Seminarski rad

Implementacija graf neuronskih mreža u NumPy-u i primena u data mining-u

Profesor:

Prof. dr Suzana Stojković

Student:

Danilo Milošević, 1732

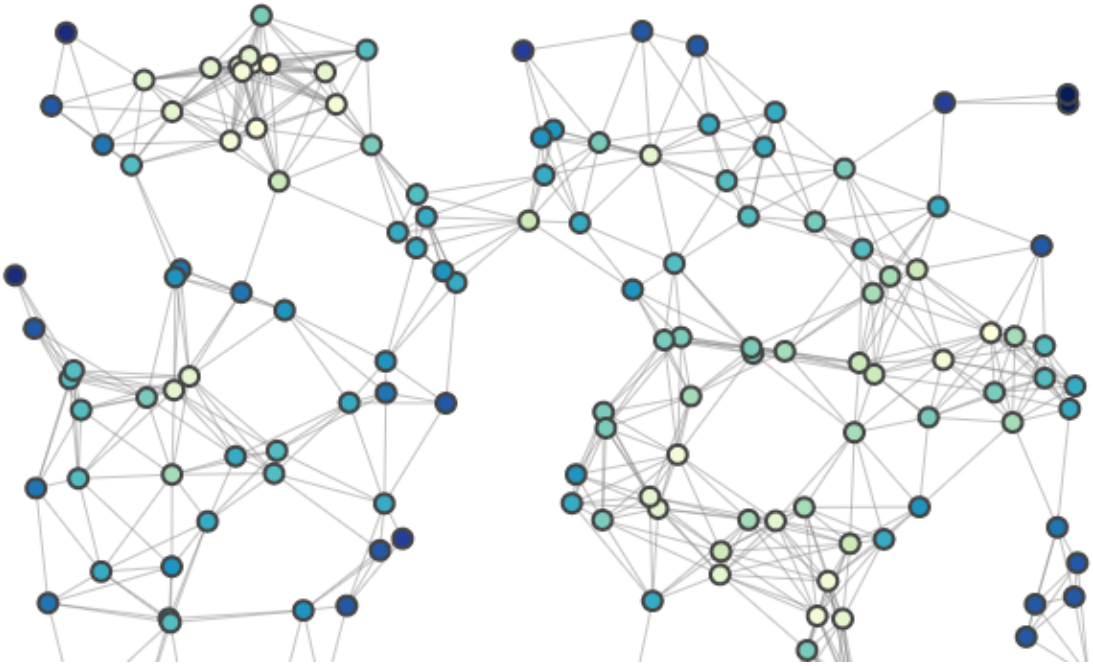
1. Uvod 3
2. Princip rada graf neuronskih mreža 4
2.1 Matematička definicija grafa, pojmovi i predstavljanje grafa 4
2.2 Arhitektura graf neuronske mreže 5
2.3 Message passing sloj 6
2.4 Problemi koje rešavaju graf neuronske mreže 7
2.5 Varijacije graf neuronskih mreža 8
3. Implementacija GCN-a u NumPy-u 10
3.1 Opis strukture implementacije 10
3.2 Opis aktivacionih funkcija i implementacija 11
3.3 Loss funkcije 13
3.4 Learning rate 14
3.5 Normalizacija matrice susedstva 14
3.6 Inicijalizacija težina neuronske mreže 15
3.7 Implementacija GCN sloja 15
3.8 Implementacija GlobalPool sloja 17
3.9 Implementacija Dense sloja 17
3.10 Implementacija droput-a 18
3.11 Implementacija graf neuronske mreže 18
3.12 Testiranje implementacije 20
4. Primena graf neuronskih mreža 22
4.1 Pregled skupa podataka 22
4.2 Treniranje Random Forest Classifier-a 23
4.3 Treniranje graf neuronske mreže 24
5. Literatura 25

1. Uvod

U toku poslednje decenije nagli razvoj veštačkih neuronskih mreža je omogućio njihovu primenu za različite zadatke i nad različitim tipovima podataka - slikama, snimcima, tekstom, audio zapisima, 3D modelima kao i nad mnogim drugim. Jednu od najbitnijih struktura podataka predstavljaju grafovi, koji se javljaju u različitim sferama kao što su programiranje, hemija, u društvenim mrežama, programskim prevodiocima kao i drugim oblastima gde se grafom mogu opisivati relacije između različitih pojmova.

Zbog ove sveprisutnosti grafova kao i nesposobnosti neuronskih mreža kao što su konvolucione mreže i transformeri da efikasno obrade grafovske podatke nastale su graf neuronske mreže koje rade nad podacima koji čine grafove.

