SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1160

Algoritmi za brzo učenje na neprijateljskim primjerima

Dominik Jambrović

Zagreb, svibanj 2023.

Zahvaljujem svojoj obitelji radi podrške tijekom studiranja, kao i prof. dr. sc. Siniši Šegviću te mag. ing. Ivanu Grubišiću na pomoći tijekom izrade završnog rada.

SADRŽAJ

1. Uvod

Velik broj problema s kojima se danas susrećemo takve su prirode da ne znamo kako ih riješiti koristeći klasičan, algoritamski pristup. Razlog tome često leži u činjenici da ne znamo ni kako mi sami rješavamo te probleme, a jedan od najčešćih primjera za to je raspoznavanje tj. klasifikacija slika. Jedno od mogućih rješenja takvih problema je korištenje umjetnih neuronskih mreža - mreža sastavljenih od velikog broja povezanih jedinica (neurona) koje obavljaju veoma jednostavne operacije.

Razvojem dubokih neuronskih mreža došlo je do ubrzanog napretka u području računalnog vida. Računalni vid područje je umjetne inteligencije koje se bavi problemima poput klasifikacije 2D slika. Sve većom popularizacijom i korištenjem dubokih modela u sustavima različitih namjena, u pitanje se dovodi sigurnost takvih modela - ako model želimo koristiti u automobilima s ciljem detekcije pješaka i vozila, model mora moći dobro generalizirati, kao i biti robustan. Takav model ne bi smio mijenjati svoje odluke na temelju veoma malih promjena na ulazu - na primjeru prometa, želimo da model točno detektira pješaka, bez obzira nosi li on kapu ili ne.

Kako bi ostvarili robusnost modela, predložene su brojne tehnike, a jedna od najpopularnijih je robusno učenje tj. učenje na neprijateljskim primjerima. Kada su u pitanju modeli koji brzo uče, robusno učenje prihvatljivo je rješenje za postizanje robusnih modela otpornih na napade. Ipak, kada su u pitanju veći modeli za koje učenje traje veoma dugo, obično robusno učenje često je neprihvatljivo. U tu svrhu, razvijene su metode koje ubrzavaju robusno učenje. U ovome radu, razmatrat ćemo tri takve metode: besplatno robusno učenje [14], brzo robusno učenje [16] i nadogradnju na brzo robusno učenje (FastAdv+ i FastAdvW, [8]).

Uz to, razmatrat ćemo i otpornost naučenih modela na zatrovane podatke. Zatrovani podatci ulazi su izmijenjeni s ciljem navođenja modela na neočekivano ponašanje. Uvođenjem takve ranjivosti u model, napadači mogu neprimijećeno postići proizvoljne ciljeve poput izbjegavanja detekcije ili pogrešne klasifikacije.

2. Neuronske mreže

2.1. Općenito o neuronskim mrežama

Umjetne neuronske mreže veoma su popularan alat za rješavanje kompleksnih problema za koje je teško modelirati ili formalizirati znanje. Predstavnik su konektivističkog pristupa umjetnoj inteligenciji [1] koji se zasniva na oblikovanju sustava inspiriranih građom mozga.

Problemi koje rješavamo umjetnim neuronskim mrežama svrstavaju se u dvije glavne kategorije:

1. klasifikacija
2. regresija

Kada su u pitanju klasifikacijski problemi, cilj nam je svrstati ulaz u jedan od mogućih razreda. Pritom je na izlazu često korišteno jednojedinično kodiranje - ako ulaze svrstavamo u 10 razreda, u izlaznom sloju mreže bit će 10 neurona, a aktivacija jednog od njih predstavljat će rezultat klasifikacije.

S druge strane, kod regresijskih problema cilj nam je što bolje aproksimirati neku, modelu nepoznatu, funkciju. Za ovakve probleme, često nam je dovoljan jedan neuron u izlaznom sloju. Taj neuron na izlazu bi trebao davati predviđenu vrijednost funkcije za neki, do sada neviđeni, ulaz.

Da bi neuronska mreža mogla rješavati takve probleme, važno nam je da može učiti na temelju predočenih podataka. Učenje neuronske mreže odvija se izmjenom težina pojedinih neurona (time znanje implicitno ugrađujemo u našu mrežu). Kako bismo detaljnije mogli govoriti o učenju i arhitekturama neuronskih mreža, važno je ukratko opisati neuron - osnovnu gradivnu jedinicu svake mreže.

2.2. Umjetni neuron

Umjetni neuroni predstavljaju jednostavne procesne jedinice koje modeliraju ponašanje prirodnih neurona. Osnovni neuron akumulira vrijednosti na ulazu pomnožene težinama, akumuliranoj vrijednosti dodaje pomak te na kraju istu propušta kroz prijenosnu (aktivacijsku) funkciju. Ponašanje jednog neurona možemo modelirati jednadžbom:

$$o = f(\sum_{i=1}^{n} x_i \cdot w_i + b) \tag{2.1}$$

pri čemu x označava pojedine ulaze, w težine na pripadnim ulazima, b pomak te f prijenosnu funkciju.

2.3. Prijenosne funkcije

Neki od najranijih modela umjetnog neurona kao prijenosnu funkciju koristili su funkciju identiteta (ADELINE-neuron) te funkciju skoka (TLU-perceptron). S vremenom su korištene i razvijene brojne druge prijenosne funkcije poput sigmoidalne funkcije definirane kao:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Sigmoidalna funkcija značajna je zbog svojstva derivabilnosti nad cijelom svojom domenom. Ovo svojstvo važno je za brojne optimizacijske postupke. Uz nju, danas je veoma značajna i zglobnica (ReLU - engl. *Rectified Linear Unit*) koju možemo prikazati na sljedeći način:

$$relu(x) = max(0, x) \tag{2.3}$$

Osim navedenih prijenosnih funkcija, korištena je i softmax funkcija koju jednadžbom možemo prikazati kao:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.4}$$

Softmax funkcija poopćenje je sigmoidalne funkcije, a često se koristi kao zadnja prijenosna funkcija u neuronskim mrežama korištenim za klasifikaciju. Korištenjem te funkcije u zadnjem sloju, na izlazu mreže imat ćemo vjerojatnosti klasifikacije u pojedini od razreda. Ovo svojstvo korisno nam je kada kao funkciju gubitka koristimo unakrsnu entropiju.

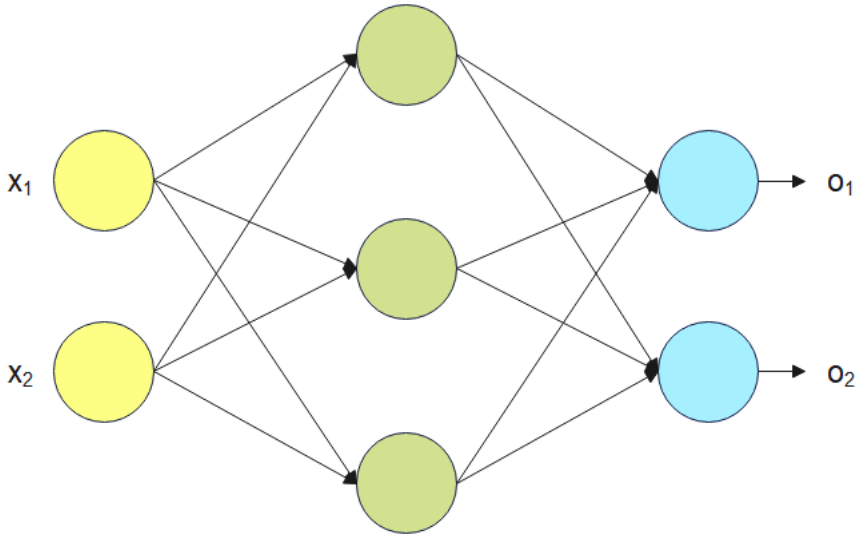
Važno je primijetiti da ako je prijenosna funkcija linearna, cijeli neuron može postići isključivo linearnu transformaciju. Kako bi umjetnim neuronima mogli modelirati kompleksnije funkcije, koristimo nelinearne prijenosne funkcije poput sigmoidalne funkcije i zglobnice. Pritom je za duboke neuronske mreže s velikim brojem slojeva često korištena upravo zglobnica - sigmoidalna funkcija za takve mreže nije prikladna zbog problema nestajućeg gradijenta (engl. *vanishing gradient problem*) koji nastaje tijekom učenja temeljenog na gradijentnim metodama.

2.4. Arhitektura umjetne neuronske mreže

Kada je za neki problem potrebno koristiti više od jednog osnovnog neurona, neurone povezujemo u mrežu. Pritom kažemo da se neuronska mreža sastoji od nekolicine slojeva:

1. ulazni sloj
2. skriveni slojevi
3. izlazni sloj

Iako ulazni sloj predstavljamo neuronima, oni, za razliku od neurona u skrivenim slojevima i izlaznom sloju, ne obavljaju nikakve transformacije - možemo reći da predstavljaju ulazni podatak. Veličina ulaznog sloja govori nam o dimenzijama ulaznih podataka, a veličina izlaznog sloja u slučaju problema klasifikacije često nam govori o broju razreda u koje klasificiramo ulaz.



Slika 2.1: 2x3x2 arhitektura umjetne neuronske mreže

Na slici 2.1 moguće je vidjeti primjer arhitekture umjetne neuronske mreže. Neuroni označeni žutom bojom predstavljaju ulazni sloj, neuroni označeni zelenom bojom skriveni sloj, a neuroni označeni plavom bojom izlazni sloj. Ovakvu arhitekturu mreže skraćeno možemo označiti kao 2x3x2 neuronsku mrežu. Pritom brojke označavaju broj neurona u pojedinom sloju (ulazni sloj je prvi sloj mreže).

