Primjena pristupa meta-učenja kao metoda brzog učenja neuronskih mreža

Strapač, Luka

Sveučilište Josipa Jurja Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Sveučilišni diplomski studij matematike
modul: matematika i računarstvo

Primjena pristupa meta-učenja kao metoda brzog učenja neuronskih mreža

Diplomski rad

Mentor:
doc. dr. sc. Domagoj Ševerdija

Student:
Luka Strapač

Osijek, 2024

Sadržaj

1 | Uvod

Ukoliko promatramo razdoblje unazad 15 godina do danas, umjetna inteligencija je doživjela najveći uzlet, te je tema broj 1 u znanosti. Teško je pronaći sustav koji ne koristi neku vrstu umjetne inteligencije, bili to algoritmi strojnog učenja ili neuronske mreže dubokog učenja. Posebno je to u zadnjih par godina primjetno u području dubokog učenja, zahvaljujući sve složenijim mrežnim arhitekturama, iznimnom napretku u razvoju grafičkih kartica i velikim skupovima podataka. Tako su modeli dubokog učenja postali glavni alat za rješavanje zadataka iz područja obrade prirodnog jezika, prepoznavanja govora, računalnog vida i još nekolicine drugih. Recept za uspjeh u rješavanju zadatka se sveo na dobar model i dovoljno velik skup podataka.

Međutim, postavlja se pitanje što ako se pred model postavi zadatak s kojim se tijekom faze treniranja nije susreo. S velikom sigurnošću možemo tvrditi da ga neće moći riješiti bez ponovnog treniranja, što je vremenski vrlo zahtjevno. Razlog tome je problem loše generalizacije na novi zadatak, s kojim se bore i današnji modeli. Motivirani ovim ozbiljnim problemom i nedostatkom modela, dolazimo do koncepta koji bi ga trebao riješiti zvanog meta-učenje (engl. meta-learning). Ovaj koncept se temelji na kreiranju modela koji su sposobni brzo se prilagoditi na novi zadatak, a proces prilagodbe se događa tijekom faze testiranja uz minimalan broj primjera. U ovom radu ćemo stoga detaljno proučiti ovaj koncept, te ćemo na kraju zaključiti da li meta-učenje zaista je rješenje problema generalizacije.

Struktura diplomskog rada izgleda ovako: U poglavlju 2 motivirat ćemo istraživanje koncepta meta učenja, opisati pojmove vezane uz njega, te se s njim i upoznati. Kroz poglavlje 3 detaljno ćemo opisati pristupe meta-učenja. U zadnjem, 4. poglavlju, implementirat ćemo faze meta-treniranja i meta-testiranja koristeći learn2learn paket.

2 | Meta-učenje

2.1 Motivacija

Transformer modeli su u rekordno kratkom vremenu poslali u prošlost modele koji su do tada imali najbolje rezultate, kao što su CNN i RNN. U pozadini jako puno aplikacija koristi se neki transformer model. Na primjer, kada napišemo nešto u Google tražilici i pritisnemo "Search", upisani tekst se prosljeđuje transformer modelu zvanom BERT (Bidirectional Encoder Representations from Transformers).

Prevođenje s jednog jezika na drugi, chatbot i prepoznavanje govora su primjeri zadataka koje transformer modeli vrlo uspješno rješavaju. Također, ovi modeli nalaze široku primjenu i izvan računalne znanosti i obrade prirodnog jezika. Korišteni su u medicini u svrhu medicinske dijagnostike, u financijama za otkrivanje prevara, klimatologiji za predviđanje vremena i u još puno drugih znanstvenih disciplina.

Međutim, činjenica je da u svrhu rješavanja bilo kojeg zadatka, iz bilo kojeg područja, transformer modelima treba skup podataka i treniranje na tom skupu podataka. To iziskuje puno truda (kreiranje skupa podataka), a zatim i dosta vremena (proces treniranja). Postavlja se pitanje što u situacijama u kojima je skup podataka ograničen na par primjera, ili je samo prikupljanje podataka za određeni zadatak vremenski i financijski zahtjevno. Tada bi vjerojatno htjeli da model nauči izvršavati taj zadatak sa što manje primjera. To bi nam trebao biti cilj, neovisno o raspoloživosti i veličini skupa podataka, jer to je način na koji čovjek uči - u kratkom vremenu i koristeći manji broj primjera.

Uzmimo za primjer dijete od 5 godina koje još nije imalo prilike susresti se s mačkama i psima. Pokažimo mu 2 psa i zatim 2 mačke. Potom, stavimo pred njega trećeg psa i zatražimo da nam kaže što je to. Gotovo sigurno ćemo dobiti točan odgovor. To želimo i od našeg modela - brzo i efikasno učenje na par primjera. Ako je model u mogućnosti učiti na taj način, brzo se adaptirajući na novi zadatak koji se pred njega postavlja, onda možemo reći da je dobar u generalizaciji.

U svrhu postizanja dobre generalizacije modela, koncept meta-učenja se počeo primjenjivati intenzivnije unazad par godina, iako spominjanja ovog koncepta datiraju iz prošlog stoljeća. U nastavku rada detaljno ćemo opisati ovaj koncept, navesti pristupe koji se koriste pri učenju te navesti primjere primjene koncepta u raznim situacijama.

2.2 Osnovni pojmovi

Prije upoznavanja s konceptom meta-učenja, uvest ćemo par pojmova koji će nam biti potrebni u nastavku rada.

Prijenosno učenje

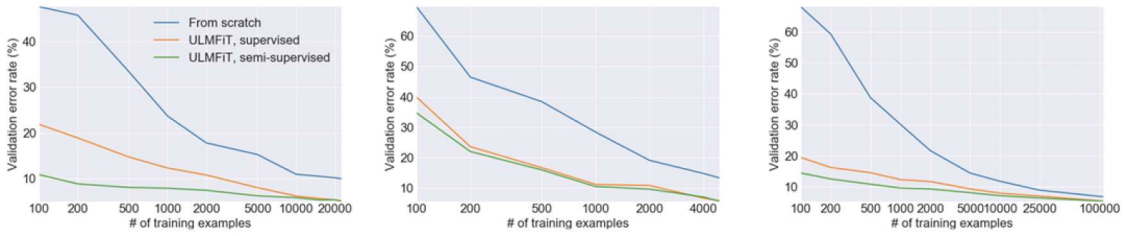
Ovaj koncept u strojnom učenju se odnosi na korištenje prenosivog znanja iz jednog zadatka u srodnim ali različitim zadacima. Sljedeća rečenica to konkretnije opisuje:

> Riješi ciljani zadatak $\mathcal{T}_b$ nakon što riješiš zadatak/zadatke $\mathcal{T}_a$, prenošenjem znanja naučenog iz $\mathcal{T}_a$.

Pri tome, za zadatak podrazumijevamo kombinaciju specifičnog skupa podataka i funkcije cilja koju model pokušava optimizirati.

Najpopularniji pristup koncepta prijenosnog učenja (engl. transfer learning) je fino ugađanje. To je tehnika kojom se koristi predtrenirani model (uglavnom predtreniran na nekom velikom korpusu) za rješavanje nekog specifičnog zadatka. Pritom se model, prije samog rješavanja zadatka, dotrenira na skupu podataka specifičnom za zadatak, koji je manji od skupa podataka na kojem je model bio predtreniran.

Cilj finog ugađanja, a samim time i prijenosnog učenja je iskoristiti znanje dobiveno predtreniranjem na većem skupu podataka. Time, umjesto treniranja modela od nule, krećemo s modelom koji ima predtrenirane parametre koji će moći uhvatiti općenitosti iz skupa podataka specifičnog za zadatak.



Slika 2.1: Fino ugađanje u ovisnosti o veličini skupa podataka na 3 različita skupa podataka (preuzeto iz [6])

Na slici 2.1 vidimo rezultat finog ugađanja na 3 različita skupa podataka. Korištena metoda prijenosnog učenja je ULMFiT (skraćeno od *Universal Language Model Fine-Tuning*). Ova metoda koristi predtrenirane jezične modele. Na njih primjenjuje fino ugađanje kako bi postigla visoke rezultate na zadacima s malim brojem označenih podataka. Plava krivulja prikazuje rezultat treniranja od nule, narančasta krivulja nadzirani predtrenirani model i zelena krivulja polunadzirani predtrenirani model.

Ono što možemo zaključiti iz sva 3 grafa je da je stopa pogreške validacije vrlo visoka na vrlo malim skupovima podataka, te da je fino ugađanje na takvim skupovima podataka loša opcija. U tim slučajevima, pokazat ćemo, meta-učenje će biti od presudne važnosti.

Višezadaćno učenje

Višezadaćno učenje (engl. multitask learning) je podkategorija koncepta prijenosnog učenja, a koja se odnosi na rješavanje srodnih zadataka istovremeno. Poboljšava generalizaciju svakog pojedinog zadatka iskorištavajući međupovezanost kroz više zadataka sa njihovim sličnostima i razlikama. Model na taj način uči istovremeno kroz više zadataka, djeleći njihove zajedničke karakteristike.

Dvije glavne tehnike koncepta višezadaćnog učenja su čvrsto i meko dijeljenje parametara.

Čvrsto dijeljenje parametara se odnosi na dijeljenje skrivenih slojeva među svim zadacima (u potpunosti dijelimo parametre među zadacima), dok je tek nekoliko izlaznih slojeva specifično zadatku (nije dijeljeno među svim zadacima). Ova tehnika značajno smanjuje mogućnost prenaučenosti na specifičan zadatak.

S druge strane, kod tehnike mekog dijeljenja parametara ne dijele se svi parametri, već samo u određenim dijelovima modela. S ovom tehnikom daje se modelu mogućnost da se prilagodi na specifičan zadatak, istovremeno iskorištavajući dijeljeno znanje.

Razlika između koncepata prijenosnog učenja i višezadaćnog učenja je što prijenosno učenje troši više vremena na ciljani, a manje na izvorni zadatak, dok višezadaćno učenje troši jednako vremena na sve zadatke.

Učenje s primjerima

Ova tehnika se fokusira na učenje s minimalnim brojem primjera, protivno principu učenja modela na velikim skupovima podataka. Učenje bez i s jednim primjerom (engl. zero-shot i one-shot learning) su posebni slučajevi s 0, odnosno 1 primjerom po klasi. Slučajevi u kojima je učenje s primjerima (engl. few-shot learning) od velike pomoći su nemogućnost pristupa većoj količini podataka za određeni problem, izbjegavanje većih skupova podataka zbog velikih troškova resursa i računanja te kad se želimo brzo adaptirati na novi zadatak.

Iako se na prvi pogled koncepti učenja s primjerima i meta-učenja čine sličnima, meta-učenje je puno širi koncept, što ćemo vidjeti u nastavku rada.

2.3 Definicija

Za razliku od standardnih koncepata strojnog učenja, gdje je fokus na poboljšanju predikcije modela kroz trening na što većem skupu podataka, ovaj koncept se fokusira na poboljšanje učenja unutarnjeg algoritma modela kroz više etapa učenja. Način na koji se pokušava postići to poboljšanje može se podijeliti u 2 faze - faza osnovnog učenja i faza meta-učenja. Tijekom faze osnovnog učenja, unutarnji algoritam rješava neki zadatak, na primjer, zadatak translacije teksta. Tijekom faze meta-učenja, vanjski algoritam ažurira unutarnji tako da se vanjska funkcija cilja poboljša. Vanjska funkcija cilja može na primjer biti sposobnost generalizacije unutarnjeg algoritma. Kroz etape učenja, dakle, mijenjamo unutarnji algoritam kako bi vanjski algoritam naučio unutarnji algoritam. Zato se često uz meta-učenje veže izraz "learning to learn".

Meta-učenje se može formalno definirati s više gledišta, no dva najčešća su distribucija zadatka i optimizacija u dvije razine.

