–naprama–**mnogo** — rijedak slučaj. Primjer je generiranje rečenice iz početne riječi i slično.
- **jedan**–naprama–**jedan** — osnovna neuronska mreža.

Napomena 4.2.1. Prije nego opišemo navedene arhitekture definirajmo neke oznake. Ranije smo spomenuli kako se skup podataka sastoji od primjera. U ovom slučaju, primjeri su nizovi. S $x_1, x_2, \ldots, x_t$ označavamo elemente niza, odnosno primjera x. Svaki od elemenata može biti vektor. U tom slučaju, svi su jednakih dimenzija, tj. $x_t \in \mathbb{R}^n$, za svaki t. Radi lakše čitljivosti, ne koristimo standardnu oznaku za vektor kao $\vec{x}$. Analogno ćemo indeksirati varijable i slojeve. U slučaju da varijable ne sadrže indekse, znači da se radi o privremenoj varijabli koja se ne prenosi kroz iteracije.

Napomena 4.2.2. Uvest ćemo oznaku za konkatenaciju vektora po recima. Neka su $x \in \mathbb{R}^n$ i $y \in \mathbb{R}^k$ vektori, uvažavajući prethodnu napomenu. Tada definiramo konkatenirani vektor s $[x;\ y] \in \mathbb{R}^{(n+k)}$.

Dodatno, sa $*$ ćemo označavati vektorsko množenje po elementima, a sa σ ćemo jednostavnije označavati *sigmoidalnu* funkciju. Primijetimo da je ona svojim najvećim dijelom približno jednaka 1 ili 0, što ćemo koristiti kao vrata.

GRU

Ova arhitektura sadrži tzv. vrata, po kojima je dobila naziv **povratna jedinica s vratima** (engl. *gated recurrent unit*). Vrata ćemo označavati s G, a svrha im je upravljanje protokom informacija. Ova arhitektura sadrži dvoja vrata G_u i G_r. Također, sadrži i tzv. memorijsku ćeliju, koju ćemo označavati s c_t. Uloga memorijske ćelije je da prenosi informacije o prethodnim elementima niza. Vrata G_u služe obnovi ćelija. Ako su vrata jednaka 1, ćelija će biti obnovljena s novim informacijama, a ako su jednaka 0, ćelija će ostati ista. Vrata G_r služe za određivanje relevantnosti pojedinih elemenata vektora. Pogledajmo sada izračun jedne iteracije t:

$$G_u = \sigma(W_u[c_{t-1};\ x_t] + b_u), \tag{4.2}$$

$$G_r = \sigma(W_r[c_{t-1};\ x_t] + b_r), \tag{4.3}$$

$$C = \sigma(W_c[G_r * c_{t-1};\ x_t] + b_c), \tag{4.4}$$

$$c_t = G_u * C + (1 - G_u) * c_{t-1}, \tag{4.5}$$

$$\hat{y}_t = softmax(c_t). \tag{4.6}$$

Prokomentirajmo jednakosti. U prvoj jednakosti (4.2) izračunavamo vrata obnove za cijeli vektor. Zatim vrata za relevantnost, s kojima ćemo izračunati potencijalno novu vrijednost C. Ako su vrata obnove G_u za pojedine elemente jednaka 1, c_t će poprimiti novu vrijednost, a za 0 ostaje ista. Na kraju, u (4.6) računamo izlaznu vrijednost iteracije. Napomenimo da se u prvoj iteraciji c_0 inicijalizira nulama.

LSTM

Sada ćemo opisati malo složeniju arhitekturu. Naziv je skraćenica od engleskog *Long-short term memory*. Ovdje se u računanju idućeg elementa prosljeđuju dvije vrijednosti c_t i a_t, umjesto jedne. LSTM sadrži troja vrata: G_u skaliraju utjecaj novo izračunate ćelije C, G_f određuju hoće li prethodno izračunati c_{t-1} zaboraviti dio informacija, a vrata G_o određuju koje će informacije biti proslijeđene idućem sloju kao varijabla a_t. Za kraj, prosljeđujemo prema izlazu informaciju $\hat{y}_t$. Opis LSTM jedinice dan je s:

$$G_u = \sigma(W_u[a_{t-1};\ x_t] + b_u), \tag{4.7}$$

$$G_f = \sigma(W_f[a_{t-1};\ x_t] + b_f), \tag{4.8}$$

$$G_o = \sigma(W_o[a_{t-1};\ x_t] + b_o), \tag{4.9}$$

$$C = \sigma(W_c[a_{t-1};\ x_t] + b_c), \tag{4.10}$$

$$c_t = G_u * C + G_f * c_{t-1}, \tag{4.11}$$

$$a_t = G_o * tanh(c_t) \tag{4.12}$$

$$\hat{y}_t = softmax(a_t). \tag{4.13}$$

Za kraj, iako je LSTM osmišljen dosta ranije od GRU-a, obje arhitekture su i danas podjednako korištene. GRU je jednostavniji i time brži, no LSTM ima veću snagu za kompleksnije probleme. Nema konsenzusa koji je model bolji za koju namjenu. LSTM se povijesno dokazao. Korišten je za NLP i digitalnu obradu govora kao dvosmjerni model.