Za ovakvu mrežu kažemo da je unaprijedna potpuno-povezana mreža. Pojam unaprijedna mreža označava to da ne postoje veze iz dubljih slojeva prema plićim slojevima, a pojam potpuno-povezana mreža označava to da svaki neuron ima vezu sa svakim neuronom iz prethodnog sloja. Uz to, svi neuroni imaju i dodatnu težinu zvanu pomak (nije prikazano na slici 2.1). Djelovanje jednog sloja mreže sažeto možemo prikazati kao:

$$\boldsymbol{h}_i = f(\boldsymbol{W}_i \cdot \boldsymbol{h}_{i-1} + \boldsymbol{b}_i) \tag{2.5}$$

pri čemu W_i predstavlja težine trenutnog sloja, b_i pomake trenutnog sloja, h_{i-1} izlaz iz prethodnog sloja, f prijenosnu funkciju primijenjenu na svaki element te h_i izlaz iz trenutnog sloja. Korištenjem takve formule za svaki sloj mreže, na kraju ćemo dobiti izlaz mreže za neki proizvoljni ulaz. Ovo nazivamo unaprijednim prolazom.

2.5. Učenje neuronske mreže

Kako bismo mogli koristiti proizvoljnu mrežu za probleme klasifikacije ili regresije, potrebno ju je prvo naučiti. Kao što je već prethodno rečeno, učenje neuronske mreže odgovara izmjeni težina pojedinih neurona, a najčešće se postiže algoritmom propagacije pogreške unatrag [3]. Da bismo mogli znati kako trebamo izmijeniti težine neurona, prvo trebamo znati koliko naš model griješi. Mjera greške naziva se gubitak, a računa se na temelju izlaza modela i očekivanog (točnog) izlaza. Za izračun gubitka često je korištena unakrsna entropija koju možemo definirati kao:

$$H(P^*|P) = -\sum_i P^*(i) \cdot \log P(i) \tag{2.6}$$

pri čemu $P^*(i)$ označava distribuciju očekivanog izlaza, a $P(i)$ distribuciju izlaza modela. Jednom kada znamo iznos gubitka, koristeći optimizatore poput stohastičkog gradijentnog spusta (SGD) ili Adam optimizatora [6] možemo poboljšati naš model. Pritom nam je za optimizacijski postupak veoma često potreban gradijent funkcije gubitka po parametrima modela, a izračunavamo ga uzastopnom primjenom pravila ulančavanja koje u svojem najjednostavnijem obliku možemo definirati kao:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{2.7}$$

U slučaju stohastičkog gradijentnog spusta, iterativno ažuriramo težine modela na temelju iznosa gradijenta funkcije gubitka, kao i stope učenja. Sažeto postupak stohastičkog gradijentnog spusta možemo prikazati jednadžbom:

$$w_{i+1} := w_i - \eta \cdot \nabla_w \cdot L(w_i, x) \tag{2.8}$$

pri čemu w_i označava jednu od težina modela u trenutnoj iteraciji, w_{i+1} istu težinu u sljedećoj iteraciji, $L(w_i, x)$ funkciju gubitka, a η stopu učenja. Stopa učenja mali je pozitivni broj pomoću kojeg možemo kontrolirati iznos promjene težina po iteracijama. Iterativnim ponavljanjem ovakvog postupka minimiziramo iznos funkcije gubitka, time dobivajući što bolju mrežu. Kod stohastičkog gradijentnog spusta, iteracije mogu biti predstavljene pojedinačnim ulazima ili mini-grupama. Zbog činjenice da SGD pohranjuje gradijent za male ulaze, ovaj optimizacijski postupak zahtijeva malo memorije.

Nakon učenja mreže na skupu za učenje, evaluirat ćemo performanse mreže na neviđenom skupu zvanom skup za testiranje (engl. *test set*). Često korištena mjera za kvalitetu modela je točnost definirana kao:

$$accuracy = \frac{correct}{total} \tag{2.9}$$

pri čemu $correct$ označava broj točno klasificiranih primjera, $total$ ukupan broj primjera, a $accuracy$ točnost. Uz točnost, postoje i brojne druge mjere kvalitete modela. Neke od njih su preciznost (engl. *precision*), odziv (engl. *recall*) i matrica zabune (engl. *confusion matrix*).

2.6. Duboke neuronske mreže

Ako želimo rješavati složenije probleme koristeći umjetne neuronske mreže sa samo jednim skrivenim slojem, suočit ćemo se s problemom - broj neurona potreban kako bi umjetna neuronska mreža mogla obavljati svoju zadaću bit će prevelik. Uz to, korištenjem širokog modela s velikim brojem neurona u skrivenom sloju teško ćemo postići svojstvo generalizacije jer će se model lako prenaučiti i zapamtiti ulazne podatke.

Zbog tih razloga, veoma su popularne duboke neuronske mreže [3]. Duboke neuronske mreže, za razliku od mreža sa samo jednim skrivenim slojem, imaju nekolicinu skrivenih slojeva. Pritom za rješavanje složenijih problema duboke mreže trebaju imati značajno manje neurona po sloju naspram mreže sa samo jednim skrivenim slojem. Zasebni slojevi mreže naučit će prepoznavati zasebne značajke ulaza, a njihovom kombinacijom mreža će moći postići uspješnu klasifikaciju.

Ipak, postoje i određene mane dubokih neuronskih mreža. Jedna od mana činjenica je da je za propagaciju pogreške unazad kod ovakvih mreža potrebno množiti gradijente. U slučaju da kao prijenosnu funkciju koristimo sigmoidalnu funkciju, ovo lako vodi do problema nestajućeg gradijenata zbog kojega težine neurona u plitkim slojevima nećemo moći ispravno izmijeniti. Uz problem nestajućeg gradijenta, postoji i problem eksplodirajućeg gradijenta (engl. *exploding gradient problem*) koji se pojavljuje kod nekih drugih prijenosnih funkcija od kojih je najpoznatija upravo zglobnica (ReLU). Još jedna mana dubokih neuronskih mreža činjenica je da kako bismo kvalitetno naučili duboku mrežu moramo imati veoma velik skup podataka.

2.7. Konvolucijske mreže

Za probleme s velikim dimenzijama ulaza, potpuno-povezane mreže imaju izuzetno velik broj parametara tj. težina neurona. Učenje ovakvih modela zahtijeva veliku količinu memorije, a i općenito je sporo. Uz to, potpuno-povezane mreže osjetljive su na translaciju ulaza: ako učimo mrežu da klasificira slike vozila te nakon učenja mreži predočimo translatiranu sliku iz skupa za učenje, mreža tu sliku neće nužno moći ispravno klasificirati jer je za nju to potpuno novi podatak. Ovo svojstvo proizlazi iz činjenice da je svaki neuron ovisan o svakom neuronu iz prethodnog sloja.

Kako bi doskočili ovim problemima, uvedene su konvolucijske mreže [12]. Za razliku od potpuno-povezanih mreža, ovdje aktivacija neurona ne ovisi o svim neuronima iz prethodnog sloja, već samo o malom rasponu neurona iz prethodnog sloja. Time konvolucijska mreža ima značajno manje parametara naspram potpuno-povezane mreže iste dubine, a postiže i svojstvo translacijske invarijantnosti. Ova svojstva konvolucijska mreža postiže zamjenom standardnog množenja matrica operacijom konvolucije s jezgrom (engl. *kernel*). Općenito govoreći, operaciju konvolucije za dvije funkcije f, g možemo definirati kao:

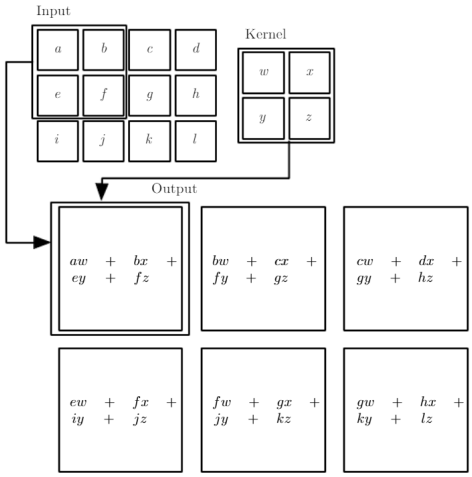
$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau) \cdot g(t - \tau) d\tau \tag{2.10}$$

Operacija konvolucije opisuje nam kako se izgled jedne funkcije mijenja pod utjecajem druge funkcije, a definirana je kao integral umnoška funkcija nakon što je jedna reflektirana i translatirana. Uz operaciju konvolucije postoji i unakrsna korelacija definirana kao:

$$(f \star g)(t) := \int_{-\infty}^{\infty} f(\tau) \cdot g(t + \tau) d\tau \tag{2.11}$$

Važno je primijetiti da je glavna razlika između te dvije operacije izostajanje reflektiranja jedne od funkcija u slučaju operacije unakrsne korelacije. Kada kod konvolucijskih mreža govorimo o konvoluciji, gotovo uvijek se zapravo misli na unakrsnu korelaciju.