Distribucija zadatka naglašava učenje preko skupa zadataka kako bi se potakla bolja sposobnost generalizacije na svaki pojedini zadatak. Meta-učenje s ovog gledišta može biti oblikovano kao

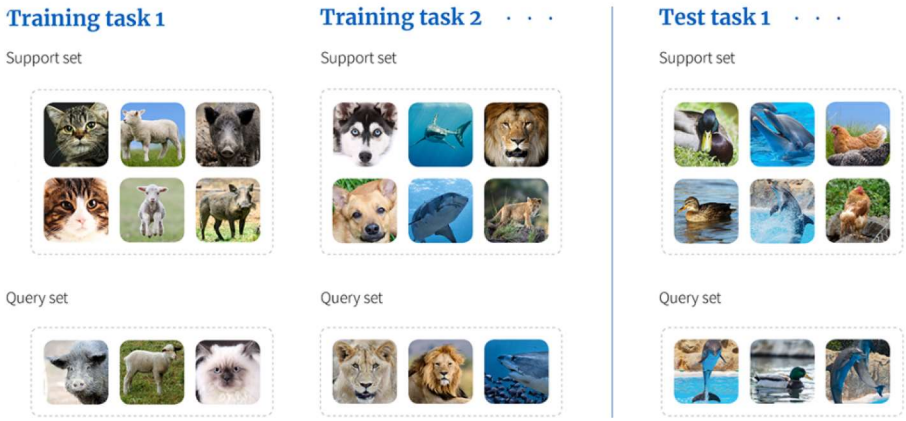
$$\min_{w} \mathbb{E}_{T\sim p(T)} \mathcal{L}(D; w) \tag{2.1}$$

gdje je w generičko meta-znanje dobiveno kroz sve zadatke, $p(T)$ distribucija svih zadataka T, a zadatak je skup koji sadrži skup podataka D i funkciju gubitka $\mathcal{L}$, odnosno, $T = \{D, \mathcal{L}\}$. $\mathcal{L}(D; w)$ mjeri učinak modela treniranog na skupu podataka D koristeći meta-znanje w.

Kao i u većini koncepata u strojnom učenju, tako i u konceptu meta-učenja postoje dvije faze: meta-treniranje i meta-testiranje. Međutim, dizajn skupova podataka je bitno drugačiji. Prilikom meta-treniranja, skup S izvornih zadataka je predstavljen kao $D_{source} = \{(D^{train}_{source}, D^{val}_{source})^{(i)}\}_{i=1}^{S}$, gdje je D^{train}_{source} potporni skup (engl. support set), a D^{val}_{source} skup za validaciju (engl. query set). U meta-testiranju skup G ciljnih zadataka je označen kao $D_{target} = \{(D^{train}_{target}, D^{test}_{target})^{(i)}\}_{i=1}^{G}$, gdje je D^{train}_{target} potporni skup, a D^{test}_{target} skup za testiranje (engl. query set).

Izraz k-way n-shot kojim definiramo postavke problema u nekom zadatku meta-učenja, označava k klasa s n primjera po klasi.

Na slici 2.2 prikazan je 3-way 2-shot zadatak klasifikacije slika. Točnije, imamo 3 klase sa po 2 primjera u potpornom skupu.



Slika 2.2: Vizualizacija skupova zadataka u meta-treniranju i meta-testiranju (preuzeto iz [16])

Korak meta-treniranja možemo računati kao

$$w^* = \arg\max_{w} \log\, p(w|D_{source}). \tag{2.2}$$

U fazi meta-testiranja naučeno meta-znanje w^* koristimo da bi trenirali model na svakom idućem ciljnom zadatku i:

$$\theta^{*(i)} = \arg\max_{\theta} \log\, p(\theta|w^*, D_{target}^{train\,(i)}). \tag{2.3}$$

Standardni koncepti strojnog učenja koriste optimizaciju u jednoj razini prilikom treniranja modela. Pod jednom razinom podrazumijevamo da imamo jednu funkciju gubitka koju trebamo optimizirati. Točnije, za neki skup podataka $\mathcal{D} = \{(x_1, y_1), ..., (x_N, y_N)\}$ treniramo model $\widehat{y} = f_\theta(x)$ tako da optimiziramo parametre modela, θ, rješavajući:

$$\theta^* = argmin_\theta \mathcal{L}(\mathcal{D}; \theta). \tag{2.4}$$

Optimizirajući parametre modela minimiziramo (optimiziramo) funkciju gubitka $\mathcal{L}$ koja mjeri razliku predviđanja modela i pravih vrijednosti izlaza za dani ulaz.

Za razliku od opisane optimizacije u jednoj razini, meta-učenje koristi optimizaciju u dvije razine. Optimizacija u dvije razine sastoji se od unutarnje petlje, koja se odnosi na treniranje na osnovnom modelu kao u standardnim konceptima strojnog ili dubokog učenja, te vanjske petlje koja označava treniranje po konceptu meta-učenja.

Iz ovoga slijedi ideja koncepta: unutarnja optimizacija ovisi o preddefiniranom pristupu učenju w, koji je preddefiniran od strane vanjske optimizacije i ne može biti promijenjen unutarnjom optimizacijom tijekom unutarnje petlje.

Korak meta-treniranja računamo na sljedeći način:

$$w^* = \arg\min_{w} \sum_{i=1}^{S} \mathcal{L}^{meta}(\theta^{*(i)}(w), w, D_{source}^{val(i)}) \tag{2.5}$$

$$\theta^{*(i)}(w) = \arg\min_{\theta} \mathcal{L}^{task}(\theta, w, D_{source}^{train(i)}) \tag{2.6}$$

gdje je $\mathcal{L}^{meta}$ vanjska, a $\mathcal{L}^{task}$ unutarnja funkcija cilja.

S gledišta optimizacije u dvije razine, w možemo promatrati kao neki hiperparametar ili parametrizaciju funkcije gubitka za unutarnju optimizaciju. w kao hiperparametar kontrolira proces meta-učenja. Tako, na primjer, može predstavljati stopu učenja u procesu meta-treniranja, ili broj koraka u ažuriranju gradijenata. w kao parametrizacija funkcije gubitka za unutarnju optimizaciju znači da je w uključen u proces optimizacije tokom meta-treniranja. Primjer takvog meta-znanja mogu biti početne vrijednosti parametara koje naučene omogućuju brzu adaptaciju na nove zadatke.

Primijetimo da smo naveli dva načina za računati korak meta-treniranja.

Prvi način, (2.2) i (2.3), se više bazira na vjerojatnosti, koristi metode kao što je Bayesov zaključak. U ovom načinu cilj je pronaći meta-parametre w takve da, primijenjene na podatke D_{source}, opisuju podatke što je moguće vjerojatnije. Vjerojatnosna funkcija $p(w|D_{source})$ korištena u (2.2) predstavlja vjerojatnost meta-parametara w uz dane izvorne podatke D_{source}. Da bi pronašli optimalne w^*,

maksimiziramo vjerojatnosnu funkciju kako je navedeno u (2.2). U fazi meta-testiranja koristimo te optimalne w^* za nove neviđene zadatke. Dakle, prvo u fazi meta-treniranja optimiziramo w na izvornim podacima D_{source}, da bi u fazi meta-testiranja tako optimizirane w koristili za optimiziranje θ (parametri osnovnog modela) za nove zadatke, kako je prikazano u (2.3).

Drugi način, odnosno, (2.5) i (2.6), je tradicionalniji, optimizacija temeljena na gradijentu. Pokušava se pronaći optimalni w minimizirajući meta-gubitak $\mathcal{L}^{meta}$ kroz skup zadataka. Parametri θ se optimiziraju na svakom zadatku i ($i \in S$, S skup zadataka) pojedinačno kojem pripada njegov specifični trening skup $D_{source}^{train(i)}$. $\mathcal{L}^{task}$ je minimiziran u prvom koraku meta-treniranja za taj trening skup. Nakon toga, u drugom koraku meta-treniranja, provodi se validacija θ nad $D_{source}^{val(i)}$ skupom, koji je ponovno, specifičan za taj zadatak i. Validacijski gubitak koji dobijemo agregira se u meta-gubitak, a cilj ovog načina je pronaći meta-parametre w^* minimizirajući taj agregirani meta-gubitak kroz zadatke, što je prikazano u (2.5).

Kroz poglavlje 3 upoznat ćemo se s raznim pristupima. Neki od njih će koristiti prvi, a neki drugi opisani način računanja koraka meta-treniranja.

2.4 Učenje u postavkama s primjerima

U potpoglavlju 2.2 smo opisali učenje s primjerima. Između ostalog, spomenuli smo da postoji i učenje bez primjera, što je posebni slučaj s 0 primjera po klasi. Logično pitanje za postaviti je, na koji točno način model može naučiti iz 0 primjera po klasi. Može, zahvaljujući meta-podacima koje model meta-učenja ima o toj klasi i iz kojih će moći učiti.

Rekli smo da je jedan od glavnih ciljeva koncepta meta-učenja, učenje na što manje primjera po klasi. Kako bi naš model učio na što manje primjera, moramo ga na to natrenirati (naučiti).

Stoga, ukoliko imamo neki skup podataka $\mathcal{D}$, iz njega ćemo uzorkovati po par primjera iz svake klase i spremiti ih u potporni skup. Isto tako, par primjera spremamo u validacijski skup. Zatim treniramo model na potpornom skupu, te nakon treniranja testiramo na validacijskom skupu. Postupak ponavljamo više puta kako bi postigli da model nauči kako učiti iz manjeg skupa, odnosno, na manje primjera.

3 | Pristupi

U ovom poglavlju navest ćemo i opisati pristupe meta-učenja koji će, prema [5], biti podijeljeni u 3 kategorije:

1. *pristupi temeljeni na modelu* - učenje u vanjskoj petlji primjenjuje se na meta-znanje w koje sadrži parametre modela; koriste se modeli s memorijom kako bi pohranili informacije iz više zadataka

2. *pristupi temeljeni na metrici* - učenje u vanjskoj petlji odgovara učenju metrike; učenje u unutarnjoj petlji ne zahtijeva učenje novih parametara za novi zadatak, koristi se naučena metrika za usporedbu podataka u danom skupu

3. *pristupi temeljeni na optimizaciji* - tretiraju zadatke unutarnje petlje kao optimizacijski problem i usredotočeni su na izvlačenje meta-znanja w kroz te zadatke u svrhu poboljšanja optimizacijskog algoritma

3.1 Pristupi temeljeni na modelu

Fokus ovih pristupa je na dizajniranju modela koji će nam omogućiti efikasno i brzo ažuriranje parametara kroz par trening koraka.

Postoje 2 pristupa, prvi temeljen na Neuronskom Turingovom stroju (prema [11]), a drugi temeljen na korištenju dviju vrsta težina - spore i brze - u svrhu dohvaćanja stare memorije (prema [8]). U sljedećem pododlomku ćemo opisati prvi pristup te ga u odlomku 4 i implementirati.

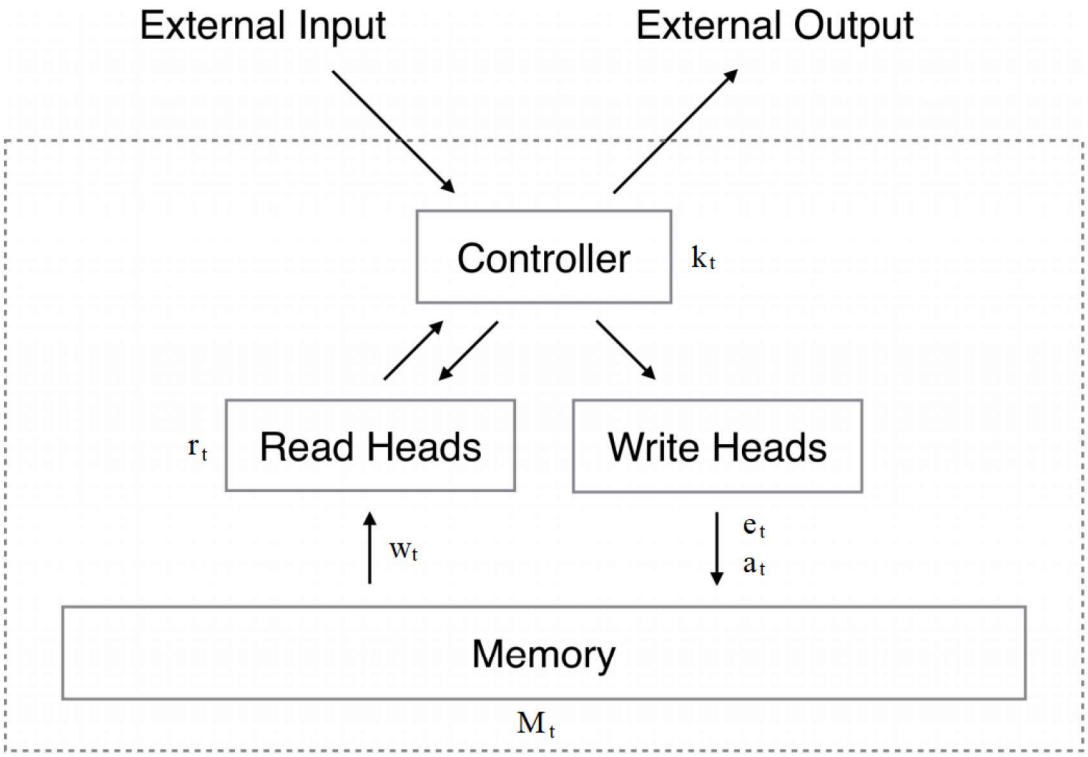
3.1.1 Neuronske mreže s proširenim pamćenjem (MANN)

U centru ovog pristupa nalazi se vanjska memorija, koja daje dodatnu snagu modelu u procesu prilagodbe na novi zadatak, na način da novu informaciju akumulira u radnu memoriju.