Dodatno, navedene arhitekture možemo kombinirati s konvolucijama, potpuno povezanim slojevima ili višestrukom nadogradnjom njih samih. Tada dobivamo duboke neuronske mreže s povratnim vezama.

Iako smo vidjeli da ResNet može imati i stotinjak slojeva, ovdje se to ne preporučuje. Ove arhitekture sporije uče te za sada nije zabilježena uspješnost s dubinama 10 i više. Najčešće implementacije imaju oko 3 sloja.

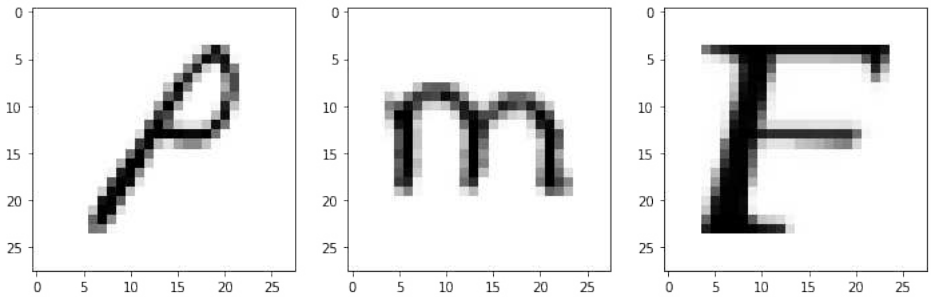
Poglavlje 5

Rješenje problema: klasifikacija slika

U ovom poglavlju ćemo opisati izradu klasifikacijskog modela slika. Podaci su preuzeti s internetske stranice Kaggle [14]. Kaggle sadrži razne podatke dostupne javnosti. Često su organizirana natjecanja u obradi podataka. U sklopu toga objavljuju se kodovi s rješenjima i naučeni modeli. Za okruženje je korišten *Google colaboratory*, koji je besplatno okruženje *Jupyter notebook*-a u oblaku.

5.1 O podacima i cilju

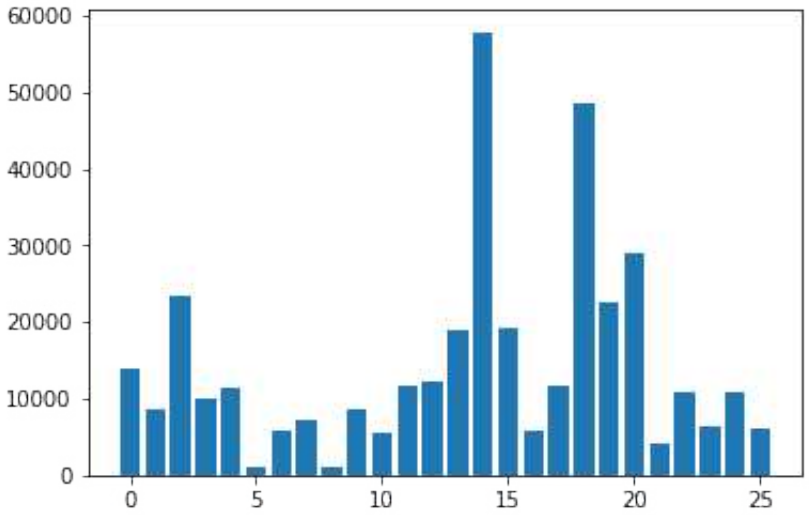
Preuzeti podaci sadrže 372 450 slika dimenzija $28 \times 28 \times 1$ te pripadajuće oznake. Na slikama se nalaze većinom rukom pisana slova. Neki primjeri, kao što možemo vidjeti na slici 5.1, sadrže strojno tiskana slova. Slova su iz engleske abecede, što znači da imamo 26 klasa. Klase su označene brojevima iz skupa $\{0, 1, \ldots, 25\}$.



Slika 5.1: Prikaz podataka

Ovaj skup podataka je namijenjen za problem klasifikacije. Na navedenoj stranici [14] je moguće pronaći javno dostupna rješenja te gotove modele s naučenim težinama. Tako postoje arhitekture čija točnost dostiže 99,82% te 98,89%, nakon samo 10 epoha učenja.

Pogledajmo distribuciju podataka po klasama na histogramu, slika 5.2. Vidimo da neke klase sadrže više, a neke manje podataka. Zato je moguće da model nauči samo klase koje se pojavljuju često te tako postigne visoku točnost.