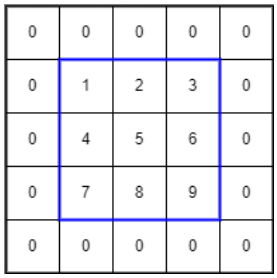
Kako bismo što jednostavnije objasnili konvoluciju, koristit ćemo primjer s 2D konvolucijom. U tom slučaju, jezgra s kojom se provodi konvolucija mala je kvadratna matrica s težinama koje učenjem izmjenjujemo. Skalarnim produktom dijelova ulazne matrice i jezgre dobivamo izlaz konvolucijskog sloja. Pritom se jezgra pomiče po ulaznoj matrici, a rezultati skalarnog produkta zapisuju se u novu matricu koju zovemo mapa značajki.



Slika 2.2: Primjer 2D konvolucije. Preuzeto iz [3]

Na slici 2.2 možemo vidjeti rezultat 2D konvolucije s ulaznom matricom veličine 3x4 i jezgrom dimenzija 2x2. Mapa značajki nastala kao rezultat ove konvolucije dimenzija je 2x3. Primijetimo da će izlaz konvolucijskog sloja uvijek biti manjih dimenzija naspram ulaza. Uz to, vrijednosti na rubovima matrice ulaza manje će doprinijeti rezultatu naspram vrijednosti koje su dalje od rubova.

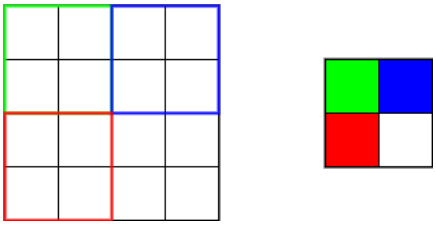
Kako bismo imali veću kontrolu nad dimenzijama izlaza, kao i utjecajem vrijednosti na rubovima matrice ulaza, često koristimo nadopunjavanje nulama (engl. *zero padding*).



Slika 2.3: Primjer nadopunjavanja 3x3 matrice nulama

Na slici 2.3 moguće je vidjeti 3x3 matricu nadopunjenu nulama. U slučaju da nad takvom matricom primijenimo konvoluciju s jezgrom dimenzija 2x2, mapa značajki na izlazu bila bi dimenzija 4x4. Da smo konvoluciju primijenili nad matricom bez nadopunjavanja, mapa značajki bila bi dimenzija 2x2, a vrijednosti pri rubovima matrice manje bi doprinosile istoj. Možemo reći da nule čine okvir oko originalne matrice, time osiguravajući veće dimenzije izlaza.

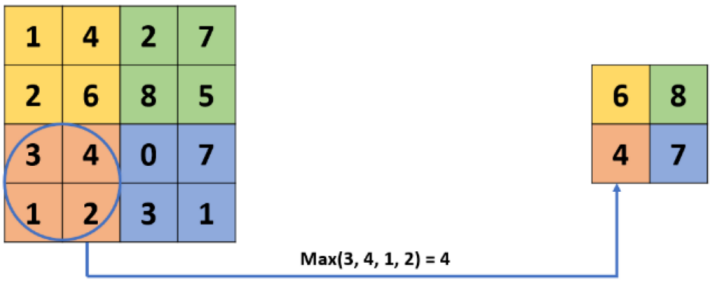
Uz nadopunjavanje nulama, kod konvolucijskih slojeva često se mijenja i korak (engl. *stride*). Na slici 2.2 korak je 1, a definira za koliko se ćelija horizontalno i vertikalno pomiče jezgra. U slučaju da povećamo korak, izlaz konvolucije bio bi manjih dimenzija, a time bi se i ubrzao postupak računanja izlaza.



Slika 2.4: Primjer konvolucije s korakom 2

Na slici 2.4 moguće je vidjeti konvoluciju s korakom 2. Za razliku od standardnog kretanja jezgre, ovdje se jezgra nakon svakog izračuna pomiče za 2 ćelije.

Osim samih konvolucijskih slojeva, konvolucijske mreže u sebi sadrže i slojeve sažimanja. Najčešći razlog za upotrebu slojeva sažimanja je smanjivanje dimenzija podataka, a time i skraćivanje vremena potrebnog za učenje mreže.

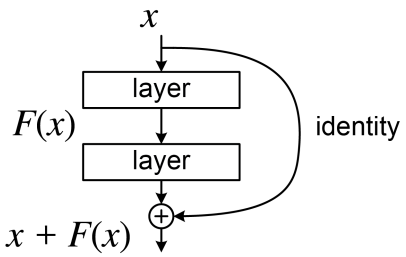


Slika 2.5: Primjer sažimanja maksimalnom vrijednošću. Preuzeto iz [2]

Na slici 2.5 moguće je vidjeti primjer korištenja sloja sažimanja maksimalnom vrijednošću. Mapa značajki dimenzija 4x4 korištenjem sažimanja maksimalnom vrijednošću 2x2 (engl. *2x2 max pooling*) smanjena je na dimenzije 2x2, efektivno smanjujući dimenzije ulaza za faktor 2. Uz sažimanje maksimalnom vrijednošću, veoma je popularno i sažimanje usrednjavanjem, no postoje i brojne druge varijante koje se koriste za slojeve sažimanja. Kao i operacija konvolucije, i slojevi sažimanja doprinose translacijskoj invarijantnosti, osiguravajući da mreža lako može prepoznati neku značajku bez obzira na njenu točnu lokaciju.

2.8. Rezidualne mreže

Rezidualne mreže [5] vrsta su dubokih neuronskih mreža koje koriste rezidualne blokove. Općenito govoreći, blok u kontekstu neuronskih mreža označava niz od nekoliko slojeva. Pojedini blokovi mogu se kombinirati kako bi sačinjavali složeniju mrežu. Rezidualni blokovi najčešće se sastoje od nekoliko konvolucijskih slojeva, a njihova glavna karakteristika postojanje je preskočnih veza.



Slika 2.6: Primjer rezidualnog bloka. Preuzeto iz [5]

Na slici 2.6 možemo vidjeti rezidualni blok sačinjen od dva sloja. Preskočna veza ulaz u blok bez ikakvih transformacija prenosi na izlaz. Ovakvim strukturiranjem mreže, cilj mreže postaje modelirati rezidualnu funkciju $F(x)$ koja mjeri razliku izlaza naspram ulaza. Rezidualni blok možemo prikazati jednadžbom:

$$f(\boldsymbol{x}) = F(\boldsymbol{x}) + \boldsymbol{x} \tag{2.12}$$

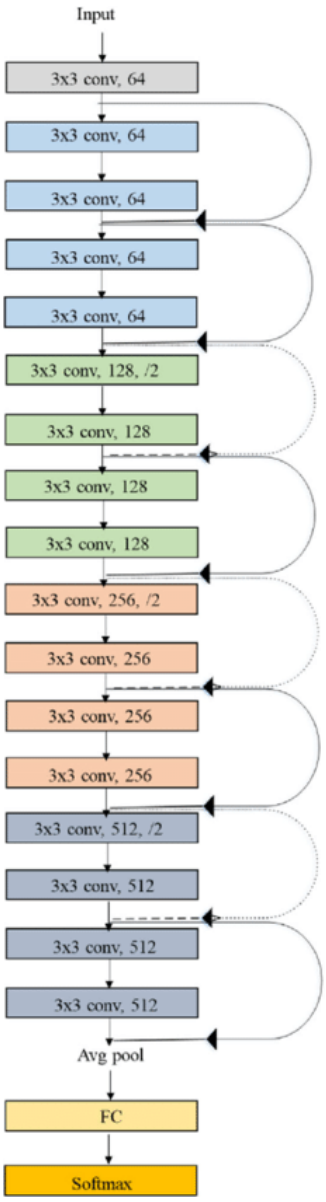
pri čemu $F(x)$ predstavlja rezidualnu funkciju, x prenošenje ulaza na izlaz preskočnom vezom, a $f(x)$ izlaz bloka. U slučaju da slojevi između mijenjaju dimenzije podataka, preskočna veza morat će raditi linearnu projekciju kako bi podatci pri zbrajanju bili istih dimenzija. Tada rezidualni blok možemo prikazati na sljedeći način:

$$f(\boldsymbol{x}) = F(\boldsymbol{x}) + \boldsymbol{W}_s \cdot \boldsymbol{x} \tag{2.13}$$

pri čemu W_s označava matricu korištenu za linearnu projekciju ulaza x. Korištenje slojeva s preskočnim vezama motivirano je željom za učenjem funkcije identiteta. Klasične duboke mreže s velikim brojem slojeva teško će naučiti funkciju identiteta, dok korištenje preskočnih veza dubokim mrežama značajno olakšava učenje te funkcije. Ovo potječe od činjenice da je za uspješno modeliranje funkcije identiteta $f(x)$ za rezidualnu funkciju $F(x)$ potrebno samo postaviti težine jezgri na 0.

2.9. ResNet18

U okviru ovog rada, za provođenje svih eksperimenata koristit ćemo ResNet18 mrežu kako bismo što bolje mogli usporediti učinak različitih eksperimenata. ResNet18 rezidualna je neuronska mreža, a sastoji se od ukupno 18 slojeva.



Slika 2.7: Arhitektura ResNet18. Preuzeto iz [13]

Kao što možemo vidjeti na slici 2.7, ResNet18 sastoji se od 8 rezidualnih blokova. Svaki od tih blokova sastoji se od 2 konvolucijska sloja i jedne preskočne veze. Rezidualnim blokovima prethodi jedan konvolucijski sloj, a nakon njih dolazi jedan potpuno-povezani sloj sa softmax prijenosnom (aktivacijskom) funkcijom.