Cilj ovog rada je proučiti princip rada graf neuronskih mreža kao i njihovih varijacija (graf konvolucione neuronske mreže, relacione graf neuronske mreže, primena *attention*-a u graf neuronskim mrežama), implementirati *GNN* u *NumPy*-u a zatim i primeniti nad skupom podataka mrežu. Takođe, na kraju rada ćemo razmotriti i moguće primene graf neuronskih mreža u različitim oblastima, njihove prednosti kao i nedostatke.



Graf neuronska mreža

2. Princip rada graf neuronskih mreža

2.1 Matematička definicija grafa, pojmovi i predstavljanje grafa

Graf G predstavlja uređeni par $G = (V, E)$, pri čemu je V konačan neprazan skup dok je E binarna relacija nad skupom V . Elementi skupa V , $v \in V$ predstavljaju čvorove grafa, dok su elementi skupa E, $e \in E$ grane u grafu G .

Na osnovu tipa relacije E razlikujemo dve vrste grafova

1. Usmereni grafovi - gde su elementi $e \in E$ uređeni parovi (v_1, v_2), $v_1, v_2 \in V$, tako da su čvorovi povezani samo u jednom smeru, odnosno grana ide od čvorav_1 ka čvoru v_2

2. Neusmereni grafovi - gde su elementi $e \in E$ skupovi , tako da su čvorovi $\{v_1, v_2\}$, v_1, $v_2 \in V$ povezani u oba smera.

U nastavku rada i implementaciji podrazumevaćemo da radimo sa neusmerenim grafovima.

Čvorovi mogu biti bilo koji matematički objekti, a u graf neuronskim mrežama čvorovi će biti opisani n dimenzionalnim vektorima koji predstavljaju ulazne feature-e, odnosno

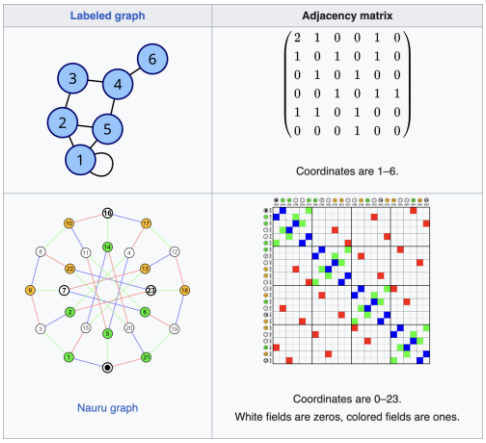
$$v \in \mathbb{R}^n, v \in V$$

Pored toga i same grane mogu biti opisane vektorima, koji predstavljaju informacije o odnosima između čvorova u grafu.

Grafovi mogu biti definisani na računaru na dva načina

1. Navođenjem čvorova i grana – grafovi mogu biti opisani direktnim nabrojavanjem čvorova i grana u grafu. Prednost ovakve reprezentacije na računaru je efikasnija upotreba memorije. Ukoliko je graf redak, tj ima mnogo manje ivica nego što je moguće (ukoliko imamo n čvorova graf može imati $n \cdot (n-1)$ grana) ovakva reprezentacija zahteva mnogo manje memorije nego matrična reprezentacija. Nedostatak ovakvog predstavljanja je otežano definisanja operacija koje se koristi u neuronskim mrežama kao što su množenje matrica.

2. Matricama susedstva - iako zahteva više memorije kod retkih grafova, matrična reprezentacija omogućava lakše definisanje operacija kao i izvršenje na grafičkim karticama. Matrica susedstva je matrica $A \in \mathrm{M}_{NxN}$, $N = card(V)$, pri čemu je $A_{i,j}$ 1 ukoliko su čvorovi i i j povezani, a u suprotnom 0. Zbog ovih prednosti ćemo u implementaciji koristiti ovu reprezentaciju.



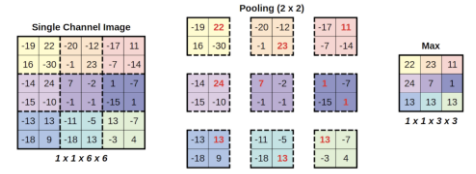
Matrica susedstva

2.2 Arhitektura graf neuronske mreže

Najosnovnija graf neuronska mreža koristi višeslojni perceptron pomoću kog se na osnovu trenutne reprezentacije (vrednosti) čvora predviđa labela čvora, a isto se može primeniti i nad ivicama kao i nad celim grafom (više o tome u oblasti 2.4). Arhitekturu graf neuronskih mreže čine 3 tipa slojeva:

- **Permutacioni ekvivalentni sloj** - permutaciono ekvivalentni sloj mapira trenutnu reprezentaciju grafa na novu reprezentaciju grafa. Ovo je predstavljeno procesom koji se naziva message passing, pri čemu je vrednost svakog čvora ažurirana vrednošću njegovih direktnih suseda. Ponavljanjem ovih slojeva omogućava se da se u svakom čvoru akumuliraju podaci daljih čvorova, odnosno čvor dobija podatke o strukturi grafa. Ovako akumulirani podaci omogućavaju bolje predviđanje labela.