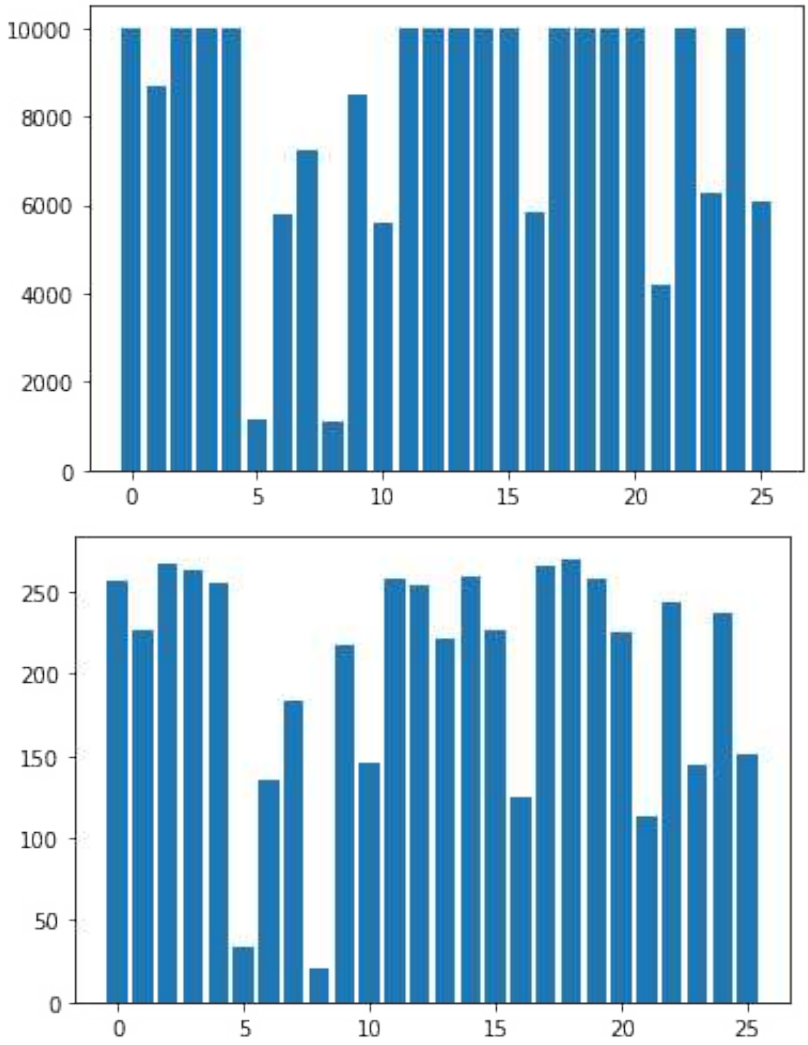


Slika 5.2: Histogram prikazuje distribuciju preuzetih podataka prema klasama

S obzirom da skup sadrži značajnu količinu podataka, da distribucija po klasama nije jednolika te da su javno objavljeni modeli postigli visoku točnost, cilj ovog rada nije postizanje bolje točnosti i uspoređivanje s drugima. **Cilj** je pokazati da dublja mreža, s dodavanjem slojeva može imati bolju predikciju.

Kako bi sve klase bile ravnopravne, da ne bi bilo prejednostavno učiti te da epohe prilikom učenja kraće traju, potrebno je reducirati skup podataka. Prvo su uzeti svi podaci čije klase sadrže manje od 10 000 primjera, a zatim je u svakoj od preostalih klasa birano 10 000 slučajnih primjera. Tako smo stvorili skup s 210 369 podataka, čija je distribucija vidljiva na slici 5.3. S obzirom na visinu točnosti spomenutih modela i smanjen broj podataka, kao **dodatni cilj** je određena točnost od 97%.

Prije početka modeliranja, potrebno je podijeliti podatke na trening, validacijske i test podatke. Određeno je 95% za učenje te po 2,5% za validaciju i testiranje. Kako su histogrami za validaciju i testiranje podjednaki, ovdje prikazujemo samo validacijski, slika 5.3. Sve podatke je potrebno standardizirati. To je učinjeno dijeljenjem vrijednosti piksela s 255. Na taj način modelu olakšavamo učenje.



Slika 5.3: Histogrami prikazuju distribuciju podataka

(gore) *Distribucija odabranih podataka iz skupa preuzetih podataka.* ***(dolje)*** *Distribucija validacijskih podataka. Podjednako izgleda distribucija testnih podataka.*

5.2 Izrada modela i učenje

Nakon što smo sakupili i pretprocesirali podatke te zadali cilj, možemo započeti s modeliranjem. Koristimo programski jezik Python. Da bismo mogli modelirati potrebno je bilo odabrati biblioteku za implementaciju neuronskih mreža. Odabran je

Tensorflow [1] koji sadrži Keras-ov API [3] te ima mogućnost učenja pomoću GPU-a. Pri izboru važno je bilo da biblioteka bude otvorenog koda s aktivnom zajednicom, jednostavnost implementacije neuronskih mreža te brzina izvođenja.