3. Brzo učenje s neprijateljskim primjerima

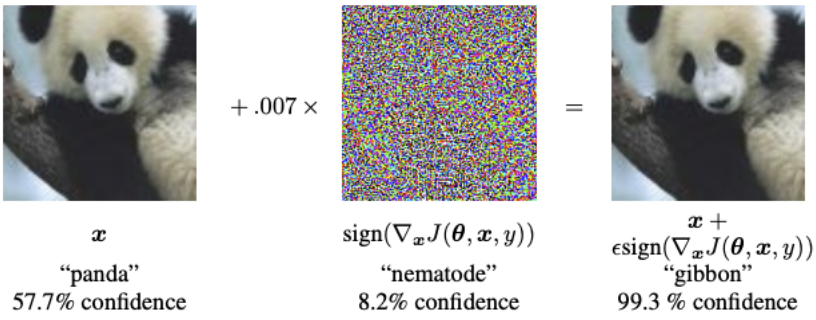
3.1. Neprijateljski primjeri

Kada učimo modele da rješavaju razne probleme, cilj nam je da modeli pokažu svojstvo generalizacije. U slučaju problema klasifikacije, to znači da bi modeli trebali dobro klasificirati i do sada neviđene primjere. Mogućnost generalizacije određenog modela najčešće provjeravamo koristeći unakrsnu provjeru (engl. *cross-validation*). Pritom skup podataka dijelimo na 2 ili 3 skupa:

1. skup za učenje (engl. *train set*)
2. skup za provjeru (engl. *validation set*)
3. skup za testiranje (engl. *test set*)

Skup za učenje modelu predočavamo kako bi na temelju njega mogao poboljšavati svoje parametre tj. učiti. S druge strane, skup za provjeru koristimo za poboljšavanje hiperparametara modela (npr. stopa učenja modela). U slučaju da smo sigurni u dobar odabir hiperparametara modela, korištenje ovog skupa nije nužno. Nakon postupka učenja, koristeći skup za testiranje evaluiramo model.

Općenito govoreći, model će na skupu za testiranje postizati lošije rezultate naspram rezultata na skupu za učenje. Ako model dobro generalizira, razlika između rezultata na ta dva skupa neće biti veoma značajna. Ipak, u zadnjih desetak godina otkriveni su ulazi koje i modeli koji inače pokazuju mogućnost generalizacije loše klasificiraju. Takve ulaze zvat ćemo neprijateljski primjeri [4].



Slika 3.1: Primjer neprijateljskog primjera. Preuzeto iz [4]

Na slici 3.1 moguće je vidjeti jedan neprijateljski primjer. Originalni ulaz kojeg model ispravno klasificira izmijenjen je za iznos perturbacije. U ovom slučaju iznos perturbacije dobiven je kao umnožak malog pozitivnog koeficijenta ϵ i predznaka gradijenta gubitka po ulazu. Iako je izmijenjeni ulaz ljudima gotovo identičan, model ovaj ulaz krivo klasificira, a pritom je veoma siguran u svoju odluku. Ovo svojstvo iskazali su brojni modeli te je zbog toga u zadnjih desetak godina porastao interes za proučavanjem neprijateljskih primjera.

3.2. Načini generiranja neprijateljskih primjera

Neprijateljske primjere moguće je generirati na različite načine. Načine generiranja neprijateljskih primjera zovemo napadima, a možemo ih podijeliti na usmjerene i neusmjerene napade. Usmjereni napadi kao cilj imaju generiranje neprijateljskih primjera koje će napadnuti model klasificirati u točno jedan ciljni razred, dok neusmjereni napadi kao cilj imaju generiranje neprijateljskih primjera koje će napadnuti model što lošije klasificirati, neovisno o konkretnom razredu. Pronalaženje neprijateljskih primjera zapravo je optimizacijski problem koji možemo prikazati sljedećim izrazom:

$$\underset{\delta \in [-\epsilon, \epsilon]}{max} L(x + \delta, y; \theta) \tag{3.1}$$

pri čemu δ označava iznos perturbacije u intervalu određenim s granicama $[-\epsilon, \epsilon]$, $L(x, y; \theta)$ funkciju gubitka, a θ parametre modela. Drugim riječima, cilj nam je naći perturbaciju omeđenu određenim iznosom takvu da je iznos funkcije gubitka za ulaz izmijenjen tom perturbacijom maksimalan. Iznos perturbacije najčešće je omeđen L_∞-normom, no isti može biti omeđen i L_2-normom ili L_1-normom.

Postoje brojni načini generiranja neprijateljskih primjera. Neki od najpoznatijih su FGSM (engl. *Fast Gradient Sign Method*) [4] i PGD (engl. *Projected Gradient Descent*) [10] napadi. Primjer sa slike 3.1 zapravo je neprijateljski primjer generiran koristeći FGSM napad, a neprijateljske primjere generirane koristeći taj napad sažeto možemo opisati jednadžbom:

$$x := x + \epsilon \cdot sign(\nabla_x L(x, y; \theta)) \tag{3.2}$$

pri čemu ϵ predstavlja mali pozitivan broj koji određuje veličinu perturbacije, a $sign(x)$ predznak od x. FGSM napad originalnom ulazu pridodaje iznos istog predznaka kao i gradijent funkcije gubitka, time osiguravajući najveći mogući rast iznosa funkcije gubitka. U slučaju neusmjerenog napada, napad bi pokušavao što više smanjiti iznos funkcije gubitka s obzirom na proizvoljan razred.

Uz FGSM napad, postoji i napad koji iterativno stvara značajno učinkovitije neprijateljske primjere. Tu vrstu napada zovemo PGD napad, a zasniva se na istoj ideji kao i FGSM napad. Za razliku od FGSM-a, PGD napad iterativno ponavlja postupak, time pokušavajući odrediti najbolje rješenje optimizacijskog problema 3.1. Uz to, PGD napad tijekom generiranja neprijateljskih primjera osigurava da ukupna perturbacija nije veća od unaprijed definiranog koeficijenta ϵ.

Algorithm 1 Generiranje neprijateljskih primjera koristeći PGD napad

Ulaz: x – ulazne slike, y – ispravne oznake ulaznih slika, ϵ – ograničenje perturbacije, α – veličina koraka po iteraciji, K – broj iteracija za generiranje neprijateljskih primjera

Izlaz: x_{adv} – neprijateljski primjeri generirani PGD napadom

---

$\delta = U(-\epsilon, \epsilon)$

for $(i = 1 \dots K)$ **do**

  $grad = \nabla_\delta L(x + \delta, y; \theta)$

  $\delta = \delta + \alpha \cdot sign(g)$

  $\delta = clamp(\delta, -\epsilon, \epsilon)$

end for

$x_{adv} = x + \delta$

return x_{adv}

Pseudokod 1 prikazuje način generiranja neprijateljskih primjera koristeći PGD napad. Perturbacija δ na početku je inicijalizirana koristeći uniformnu distribuciju s rasponom $[-\epsilon, \epsilon]$, a na kraju svake iteracije iznos perturbacije ograničava se koristeći funkciju *clamp*. Naravno, postoje brojni načini inicijalizacije iznosa perturbacije, no nasumična inicijalizacija pokazala se dosta uspješnom. Važno je istaknuti da pseudokod za ograničavanje iznosa perturbacije koristi L_∞-normu. PGD napad s K iteracija za generiranje neprijateljskih primjera zvat ćemo K-PGD (npr. 20-PGD).

3.3. Učenje s neprijateljskim primjerima

Ako želimo da modeli koje učimo budu otporni na neprijateljske primjere tj. ispravno ih klasificiraju, ne možemo se ograničiti na klasično (prirodno) učenje modela. Svojstvo otpornosti na neprijateljske primjere zvat ćemo robusnost, a jedan od mogućih načina za postizanje robusnih modela učenje je s neprijateljskim primjerima (robusno učenje). Kada govorimo o učenju s neprijateljskim primjerima, općenita ideja je da se tijekom učenja generiraju neprijateljski primjeri prilagođeni samome modelu. Model tada ne uči na originalnim ulaznim podatcima, već isključivo na generiranim neprijateljskim primjerima. U tom slučaju neprijateljske primjere možemo smatrati načinom augmentacije ulaznih podataka - na temelju originalnih ulaznih podataka stvaramo nove podatke koje predočavamo modelu.

Robusnim učenjem možemo postići puno višu točnost modela na skupu za testiranje sačinjenom od neprijateljskih primjera naspram modela učenih prirodnim učenjem. Nažalost, robusno učenje ima svoju cijenu. Robusni modeli gotovo uvijek će na originalnom skupu za testiranje imati nižu točnost naspram modela učenih prirodnim učenjem, a i učit će se dulje. Ako za generiranje neprijateljskih primjera tijekom robusnog učenja koristimo FGSM metodu, učenje će trajati otprilike dvostruko dulje naspram prirodnog učenja jer je za generiranje neprijateljskih primjera FGSM metodom potreban dodatan unaprijedni prolaz kroz mrežu. Ako za generiranje neprijateljskih primjera koristimo PGD metodu, duljina trajanja učenja ovisit će o broju iteracija korištenih za generiranje neprijateljskih primjera. Koristimo li 10 iteracija za generiranje neprijateljskih primjera, učenje će trajati otprilike 11 puta dulje naspram prirodnog učenja. Općenito govoreći, kako bismo model učili koristeći neprijateljske primjere generirane K-PGD metodom, učenje će trajati otprilike K + 1 puta dulje naspram prirodnog učenja [14].