- **Lokalni pooling sloj** - kod graf neuronskih mreža se često koriste pooling slojevi koji se javljaju i u drugim mrežama, kao što su konvolucione neuronske mreže. Pooling sloj smanjuje dimenzije ulazne matrice, tako što vrši agregaciju vektora u matrici. Primer je max pooling sloj koji deli matrice na jednake blokove a zatim svaki blok menja najvećom vrednošću tog bloka. Kod graf neuronskih mreža ova operacija omogućava smanjenje grafa.



- **Globalni pooling sloj** - kod nekih problema želimo da vršimo klasifikaciju celokupnog grafa. Tada je potrebno akumulirati reprezentacije svih čvorova i grana na osnovu koje vršimo klasifikaciju grafa.

2.3 Message passing sloj

Message passing slojevi kod graf neuronskih mreža predstavljaju permutaciono invarijantnu operaciju, što znači da ukoliko se promene oznake čvorova (a time i matrica susedstva) rezultat operacije ostaje isti, tj operacija nije osetljiva na oznake čvorova. Ovaj sloj se primenjuje tako što svaki od čvorova u grafu šalje svom susedu svoju reprezentaciju na osnovu kojih čvorova ažuriraju svoju reprezentaciju. Na ovaj način, 1 ovakav sloj omogućava čvorovima da prikupe informacije o čvorovima na distanci od 1 "skoka".Ređanjem n ovakvih slojeva, dobijaju se informacije o čvorovima koji su n skoka udaljeni.

Kako bi se omogućilo brže računanje, ova operacija se može definisati na sledeći način

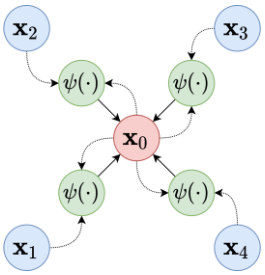
$$\hat{A} = \widetilde{D}^{-\frac{1}{2}} \tilde{A} \widetilde{D}^{-\frac{1}{2}}$$

pri čemu $\hat{A}$ predstavlja matricu koja predstavlja čvorove nakon prosleđivanja poruka, $\tilde{A}$ je matrica susedstva sa dodatim sopstvenim granama (grana koja spaja čvor sa samim sobom) a matrica $\widetilde{D}$ je dijagonalna matrica pri čemu element $d_{i,i}$ predstavlja broj ulaznih/izlaznih grana čvora i, pri čemu se određuje koren njene inverze.

Međutim, ubacivanjem previše ovakvih slojeva dovodi do 2 problema

- **Oversmoothing** - ukoliko previše slojeva ubacimo biće puno razmena poruka između čvorova pa će se njihove reprezentacije previše izmešati i postati približno jednake, čime se gube specifičnosti čvorova
- **Oversquashing** - ponavljanjem prosleđivanja poruka u svaki čvor stižu poruke iz dalekih delova grafa. Informacije o struktura grafa se time pokušavaju sačuvati u reprezentaciji čvorova, koja često nije dovoljna da opiše složene strukture koje se javljaju u grafovima

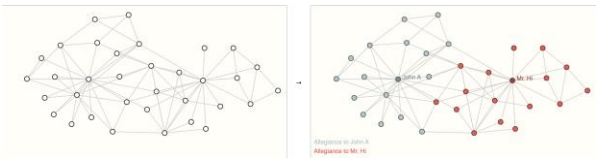
Ovi problemi se mogu donekle izbeći tehnikama kao što su skip konekcije kao u rezidualnim neuronskim mrežama.



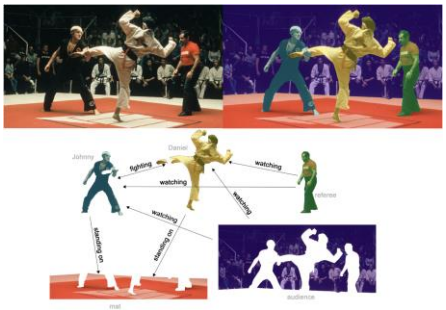
2.4 Problemi koje rešavaju graf neuronske mreže