Keras ima dva načina izrade modela: **funkcijski** i **sekvencijalni**. Iako je sekvencijalni jednostavniji, jer se slojevi nadograđuju nizanjem, korišten je funkcijski. Funkcijski omogućuje spremanje sloja u varijablu. Time olakšavamo izgradnju kompliciranih mreža gdje se slojevi isprepliću, što je pogodno za konstrukciju blokova kao što su *Inception* blok te rezidualni blok, koji je korišten za ResNet.

Jednostavan model

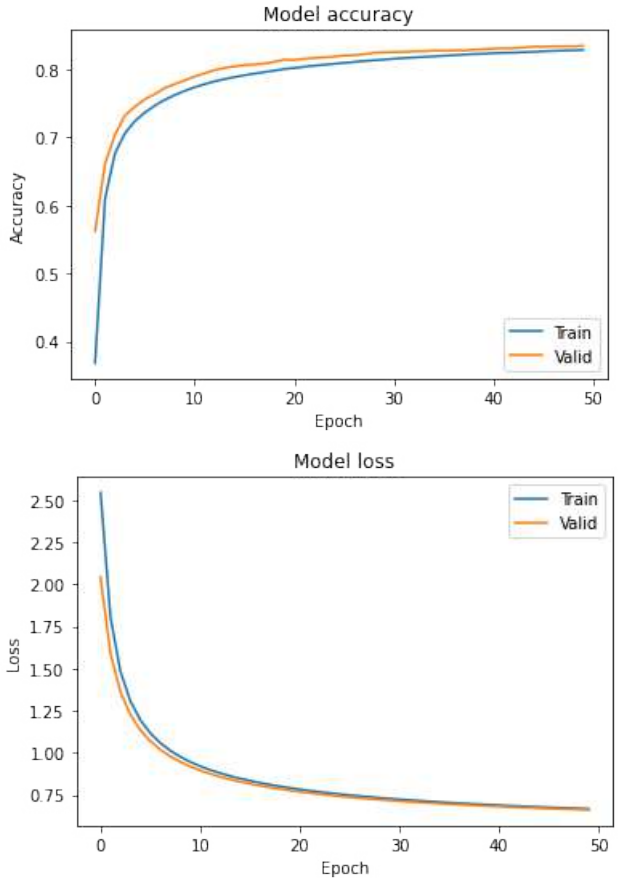
Pri izradi modela dobro je, za početak, izraditi jednostavan model. Arhitekturu prve mreže možemo vidjeti na slici 5.4. Keras zahtijeva da zadamo ulazni oblik, nakon čega se spljošte dimenzije na veličinu 784. Jedini sloj s težinama je potpuno povezan sloj koji sadrži 26 neurona. *Softmax* aktivira prethodne neurone te daje konačan rezultat. Na slici 5.4 možemo vidjeti da model ima 20 410 parametara koje je potrebno naučiti.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 26)	20410
activation (Activation)	(None, 26)	0

Total params: 20,410
Trainable params: 20,410
Non-trainable params: 0

Slika 5.4: Slika prikazuje tablicu sa slojevima prvog modela

Kod Kerasa, model je zapravo objekt nad kojim pozivamo metode. Metoda *compile* kao argumente prima optimizacijski algoritam, vrstu funkcije gubitka, mjere uspješnosti ili greške i dr. Izabrali smo gradijentni spust s grupama, *categorical cross-entropy* kao funkciju gubitka te točnost kao mjeru uspješnosti. Metoda *summary* daje tablični prikaz modela, a metoda *fit* prima potrebne podatke za učenje i validaciju, broj epoha, veličinu grupa i ostale argumente. Ona pokreće učenje modela. Određeno je 50 epoha s 512 primjera po grupi. Učenje je trajalo ukupno 100 sekundi.



Slika 5.5: Slika prikazuje grafove točnosti i troška kroz epohe

(gore) *Za trening podatke točnost započinje s* 0,3686*, u drugoj epohi se penje na* 0,61 *te nastavlja rasti do zadnje epohe s* 0,8284*. Za validacijski skup je prvo mjerenje jednako* 0,5622*, u četvrtoj epohi dolazi do* 0,7321 *te nastavlja rasti do* 50*-te epohe.* ***(dolje)*** *Za trening, trošak započinje s* 2,5441 *te konvergira do* 0,6676*. Validacijski skup započinje s* 2,0428 *te u* 50*. epohi poprima vrijednost* 0,6616*.*

Dvije metode koje su nam također od značaja su *evaluate* i *predict*. Prvom možemo izmjeriti trošak i ostale mjere koje smo zadali pri pozivu metode *compile*, a druga računa predikcije primjera nekog skupa. Na testnom skupu model je postigao trošak od **0,6679** te točnost od **0,8260**. Drugim riječima, 82,6% je točno prediktiranih primjera.