Neuronski Turingov stroj

Neuronski Turingov stroj (eng. Neural Turing machine - u nastavku NTM) je neuronska mreža koja je pomoću mehanizama baziranih na pozornosti spojena s vanjskom memorijom te je s njom u interakciji. Inspirirana Turingovim strojem, NTM koristi glave čitanja (pisanja) u radu s memorijom u svrhu čitanja (pisanja) memorije na određenu lokaciju, a manje na ostala mjesta. Opisano je prikazano

na slici 3.1, uz napomenu da je svaka komponenta arhitekture sa slike diferencijabilna.



Slika 3.1: NTM arhitektura (preuzeto iz [4]; izmijenjeno)

Težine pozornosti su određene s mehanizmima adresiranja temeljenih na sadržaju, odnosno lokaciji. Mehanizam temeljen na sadržaju fokusira se na lokacije na način da računa sličnost između memorijskih vrijednosti u trenutku t i vrijednosti koju traži neuronska mreža (upiti nad memorijom) u tom trenutku. Što je veća sličnost vrijednosti (sadržaja), to je veća težina pozornosti dodijeljena određenoj lokaciji. Za računanje sličnosti između tih vrijednosti koristimo kosinusovu sličnost koju definiramo s

$$K(k_t, \mathrm{M}_t(i)) = \frac{k_t \cdot \mathrm{M}_t(i)}{||k_t||\ ||\mathrm{M}_t(i)||}, \tag{3.1}$$

gdje je M_t matrica memorije dimenzije $N \times M$ koja označava sadržaj memorije u trenutku t, a k_t ključ generiran od strane upravljača (engl. controller) što može biti neuronska mreža. S t označavamo trenutak u vremenu, a $\mathrm{M}_t(i)$ je i-ti redak matrice, $\forall i \in \{1, ..., N\}$.

Ovaj mehanizam generira normaliziranu težinu w_c^t kroz kosinusovu sličnost K između vektora ključa k_t i vektora $\mathrm{M}_t(i)$, gdje β_t označava pozitivnu jačinu ključa:

$$w_c^t(i) \leftarrow \frac{exp(\beta_t K[k_t, \mathrm{M}_t(i)])}{\sum_j exp(\beta_t K[k_t, \mathrm{M}_t(j)])}$$

S druge strane, mehanizam temeljen na lokaciji kontrolira iteracije kroz lokacije u memoriji.

Vektor čitanja r_t dobivamo kao konveksnu kombinaciju vektora $\mathrm{M}_t(i)$ s težinom i-tog elementa $w_t(i)$ u memoriji:

$$r_t \leftarrow \sum_i w_t(i)\mathrm{M}_t(i),$$

gdje je $\sum_i w_t(i) = 1, 0 \leq w_t(i) \leq 1, \forall i$. w_t je vektor težina na N lokacija u trenutku t. N je broj memorijskih lokacija, a M veličina vektora na svakoj lokaciji. Dobiveni vektor r_t je veličine M, a sadrži težinske kombinacije vektora $\mathrm{M}_t(i)$.

Pisanje je podijeljeno u 2 dijela - brisanje i dodavanje. U koraku brisanja, uz danu težinu w_t postavljenu glavom pisanja u trenutku t zajedno sa vektorom brisanja e_t, sadržaj memorije M_{t-1} iz prošlog vremenskog koraka je promijenjen na sljedeći način:

$$\tilde{\mathrm{M}}_t(i) \leftarrow \mathrm{M}_{t-1}(i)[1 - w_t(i)e_t].$$

U koraku dodavanja, glava pisanja proizvede vektor dodavanja a_t koji se koristi da bi se napravile promjene u memoriji nakon koraka brisanja:

$$\mathrm{M}_t(i) \leftarrow \tilde{\mathrm{M}}_t(i) + w_t(i)a_t.$$

Da bi enkodiranje novih informacija bilo brzo, trebaju biti ispunjena 2 nužna uvjeta:

1. reprezentacija podataka koji su pohranjeni u memoriji je stabilna (za pouzdano pristupanje memoriji, mora ostati nepromijenjena kroz vremenske trenutke, na primjer korištenje konzistentnih vektorskih prostora) i adresabilna po elementu (za selektivno pristupanje relevantnim podacima)

2. broj parametara ne ovisi o veličini memorije.

Navedeni uvjeti isključuju korištenje neuronskih mreža kao što su RNN i LSTM. Razlog tome je da ove neuronske mreže nemaju vanjsku nego unutarnju memoriju koja se konstantno ažurira te je za očekivati i da se reprezentacija podataka promijeni. Također, sve informacije su sadržane u jednoj ćeliji, pa je dohvaćanje specifičnih elemenata memorije vrlo kompliciranо. Posljednji razlog, povećanje kapaciteta memorije obično zahtjeva povećanje broja LSTM jedinica, što znači veći broj parametara mreže.

Motivirani traženjem novih pogodnih modela, a inspirirani NMT neuronskom mrežom dolazimo do prvog pristupa meta-učenja baziranog na modelu - neuronske mreže s proširenim pamćenjem (Memory-augmented neural network - u nastavku MANN).

Neuronske mreže s proširenim pamćenjem

Ta arhitektura opremljena resursima vanjske memorije ima mogućnosti brzog i efektivnog enkodiranja i dohvaćanja informacija, kao i raditi točna predviđanja te prilagoditi se novim zadacima u postavkama učenja s primjerima.

Kako MANN spada u model meta-učenja, zadatak mu je naučiti meta-parametre u svrhu minimiziranja funkcije troška kroz više zadataka, te naći parametar θ koji će smanjiti očekivanu funkciju troška kroz distribuciju skupova podataka $p(y_t|x_t, D_{1:t-1}; \theta)$. Stoga, i iz (2.1) i (2.6) slijedi:

$$\theta^* = \text{argmin}_\theta \mathbb{E}_{D\sim p(D)}[\mathcal{L}(D;\theta)],$$

gdje je $D = \{d_t\}_{t=1}^T = \{(x_t, y_t)\}_{t=1}^T$ tako da

$$(x_1, null), (x_2, y_1), ..., (x_T, y_{T-1}). \tag{3.2}$$

$p(D)$ označava distribuciju skupova podataka, a $p(y_t|x_t, D_{1:t-1}; \theta)$ je prediktivna distribucija koja predstavlja vjerojatnosnu distribuciju mogućih vrijednosti za y_t uz dani trenutni ulaz x_t, podatke iz prethodnih koraka $D_{1:t-1}$ u svrhu razumijevanja obrazaca, te parametre modela θ. U (3.2) možemo primijetiti da je y_{t-1} pomaknut za jedan korak, tako da je y_{t-1} odgovarajuća oznaka u vremenu $t-1$, a također i dio ulaza u vremenu t.

Iz tog razloga, sustav zadržava trenutni ulaz u memoriji, dok se ne prikaže ispravna oznaka u sljedećem vremenskom koraku. Kada se prikaže, dohvaćamo pripadnu staru oznaku kako bi napravili dobro predviđanje. Kako bi dohvatili staru memoriju, računamo kosinusovu sličnost prema (3.1).

Vektor čitanja, u oznaci r_t, s vektorom težine čitanja w_t^r računamo na sljedeći način:

$$r_t \leftarrow \sum_i w_t^r(i)\mathrm{M}_t(i),$$

a vektor težine čitanja:

$$w_t^r(i) \leftarrow \frac{exp(K(k_t,\mathrm{M}_t(i)))}{\sum_j exp(K(k_t,\mathrm{M}_t(j)))}.$$

U opisanom NTM memorija je bila adresirana i prema sadržaju i prema lokaciji. Taj način adresiranja je imao prednosti kod zadataka predikcije baziranih na nizu. Kod drugih zadataka, ovaj način nije optimalan. Iz tog razloga, MANN uvodi novi način pisanja u memoriju, zvan modul najmanje nedavno korištenog pristupa (LRUA), koji je pisač u memoriju potpuno temeljen na sadržaju.

Zamjena predmemorije, tehnika koja je motivirala LRUA, adresira ili najmanje korištenu ili najskorije korištenu memoriju. LRUA adresira memoriju na dvije lokacije - ili najmanje korištena memorijska mjesta za često korištene informacije ili najskorije korištena memorijska mjesta koja se neće uskoro koristiti (ažuriranje memorije s novim, možda i bitnijim informacijama).

Vektor pisanja, w_t^w, odražava interpolaciju između ove dvije opcije adresiranja. Tu interpolaciju možemo izraziti pomoću sigmoid vrata tako da izračunamo konveksnu kombinaciju čitanih težina i najmanje čitanih težina u prošlom vremenskom koraku:

$$w_t^w \leftarrow \sigma(\alpha) w_{t-1}^r + (1 - \sigma(\alpha)) w_{t-1}^{lu},$$

gdje je α hiperparametar, a w_{t-1}^{lu} najmanje čitane težine u prošlom vremenskom koraku.

Težine upotrebe, w_t^u, su ažurirane u svakom vremenskom koraku smanjivanjem prethodnih težina upotrebe i dodavanjem trenutnih težina čitanja i pisanja:

$$w_t^u \leftarrow \gamma w_{t-1}^u + w_t^r + w_t^w,$$

gdje je γ parametar smanjivanja.

Najmanje korištena težina, u oznaci w_t^{lu}, izražena je težinom upotrebe w_t^u u vremenu t:

$$w_t^{lu}(i) = \begin{cases} 0, & \text{if } w_t^u(i) > m(w_t^u, n) \\ 1, & \text{if } w_t^u(i) \leq m(w_t^u, n) \end{cases},$$

gdje je $m(v, n)$ n-ti najmanji element vektora n.

Prije pisanja u memoriju, najmanje korištena memorijska lokacija se izračuna iz w_{t-1}^u i postavi na 0. Nakon toga, pisanje u memoriju dolazi u skladu s izračunatim vektorom pisanja:

$$\mathrm{M}_t(i) \leftarrow \mathrm{M}_{t-1}(i) + w_t^w(i) k_t, \forall i.$$

3.2 Pristupi temeljeni na optimizaciji

Optimizacijski algoritmi temeljeni na gradijentu su najčešće korišteni optimizacijski algoritmi u neuronskim mrežama, koji mogu efikasno obrađivati velike skupove podataka. Međutim, da bi se stekao optimalan rezultat, potrebno je proći kroz puno iterativnih koraka sa mnoštvom primjera što uzrokuje sporu konvergenciju. Glavna ideja pristupa temeljenih na optimizaciji je rješavanje ovog problema, odnosno, brza konvergencija na par trening primjera uz održavanje dobre sposobnosti generalizacije.

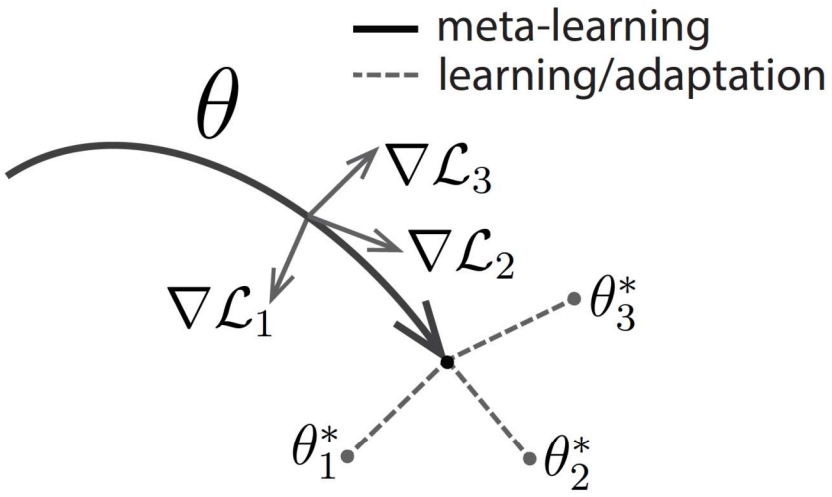
Tri su najpoznatija pristupa temeljena na optimizaciji. Prvi, LSTM meta-učenik, temeljen na LSTM mreži (prema [10]), drugi, model-agnostic meta-learning (MAML) inspiriran tehnikom finog ugađanja (prema [3]), te treći pristup, Reptile, kao poboljšana varijacija MAML pristupa (prema [9]). Kroz rad ćemo se teorijski upoznati s MAML te ćemo ga na kraju i iskoristiti u implementaciji.