Postoje 3 vrste problema za koje se mogu primeniti graf neuronske mreže

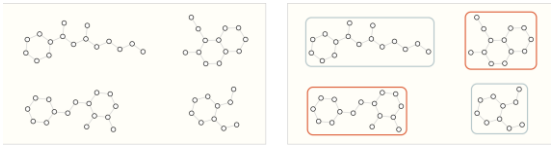
1. **Problemi na nivou čvorova -** odnose se na probleme gde je potrebno vršiti klasfikaciju čvorova ili regresiju za predviđanje vrednosti čvora. Najpoznatiji primer je nad skupom podataka koji predstavlja karate klub, pri čemu su čvorovi članovi a grane odnosi između članova. Graf neuronske mreže se u ovom slučaju koriste za predviđanje ko će se kom klubu pridružiti nakon podele kluba na dva kluba. Drugi poznati primeri su predviđanje napadača u grafu koji predstavlja mrežu, predviđanje zakrčenja u saobraćaju gde su čvorovi mesta gde su postavljene kamere a grane putevi, kao i mnogi drugi



2. **Problemi na nivou grana** - koriste se za predviđanje relacija između čvorova u grafu. Jedan primer su sistemi predlaganja (recommendation sistemi) gde grane povezuju korisnike i artikle koji ih možda interesuju. Još jedan primer ovakvih problema je kod računarskog vida, gde drugom mrežom (npr CNN) izdvajamo feature-e na slici (osobe, objekte...) a zatim graf neuronskom mrežom se predviđaju odnosi između objekata.



3. **Problemi na nivou grafa -** odnose se na klasifikaciju celog grafa. Interesantan primer je u hemiji, gde se pomoću graf neuronskih mreža mogu predvideti hemijske osobine (miris, reaktivnost, lekovitost...) na osnovu grafovske reprezentacije molekula.



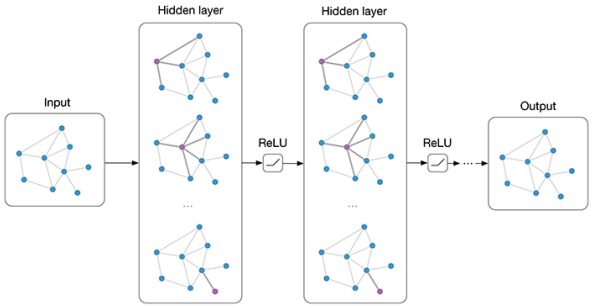
2.5 Varijacije graf neuronskih mreža

Postoje mnoge varijacije i arhitekture graf neuronskih mreža ali ovde ćemo dati opis tri najbitnije varijante.

Graf konvolucione neuronske mreže

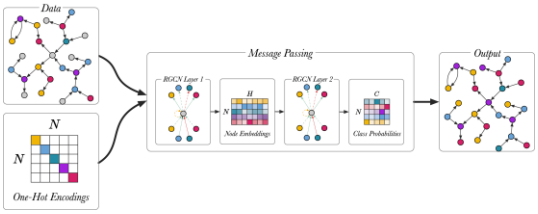
Ova vrsta graf neuronskih mreža funkcioniše tako što svaki čvor prvenstveno prikupi reprezentacije svih susednih čvorova, na osnovu neke funkcije vrši agregaciju kao što je maksimum ili prosek (local pooling layer) a zatim tako dobijenu reprezentaciju koristi kao ulaz u neuronsku mrežu. Ova procedura se sastoji od više koraka

1. Prikupljanje vektora koji predstavljaju susedne čvorove
2. Agregacija prikupljenih vrednosti, koja se često računa kao težinski prosek prikupljenih reprezentacija pri čemu najveću težinu imaju čvorovi sa manjim brojem povezanih grana.
3. Propuštanje kroz prvi sloj neuronske mreže i dobijanje reprezentacije sa manje dimenzija. Svi čvorovi dele istu neuronsku mrežu.
4. Za svaki sledeći sloj u mreži se ponavljaju operacije 1-3, odnosno opet se prikupljaju, sada manje dimenzionalni, vektori koji predstavljaju susedne čvorove, vrši se agregacija i zatim propuštanje kroz sledeći sloj sve dok se ne stigne do željenog broja dimenzija. Za primer binarne klasifikacije to bi bio jedan broj koji predstavlja verovatnoću da čvor pripada jednoj ili drugoj klasi.



Relacione graf konvolucione neuronske mreže

Nedostatak graf konvolucionih neuronskih mreža je u tome što podrazumevaju da sve veze u grafu predstavljaju istu relaciju. U realnim primerima to ne važi uvek - u grafu koji predstavlja društvenu mrežu grane mogu predstavljati činjenicu da je neko postavio komentar ali mogu predstavljati i relacije kao što su "prijateljstva". Zato su uvedene relacione graf konvolucione neuronske mreže koje mogu analizirati i kompleksnije grafove.