Model s jednim konvolucijskim slojem

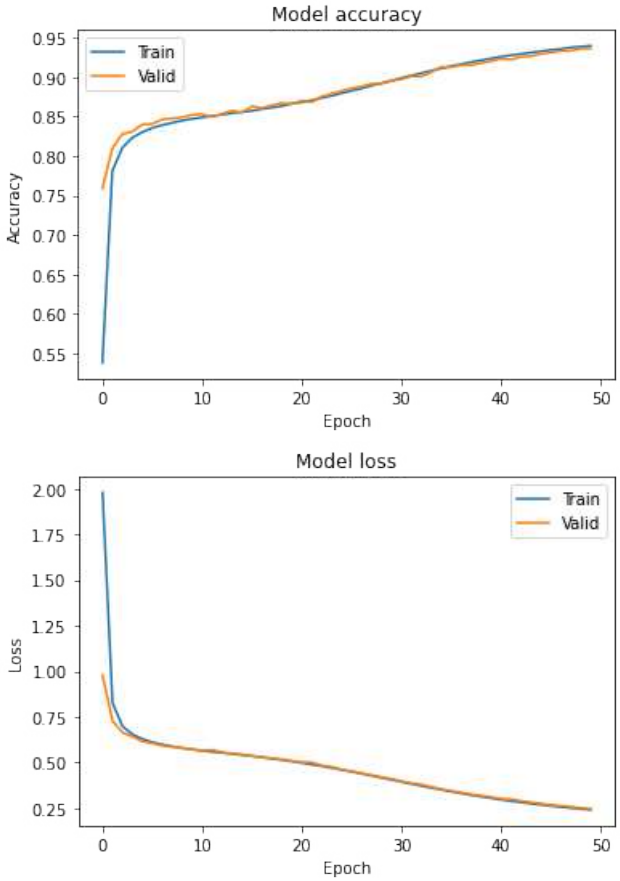
Nakon što smo napravili jednostavan model, želimo vidjeti kako će utjecati dodavanje konvolucijskog sloja. Pogledajmo sliku 5.6. Promjene koje uočavamo su dodavanje konvolucije kao prvog parametriziranog sloja. On sadrži 32 filtera dimenzija 3×3 s korakom 1. Dodano je dopunjenje, tako da dimenzija ostane ista. Aktivacija je ReLU, a nju slijedi sažimanje maksimuma dimenzija 2×2 s korakom 2.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
activation_2 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 26)                163098
_________________________________________________________________
activation_3 (Activation)    (None, 26)                0
=================================================================
Total params: 163,418
Trainable params: 163,418
Non-trainable params: 0
```

Slika 5.6: Slika prikazuje tablicu sa slojevima drugog modela

Kraj je ostao isti kao i u prethodnom modelu. Također je ista konfiguracija algoritma učenja te mjera uspješnosti i funkcije troška. Proces samog učenja s 50 epoha možemo vidjeti na slici 5.7. Vidimo da točnost raste, a trošak pada. Vjerojatno je bilo još prostora za učenje, jer ne vidimo zaravnavanje.

Skup podataka za testiranje je postigao trošak od **0,2340** te **0,9403** za točnost. Napomenimo kako je učenje trajalo 250 sekundi, 5 sekundi po epohi.



Slika 5.7: Slika prikazuje grafove točnosti i troška kroz epohe za drugi model

(gore) *Za trening podatke točnost započinje s* 0,5376*, nastavlja rasti do zadnje epohe s* 0,9398*. Za validacijski skup je prvo mjerenje jednako* 0,7587*, nastavlja rasti do* 50*-te epohe, gdje poprima* 0,9361*.* ***(dolje)*** *Za trening, trošak započinje s* 1,9778 *te konvergira do* 0,2414*. Validacijski skup započinje s* 0,9786 *te u* 50*. epohi poprima vrijednost* 0,2476*.*

Tri konvolucijska sloja

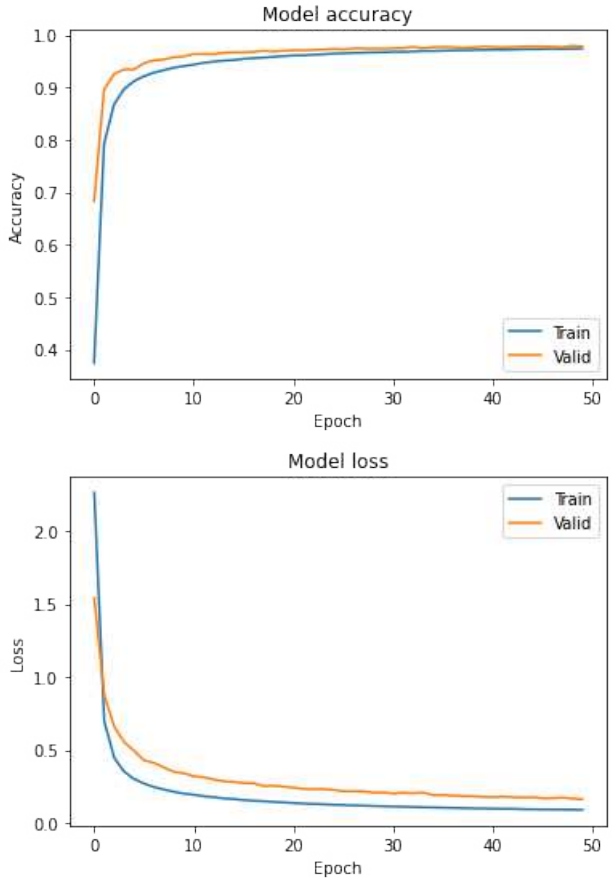
Za razliku od prethodnog, ovaj model ima 3 konvolucijska sloja. Broj filtera raste prema dubini. Prvi sadrži 32, drugi 64, a treći sloj 128 filtera, po uzoru na VGG16. Prije sažimanja maksimuma, dodan je *dropout* sloj s parametrom jednakim 0,3. To znači da nasumično deaktivira 30% neurona prethodnog sloja. Time se poboljšava generalizacija. Ukupan broj parametara je 122 650. Učenje je trajalo 50 epoha po 5 sekundi, što je zapravo 250 sekundi, jednako koliko i prethodni model.

Na testnim podacima postiže točnost jednaku **0,9829**, odnosno 98,29%. Time smo uspjeli nadmašiti ciljanih 97% s gotovo upola manje podataka. Trošak je jednak **0,1588**.

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_3 (Conv2D)	(None, 28, 28, 32)	320
activation_4 (Activation)	(None, 28, 28, 32)	0
dropout (Dropout)	(None, 28, 28, 32)	0
max_pooling2d_2 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_4 (Conv2D)	(None, 14, 14, 64)	18496
activation_5 (Activation)	(None, 14, 14, 64)	0
dropout_1 (Dropout)	(None, 14, 14, 64)	0
max_pooling2d_3 (MaxPooling2	(None, 7, 7, 64)	0
conv2d_5 (Conv2D)	(None, 7, 7, 128)	73856
activation_6 (Activation)	(None, 7, 7, 128)	0
dropout_2 (Dropout)	(None, 7, 7, 128)	0
max_pooling2d_4 (MaxPooling2	(None, 3, 3, 128)	0
conv2d_6 (Conv2D)	(None, 1, 1, 26)	29978
flatten_2 (Flatten)	(None, 26)	0