Algorithm 2 Učenje s neprijateljskim primjerima

Ulaz: x – ulazne slike, y – ispravne oznake ulaznih slika, N – broj epoha, M – veličina skupa za učenje, K – broj iteracija za generiranje neprijateljskih primjera, η – stopa učenja, ϵ – ograničenje perturbacije, α – veličina koraka po iteraciji

---

$\theta = initializeModelParams()$
for $(ep = 1 \dots N)$ **do**
  for $(i = 1 \dots M)$ **do**
    $x_{adv} = generateAdversarial(x_i, y_i, \epsilon, \alpha, K)$
    $grad = \nabla_\theta L(x_{adv}, y_i; \theta)$
    $\theta = \theta - \eta \cdot grad$
  end for
end for

Pseudokod 2 prikazuje općeniti algoritam za učenje s neprijateljskim primjerima. Pritom inicijalizaciju parametara modela θ prikazujemo pozivom funkcije *initializeModelParams*. Radi općenitosti, generiranje neprijateljskih primjera prikazano je pozivom funkcije *generateAdversarial*. Ta funkcija mogla bi generirati neprijateljske primjere koristeći FGSM metodu, PGD metodu ili neki alternativan način, no to nam za općeniti prikaz nije važno.

3.4. Brzo učenje s neprijateljskim primjerima

U slučaju da model koji učimo nije veoma složen, obično učenje s neprijateljskim primjerima prihvatljiv je način za postizanje robusnih modela otpornih na napade. Ipak, ako učimo složene modele na velikim skupovima podataka, obično učenje s neprijateljskim primjerima presporo je. Uz to, rijetko tko uopće posjeduje dovoljnu količinu računalnih resursa za robusno učenje složenijih modela. Kako bismo riješili ili barem umanjili ovaj problem, predložena su brojna rješenja čijom bi se primjenom ubrzalo robusno učenje. Razmotrit ćemo tri takva rješenja.

3.4.1. Besplatno učenje

Jedno od prvih predloženih rješenja za problem duljine trajanja učenja robusnih modela naziva se besplatno učenje (engl. *free adversarial training*) [14]. Uočeno je da modeli učeni koristeći neprijateljske primjere generirane PGD metodom imaju visoku robusnost, no, kao što je već rečeno, zahtijevaju puno vremena za učenje.

Osnovna ideja besplatnog učenja je sljedeća: unatražnim prolazom osim iznosa gradijenta po parametrima modela izračunamo i iznos gradijenta po ulazu potreban za generiranje neprijateljskih primjera. Kako bi se neprijateljski primjeri iterativno izmjenjivali, na svakoj mini-grupi model uči nekoliko puta zaredom. Ovaj hiperparametar zovemo ponavljanje (engl. *replay*). Važno je uočiti da, kako bismo i dalje trebali otprilike jednako vremena za učenje modela, moramo smanjiti broj epoha za faktor ponavljanja. Ako svaku mini-grupu unutar jedne epohe ponavljamo 8 puta, broj epoha bit će 8 puta manji naspram broja epoha kod modela učenog prirodnim učenjem. Nažalost, povećavanjem broja ponavljanja dolazi do degradacije performansi modela na običnom testnom skupu - u pitanju je kompromis između željene razine robusnosti i točnosti na običnom testnom skupu.

Algorithm 3 Besplatno učenje s neprijateljskim primjerima. Prilagođeno iz [16]

Ulaz: x – ulazne slike, y – ispravne oznake ulaznih slika, N – broj epoha, M – veličina skupa za učenje, K – broj ponavljanja (engl. *replay*), η – stopa učenja, ϵ – ograničenje perturbacije

---

$\theta = initializeModelParams()$
$\delta = 0$
for $(ep = 1 \dots N/K)$ **do**
  for $(i = 1 \dots M)$ **do**
    for $(j = 1 \dots K)$ **do**
      $gradAdv, grad = \nabla L(x_i + \delta, y_i; \theta)$
      $\delta = \delta + \epsilon \cdot sign(gradAdv)$
      $\delta = clamp(\delta, -\epsilon, \epsilon)$
      $\theta = \theta - \eta \cdot grad$
    end for
  end for
end for

Pseudokod 3 prikazuje algoritam za besplatno učenje s neprijateljskim primjerima. Kao i u prethodnom algoritmu, parametri modela θ inicijaliziraju se pozivom funkcije *initializeModelParams*. Važno je uočiti da se iznosi gradijenata potrebni za generiranje neprijateljskih primjera, ali i izmjenu parametara modela izračunavaju u istom unatražnom prolazu, time osiguravajući da generiranje neprijateljskih primjera ne usporava postupak učenja. Uz to, perturbacija δ inicijalizira se na 0 samo na početku postupka učenja - kada započinje izračun perturbacije za sljedeći ulaz, prethodni iznos služi kao "iznos za zagrijavanje" (engl. *warmup*).

Ovakvim načinom robusnog učenja, duljina trajanja učenja modela otprilike je jednaka kao i duljina trajanja učenja modela prirodnim učenjem. Uz to, rezultati rada [14] pokazuju da je robusnost naučenih modela usporediva s robusnošću modela učenih koristeći neprijateljske primjere generirane PGD metodom. Kao što je već rečeno, visoku robusnost modela učenih koristeći besplatno učenje možemo postići izmjenom broja ponavljanja, ali po cijeni smanjenja performansi modela na običnom testnom skupu.

3.4.2. Brzo učenje

Za modele učene koristeći FGSM metodu dugo se smatralo da nisu otporni na neprijateljske primjere generirane koristeći iterativne metode poput PGD-a. Takvi modeli nisu nam veoma korisni jer često imaju nižu točnost na običnim podatcima, a ne nude pravu mjeru robusnosti - napadač jednostavno može generirati neprijateljske primjere koje će model krivo klasificirati. Ipak, rad [16] predlaže da problem ne leži u korištenju FGSM metode, već njenoj inicijalizaciji. Standardno se perturbacija za FGSM metodu inicijalizira na 0 ili na granicu perturbacije ϵ, bilo s pozitivnim ili negativnim predznakom. U slučaju da za inicijalizaciju iskoristimo nasumičnu inicijalizaciju koristeći uniformnu distribuciju s granicama $[-\epsilon, \epsilon]$, modeli učeni koristeći FGSM metodu postižu svojstvo robusnosti čak i protiv iterativnih napada.

Ako učenje koristeći FGSM metodu s nasumičnom inicijalizacijom kombiniramo s nekim od mogućih optimizacija poput korištenja cikličke stope učenja i računanja u mješovitoj preciznosti, dobivamo tzv. brzo učenje (engl. *fast adversarial training*) [16]. Predložene optimizacije omogućavaju značajno ubrzanje učenja modela, a nisu implementacijski zahtjevne. Ciklička stopa učenja [15] označava promjenu stope učenja kroz epohe ili iteracije - nakon svakog koraka, stopa učenja će se povećavati ili smanjivati unutar unaprijed definiranih granica. Uvođenje cikličke stope učenja korisno je za brže konvergiranja modela, time ubrzavajući učenje. Uz cikličku, postoje i brojne druge poput linearne i kosinusne, no brzo učenje koristi upravo cikličku stopu učenja.

Razvoj tenzorskih jezgri grafičkih kartica omogućilo je korištenje računanja u mješovitoj preciznosti - umjesto da se svi izračuni obavljaju koristeći 32-bitne brojeve s pomičnim zarezom, neki izračuni obavljaju se koristeći 16-bitne brojeve. Korištenjem računanja u mješovitoj preciznosti, značajno se može ubrzati učenje, ali i smanjiti potrebna količina memorije za učenje modela. Naravno, da bismo isto mogli koristiti, grafičke kartice na kojim učimo modele moraju imati tenzorske jezgre. Ako koristimo starije grafičke kartice, učenje neće biti brže.

Algorithm 4 Brzo učenje s neprijateljskim primjerima. Prilagođeno iz [16]

Ulaz: x – ulazne slike, y – ispravne oznake ulaznih slika, N – broj epoha, M – veličina skupa za učenje, η – stopa učenja, ϵ – ograničenje perturbacije, α – veličina koraka

---

$\theta = initializeModelParams()$
for $(ep = 1 \dots N)$ **do**
  for $(i = 1 \dots M)$ **do**
    $\delta = U(-\epsilon, \epsilon)$
    $gradAdv = \nabla_\delta L(x_i + \delta, y_i; \theta)$
    $\delta = \delta + \alpha \cdot sign(gradAdv)$
    $\delta = clamp(\delta, -\epsilon, \epsilon)$
    $grad = \nabla_\theta L(x_i + \delta, y_i; \theta)$
    $\theta = \theta - \eta \cdot grad$
  end for
end for

Pseudokod 4 prikazuje algoritam za brzo učenje s neprijateljskim primjerima. Kao i prije, parametri modela θ inicijaliziraju se pozivom funkcije *initializeModelParams*. Početni iznos perturbacije δ inicijalizira se koristeći uniformnu distribuciju s rasponom $[-\epsilon, \epsilon]$. Za razliku od besplatnog učenja, kod brzog učenja potrebna su dva unaprijedna prolaza, kao i dva zasebna izračuna iznosa gradijenta za svaki ulaz. Zbog ovoga brzo učenje nije jednako brzo kao i prirodno učenje modela, ali zato nudi određenu mjeru robusnosti.

Prema radu [16], brzo učenje modela nudi gotovo jednaku mjeru robusnosti protiv iterativnih napada kao i modeli učeni koristeći PGD metodu, ali sa značajno kraćim vremenom učenja. Čak i ako već navedene optimizacije primijenimo na besplatno učenje i učenje koristeći PGD metodu, da bismo postigli jednaku mjeru robusnosti modele je potrebno učiti dulje nego modele učene koristeći brzo učenje.