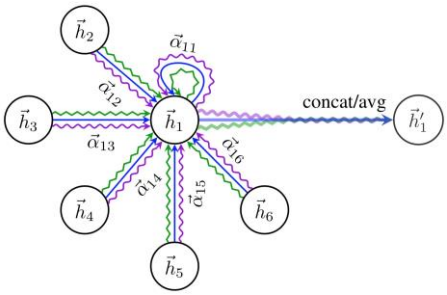
Kod graf konvolucione neuronske mreže svi čvorovi koriste zajedničku matricu težina. Kod relacionih mreža postoje više matrica težina

- Prvenstveno, svaka vrsta relacije u grafu dobija svoju posebnu matricu težina koja se uči vremenom.
- Pored toga, kod ove vrste mreža se koristi i posebna matrica W_0 koju čvor koristi za agregaciju sopstvene reprezentacije

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right),$$

Graf neuronske mreže sa attention-om

Kao što smo spomenuli, kod konvolucionih i relacionih graf neuronskih mreža u svakom koraku se vrši agregacija reprezentacija pri čemu se radi težinski prosek na osnovu broja grana datog čvora. Ovakva agregacija je jednostavna i nepromenjiva, tj mreža nije u mogućnosti da nauči vremenom koje reprezentacije su bitnije jer je agregacija unapred definisana. Zato je uveden attention mehanizam gde se funkcija agregacije uči vremenom. Data funckija će koristiti reprezentacije svih susednih čvorova kako bi odredila težinu odnosno "bitnost" informacija susednih čvorova. Nad rezultati funkcije se zatim primenjuje softmax, kako bi sve težine u zbiru dale 1.



3. Implementacija GCN-a u NumPy-u

3.1 Opis strukture implementacije

Za implementaciju graf konvolucione neuronske mreže koristićemo samo Numpy u Pythonu. Svi slojevi, aktivacione funckije, loss funkcije i sama mreža će biti implementirane korišćenjem osnovnih operacija. Projekat je podeljen u više fajlova, gde ćemo svaki od njih objasniti detaljnije.

1. helpers.py - u okviru ovog fajla su definisane pomoćne metode koje ćemo koristiti za definisanje slojeva i cele mreže. Ovde su definisane i implementirane različite aktivacione funkcije koje ćemo objasniti, loss funkcije, learning rate scheduleri kao i pomoćna funkcija za normalizaciju matrice susedstva. Ovde je implementirana i dropout funkcija za regularizaciju.
2. layers.py - ovde definišemo sam GCN sloj i sve potrebne funkcionalnosti. To uključuje
    - Inicijalizacija sloja
    - Forward funkcija koja prosledjuje ulazni signal i generiše izlaz
    - Backward funckija koja služi za vršenje backpropagacije na osnovu koje je trenirana mreža
    - Funkcije za inicijalizaciju težina korišćenjem Glorot i He inicijalizacije

    Pored toga, ovde je definisan i BatchNormalization sloj.
3. models.py - na osnovu slojeva definisanih u layers.py konstruišemo graf neuronsku mrežu. U okviru klase GCN potrebno je inicijalizovati graf neuronsku mrežu izborom broja neurona u skrivenom sloju, broja neurona u zadnjem sloju tj dimenzija izlaza, aktivacione funkcije, funkcije learning rate-a kao i načina inicijalizacije težina mreže. Tu je definisana i forward funkcija koja prosleđuje ulaz kroz mrežu na osnovu koje se dobija izlaz. Za trening mreže su definisane funkcije kojima se računa gradient težina i primenjuje u procesu treninga.
4. test.py - ovde je instancirana mreža i trenirana nad skupom podatka. Za testiranje ispravnosti mreže ćemo generisati nasumične podatke i trenirati mrežu. Kako performanse mreže ne mogu biti kvalitetno testirane nad nasumičnim podacima koristićemo u sledećem poglavlju više skupova podataka kako bi odredili performanse i moguće primene mreža.

U folderu data se nalaze skupovi podataka nad kojim je vršeno testiranje a u results se nalaze rezultati mreža kao što su preciznost i vizuelizacije grafova.

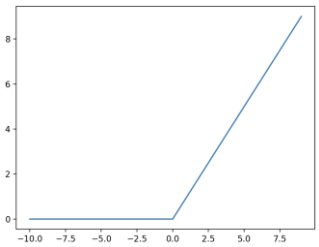
3.2 Opis aktivacionih funkcija i implementacija

Za potrebe neuronske mreže koju implementiramo, realizovaćemo 6 aktivacionih funkcija koje ćemo isprobati tokom treniranja i razmotrićemo kako utiču na vreme treniranja i performanse mreže. Implementiramo sledeće aktivacione funkcije

- ReLU - Rectified Linear Unit - ReLu predstavlja jednostavnu aktivacionu funkciju čiji matematičku izraz glasi

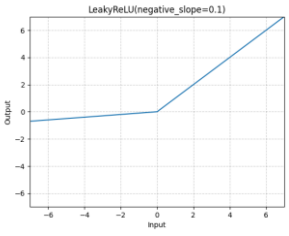
  $$ReLU(x) = \max(0, x)$$

  Jednostavnost ove funkcije je njena prednost s obzirom da je samo računanje dosta brža od drugih aktivacionih funkcija. Nedostatak je što funkcija nije glatka, to jest izvod funkcije u tački 0 nije jednak ukoliko x teži 0 sa leve i sa desne strane. Još jedan nedostatak je što se negativne vrednosti gube.