activation_7 (Activation)	(None, 26)	0

Total params: 122,650
Trainable params: 122,650
Non-trainable params: 0

Slika 5.8: Slika prikazuje tablicu sa slojevima trećeg modela



Slika 5.9: Slika prikazuje grafove točnosti i troška kroz epohe za treći model

(gore) *Za trening podatke točnost započinje s* 0,5376*, nastavlja rasti do zadnje epohe s* 0,9398*. Za validacijski skup je prvo mjerenje jednako* 0,7587*, nastavlja rasti do* 50*-te epohe, gdje poprima* 0,9361*.* ***(dolje)*** *Za trening, trošak započinje s* 1,9778 *te konvergira do* 0,2414*. Validacijski skup započinje s* 0,9786 *te u* 50*. epohi poprima vrijednost* 0,2476*.*

Model nalik ResNet-u

Ova neuronska mreža je inspirirana ResNet-om. Možemo gledati na nju kao da se sastoji od 3 bloka. Taj blok možemo vidjeti na slici 5.10. Ako pogledamo bolje, blok možemo podijeliti na dva podbloka. Svaki podblok je jedan rezidualni sloj. Svaka konvolucijska mreža ima 64 filtera te ne djeluje na veličinu dimenzija, ostavlja ih istima. Prvi rezidualni podblok je do prve aktivacijske funkcije. On ne sadrži *dropout* ni sažimanje.

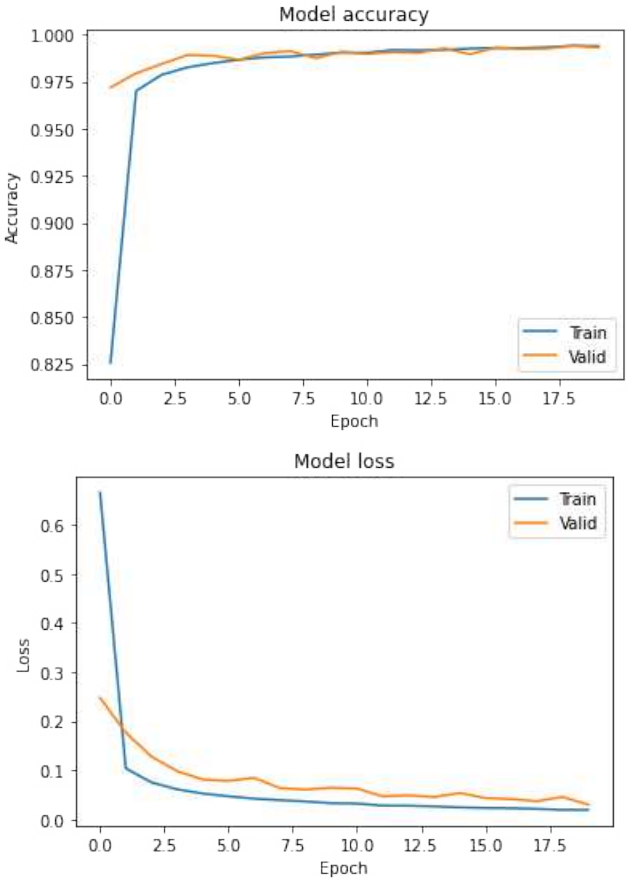
Idući rezidualni podblok sadrži dva uzastopna konvolucijska sloja, jednaka prethodnim. Prvi sloj implicitno sadrži ReLU aktivacijsku funkciju, dok drugi ne sadrži aktivaciju. Nakon zbrajanja s prethodnim podblokom slijedi *dropout* s 30% učinkom te sažimanje maksimuma dimenzija 2×2 s korakom 2. Primijetimo da sažimanje smanjuje dimenzije, dok ih konvolucija ostavlja istima.

conv2d_20 (Conv2D)	(None, 14, 14, 64)	36928	max_pooling2d_12[0][0]
conv2d_21 (Conv2D)	(None, 14, 14, 64)	36928	conv2d_20[0][0]
add_2 (Add)	(None, 14, 14, 64)	0	conv2d_21[0][0] max_pooling2d_12[0][0]
activation_18 (Activation)	(None, 14, 14, 64)	0	add_2[0][0]
conv2d_22 (Conv2D)	(None, 14, 14, 64)	36928	activation_18[0][0]
conv2d_23 (Conv2D)	(None, 14, 14, 64)	36928	conv2d_22[0][0]
add_3 (Add)	(None, 14, 14, 64)	0	conv2d_23[0][0] activation_18[0][0]
activation_19 (Activation)	(None, 14, 14, 64)	0	add_3[0][0]
dropout_13 (Dropout)	(None, 14, 14, 64)	0	activation_19[0][0]
max_pooling2d_13 (MaxPooling2D)	(None, 7, 7, 64)	0	dropout_13[0][0]

Slika 5.10: Slika prikazuje tablicu sa slojevima od kojih se sastoji jedan blok s dva rezidualna podbloka

Postavke za učenje su jednake prethodnim modelima, s dvije iznimke. Za optimizacijski algoritam izabran je **Adam** te je zadano 20, umjesto 50 epoha. Povijest učenja se može vidjeti na slici 5.11.

Za kraj, recimo kako je trošak na testnim podacima jednak **0,0296**, a točnost **0,9932** čime su nadmašena očekivanja. Učenje je trajalo 20 epoha po 25 sekundi.



Slika 5.11: Slika prikazuje grafove točnosti i troška kroz epohe za četvrti model

(gore) *Trening podaci su na početku imali točnost jednaku* 0,8255*, a završili su s* 0,9939*. Validacija je započela s* 0,9719*, a završila s* 0,9932*.* ***(dolje)*** *Za trening, trošak započinje s* 0,6650 *te konvergira do* 0,0191*. Validacijski skup započinje s* 0,2472 *te u* 20*. epohi poprima vrijednost* 0,0297*.*

5.3 Zaključak

Cilj ovog poglavlja bio je na jednostavnom primjeru klasifikacije slika s rukom pisanim slovima, pokazati da se s dubljom neuronskom mrežom postižu bolji rezultati. Drugi cilj bio je postići 97% točnost na testnim podacima. To je bilo ostvareno modelom s tri konvolucijska sloja. Postignuta je točnost 98,29%. Na iznanađenje, četvrti model je postigao visokih 99,32%.