Nažalost, korištenje brzog učenja s neprijateljskim primjerima ima i svoju manu. U slučaju da modele učimo velik broj epoha, u jednom trenutku gotovo sigurno će doći do značajnog pada u točnosti na neprijateljskim primjerima. Ovu pojavu zovemo katastrofalna prenaučenost (engl. *catastrophic overfitting*), a jedan od mogućih načina za sprječavanje iste je korištenje ranog završetka (engl. *early stopping*). Kako bismo na vrijeme mogli zaustaviti učenje modela, nakon svake epohe evaluiramo točnost modela na neprijateljskim primjerima generiranim iz nasumične mini-grupe. U slučaju da je točnost u trenutnoj epohi značajno manja od točnosti u prethodnoj epohi, zaustavljamo učenje modela i kao najbolji model vraćamo model iz prethodne epohe. Zbog ovog svojstva, korištenje brzog učenja s neprijateljskim primjerima nije prikladno za modele koje trebamo učiti velik broj epoha.

3.4.3. Nadogradnje brzog učenja

Kako bi brzo učenje s neprijateljskim primjerima bilo primjenjivo za proizvoljno dugo učenje, potrebno je riješiti problem katastrofalne prenaučenosti. Taj problem pojavljuje se kod svih postupaka koji koriste FGSM metodu za učenje, no manje je izražen kod brzog učenja nego kod običnog učenja s neprijateljskim primjerima generiranim koristeći FGSM metodu. Rad [8] otkriva nam da brzo učenje u manjoj mjeri pati od problema katastrofalne prenaučenosti zbog nasumične inicijalizacije perturbacije. Kao i kod modela učenih običnim FGSM-om, i modeli učeni koristeći brzo učenje susreću se s katastrofalnom prenaučenošću, ali se od iste u periodu od nekoliko mini-grupa mogu oporaviti te nastaviti dalje normalno učiti.

Ključ u oporavku leži u nasumičnoj inicijalizaciji koja pomaže u stvaranju kvalitetnih neprijateljskih primjera. Ipak, zbog nasumičnosti nemamo garanciju da će takav postupak uvijek biti dovoljan za oporavak pa nakon određene količine vremena i dalje dođe do katastrofalne prenaučenosti. Ako bismo mogli garantirati da će se model tijekom učenja uvijek moći oporaviti od katastrofalne prenaučenosti u periodu od nekoliko mini-grupa, mogli bismo proizvoljno dugo učiti model. U radu [8] oporavak se postiže kontinuiranim praćenjem točnosti na neprijateljskim primjerima generiranim iz nasumične mini-grupe svakih s mini-grupa. U slučaju da je točnost u trenutnom koraku značajno manja od točnosti u prethodnom koraku, do sljedeće provjere točnosti umjesto FGSM metode s nasumičnom inicijalizacijom, za generiranje neprijateljskih primjera koristimo PGD metodu. Ovakvim postupkom model se lako može oporaviti od pojave katastrofalne prenaučenosti pa stoga može i dulje učiti. Opisani način učenja nazivamo *FastAdv+* (engl. *Fast Adversarial Training Plus*).

Algorithm 5 *FastAdv+* učenje s neprijateljskim primjerima. Prilagođeno iz [8]

Ulaz: x – ulazne slike, y – ispravne oznake ulaznih slika, N – broj epoha, M – veličina skupa za učenje, η – stopa učenja, ϵ – ograničenje perturbacije, α – veličina koraka, c – prag detekcije, s – frekvencija detekcije

---

$\theta = initializeModelParams()$
$accLast = 0$
$accValid = 0$
for $(ep = 1 \dots N)$ **do**
  for $(i = 1 \dots M)$ **do**
    if $(accLast > accValid + c)$ **then**
      $x_{adv} = PGD(x_i, y_i, \epsilon, \alpha, K)$
    else
      $x_{adv} = RandomFGSM(x_i, y_i, \epsilon, \alpha, K)$
    end if
    $grad = \nabla_\theta L(x_{adv}, y_i; \theta)$
    $\theta = \theta - \eta \cdot grad$
    if $(i\%s == 0)$ **then**
      $accLast = accValid$
      $accValid = evaluateRobustness(x_{rand}, y_{rand}, \epsilon, \alpha, K)$
    end if
  end for
end for

Pseudokod 5 prikazuje algoritam *FastAdv+* za učenje s neprijateljskim primjerima. Kao i prije, parametri modela θ inicijaliziraju se pozivom funkcije *initializeModelParams*. Kako bismo pratili točnost modela na neprijateljskim primjerima, koristimo varijable *accLast* i *accValid* pri čemu prva varijabla predstavlja stariju točnost s kojom uspoređujemo, a druga varijabla točnost iz prethodnog koraka. U slučaju da je u prethodnom koraku točnost pala ispod vrijednosti određene pragom c, za generiranje neprijateljskih primjera koristit ćemo PGD metodu, a inače ćemo za generiranje neprijateljskih primjera koristiti metodu *RandomFGSM* koja predstavlja FGSM metodu s nasumičnom inicijalizacijom. Nakon provođenja učenja, u slučaju da smo na iteraciji određenoj frekvencijom detekcije s, evaluirat ćemo točnost modela na neprijateljskim primjerima pozivom funkcije *evaluateRobustness*. U originalnome radu [8], za provjeru je korišten PGD metoda, a isto je reproducirano i u ovome radu.

Ovakav način učenja omogućava nam proizvoljno dugo učenje modela, no i dalje ne dostiže jednaku mjeru robusnosti kao i najnovije varijante učenja s PGD metodom. Kako bismo dodatno unaprijedili *FastAdv+* učenje, predložena je nadogradnja: zadnjih nekolicinu epoha model ćemo učiti isključivo koristeći PGD metodu za generiranje neprijateljskih primjera. Ova nadogradnja motivirana je hipotezom da je na početku treniranja modelu dovoljno imati slabije neprijateljske primjere za učenje, dok je za kasnije epohe potrebno koristiti jače neprijateljske primjere kako bi model dodatno mogao učiti. Opisani način učenja zovemo *FastAdvW* (engl. *Fast Adversarial Training Warmup*), a prema rezultatima rada [8] modeli učeni koristeći *FastAdvW* način učenja postižu bolje rezultate čak i naspram najnovijih varijanta učenja s PGD metodom, pritom zahtijevajući značajno manje vremena za učenje.

4. Zatrovani podatci

4.1. Općenito o zatrovanim podatcima

Kako bismo u nekom sustavu koristili modele dubokog učenja za postizanje određene funkcionalnosti, modele je prvo potrebno naučiti na velikom skupu podataka. Iako danas često imamo goleme količine podataka koje možemo koristiti za učenje modela, sve podatke trebalo bi provjeravati. U slučaju da koristimo neprovjerene podatke, izlažemo naš sustav brojnim prijetnjama. Jedna od takvih prijetnji ubacivanje je zatrovanih podataka u skup za učenje modela.

Zatrovani podatci namjerno su dizajnirani podatci čiji cilj je zavaravanje sustava. Slični su neprijateljskim primjerima, ali postoje i određene razlike koje se većinom očituju u namjeri korištenja istih. Dok neprijateljske primjere koristimo kako bi za neki konkretan ulaz izmijenili klasifikaciju nakon što je model već naučen, zatrovane podatke ubacujemo u skup za učenje modela kako bismo izmijenili decizijsku granicu modela - ako model učimo na zatrovanom skupu podataka, kada isti nakon učenja koristimo za klasifikaciju, njegove odluke za određene ulaze bit će drugačije nego što bi bile da smo model učili na čistom skupu. Uz to, zatrovani podatci mogu biti ručno dizajnirani, dok neprijateljske primjere većinom generiramo koristeći napade zasnovane na gradijentu funkcije gubitka modela po ulazu.

Slika 4.1: Pojednostavljeni primjer zatrovanih podataka. Preuzeto iz [11]

Na slici 4.1 moguće je vidjeti pojednostavljene primjere zatrovanih podataka. Na originalne slike dodan je bijeli pravokutnik u nadi da će model naučiti prepoznavati takav pravokutnik i na temelju njegove prisutnosti kategorizirati sve ulaze u pojedini razred. Osim modifikacije samih ulaznih slika, radi se i modifikacija ispravnih oznaka: za sve modificirane slike napadač bi mogao postaviti istu oznaku. U slučaju slike 4.1, napadač bi mogao za sve tri slike dodijeliti razred *dog* kao ispravnu oznaku. Ako nakon učenja model na nekoj slici prepozna bijeli pravokutnik, lako je moguće da će ju klasificirati u razred *dog* iako ona zapravo pripada nekom drugom razredu. Naravno, ovaj primjer veoma je pojednostavljen - u stvarnosti izmjene originalnih slika mogu biti veoma suptilne, baš kao i promjene prisutne kod neprijateljskih primjera. Izmijenjeni dio ulaza zvat ćemo okidačem (engl. *trigger*). U slučaju da je na jednostavan način moguće ubaciti podatke u skup za učenje našeg modela, napadači tu ranjivost mogu iskoristiti za razne ciljeve. Općenito govoreći, ubacivanje zatrovanih podataka kao posljedicu može imati pojavu jedne od dviju značajnih mana sustava:

1. Pad točnosti modela
2. Ugradnja stražnjih vrata u model

U prvome slučaju, napadač ubacivanjem zatrovanih podataka želi degradirati performanse modela, time čineći njegov rad nepouzdanijim. U drugome slučaju, napadač pažljivim dizajnom zatrovanih podataka može do određene mjere manipulirati ponašanjem modela. Ako model nauči klasificirati sve zatrovane podatke u određeni razred te nakon učenja kao ulaz dobije novi zatrovani podatak ili podatak koji ima određenu mjeru sličnosti s naučenim okidačem, postoji velika vjerojatnost da će isti krivo klasificirati.