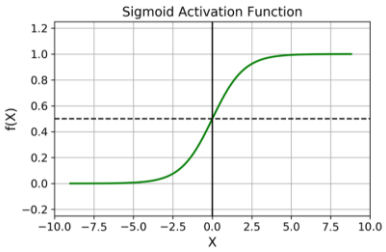
- LeakyReLU - Rešava neke od problema ReLU-a. Prvenstveno omogućava da se zadrže negativne vrednosti. Matematička definicija glasi

  $$LeakyReLU(x) = kx,\ x < 0$$
  $$LeakyReLU(x) = x,\ x \geq 0$$

- Sigmoid - Jedna od prvih upotrebljenih aktivacionih funkcija, s obzirom da se javlja u prirodi kod aktiviranja neurona. Nedostatak ove aktivacione funkcije je u eksplodirajućim/nestajućim gradijentima, s obzirom da kako apsolutna vrednost ulaza raste gradijent vrlo brzo dostiže vrednost približno 0. Pored svojih nedostatak i dalje se često koristi kod nekih neruonskih mreža kao što su LSTM-ovi.

  $$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

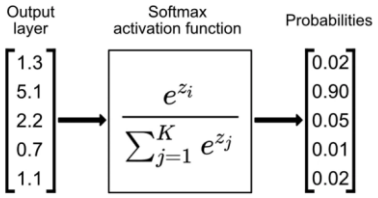


Postoji i varijacija sigmoid funkcije koja predstavlja glatku verziju ReLU-a - Swish koja je definisana kao

$$SWISH_\beta(x) = \beta Sigmoid(\beta x)$$

- Softmax - Koristi se kod klasifikacije sa više od 2 klase. Na osnovu ulaznog vektora x ova funkcija generiše vektor verovatnoće, tj vektor čiji zbir elemenata je jednak 1.

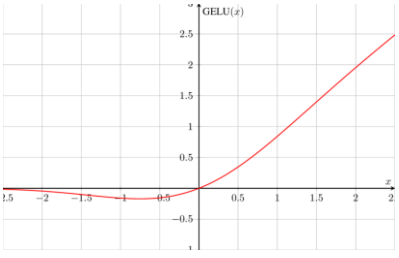
$$Soft(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$



- GELU - Gaussian Error Linear Unit. Predstavlja glatku aproksimaciju funkcije kumulativne raspodele verovatnoće. U svom originalnom zapisu je vrlo kompleksna i teška za računanje, te je previše sporo za treniranje neuronske mreže, međutim sledećom aproksimacijom je moguće implementirati GELU.

$$GELU_{\tanh}(x) = 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

Iako sporija, dovodi do dosta brže konvergencije.



Implementacije ovih aktivacionih funkcije su date na slici ispod. Realizovane su kao statičke metode klase `ActivationFunctions` koje se kasnije koriste u okviru neuronske mreže. Bitno je napomenuti da se softmax funkciji prosleđuje i parametar koji predstavlja osu, odnosno ukoliko su ulazni vektori višedimenzionalni moramo izabrati po kojoj dimenziji radimo softmax.

```python
class ActivationFunctions:
    @staticmethod
    def relu(x):
        return np.maximum(0, x)
    @staticmethod
    def gelu(x):
        return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * np.power(x, 3))))
    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))
    @staticmethod
    def swish(x):
        return x * ActivationFunctions.sigmoid(x)
    @staticmethod
    def leaky_relu(x, negative_slope=0.2):
        return np.maximum(negative_slope * x, x)
    @staticmethod
    def softmax(x, axis=-1):
        exp_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
        return exp_x / np.sum(exp_x, axis=axis, keepdims=True)
```

3.3 Loss funkcije

Kako bi trenirali neuronsku mrežu moramo definisati meru kvaliteta izlaza neuronske mreže. Česta mera kod problema klasifikacije gde labele predstavljaju verovatnoće je cross-entropy loss koji meri razliku između dve raspodela verovatnoća - tačne raspodele i predviđene. Definisana je izrazom

$$H = -\sum p(x)\log\big(p(x)\big)$$

Za treniranje mreže je potrebno koristiti i izvod loss funkcije pa zato implementiramo funkciju koja računa cross-entropy loss kao i njen izvod

```python
class LossFunctions:
    @staticmethod
    def cross_entropy_loss(predictions, targets):
        # Avoid log(0) by clipping predictions
        epsilon = 1e-6
        predictions = np.clip(predictions, epsilon, 1. - epsilon)
        n = targets.shape[0]
        return -np.sum(targets * np.log(predictions)) / n
    @staticmethod
    def cross_entropy_loss_derivative(predictions, targets):
        # Derivative of cross-entropy loss with respect to predictions
        epsilon = 1e-6
        predictions = np.clip(predictions, epsilon, 1. - epsilon)
        n = targets.shape[0]
        return (predictions-targets)/n
```