Vratimo se sada na prvotni cilj. Pogledajmo tablicu 5.1. Možemo primijetiti da s većim brojem slojeva trošak pada. On govori koliko je naš model pouzdan. Ako imamo dva modela s istom točnošću, ali dva različita troška, tada je onaj s manjim troškom bolji. Također, pogledamo li točnosti, vidjet ćemo da rastu s veličinom modela — što je bio i cilj pokazati. Treba napomenuti da to ne znači da takvo pravilo uvijek vrijedi. Da smo napravili model s iznimno visokim kapacitetom, lagano bi se moglo dogoditi da model prenauči. Tada bi točnost na testnim podacima pala.

Grafovi točnosti i troška su pokazivali trend. Da smo nastavili trenirati još epoha vjerojatno bismo dobili bolje rezultate. No, ako predugo treniramo, teorija kaže da se također može dogoditi prenaučenost.

Vrsta modela	**Trošak** / *Loss*	**Točnost** / *Accuracy*
Jednostavni	0,6679	0,8260
S jednom konvolucijom	0,2340	0,9403
S tri konvolucije	0,1588	0,9829
Nalik ResNet-u	0,0296	0,9932

Tablica 5.1: Tablica s troškom i točnošću izračunatih nad testnim podacima.

Za kraj, trebamo biti zadovoljni rezultatima, jer je izrađen model koji ima zadovoljavajuću točnost. Inače, gotovo je nemoguće postići model s greškom jednakom nula. Prvi razlog je što postoji Bayesova greška koja definira teorijsku granicu minimalne greške, čak i ako raspolažemo s beskonačno podataka s točnom distribucijom. Do toga dolazi jer značajke ne sadrže sve potrebne informacije kojima bismo opisali zavisnu varijablu. Druga mogućnost je da podaci sadrže šum ili što imamo konačno mnogo podataka. Također je moguć nesrazmjer podataka, kao što smo imali u primjeru s madežima, velika količina benignih, a mala malignih.

Bibliografija

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu i Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015, `http://tensorflow.org/`, Software available from tensorflow.org.

[2] M. Bošnjak, *Neuronske mreže*, `https://web.math.pmf.unizg.hr/nastava/su/index.php/download_file/-/view/166/`, posjećeno 17. 2. 2020.

[3] François Chollet i ost., *Keras*, `https://keras.io`, 2015.

[4] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li i L. Fei-Fei, *ImageNet: A Large-Scale Hierarchical Image Database*, CVPR09, 2009.

[5] Ian Goodfellow, Yoshua Bengio i Aaron Courville, *Deep Learning*, MIT Press, 2016, `http://www.deeplearningbook.org`.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren i Jian Sun, *Deep Residual Learning for Image Recognition*, arXiv e-prints (2015), arXiv:1512.03385.

[7] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama i Trevor Darrell, *Caffe: Convolutional Architecture for Fast Feature Embedding*, arXiv preprint arXiv:1408.5093 (2014).

[8] Alex Krizhevsky, Ilya Sutskever i Geoffrey E Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in Neural Information Processing Systems 25 (F. Pereira, C. J. C. Burges, L. Bottou i K. Q. Weinberger, ur.), Curran Associates, Inc., 2012, str. 1097–1105,

http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio i Patrick Haffner, *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE, sv. 86, 1998, str. 2278–2324, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