3.2.1 MAML

MAML (engl. *Model-agnostic meta-learning*) je pristup meta-učenja koji je neovisan o modelu u smislu da će funkcionirati na svim modelima koji u procesu treniranja koriste gradijentni spust. Jedini uvjet je da funkcija gubitka bude dovoljno glatka kako bi se mogla optimizirati nekim pristupom baziranim na gradijentu. Cilj ovog pristupa je treniranje modela na način da ga nakon završetka treniranja bude lako fino ugoditi (engl. fine-tune) na novim zadacima, istovremeno izbjegavajući prenaučenost (engl. overfitting).

Način na koji to pokušava postići je maksimiziranjem osjetljivosti funkcija gubitaka novih zadataka s obzirom na parametre. MAML dijeli parametre modela s obzirom na njihovu osjetljivost u različitom broju zadataka i trenira one koji su osjetljivi na promjene u zadatku. Rezultat toga je da male promjene u zadacima dovode do znatnog poboljšanja u funkciji gubitka kroz različite zadatke i signaliziraju da se promijeni smjer gradijentnog spusta.

Na slici 3.2 vizualno je prikazano kako MAML postiže brzu adaptaciju na novi zadatak kroz reprezentaciju θ.



Slika 3.2: MAML pristup (preuzeto iz [3])

Opišimo oznake sa slike 3.2.

1. θ predstavlja inicijalne parametre modela koji se uče u fazi meta-treniranja, u koraku meta-optimizacije. Iz tog razloga, zovemo ih i meta-parametrima, te služe kao dobra početna točka osnovnom učeniku. Također su dijeljeni među svim zadacima $\mathcal{T}_i$ tokom faze meta-treniranja, a kasnije i u fazi meta-testiranja.

2. $\nabla\mathcal{L}_1$, $\nabla\mathcal{L}_2$ i $\nabla\mathcal{L}_3$ predstavljaju gradijente funkcija gubitka od 3 različita zadatka, po konvenciji, $\mathcal{T}_1$, $\mathcal{T}_2$ i $\mathcal{T}_3$. Ukazuju na promjenu funkcije gubitka zadatka s obzirom na parametre θ, te se iz tog razloga koriste za prilagodbu parametara, kako bi parametri odgovarali svim zadacima najbolje moguće.

3. θ_1^*, θ_2^* i θ_3^* optimalni su parametri specifični zadacima, po konvenciji, $\mathcal{T}_1$, $\mathcal{T}_2$ i $\mathcal{T}_3$. Finim ugađanjem inicijalnih parametara θ na pojedini zadatak želimo se približiti što bliže tim parametrima. Približavanjem tim parametrima ćemo minimizirati gubitke, po konvenciji, $\mathcal{L}_1$, $\mathcal{L}_2$ i $\mathcal{L}_3$.

4. "meta-learning" krivulja označava putanju parametara θ tokom faze meta-treniranja. Oblik krivulje definiraju gradijenti zadataka s kojima se model susreće u procesu meta-učenja.

5. "learning/adaptation" isprekidane linije predstavljaju učenje specifično zadatku, odnosno prilagodbu na zadatak. U ovoj fazi učenja inicijalne parametre θ želimo, kako je opisano u točki 3., dovesti dovoljno blizu, po konvenciji, θ_1^*, θ_2^* i θ_3^* djelujući gradijentnim spustom na funkcije gubitka $\mathcal{L}_1$, $\mathcal{L}_2$ i $\mathcal{L}_3$.

Nadalje, pretpostavimo da želimo pronaći skup parametara θ koji su vrlo prilagodljivi, u smislu da s minimalno koraka ažuriranja (tih parametara) na novom zadatku postižemo visoke rezultate. Tijekom faze meta-učenja (na slici 3.2 podebljana strelica), MAML optimizira za skup parametara tako da kada se primjenjuje gradijentni korak za određeni zadatak $\mathcal{T}_i$ (na slici 3.2 isprekidane linije), parametri će biti blizu optimalnih parametara θ_i^* za zadatak $\mathcal{T}_i$.

Ukoliko imamo model reprezentiran s parametriziranom funkcijom f_θ i parametrima θ koji se treba adaptirati na novi zadatak $\mathcal{T}_i$, s nekim skupom podataka $(D_{train}^{(i)}, D_{test}^{(i)})$, parametri modela θ postaju θ_i'. Tako prilagođeni parametri modela se ažuriraju kroz jedan ili više gradijentnih koraka na zadatku $\mathcal{T}_i$. Jedan korak ažuriranja se računa na sljedeći način:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta), \tag{3.3}$$

gdje je α fiksan hiperparametar ili meta-naučen.

Parametri modela su trenirani optimizirajući $f_{\theta_i'}$, s obzirom na θ kroz zadatke uzorkovane iz $p(\mathcal{T})$. Funkcija meta-cilja za optimizaciju na više zadataka $\mathcal{T}_i$ je dana s:

$$\theta^* = \min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) = \min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}) \tag{3.4}$$

Meta-optimizacija se izvršava preko stohastičkog gradijentnog spusta (engl. stochastic gradient descent, u nastavku SGD). Ažuriraju se početni parametri θ računanjem gradijenata s obzirom na optimalne parametre iz prethodnog koraka:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}), \tag{3.5}$$

gdje je β veličina meta-koraka. Opisano sada možemo uklopiti u algoritam.

Napomenimo odmah jednu bitnu činjenicu koja je implicitno dana u algoritmu 1, ukoliko razmišljamo o načinu kako meta-učenje funkcionira. Iako se radi o istim zadacima u linijama 10 i 11 algoritma, ti zadaci ne koriste iste primjere. Naime, ukoliko se vratimo na potpoglavlje 2.3, sliku 2.2 možemo uočiti da svaki zadatak ima potporni i testni skup. Potporni skup koristi se u algoritmu u koraku adaptacije (linija 10) gdje se eksplicitno navodi broj primjera K. Testni skup koristi se u meta-ažuriranju implicitno. Ta dva skupa su odijeljena i imaju potpuno različite primjere, te se zbog toga primjeri koji su viđeni u fazi adaptacije neće pojaviti u fazi meta-ažuriranja. Time meta-učenje pokušava simulirati scenarije iz stvarnog života u kojima stečeno znanje koristimo u rješavanju nekih novih zadataka.

Algoritam 1 MAML - generalna trening metoda

1: **Ulaz** : $p(\mathcal{T})$, distribucija zadataka
2: **Ulaz** : α, β hiperparametri za veličinu koraka
3: **Ulaz** : D, skup podataka za treniranje
4: **Izlaz** : θ, ažurirani parametri
5: $\theta \leftarrow$ nasumična inicijalizacija
6: **while** ne ispunjava kriterij zaustavljanja **do**
7: Nasumično izaberi nakupinu zadataka $\mathcal{T}_i$ iz $p(\mathcal{T})$
8: **for** svaki $\mathcal{T}_i$ **do**
9: Izračunaj $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ s obzirom na K trening primjera
10: Izračunaj adaptirane parametre preko gradijentnog spusta $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
11: Ažuriraj inicijalne parametre θ s $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$

Dakle, potporni skup koristi se za adaptaciju na specifičan zadatak, a testni skup za evaluaciju generalizacije kroz zadatke.

Primijetimo da se stohastičnost u meta-optimizaciji odnosi na uzorkovanje zadataka iz distribucije $p(\mathcal{T})$ (linija 7 u algoritmu 1).

Promotrimo liniju 11. algoritma 1 u kojoj se odrađuje korak meta-ažuriranja s formulom 3.5. Prilikom računanja gradijenta $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ deriviramo gubitak specifičan zadatku po θ. Međutim, gubitak $\mathcal{L}_{\mathcal{T}_i}$ se računa s obzirom na parametre θ_i' koji ovise o θ u formuli za računanje adaptiranih parametara (vidi 3.3) u liniji 10. Ta ovisnost parametara θ_i' o θ dovodi do potrebe za derivacijom drugog reda. Naime, računajući $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ diferenciramo kroz formulu za adaptaciju parametara iz linije 10. To znači da moramo izračunati $\frac{\partial}{\partial \theta}(\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta))$ što je derivacija drugog reda.

Poznato je da su derivacije drugog reda izrazito računalno zahtjevne (računanje Hessijanove matrice - parcijalnih derivacija drugog reda), pogotovo u dubokom učenju kad modeli imaju veliki broj parametara. Posljedično, meta-treniranje MAML metodom može biti jako sporo. Također, javlja se potencijalni problem s memorijom jer svaki prolazak kroz mrežu zahtijeva pamćenje gradijenta u svrhu kasnije propagacije unazad. Ipak, najveći problem derivacija drugog reda je usporena i nestabilna konvergencija, do koje dolazi jer računalno kompleksnije operacije uzrokuju šumove u procjeni gradijenata.

Spomenimo ukratko metode nastale kako bi ponudile rješenje problema druge derivacije u MAML metodi.

Prva je FOMAML (engl. First-Order MAML) koja je pojednostavljena verzija MAML iz razloga što kompletno ignorira derivaciju drugog reda. Ova metoda u koraku meta-ažuriranja aproksimira gradijent na način da računa gradijent s obzirom na θ_i', umjesto θ. Odnosno,

$$\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \approx \nabla_{\theta_i'} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.6}$$

Iako je FOMAML aproksimacija, pokazalo se da u praksi funkcionira dobro.

Druga metoda je Reptile, koja također izbjegava derivaciju drugog reda. U koraku meta-ažuriranja, umjesto računanja gradijenta funkcije gubitka s obzirom na

parametre $\theta_i^{'}$, Reptile ažurira inicijalne parametre kao razliku između trenutnih inicijalnih parametara i adaptiranih parametara:

$$\theta \leftarrow \theta + \epsilon(\theta_i^{'} - \theta) \tag{3.7}$$

ϵ je hiperparametar veličine koraka. Na taj način, Reptile "pomiče" inicijalne parametre prema parametrima specifičnim zadatku. Ideja iza ove metode je da će parametri specifični zadatku nakon određenog broja koraka adaptacije na zadatak biti vrlo blizu optimalnim parametrima za taj zadatak. Smatra se da, iako različiti, zadaci dolaze iz iste distribucije te imaju neke sličnosti koja će biti dovoljna za dobru generalizaciju kroz distribuciju zadataka. Također, pomicanje inicijalnih parametara kroz više zadataka prema $\theta_i^{'}$ kreirat će skup parametara koji nisu previše specijalizirani za nijedan zadatak zasebno, nego su u nekoj sredini zadataka odakle se mogu brzo adaptirati na bilo koji zadatak iz distribucije.

3.3 Pristupi temeljeni na metrici

Inspirirani algoritmom k-najbližih susjeda (k-NN) i K-means klasteriranjem, ovi pristupi generiraju težine preko kernel funkcije mjerenjem udaljenosti između dva uzorka. $P_\theta(y|x)$ je modelirana preko skupa poznatih oznaka y kao težinska suma oznaka iz primjera pomoćnog skupa:

$$P_\theta(y|x, S) = \sum_{(x_i, y_i) \in S} k_\theta(x, x_i) y_i$$

Ispravna metrika udaljenosti ključna je za performanse modela. Ona bi trebala predstavljati odnos između ulaza u prostoru zadatka i pomoći u rješavanju problema.