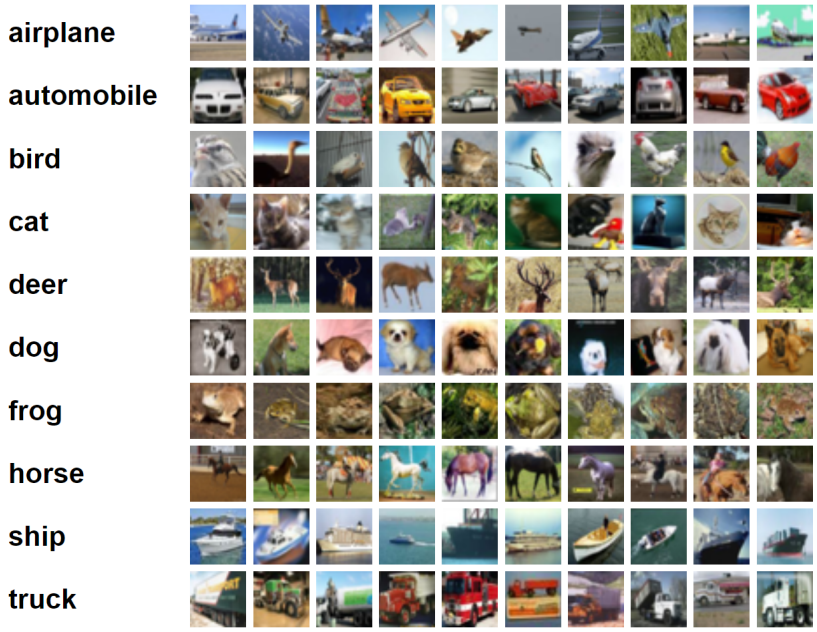
4.2. BackdoorBox

BackdoorBox Toolbox [9] skup je alata otvorenog koda koji služi za edukaciju o napadima čiji je cilj ugradnja stražnjih vrata u modele. Osim napada, u skupu alata postoje i brojne obrane. U okviru ovog rada, koristit ćemo *BadNets* alat iz skupa alata *BackdoorBox* kako bismo generirali zatrovane skupove podataka. Generirane zatrovane skupove podataka koristit ćemo za evaluaciju otpornosti robusnih modela na zatrovane podatke, kao i učenje novih modela.

5. Eksperimenti

5.1. Skup podataka CIFAR-10

Skup podataka CIFAR-10 [7] sastoji se od 60000 32x32 slika u boji. Svaka slika ima oznaku jednog od 10 razreda, a cijeli skup podataka sastoji se od 6000 slika svakog razreda. Skup podataka podijeljen je na 50000 slika u skupu za učenje i 10000 slika u skupu za testiranje.



Slika 5.1: Primjeri slika i oznaka iz skupa podataka CIFAR-10. Preuzeto iz [7]

Na slici 5.1 moguće je vidjeti razrede iz skupa podataka CIFAR-10, kao i po 10 nasumično odabranih slika iz svakog razreda. Svi razredi međusobno su isključivi tj. nijedna slika ne pojavljuje se u skupu podataka s više različitih oznaka razreda. Uz CIFAR-10 skup podataka, postoji i CIFAR-100 skup podataka koji sadrži 100 različitih razreda. Za provođenje eksperimenata u ovome radu korišten je CIFAR-10 skup podataka zbog brzine učenja, no metode korištene u radu lako se mogu primijeniti i na skup podataka CIFAR-100.

5.2. Korištene tehnologije

Za provođenje eksperimenata u ovome radu, kod je pisan u programskom jeziku Python. Osim biblioteka dostupnih u standardnom paketu biblioteka programskog jezika Python, korištene su biblioteke NumPy, Matplotlib i radni okvir PyTorch. Sami kod izvršavan je primarno na platformi Kaggle koristeći dvije grafičke kartice NVIDIA T4.

5.2.1. NumPy

NumPy je biblioteka pisana za programski jezik Python. Služi za provođenje operacija na matricama i tenzorima, a nudi i implementacije brojnih matematičkih funkcija koje se lako mogu primijeniti na matrice i tenzore. Korištenjem biblioteke NumPy, moguće je značajno ubrzati izvođenje koda vezanog uz neuronske mreže zbog opcija provođenja operacija na tenzorima, time često eliminirajući potrebu za korištenjem klasičnih petlji.

5.2.2. Matplotlib

Kao i NumPy, i Matplotlib je biblioteka pisana za programski jezik Python. Služi za jednostavno iscrtavanje grafičkih objekata. Osim klasičnih grafova, koristeći Matplotlib moguće je prikazati i slike (npr. slike učitane kao dio skupa podataka CIFAR-10). Uz iscrtavanje grafova i slika, koristeći Matplotlib iste možemo i s lakoćom pohraniti.

5.2.3. PyTorch

PyTorch je radni okvir pisan za programski jezik Python. Služi za laku izgradnju i učenje neuronskih mreža, kao i općenito provođenje izračuna s tenzorima. Pruža nam mogućnost automatske diferencijacije veoma značajnu za izračun gradijenta funkcije gubitka, kao i pristup brojnim optimizacijskim algoritmima, ali i slojevima neuronskih mreža koje s lakoćom možemo kombinirati za izgradnju vlastitih mreža.

Uz navedene mogućnosti, radni okvir PyTorch nudi nam podršku za korištenje grafičkih kartica za provođenje izračuna. U slučaju da na računalu imamo prikladnu grafičku karticu, kao i instaliranu prikladnu programsku podršku, koristeći PyTorch lako možemo prebacivati provođenje izračuna s procesora na grafičku karticu koja je uobičajeno specijalizirana za paralelno procesiranje. PyTorch nam također nudi i mogućnost provođenja izračuna sa 16-bitnim brojevima s pomičnim zarezom. Bitno je napomenuti da je za ovo potrebno imati grafičku karticu s tenzorskim jezgrama.

5.3. Brzo učenje s neprijateljskim primjerima - eksperimenti

Kako bismo usporedili učinkovitost različitih pristupa brzom učenju s neprijateljskim primjerima, učili smo nekolicinu mreža koristeći varirajuće hiperparametre i načine učenja. Sve naučene mreže arhitekture su ResNet18 i učene su 80 epoha. Kao mjeru gubitka kod svih mreža korištena je unakrsna entropija, a kao optimizator korišten je stohastički gradijentni spust sa zamahom (engl. *momentum*) iznosa 0.9 i propadanjem težina iznosa $5e^{-4}$. Korištenje zamaha ubrzava konvergenciju modela, a korištenje propadanja težina regularizacijska je tehnika koja služi za smanjivanje složenosti modela, a time i vjerojatnosti pojave prenaučenosti. Kako bi što bolje reproducirali originalni rad [16], za brzo učenje s neprijateljskim primjerima (*FastAdv*) korištena je cikličkа stopa učenja. Pritom stopa učenja maksimum dosegne na pola koraka učenja te se nakon toga do kraja učenja njen iznos smanjuje. Važno je napomenuti da se kod cikličke stope učenja iznos stope učenja mijenja nakon svake mini-grupe, a odozdo je ograničen iznosom 0. Sve ostale mreže koristile su stopu učenja s kosinusnim žarenjem. Pritom je maksimalan broj koraka postavljen na ukupan broj epoha, a iznos stope učenja mijenja se nakon svake epohe.

Kako bi se što više ubrzalo učenje, za učenje svih modela korišteno je računanje u mješovitoj preciznosti. Osim mreža učenih s neprijateljskim primjerima, za usporedbu je učena i mreža na prirodnom skupu podataka. Nakon učenja, za sve mreže izračunata je točnost na prirodnom skupu za testiranje, ali i točnost na neprijateljskim primjerima generiranim iz prirodnog skupa za testiranje koristeći PGD napad s 20 iteracija. Pritom je PGD napad kao hiperparametre imao ograničenje perturbacije (ϵ) iznosa $8/255$ te veličinu koraka (α) iznosa $1/255$. Uz točnost, mjereno je i vrijeme potrebno za učenje modela.

U tablici 5.1 stupac *LR* označava stopu učenja, stupac *Točnost* točnost na skupu za testiranje, a stupac *20-PGD* točnost na neprijateljskim primjerima generiranim iz skupa za testiranje koristeći PGD napad s 20 iteracija. Točnosti su prikazane postotkom, a vrijeme učenja prikazano je brojem minuta potrebnim za učenje pojedinog modela. Pritom su točnost, ali i vrijeme učenja zaokruženi na jedno decimalno mjesto. U stupcu *Način učenja*, *FreeAdv* predstavlja besplatno učenje, *FastAdv* brzo učenje, *FastAdv+* nadogradnju na brzo učenje, a *FastAdvW* nadogradnju na brzo učenje s korištenjem PGD metode zadnjih 10 epoha učenja. U slučaju da na kraju vrijednosti iz stupca *Način učenja* piše *Early*, korišteno je učenje s ranim završetkom.