[10] Min Lin, Qiang Chen i Shuicheng Yan, *Network In Network*, arXiv e-prints (2013), arXiv:1312.4400.

[11] T. Lipić, *PMF Machine Learning Introduction to Deep Learning*, https://www.dropbox.com/s/3r0o1ergwk342f1/SU_PMF_DL_Basics_2020_hq.pdf?dl=0, posjećeno 19. 6. 2020.

[12] A. Ng, *Deep Learning Specialization*, https://www.coursera.org/specializations/deep-learning, posjećeno 24. 6. 2020.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai i Soumith Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, Advances in Neural Information Processing Systems 32 (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox i R. Garnett, ur.), Curran Associates, Inc., 2019, str. 8024–8035, http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[14] S. Patel, *A-Z Handwritten Alphabets in .csv format*, https://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format, posjećeno 17. 2. 2020.

[15] Sebastian Ruder, *An overview of gradient descent optimization algorithms*, arXiv e-prints (2016), arXiv:1609.04747.

[16] Karen Simonyan i Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv e-prints (2014), arXiv:1409.1556.

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke i Andrew Rabinovich, *Going Deeper with Convolutions*, arXiv e-prints (2014), arXiv:1409.4842.

Sažetak

Duboke neuronske mreže su sve popularnije zahvaljujući velikim količinama podataka, većoj snazi računala te zajednici okruženoj oko softverskih alata. Koriste se za obradu računalnog vida, prirodnog jezika, tekstova te raznih predikcija.

U ovom radu najprije opisujemo općeniti pojam neuronske mreže te neophodne segmente za učenje modela, a to su: podaci, mjere uspješnosti, arhitektura modela i algoritmi učenja. Dalje se uvode i analiziraju optimizacijski algoritmi, mrežne arhitekture i njihove primjene. Na kraju se pokazuje važnost dubine neuronskih mreža, rješenjem problema klasifikacije slika rukom pisanih slova.

Summary

Deep neural networks are becoming popular because of a large amount of existing data, increased computing power as well as the community created around the software. Networks are used for computer vision, natural language processing and various predictions.

This thesis, firstly describes the general concept of a neural network and the necessary segments for model learning, of which there are four: data, performance measures, model architecture and learning algorithms. Optimization algorithms, network architectures and their applications are further introduced and analyzed. Finally, the importance of the depth is shown by solving the problem of classifying images of handwritten characters.

Životopis

Rođen sam 1995. godine u Zagrebu. Od tada sam završio Osnovnu školu Dragutina Domjanića te Gimnaziju Lucijana Vranjanina, prirodoslovno-matematički smjer. 2014. godine sam upisao preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Nakon što sam postao prvostupnik matematike 2017. godine, upisujem diplomski studij računarstva i matematike. Kao student, dodatno iskustvo stjecao sam radeći u Zagrebačkoj banci te Gideon Brothers-u. Stekavši sve preduvjete, 2019. godine sam upisao diplomski rad s temom Duboke neuronske mreže.