U pristupe temeljene na metrici najčešće se ubrajaju: Konvolucijska sijamska neuronska mreža, prvi pristup temeljen na metrici s mrežama blizancima (identične) i unakrsnom entropijom kao funkcijom gubitka (prema [7]). Podudarajuće mreže koje su neparametarski modeli (prema [15]), Prototipske mreže koje uče prototipsku reprezentaciju svake klase na osnovu koje klasificiraju nove podatke (prema [12]), te Relacijska mreža (prema [13]). U ovom radu teorijski ćemo obraditi Prototipske mreže te ih iskoristiti u implementaciji.

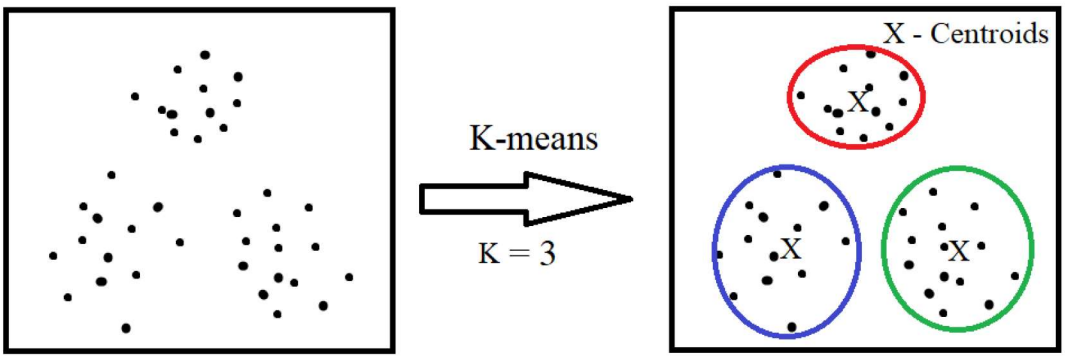
3.3.1 Prototipske mreže

Motivirane centroidima u K-means algoritmu, uvedene su Prototipske mreže za rješavanje zadataka u postavkama sa i bez primjera učenjem udaljenosti do prototipske reprezentacije svake klase.

Prije nego krenemo istraživati prototipske mreže, prođimo kroz K-means algoritam, centroide i kako su prototipske mreže motivirane s navedenim.

K-means je algoritam nenadziranog učenja koji podatke pokušava podijeliti u K klastera (grupa), minimizirajući varijancu u svakome klasteru. Na taj način K-means neoznačene podatke grupira u označene klastere, što je prikazano na

slici 3.3. Cilj algoritma je grupirati slične podatkovne točke tako da su sve unutar jednog klastera što bliže moguće, pri tom izbjegavajući dodirivanje klastera.



Slika 3.3: Grafički prikaz *K*-means algoritma

Centar klastera naziva se centroid, a predstavlja srednju vrijednost svih podatkovnih točaka unutar jednog klastera.

Algoritam 2 *K*-means algoritam

Ulaz : Podatkovne točke $X = \{x_1, ..., x_n\}$, gdje je svaki x_i D-dimenzionalni vektor
Ulaz : K, broj klastera
Ulaz : max_iters, maksimalni broj iteracija
Izlaz : centroidi $C = \{c_1, ..., c_K\}$, pridružen klaster za svaki $x_i \in X$
Inicijalizacija:
    Nasumično odaberi K točaka iz skupa podataka kao inicijalne centroide: $C = \{c_1, c_2, ..., c_K\}$
repeat
    Assignment Step:
    for $x_i \in X$ **do**
        for $c_j \in C$ **do**
            Izračunaj udaljenost između x_i i svakog centroida $c_j \in C$
        Pridruži x_i klasteru najbližeg centroida
    Update Step:
    for $j = 1$ to K **do**
        Preračunaj centroid c_j kao srednju vrijednost svih podatkovnih točaka pridruženih klasteru j

$$c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i \quad \text{gdje je} \quad C_j = \text{skup točaka pridruženih klasteru } j$$

until centroidi se ne mijenjaju (ili se mijenjaju vrlo malo) između iteracija ili se dostigao max_iters

U algoritmu 2 predstavljen je pseudokod *K*-means algoritma. Ukratko, *K*-means počinje s inicijalizacijom nasumično odabranih K točaka koje predstavljaju centre klastera. U koraku pridruživanja se svaka podatkovna točka (predstavlja jedan podatak u skupu podataka) pridjeljuje najbližem centroidu po udaljenosti, najčešće Euklidskoj. Nakon što je svaka podatkovna točka pridružena pripadajućem klasteru, formirano je K klastera. U koraku ažuriranja za svaki klaster preračunavamo vrijednost centroida s obzirom na srednju vrijednost podatkovnih

točaka pridruženih klasteru. Nova vrijednost centroida postaje novi centar klastera. Korake pridruživanja i ažuriranja ponavljamo dok nije zadovoljen neki od kriterija, najčešće nepromijenjenost centroida između iteracija.

Centroidi naučeni tokom ovog iterativnog postupka reprezentiraju karakteristike njima pripadnog klastera. Nove podatkovne točke klasificiraju se na način da ih se pridruži njima najbližem centroidu (po istoj udaljenosti kao i u algoritmu 2).

Prijeđimo sada na motivaciju za Prototipske mreže. Spomenuli smo da su one motivirane centroidima u K-means algoritmu. Temeljna ideja iza Prototipskih mreža je da postoji prostor ugrađivanja u kojemu su uzorci iste klase klasterirani, odnosno grupirani, oko jedne prototipske reprezentacije. Objasnimo pojmove prostor ugrađivanja i prototipska reprezentacija, kako bi mogli uvidjeti sličnost sa centroidima u K-means algoritmu.

Prostor ugrađivanja je nižedimenzionalni vektorski prostor u koji se mapiraju visokodimenzionalni podaci. Ovaj prostor otkriva semantičku vezu među podacima te ih s obzirom na to i mapira blizu ako su slični, odnosno dalje ako su različiti.

Prototip, odnosno prototipska reprezentacija, u kontekstu Prototipskih mreža označava vektor u prostoru ugrađivanja koji reprezentira cijelu klasu. Dobiven je kao srednja vrijednost svih ugrađenih primjera te klase u potpornom skupu.

Jasno uočavamo sličnost između pojma centroid iz K-means algoritma te pojma prototip iz Prototipskih mreža, iako autori u radu [12] ne spominju K-means u kontekstu Prototipskih mreža.

U nastavku ovog odlomka usredotočit ćemo se na Prototipske mreže. Spomenuli smo na početku odlomka da je ova metoda stvorena u svrhu rješavanja zadataka sa i bez primjera učenjem udaljenosti. Preciznije, ova metoda uči metrički prostor u kojem bi se mogla izvršiti klasifikacija računanjem udaljenosti do prototipske reprezentacije svake klase.

Za mapiranje nelinearnih veza ulaznih točaka u prostor ugrađivanja modeli ovog pristupa koriste duboke neuronske mreže. Nakon što su točke mapirane u prostor ugrađivanja, izračuna se prototip svake klase iz ulaza (a ulaz je potporni skup koji sadrži par primjera po svakoj klasi).

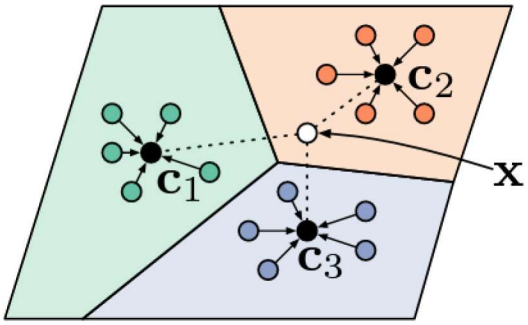
Na slici 3.4 c_k predstavlja prototip klase k, a dobiven je kao sredina potpornog skupa klase k u prostoru ugrađivanja, za $k = 1, 2, 3$. Klasifikacija nove točke, koja je prethodno mapirana u prostor ugrađivanja, na slici 3.4 označene s x, izvršava se tako da se nađe prototip najbliže klase. Konkretno u slučaju sa slike 3.4 radi se o prototipu klase 2, označenom sa c_2.

U postavkama bez primjera svaka klasa sadrži meta-podatke koji opisuju klasu na višoj razini. Tako će ti meta-podaci poslužiti umjesto primjera za računanje prototipa. Konkretno, c_k je dobiven kao ugrađeni meta-podaci v_k, svake klase k, kako je prikazano na slici 3.5. Nadalje se klasifikacija provodi kao i u postavkama s primjerima, traženjem prototipa najbliže klase.

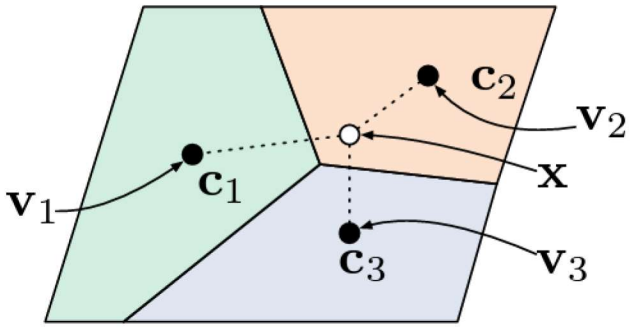
U oba slučaja, nove točke su klasificirane primjenom softmax funkcije nad udaljenosti do prototipa klasa.

Prototip svake klase $c_k \in \mathbb{R}^M$, gdje je M dimenzija prototipa, računamo preko funkcije ugrađivanja

$$f_\phi : \mathbb{R}^D \rightarrow \mathbb{R}^M.$$



Slika 3.4: Klasifikacija u postavkama s primjerima (preuzeto iz [12])



Slika 3.5: Klasifikacija u postavkama bez primjera (preuzeto iz [12])

Ova funkcija je zadužena za mapiranje D-dimenzionalnih ulaznih podataka u M-dimenzionalni prostor ugrađivanja. Ona nije unaprijed definirana, nego ju neuronska mreža uči u fazi meta-treniranja. Stoga, parametri ϕ predstavljaju težine i pristranosti neuronske mreže koja uči tu funkciju. Kroz fazu treniranja mreže, parametri se ažuriraju u svrhu optimizacije prostora ugrađivanja, kako bi primjeri iz iste klase bili što bliže, a iz različitih što dalje.

Nakon završetka treniranja, f_ϕ bi trebala moći uspješno mapirati nove zadatke u prostor ugrađivanja i time omogućiti uspoređivanje s prototipima klasa u svrhu klasifikacije novog zadatka.

U konačnici, računanje prototipa c_k je dano sljedećim izrazom:

$$c_k = \frac{1}{|S_k|} \sum_{(x_i, y_i) \in S_k} f_\phi(x_i),$$

gdje S_k predstavlja potporni skup klase k, x_i je D-dimenzionalni vektor značajki $x_i \in \mathbb{R}^D$, a y_i označava odgovarajuću oznaku $y_i \in \{1, ..., k\}$.

Distribucije po klasama se računaju koristeći softmax funkciju nad udaljenosti nove točke x do prototipa c_k:

$$p_\phi(y = k|x) = \frac{\exp(-d(f_\phi(x), c_k))}{\sum_{k'} \exp(-d(f_\phi(x), c_{k'}))},$$

pri čemu je $d : \mathbb{R}^M \times \mathbb{R}^M \rightarrow [0, +\infty]$ nenegativna funkcija udaljenosti (u radu [12] korištena Euklidska udaljenost).

Cilj je minimizacija funkcije gubitka negativne log-vjerojatnosti $J(\phi)$ prave klase k preko SGD:

$$J(\phi) = -log\ p_\phi(y = k|x)$$

Trening metoda Prototipskih mreža je dana u algoritmu 3.

Udaljenost korištena u računanju funkcije gubitka mora zadovoljiti Bregmanovu divergenciju (za objašnjenje pogledati [12]).