3.4 Learning rate

U neuronskim mrežama, nakon računanja gradijenta greške, moramo primeniti dati gradijent na težinama mreže kako bi mreža učila i dostigla bolje performanse. Međutim, ukoliko previše promenimo težine mreža može "preskočiti" najbolje rešenje a ukoliko previše sporo menjamo težine možemo biti sigurniji da će mreža naći dobro rešenje ali će duže trajati. Faktor koji utiče na to koliko brzo se težine menjaju je learning rate koji može biti statički tj unapred definisan ali je mnogo bolje menjati learning rate sa epohama. Postoji više algoritama koje možemo primeniti, međutim ovde ćemo spomenuti i implementirati dva

1. Statički learning rate - u svakoj epohi učenja learning rate će biti isti.
2. Eksponencijalno opadajući learning rate - sa povećanjem broja epohe learning rate opada po eksponencijalnom zakonu. Uvodi se i faktor koji utiče na brzinu opadanja learning rate-a k .

```python
class LearningRate:
    @staticmethod
    def static_lr(epoch, start_rate):
        return start_rate
    @staticmethod
    def exp_lr(epoch, start_rate):
        k = 0.1
        return start_rate * np.exp(-k*epoch)
```

3.5 Normalizacija matrice susedstva

Za message passing algoritam je potrebno odrediti matricu $\hat{A}$ kao što je pomenuto u 2.3. Zato je implementirana pomoćna funkcija koja implementira formulu.

```python
class MatrixHelper:
    @staticmethod
    def normalize_adjacency_matrix(A):
        I = np.eye(A.shape[0])
        A_hat = A + I
        D_hat = np.diag(np.sum(A_hat, axis=1))
        D_hat_inv_sqrt = np.linalg.inv(np.sqrt(D_hat))
        return D_hat_inv_sqrt @ A_hat @ D_hat_inv_sqrt
```

3.6 Inicijalizacija težina neuronske mreže

Pre nego što se počne sa treniranjem neuronske mreže potrebno je inicijalizovati težine neuronske mreže. Postoje više pristupa a ovde ćemo objasniti He i Glorot inicijalizaciju. Ovi algoritmi incijalizacije težina su nastali usled problema tokom treniranja neuronskih mreža. Ukoliko se težine podese na 0, što je najjednostavniji pristup, izlazi neurona će biti 0 a time i gradijenti, pa mreža neće učiti. Drugi metod je inicijalizacija težina nasumično, tako da se sempluje normalna raspodela. Ovo dovodi do problema da varijansa izlaza slojeva opada, što dovodi do problema da mreža opet ne uči. Zato se težine mogu bolje incijalizovati na sledeća dva načina

1. Glorot inicijalizacija - težine se dobijaju semplovanjem uniformne distirbucije pri čemu su granice distribucije dobijene formulom

$$r = \sqrt{\frac{6}{fan_{in} + fan_{out}}}$$

gde su i brojevi ulaza odnosno izlaza neuronske mreže.

2. He incijalizacija - sličan princip rada kao Glorot inicijalizacija međutim granice su date kao

$$r = \sqrt{\frac{6}{fan_{in}}}$$

```python
def glorot_initialization(self, input_dim, output_dim):
    limit = np.sqrt(6 / (input_dim + output_dim))
    return np.random.uniform(-limit, limit, (input_dim, output_dim))

def he_initialization(self, input_dim, output_dim):
    limit = np.sqrt(6 / (input_dim))
    return np.random.uniform(-limit, limit, (input_dim, output_dim))
```

3.7 Implementacija GCN sloja

Sloj graf konvolucione neuronske mreže je implementiran u okviru klase `GCNLayer`. Prvenstveno se vrši incijalizacija u okviru koje se bira tip inicijalizacije težina, težine se inicijalizuju, bira se algoritam za računanje learning rate-a i bira da li koristimo i bias pored težina.

```
def __init__(self, input_dim, output_dim, init_method = 'glorot', use_bias = True, start_lr = 0.1, learning_rate = helpers.LearningRate.exp_lr):
    init_method_dict = {
        'glorot':self.glorot_initialization,
        'he':self.he_initialization,
    }
    if init_method not in init_method_dict.keys():
        init_method = 'glorot'
    self.W = init_method_dict[init_method](input_dim, output_dim)
    self.use_bias = use_bias
    if self.use_bias:
        self.bias = np.zeros(output_dim)
    self.learning_rate = learning_rate
    self.start_lr = start_lr
```