Tablica 5.1: Rezultati raznih načina brzog učenja s neprijateljskim primjerima

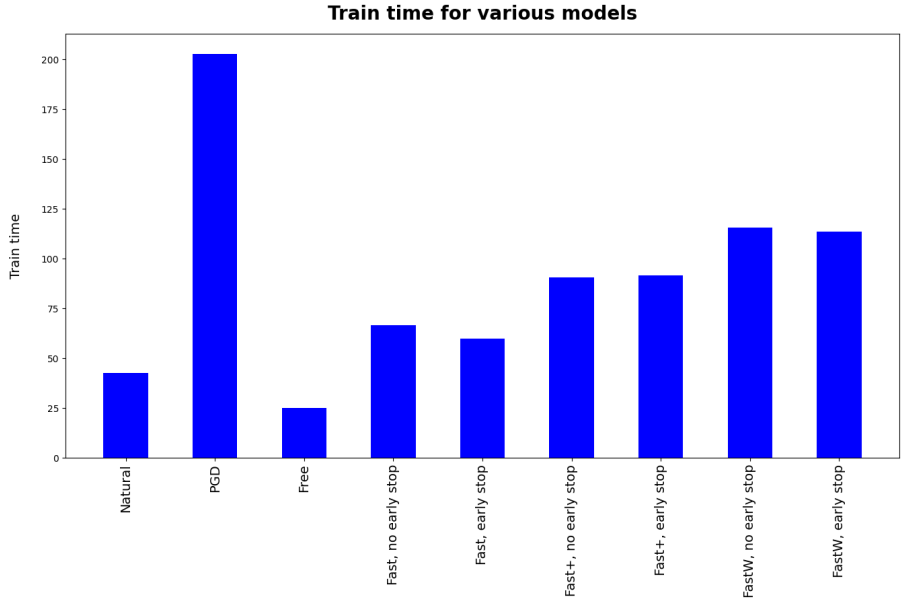
Način učenja	LR	Točnost [%]	20-PGD [%]	Vrijeme učenja [min]
Prirodno	0.01	90.4	0	43.3
Prirodno	0.02	92.3	0	42.7
PGD	0.1	83.4	43.7	202.7
FreeAdv	0.1	84.7	30.8	25
FastAdv	0.2	89.6	0	66.5
FastAdv, Early	0.2	82.2	40.2	59.8
FastAdv+	0.2	85.7	40.7	90.7
FastAdv+, Early	0.2	85.7	41.2	91.5
FastAdvW	0.2	85.1	43.2	115.5
FastAdvW, Early	0.2	85.3	42.8	113.4

Kao što možemo vidjeti u tablici 5.1, najvišu točnost na prirodnom skupu za testiranje ima model učen na prirodnom skupu podataka. Nažalost, taj model nije nimalo robustan - njegova točnost na neprijateljskim primjerima generiranim koristeći PGD metodu s 20 iteracija iznosi 0%. Kada govorimo o robusnim modelima, najvišu točnost na neprijateljskim primjerima ima model učen koristeći PGD metodu, no on se i daleko najdulje uči. Pritom su tijekom učenja s PGD metodom neprijateljski primjeri generirani koristeći PGD sa 7 iteracija.

Od načina brzog učenja s neprijateljskim primjerima izloženih u radovima [14], [16] i [8], najbolje performanse ima model učen koristeći *FastAdvW* način učenja. Njegova točnost na prirodnom skupu za testiranje druga je najviša od svih modela koji nude određenu mjeru robusnosti, a njegova točnost na neprijateljskim primjerima usporediva je s točnošću modela učenog PGD metodom. Važno je napomenuti da *FastAdvW* modeli trebaju skoro dvostruko manje vremena za učenje naspram PGD modela, ali se nažalost uče najdulje od svih predloženih varijanti brzog učenja. S obzirom da *FastAdvW* modeli zadnjih 10 epoha uče koristeći isključivo PGD metodu, takav rezultat je i očekivan. Modeli učeni koristeći *FastAdv+* način učenja dobra su alternativa korištenju *FastAdvW* modela jer se uče kraće i postižu malo bolje rezultate na prirodnom skupu za testiranje. Uočimo da korištenje ranog završetka kod *FastAdv+* i *FastAdvW* modela nema nikakav učinak na performanse. Gledajući da je glavni cilj tih načina učenja sprječavanje pojave katastrofalne prenaučenosti prisutne kod *FastAdv* modela, ovakav rezultat je očekivan, a govori nam da ove nadogradnje stvarno uspijevaju spriječiti pojavu katastrofalne prenaučenosti.

U slučaju da nam je glavni cilj brzo naučiti model koji je barem u nekoj mjeri otporan na neprijateljske primjere, kao najbolja opcija ističe se učenje koristeći *FreeAdv* način učenja. *FreeAdv* model učen je koristeći ponavljanje (engl. *replay*) iznosa 8 pa je stoga ukupno učen 10 epoha - ukupan broj epoha smanjen je za faktor jednak ponavljanju. Ovaj model nudi nam konkurentnu točnost na prirodnim podacima, ali je njegova točnost na neprijateljskim primjerima niža od svih ostalih robusnih modela. U slučaju da smo dodatno povećali parametar ponavljanja, točnost na neprijateljskim primjerima povećavala bi se. Naravno, istovremeno bi padala točnost na prirodnim podacima. Dobra strana *FreeAdv* modela svakako je vrijeme učenja - ovaj model uči se kraće čak i od modela učenog na prirodnom skupu podataka. Ovo svojstvo pripisujemo činjenici da je tijekom učenja za *FreeAdv* model potrebno provesti evaluaciju nakon epoha samo 10 puta, dok je za ostale modele potrebno provesti evaluaciju punih 80 puta.

Modeli učeni *FastAdv* načinom učenja ograničeni su pojavom katastrofalne prenaučenosti. Ovo je veoma vidljivo kod *FastAdv* modela bez ranog završetka koji postiže veoma visoku točnost na prirodnom skupu podataka, ali zato nije nimalo otporan na neprijateljske primjere. Korištenjem *FastAdv* učenja u kombinaciji s ranim završetkom, točnost na neprijateljskim primjerima podjednaka je s točnošću *FastAdv+* modela, ali je ovakav model ograničen zbog nemogućnosti proizvoljno dugog učenja.



Slika 5.2: Usporedba vremena učenja raznih načina brzog učenja s neprijateljskim primjerima

Na slici 5.2 moguće je vidjeti usporedbu vremena učenja raznih načina brzog učenja s neprijateljskim primjerima. Zbog usporedbe, na grafu je prikazano i vrijeme učenja modela učenog na prirodnim podatcima, kao i vrijeme učenja modela učenog PGD metodom. Kao što je bilo vidljivo i iz tablice 5.1, daleko najdulje vrijeme učenja ima model učen PGD metodom. S druge strane, najkraće vrijeme učenja ima *FreeAdv* model. Ako uspoređujemo modele učene koristeći *FastAdv*, *FastAdv+* ili *FastAdvW* način učenja, najkraće vrijeme učenja ima *FastAdv* model s ranim završetkom. Gledajući da *FastAdv+* model pri uočavanju pada točnosti na neprijateljskim primjerima s sljedećih mini-grupa uči koristeći PGD metodu, a *FastAdvW* model uz to zadnjih 10 epoha uči koristeći isključivo PGD metodu, ovakav rezultat je i očekivan.

6. Zaključak

Zaključak.

LITERATURA

[1] Bojana Dalbelo Bašić, Marko Čupić, i Jan Šnajder. Umjetne neuronske mreže, prezentacija, 2020.

[2] Hossein Gholamalinezhad i Hossein Khosravi. Pooling methods in deep neural networks, a review. *arXiv preprint arXiv:2009.07485*, 2020.

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

[4] Ian J Goodfellow, Jonathon Shlens, i Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, i Jian Sun. Deep residual learning for image recognition. U *Proceedings of the IEEE conference on computer vision and pattern recognition*, stranice 770–778, 2016.

[6] Diederik P Kingma i Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[8] Bai Li, Shiqi Wang, Suman Jana, i Lawrence Carin. Towards understanding fast adversarial training. *arXiv preprint arXiv:2006.03089*, 2020.

[9] Yiming Li, Mengxi Ya, Yang Bai, Yong Jiang, i Shu-Tao Xia. Backdoorbox: A python toolbox for backdoor learning. *arXiv preprint arXiv:2302.01762*, 2023.

[10] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, i Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[11] Arjun Menon. Data poisoning and its impact on the ai ecosystem, 2023.

[12] Keiron O'Shea i Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[13] Farheen Ramzan, Muhammad Usman Khan, Asim Rehmat, Sajid Iqbal, Tanzila Saba, Amjad Rehman, i Zahid Mehmood. A deep learning approach for automated diagnosis and multi-class classification of alzheimer's disease stages using resting-state fmri and residual neural networks. *Journal of Medical Systems*, 44, 12 2019. doi: 10.1007/s10916-019-1475-2.

[14] Ali Shafahi, Mahyar Najibi, Mohammad Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, i Tom Goldstein. Adversarial training for free! *Advances in Neural Information Processing Systems*, 32, 2019.

[15] Leslie N Smith. Cyclical learning rates for training neural networks. U *2017 IEEE winter conference on applications of computer vision (WACV)*, stranice 464–472. IEEE, 2017.

[16] Eric Wong, Leslie Rice, i J Zico Kolter. Fast is better than free: Revisiting adversarial training. *arXiv preprint arXiv:2001.03994*, 2020.

Algoritmi za brzo učenje na neprijateljskim primjerima

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Algorithms for fast robust training on adversarial examples

Abstract

Abstract.

Keywords: Keywords.