Algoritam 3 Prototipske mreže - trening metoda

1: **Ulaz** : $D = \{(x_1, y_1), ..., (x_N, y_N)\}, \forall y_i \in \{1, ..., K\}$, skup za treniranje
2: **Ulaz** : D_k, podskup od D s elementina (x_i, y_i) takvim da $y_i = k$
3: **Ulaz** : N, veličina uzorka od D
4: **Ulaz** : K, broj klasa u D
5: **Ulaz** : N_C, broj klasa za svaku iteraciju
6: **Ulaz** : N_S, veličina uzorka potpornog skupa za svaku klasu
7: **Ulaz** : N_Q veličina uzorka validacijskog skupa za svaku klasu
8: **Ulaz** : NASUMIČNIUZORAK(S, N), promiješani skup s N elemenata uniformno odabranih iz skupa S
9: **Izlaz** : gubitak J
10: Nasumično odaberi indekse klasa koristeći $V \leftarrow$ NASUMIČNIUZORAK$(\{1, ..., K\}, N_C)$
11: **for** $k \in \{1, ..., N_C\}$ **do**
12: Konstruiraj potporne uzorke koristeći $S_k \leftarrow$ NASUMIČNIUZORAK(D_{V_k}, N_S)
13: Konstruiraj validacijske uzorke koristeći $Q_k \leftarrow$ NASUMIČNIUZORAK$(D_{V_k} \setminus S_k, N_Q)$
14: Izračunaj prototip ugrađivanjem potpornih uzoraka koristeći $c_k \leftarrow \frac{1}{|N_C|} \sum_{(x_i, y_i) \in S_k} f_\phi(x_i)$
15: Inicijalizacija gubitka $J \leftarrow 0$
16: **for** $k \in \{1, ..., N_C\}$ **do**
17: **for** $(x, y) \in Q_k$ **do**
18: Ažuriraj gubitak koristeći $J \leftarrow J + \frac{1}{N_C N_Q} \left[d(f_\phi(x), c_k) + log \sum_{k'} exp(-d(f_\phi(x), c_{k'})) \right]$

Prokomentirajmo ažuriranje gubitka u liniji 18 algoritma 3. $\frac{1}{N_C N_Q}$ označava normalizaciju s obzirom da iteriramo prvo po svim Q_k u unutarnjoj petlji, a u vanjskoj od 1 do N_C. $d(f_\phi(x), c_k)$ je udaljenost između nove točke $f_\phi(x)$ i prototipa c_k. $log \sum_{k'} exp(-d(f_\phi(x), c_{k'}))$ sumira gubitak nad udaljenosti do svih prototipa klasa.

4 | Implementacija meta-trening i meta-test faza korištenjem learn2learn paketa

U ovom poglavlju iskoristit ćemo modele iz learn2learn paketa te provesti meta-treniranje i meta-testiranje na Omniglot skupu podataka, prokomentirati rezultate te zaključiti da li je base-learner dobro generalizirao na testnim podacima.

Skup podataka

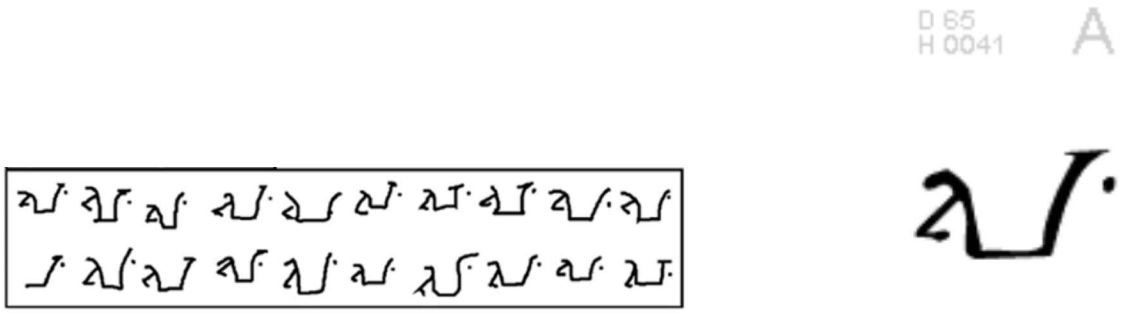
Faze meta-treniranja i meta-testiranja provest ćemo na Omniglot skupu podataka.

Raznolikost podataka omogućuje kreiranje mnoštva različitih zadataka što čini ovaj skup podataka idealnim za evaluaciju modela meta-učenja.

Omniglot je skup podataka koji se sastoji od slika 1623 različitih rukom pisanih znakova iz 50 abeceda.

Slika 4.1: Neki od znakova iz Omniglot skupa podataka

Svaki od znakova je napisan od strane 20 ljudi, što znači da imamo samo 20 primjera po klasi. Takav mali broj primjera pogodan je za učenje s primjerima. Na donjoj slici desno možemo vidjeti znak iz varijante hebrejske abecede koji odgovara slovu A, a lijevo možemo vidjeti taj znak napisan na 20 različitih načina u Omniglot skupu podataka.



(a) Slovo A na 20 načina (rukopis)

(b) Slovo A

Dakle, Omniglot skupom podataka pokušavamo naučiti model uspješno izvršiti zadatak prepoznavanja slike, odnosno prepoznavanja znaka na slici.

learn2learn paket

U svrhu boljeg razumijevanja implementacije, u ovom potpoglavlju ukratko ćemo objasniti funkcije iz learn2learn paketa koje koristimo:

1. l2l.vision.datasets - klasa pomoću koje ćemo skinuti Omniglot skup podataka i koristiti ga dalje u aplikaciji
2. l2l.data.MetaDataset - klasa koju ćemo primijeniti na Omniglot skup podataka. Omogućuje brzo indeksiranje uzoraka što je poželjno kod kreiranja zadataka s primjerima
3. l2l.data.transforms - klasa koja sadrži metode za transformaciju podataka u zadatke učenja s primjerima
4. l2l.data.TaskDataset - klasa u kojoj pravimo skup zadataka. Prosljeđujemo joj transformacije iz l2l.data.transforms, skup podataka iz l2l.data.MetaDataset te broj zadataka
5. l2l.vision.models - klasa koja sadrži vision modele, specifične skupovima podataka iz l2l.vision.datasets. Model koji ćemo koristiti iz ove klase, kao base-learner, bit će OmniglotFC, specifičan Omniglot skupu podataka
6. l2l.algorithms - klasa koja sadrži implementacije algoritama meta-učenja. Algoritam koji ćemo koristiti iz ove klase je MAML

Implementacija

U ovom potpoglavlju detaljno ćemo opisati korake implementacije, od inicijalizacije varijabli do faze meta-testiranja. Prvo ćemo primijeniti pristup temeljen na optimizaciji, te zatim, temeljen na metrici. Potom ćemo usporediti dobivene rezultate tih dvaju pristupa. Pristupi temeljeni na modelu ne mogu se naći u l2l paketu te ćemo njih preskočiti. Dijelovi programskog koda mogu se naći na [1] i [2].

Primjena MANN modela

Na početku inicijaliziramo hiperparametre na željene vrijednosti.

```
ways = 5
shots = 1
meta_lr = 0.003
fast_lr = 0.5
meta_batch_size = 32
fast_adaptation_steps = 1
num_iterations = 1000
```

Parametar ways označava da ćemo imati 5 klasa, a shots 1 primjer po zadatku. Ta dva parametra su važna kako bi mogli konstruirati meta skup podataka i iz njega transformacijama generirati skup zadataka.

meta_lr je parametar koji određuje koliko brzo će meta-učenik učiti osnovni model. Vrijednost ovog hiperparametra je manja jer želimo da osnovni model nauči što bolju generalizaciju.

fast_lr odnosi se na brzu adaptaciju na nove zadatke i zato je vrijednost ovog hiperparametra visoka.

meta_batch_size odnosi se na broj zadataka koji će biti uzorkovani u svakoj iteraciji meta treniranja, odnosno meta-testiranja.

num_iterations označava broj iteracija koji ćemo provesti u fazi meta-treniranja.

Nakon inicijalizacije parametara, učitavamo Omniglot skup podataka uz transformaciju ulaza s namjerom smanjenja računalne složenosti.

```
omniglot = l2l.vision.datasets.FullOmniglot(root='./data',
                                            transform=transforms.Compose([
                                                transforms.Resize(28,
    interpolation=LANCZOS),
                                                transforms.ToTensor(),
                                                lambda x: 1.0 - x,
                                            ]),
                                            download=True)
dataset = l2l.data.MetaDataset(omniglot)
classes = list(range(1623))
random.shuffle(classes)
```

Od tako učitanog skupa podataka, pravimo meta skup podataka iz kojeg ćemo moći napraviti skup zadataka. Lista brojeva od 0 do 1622 predstavlja broj klasa u Omniglot skupu podataka. Klase izmiješamo kako bi dobili nasumičnost i potakli bolju generalizaciju.

Primijenjujući transformacije na meta skup podataka, formiramo skup zadataka, posebno za svaku fazu procesa meta-učenja.

```
train_transforms = [
    l2l.data.transforms.FilterLabels(dataset, classes[:1100]),
    l2l.data.transforms.NWays(dataset, ways),
    l2l.data.transforms.KShots(dataset, 2*shots),
    l2l.data.transforms.LoadData(dataset),
    l2l.data.transforms.RemapLabels(dataset),
    l2l.data.transforms.ConsecutiveLabels(dataset),
    l2l.vision.transforms.RandomClassRotation(dataset, [0.0, 90.0, 180.0,
    270.0])
]
train_tasks = l2l.data.TaskDataset(dataset,
                                   task_transforms=train_transforms,
                                   num_tasks=20000)

valid_transforms = [
    l2l.data.transforms.FilterLabels(dataset, classes[1100:1200]),
    l2l.data.transforms.NWays(dataset, ways),
    l2l.data.transforms.KShots(dataset, 2*shots),
    l2l.data.transforms.LoadData(dataset),
    l2l.data.transforms.RemapLabels(dataset),
    l2l.data.transforms.ConsecutiveLabels(dataset),
    l2l.vision.transforms.RandomClassRotation(dataset, [0.0, 90.0, 180.0,
    270.0])
]
valid_tasks = l2l.data.TaskDataset(dataset,
                                   task_transforms=valid_transforms,
                                   num_tasks=1024)

test_transforms = [
    l2l.data.transforms.FilterLabels(dataset, classes[1200:]),
    l2l.data.transforms.NWays(dataset, ways),
    l2l.data.transforms.KShots(dataset, 2*shots),
    l2l.data.transforms.LoadData(dataset),
    l2l.data.transforms.RemapLabels(dataset),
    l2l.data.transforms.ConsecutiveLabels(dataset),
    l2l.vision.transforms.RandomClassRotation(dataset, [0.0, 90.0, 180.0,
    270.0])
]
test_tasks = l2l.data.TaskDataset(dataset,
                                  task_transforms=test_transforms,
                                  num_tasks=1024)
```

Uočimo kako smo filtrirali klase po fazama, to jest, od 0 do 1100 smo odvojili za fazu treniranja, od 1100 do 1200 za fazu validacije te ostatak za testiranje.

Kreirajmo sad klasu modela MANN sljedeći arhitekturu modela opisanu u 3.1.1.

```
class MANN(nn.Module):
    def __init__(self, input_size, output_size, memory_slots, memory_size):
        super(MANN, self).__init__()
        self.controller = nn.LSTMCell(input_size, 128)
        self.read_head = nn.Linear(128, memory_slots)
        self.write_head = nn.Linear(128, memory_slots)
        self.write_content_layer = nn.Linear(128, memory_size)
        self.output_layer = nn.Linear(128 + memory_size, output_size)
        self.memory_slots = memory_slots
        self.memory_size = memory_size
        self.memory = None

    def forward(self, x):
        batch_size, seq_len, _ = x.size()
        h_t = torch.zeros(batch_size, 128).to(x.device)
        c_t = torch.zeros(batch_size, 128).to(x.device)
        outputs = []

        self.memory = torch.zeros(batch_size, self.memory_slots, self.
    memory_size).to(x.device)

        for i in range(seq_len):
            input_t = x[:, i, :]
            h_t, c_t = self.controller(input_t, (h_t, c_t))

            read_weights = F.softmax(self.read_head(h_t), dim=1)

            read_content = torch.bmm(read_weights.unsqueeze(1), self.memory
    ).squeeze(1)

            write_weights = F.softmax(self.write_head(h_t), dim=1)

            write_content = torch.tanh(self.write_content_layer(h_t))

            memory_update = torch.einsum('bi,bj->bij', write_weights,
    write_content)
            self.memory = self.memory + memory_update

            final_input = torch.cat([h_t, read_content], dim=1)
            output = self.output_layer(final_input)
            outputs.append(output.unsqueeze(1))

        return torch.cat(outputs, dim=1)
```

Za kontroler mrežu smo uzeli LSTM ćeliju, koja je pogodna za obradu informacija kroz iteracije i interakciju s memorijom.

Idući korak je inicijalizacija meta modela, te optimizatora i funkcije gubitka.

```
mann = MANN(28 ** 2 + ways, ways, memory_slots=128, memory_size=40).to(
    device)
loss = nn.CrossEntropyLoss(reduction='mean')
optimizer = optim.Adam(mann.parameters(), meta_lr)
```

Za brzu prilagodbu na zadatak i evaluaciju na tom zadatku, koristimo funkciju fast_adapt prilagođenu MAML meta-učeniku:

```
def fast_adapt(batch, model, criterion, shots, ways, device):
    data, labels = batch
    data, labels = data.to(device), labels.to(device)
    data = data.view(-1, 28 * 28)

    adaptation_indices = torch.zeros(data.size(0), dtype=torch.bool)
    adaptation_indices[::2] = True
    adaptation_data, adaptation_labels = data[adaptation_indices], labels[
    adaptation_indices]
    evaluation_data, evaluation_labels = data[~adaptation_indices], labels
    [~adaptation_indices]

    combined_data = torch.cat([adaptation_data, evaluation_data], dim=0)
    combined_labels = torch.cat([adaptation_labels, evaluation_labels], dim
    =0)

    seq_len = combined_data.size(0)

    prev_label = torch.zeros(ways).to(device)

    inputs = []
    targets = []

    for t in range(seq_len):
        x_t = combined_data[t]
        y_t = combined_labels[t]

        prev_label_one_hot = prev_label.clone()
        input_t = torch.cat([x_t, prev_label_one_hot], dim=0)
        inputs.append(input_t)
        targets.append(y_t)

        prev_label = torch.zeros(ways).to(device)
        prev_label[y_t] = 1

    inputs = torch.stack(inputs)
    targets = torch.tensor(targets).to(device)

    inputs = inputs.unsqueeze(0)
```

```
    targets = targets.unsqueeze(0)

    outputs = model(inputs)
    outputs = outputs.view(-1, ways)
    targets = targets.view(-1)

    loss = criterion(outputs, targets)

    evaluation_size = evaluation_data.size(0)

    evaluation_outputs = outputs[-evaluation_size:]
    evaluation_targets = targets[-evaluation_size:]

    _, predicted = torch.max(evaluation_outputs.data, 1)
    correct = (predicted == evaluation_targets).sum().item()
    acc = correct / evaluation_size

    return loss, acc
```

U fazi meta-treniranja prolazimo kroz 1000 iteracija, u svakoj iteraciji uzorkujemo 32 zadatka i uprosječimo gubitak i točnost tih zadataka. To uzmemo za gubitak i točnost iteracije, te isti postupak ponovimo i za zadatke s validacijskog skupa.

```
for iteration in range(num_iterations):
    optimizer.zero_grad()
    meta_train_error = 0.0
    meta_train_accuracy = 0.0
    meta_valid_error = 0.0
    meta_valid_accuracy = 0.0

    for task in range(meta_batch_size):
        batch = train_tasks.sample()
        train_error, train_acc = fast_adapt(batch, mann, loss, shots, ways,
     device)
        train_error.backward()
        meta_train_error += train_error.item()
        meta_train_accuracy += train_acc

        batch = valid_tasks.sample()
        with torch.no_grad():
            val_loss, val_acc = fast_adapt(batch, mann, loss, shots, ways,
    device)
        meta_valid_error += val_loss.item()
        meta_valid_accuracy += val_acc

    for p in mann.parameters():
        if p.grad is not None:
            p.grad.data.mul_(1.0 / meta_batch_size)
```

```
    optimizer.step()
```

U fazi meta-testiranja, postupak je sličan, međutim broj iteracija je puno manji i koristimo ga samo kako bi uprosječili točnost.

```
test_iterations = 10
for iteration in range(test_iterations):
  meta_test_error = 0
  meta_test_accuracy = 0
  for task in range(meta_batch_size):
    batch = test_tasks.sample()
    evaluation_error, evaluation_accuracy = fast_adapt(batch, mann, loss,
    shots, ways, device)
    meta_test_error += evaluation_error.item()
    meta_test_accuracy += evaluation_accuracy
  average_meta_test_error = meta_test_error / meta_batch_size
  average_meta_test_accuracy = meta_test_accuracy / meta_batch_size
```

Napomenimo da je Omniglot skup podataka dizajniran za učenje s jednim primjerom, te iz tog razloga za parametre shots i fast_adaptation_steps uzimamo vrijednost 1.

Primjena MAML modela

Situacija s MAML modelom je znatno jednostavnija, jer umjesto implementacije, koristimo gotov MAML model iz learn2learn klase, te OmniglotFC kao osnovni učenik.

```
model = l2l.vision.models.OmniglotFC(28 ** 2, ways)
maml = l2l.algorithms.MAML(model, lr=fast_lr, first_order=False)
```

Opet koristimo funkciju fast_adapt, sada prilagođenu MAML meta-učeniku:

```
def fast_adapt(batch, learner, loss, adaptation_steps, shots, ways, device)
    :
    data, labels = batch
    data, labels = data.to(device), labels.to(device)
    adaptation_indices = torch.zeros(data.size(0)).byte()
    adaptation_indices[torch.arange(shots*ways) * 2] = 1
    adaptation_data, adaptation_labels = data[adaptation_indices], labels[
    adaptation_indices]
    evaluation_data, evaluation_labels = data[1 - adaptation_indices],
    labels[1 - adaptation_indices]

    for step in range(adaptation_steps):
```

```
        train_error = loss(learner(adaptation_data), adaptation_labels)
        train_error /= len(adaptation_data)
        learner.adapt(train_error)

    predictions = learner(evaluation_data)
    valid_error = loss(predictions, evaluation_labels)
    valid_error /= len(evaluation_data)
    valid_accuracy = accuracy(predictions, evaluation_labels)
    return valid_error, valid_accuracy
```

Primjena Prototipskih mreža

S obzirom da implementacija većim dijelom ostaje ista kao u prethodna dva modela, prikazat ćemo samo programski kod koji je izmijenjen, a tiče se izgradnje Prototipske mreže te njezine brze adaptacije.

S obzirom da nam l2l ne pruža direktno implementaciju Prototipske mreže, definirat ćemo sami klasu.

```
class PrototypicalNetwork(nn.Module):
    def __init__(self, base_model):
        super(PrototypicalNetwork, self).__init__()
        self.base_model = base_model

    def forward(self, x):
        x = self.base_model(x)
        return x.view(x.size(0), -1)
```

U konstruktoru klase definiramo base model koji ćemo koristiti, a to je OmniglotFC. U forward metodi izračunamo izlaz te ga spljoštimo u 2D radi daljnjih usporedbi i računanja.

Funkcija fast_adapt osim opisanog u MAML implementaciji, odrađuje i računanje prototipa klasa iz pomoćnog skupa te klasifikaciju testnog skupa pomoću najbližeg prototipa.

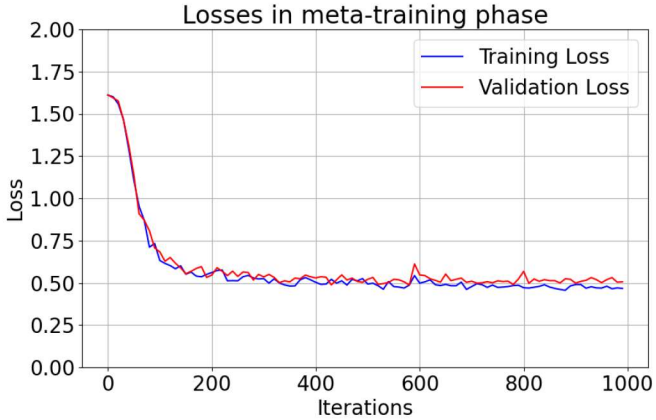
```
def pairwise_distances_logits(a, b):
    n = a.shape[0]
    m = b.shape[0]
    logits = -((a.unsqueeze(1).expand(n, m, -1) -
                b.unsqueeze(0).expand(n, m, -1))**2).sum(dim=2)
    return logits

def fast_adapt(model, batch, ways, shot, query_num, device=None):
    data, labels = batch
    data = data.to(device)
    labels = labels.to(device)
```

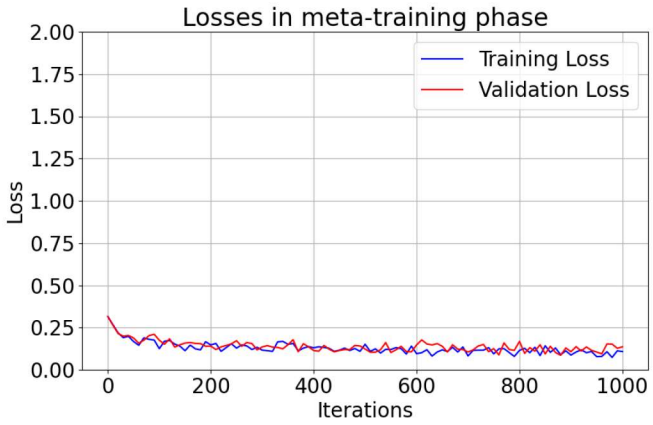
```
12
13      sort = torch.sort(labels)
14      data = data.squeeze(0)[sort.indices].squeeze(0)
15      labels = labels.squeeze(0)[sort.indices].squeeze(0)
16
17      embeddings = model(data)
18      support_indices = np.zeros(data.size(0), dtype=bool)
19      selection = np.arange(ways) * (shot + query_num)
20      for offset in range(shot):
21          support_indices[selection + offset] = True
22      query_indices = torch.from_numpy(~support_indices)
23      support_indices = torch.from_numpy(support_indices)
24      support = embeddings[support_indices]
25      support = support.reshape(ways, shot, -1).mean(dim=1)
26      query = embeddings[query_indices]
27      labels = labels[query_indices].long()
28
29      logits = pairwise_distances_logits(query, support)
30      loss = F.cross_entropy(logits, labels)
31      acc = accuracy(logits, labels)
32      return loss, acc
```

Rezultati

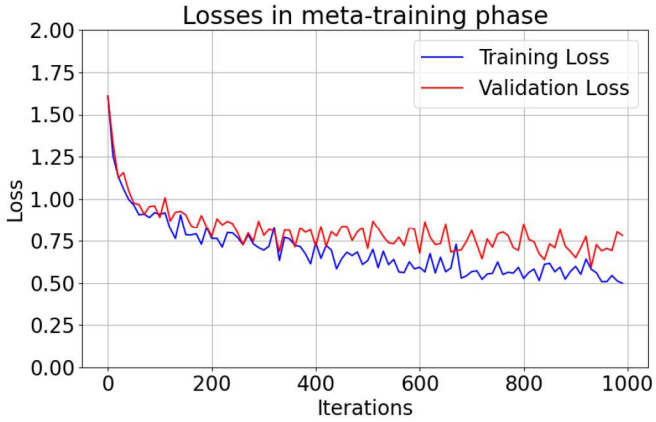
Iz priloženih grafova možemo vidjeti da se u fazi meta-treniranja modeli dobro prilagođavaju na trening podatke.



Slika 4.3: MANN gubitak (loss) u fazi meta treniranja

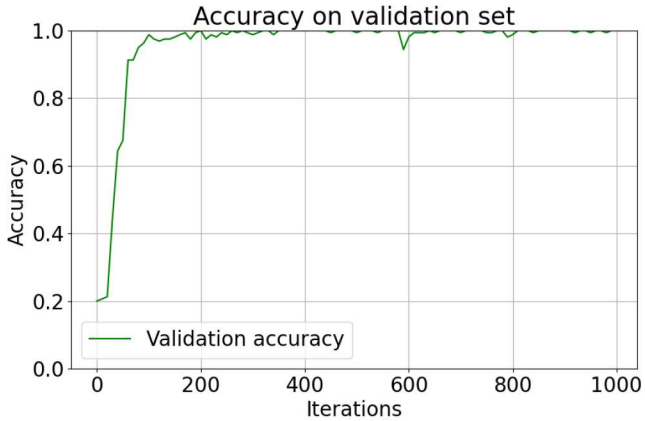


Slika 4.4: MAML gubitak (loss) u fazi meta treniranja

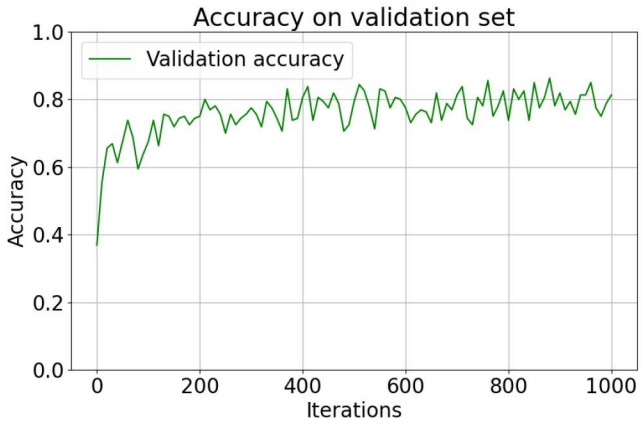


Slika 4.5: Prototipske mreže gubitak (loss) u fazi meta treniranja

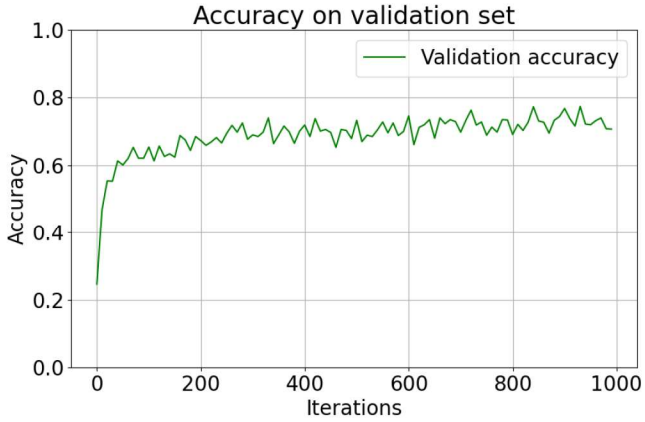
Također, s obzirom da se točnost na validacijskom skupu povećava s brojem iteracija, kako se vidi iz grafova 4.6, 4.7 i 4.8, možemo zaključiti da modeli uspjevaju generalizirati na dosad neviđene klase.



Slika 4.6: MANN točnost (accuracy) na validacijskom skupu

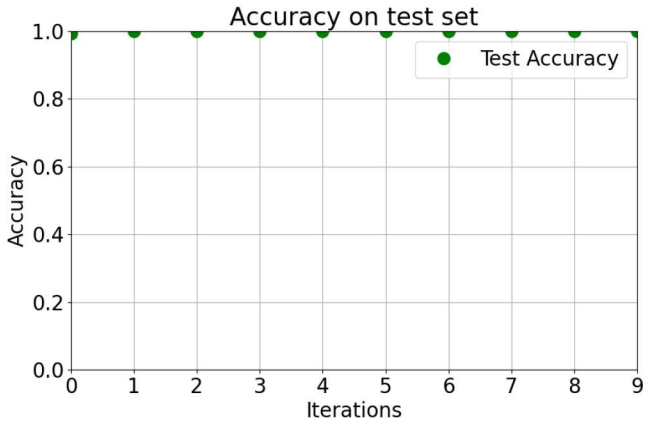


Slika 4.7: MAML točnost (accuracy) na validacijskom skupu

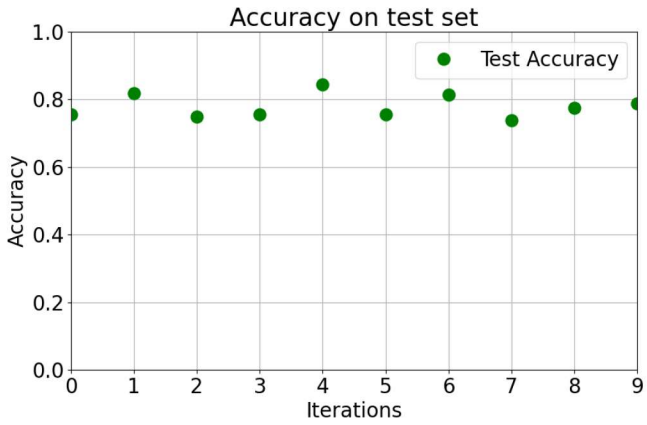


Slika 4.8: Prototipske mreže točnost (accuracy) na validacijskom skupu

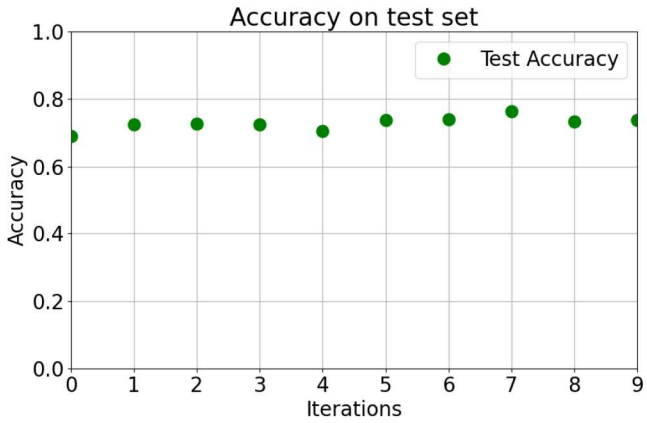
Iz ta dva razloga i uzimajući u obzir koncept meta-učenja, za pretpostaviti je da će i faza meta-testiranja na Omniglot skupu podataka dati podjednake rezultate, odnosno, da će model na osnovu jednog primjera uspjeti ostvariti točnost dobivenu kroz meta-treniranje. Pokretanjem faze meta-testiranja, dobivamo rezultate:



Slika 4.9: MANN točnost (accuracy) na testnom skupu kroz 10 iteracija



Slika 4.10: MAML točnost (accuracy) na testnom skupu kroz 10 iteracija



Slika 4.11: Prototipske mreže točnost (accuracy) na testnom skupu kroz 10 iteracija

Usporedba grafova

Iako smo zaključili da sva tri pristupa uspijevaju dobro generalizirati na Omniglot skupu podataka, navest ćemo neke zaključke iz grafova iznad.

1. Iz grafova 4.3, 4.4 i 4.5 vidimo da su početni gubitci mali brojevi, što je karakteristika modela meta-učenika (dobra inicijalizacija početnih parametara/prototipa)

2. Prototipske mreže bilježe veću razliku između trening i validacijskog gubitka u zadnjih 200-300 iteracija što mogu biti znakovi prenaučenosti. Razliku između trening i validacijskog gubitka bilježi i MANN, ali u manjoj mjeri

3. Iako ima bolje rezultate od Prototipskih mreža, MAML bilježi fluktuacije vrijednosti gubitaka i točnosti kroz zadatke, dok kod Prototipskih mreža vidimo relativno dosljedne vrijednosti (bolja otpornost Prototipskih mreža na različite zadatke). MANN ima vrlo stabilne vrijednosti gubitaka i točnosti, što je jedan od razloga za brzu konvergenciju

4. Klase MANN i Prototipskih mreža smo implementirali, dok smo za MAML iskoristili gotovu implementaciju iz l2l paketa. Implementacija iz l2l paketa je sigurno visoko optimizirana te iz tog razloga su početne vrijednosti funkcije gubitka, a onda i završne, puno niže od druga 2 pristupa. Iako MAML tokom cijelog procesa meta-treniranja ima niže vrijednosti funkcije gubitka od druga dva meta-učenika, to nije impliciralo najveću točnost. Iz navedenog ćemo se za usporedbu performansi meta-učenika fokusirati na točnost, što bi značilo da se MANN pokazao kao najprikladniji za 5-way 1-shot zadatak na Omniglot skupu podataka.

Zaključak

U ovom radu smo teorijski opisali pristup temeljen na modelu MANN, pristup temeljen na optimizaciji MAML i pristup temeljen na metrici Prototipske mreže, te ih potom implementirali. Rezultate smo detaljno obrazložili u podpoglavljima 4 i 4.

Iz rezultata se vidi da su modeli na 1000 iteracija ostvarili točnost od 100, 78 i 73 % na testnom skupu, te bi vjerojatno s većim brojem iteracija točnost bila još veća. S obzirom da su uspjeli generalizirati na neviđenim klasama istog skupa podataka, postavlja se pitanje da li bi mogli generalizirati na neviđenim klasama nekog drugog skupa podataka.

Odgovor je - do neke mjere, a mjera je sličnost Omniglota i drugog skupa podataka. U [14] naglašavaju da je slabljenje generalizacije modela proporcionalno porastu razlike među skupovima podataka. Nadalje, pretpostavljaju da to ukazuje na to da skup meta-parametara ne uspijeva dovoljno dobro uhvatiti heterogenost među zadacima.

Iako je bilo pokušaja, trenutno još uvijek nema konkretnih ideja kako bi se mogao riješiti taj problem prilagodbe na različite domene zadataka.

Zaključujemo, meta-učenje je privuklo pozornost kao moguće rješenje problema od kojeg pate i današnji modeli, loše generalizacije. Možemo reći da je problem zasad djelomično rješen, te da je prostor za istraživanje i napredak u ovom području ogroman.

Literatura

[1] Learn2learn: A pytorch library for meta-learning. https://learn2learn.net.

[2] Learn2learn: Github repository. https://github.com/learnables/learn2learn.

[3] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR, 06–11 Aug 2017.

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[5] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5149–5169, 2022.

[6] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[7] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*. Lille, 2015.

[8] Tsendsuren Munkhdalai and Hong Yu. Meta networks. In *International conference on machine learning*, pages 2554–2563. PMLR, 2017.

[9] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *CoRR*, abs/1803.02999, 2018.

[10] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International Conference on Learning Representations*, 2017.

[11] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1842–1850, New York, New York, USA, 20–22 Jun 2016. PMLR.

[12] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30, 2017.

[13] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H.S. Torr, and Timothy M. Hospedales. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[14] Anna Vettoruzzo, Mohamed-Rafik Bouguelia, Joaquin Vanschoren, Thorsteinn Rognvaldsson, and KC Santosh. Advances and challenges in meta-learning: A technical review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[15] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, and Daan Wierstra. Matching networks for one shot learning. *Advances in neural information processing systems*, 29, 2016.

[16] S. Prince W. Zi, L. S. Ghoraie. Few-shot learning i& meta-learning | tutorial. https://www.borealisai.com/research-blogs/tutorial-2-few-shot-learning-and-meta-learning-i/.

Sažetak

U ovom radu opisujemo koncept meta-učenja i njegovu ulogu kod brzog učenja i prilagodbe neuronskih mreža na nove zadatke. U današnje vrijeme se u području umjetne inteligencije još uvijek susrećemo s problemom loše generalizacije modela na nove zadatke. Zbog toga, za svaki novi zadatak moramo konstruirati novi skup podataka što može biti vrlo skupocjeno. Meta-učenje se bavi upravo tim problemom, odnosno, prilagođavanjem modela na novi zadatak uz minimalan broj primjera. Kroz rad definiramo meta-učenje, navodimo najpoznatije pristupe te implementiramo modele meta-učenika triju pristupa koristeći learn2learn paket u programskom jeziku Python.

Ključne riječi

meta učenje, strojno učenje, duboko učenje, osnovni učenik, meta-učenik, pristupi, mann, maml, prototipske mreže, učenje s primjerima, learn2learn

Applying meta-learning approaches as methods for quick learning of neural networks

Summary

In this paper we describe meta-learning concept and its role in quick learning and adaptation of neural networks on new tasks. Nowadays in artifical intelligence area we still face problem of poor generalization of models on new tasks. Consequently, for every new task we need to construct new dataset, which can be very expensive. Meta-learning deals with exactly this problem, i.e., adapting model on new task with a minimal number of examples. Through paper we define meta-learning, list the most popular approaches and implement meta learner models of 3 approaches using learn2learn package in programming language Python.

Keywords

meta-learning, machine learning, deep learning, base learner, meta learner, approaches, mann, maml, prototypical networks, few-shot learning, learn2learn

Životopis

Rođen sam 28.07.1998. u Našicama. Od 2005. do 2013. godine pohađao sam Osnovnu školu kralja Tomislava u Našicama. Nakon toga, 2013. godine upisao sam Prirodoslovno-matematičku gimnaziju, također u Našicama, koju sam završio 2017. godine uspješnim polaganjem mature. Nakon mature, 2017. godine upisao sam Preddiplomski sveučilišni studij Matematika i računarstvo na Odjelu za matematiku u Osijeku. Preddiplomski studij završavam 2021. godine završnim radom "Blogdown: kreiranje web stranica pomoću R Markdown" pod mentorstvom izv. prof. dr. sc. Danijela Grahovca. Iste godine upisujem Diplomski sveučilišni studij Matematike, smjer Matematika i računarstvo na Odjelu za matematiku u Osijeku.