Pored toga, imamo još dve funkcije - forward i backward. U forward proceduri se ulazni signal prosleđuje na izlaz neurona. Forward procedura se vrši po sledećoj formuli

$$H_{i+1} = \hat{A}\, H_i\, W + b,\ H_o = X$$

```
def forward(self, A_hat, H):
    self.A_hat = A_hat
    self.h_prev = H

    H_prime = H@self.W
    H_new = A_hat @ H_prime

    if self.use_bias:
        H_new += self.bias

    return H_new
```

Potrebno je još implementirati backward prolaz kako bi trenirali mrežu korišćenjem algoritma

$$\nabla W = H_{i-1}^{T}(\hat{A})\frac{\partial\, loss(Hi)}{\partial\, H_{i+1}}$$

pri čemu je prvobitno H_i izlaz neuronske mreže poslednjeg sloja a H_{i+1} ciljne labele.

```
def backward(self, dH, epoch):
    dH_prime = dH @ self.W.T
    self.dW = self.h_prev.T @ (self.A_hat @ dH)

    if self.use_bias:
        self.db = np.sum(dH, axis=0)

    self.W -= self.learning_rate(epoch, self.start_lr) * self.dW
    if self.use_bias:
        self.bias -= self.learning_rate(epoch, self.start_lr) * self.db

    return dH_prime
```

3.8 Implementacija GlobalPool sloja

Kako bi omogućili klasifikaciju celog grafa implementiraćemo i GlobalPoolLayer koji će na osnovu svih reprezentacija čvorova konstruisati jedinstvenu reprezentaciju koju možemo koristiti kao ulaz višeslojnog perceptrona kojim predviđamo klasu grafa. Ovaj sloj nema težine, već na ulazima primenjuje samo operaciju sumiranja, maksimalnog elementa ili proseka (metoda se bira pri konstrukciji sloja). Kako sloj nema težine onda na se na njemu ne primenjuje backpropagation, već samo vraća 0.

```
class GlobalPoolLayer:
    def sum(self, X):
        return np.sum(X, axis = 0, keepdims=True)
    def max(self, X):
        return np.max(X, axis = 0, keepdims=True)
    def avg(self, X):
        return np.mean(X, axis = 0, keepdims=True)

    def __init__(self, method='sum'):
        if method not in ['sum','avg','max']:
            print("Unknown global pool method")
            return
        method_map = {
            'sum':self.sum,
            'max':self.max,
            'avg':self.avg,
        }
        self.method = method_map[method]

    def forward(self, A_hat, H):
        self.input = H
        return self.method(H)

    def backward(self, dH, epoch):
        return np.zeros_like(self.input)
```

3.9 Implementacija Dense sloja

Kako bi koristili reprezentaciju grafa za predviđanje labele, potreban nam je Dense sloj. Kao i kod GCN sloja, i ovde inicijalizujemo težine pomoću He ili Glorot inicijalizacije, kao i learning rate metodu. Forward će samo računati

$$Y = XW + b$$

dok će backward funkcija odrediti gradijent i modifikovati težine i bias.

3.10 Implementacija droput-a

Kako se mreža ne bi overfit-ovala potrebno je na neki način izvršiti regularizaciju. Jedna opcija je l1 ili l2 regularizacija, ali ćemo u ovom slučaju koristiti dropout. Dropout funkcioniše tako što se u svakoj epohi tokom treninga neki neuroni u mreži isključe. Na taj način se mreža forsira da nauči reprezentaciju podataka ne oslanjajući se previše na susedne neurone. Za svaki neuron postoji verovatnoća p odnosno u kodu `dropout_rate` da će on biti isključen. Tokom inferencije, odnosno korišćenja mreže nakon treninga, dropout se ne primenjuje.

```python
class Regularization:
    @staticmethod
    def dropout(X, dropout_rate=0.5, training=True):
        if not training or dropout_rate == 0.0:
            return X
        mask = np.random.binomial(1, 1 - dropout_rate, size=X.shape)
        return X * mask / (1 - dropout_rate)
```

3.11 Implementacija graf neuronske mreže

Sada kada imamo GCN sloj implementiran, implementacija cele graf neuronske mreže je jednostavna. Prvo moramo izabrati veličinu ulaza, izlaza, broj skrivenih slojeva i broj neurona u njima kao i aktivacionu funkciju, funkciju promene learning rate-a i inicijalizaciju težina, kao i da li poslednji sloj koristi global pool za klasifikaciju grafova. Pored toga, ovde specifiramo i dropout rate, tj verovatnoću da će neuron biti isključen u toku jedne epohe tokom treninga. To je urađeno u konstruktoru klase `GCN`.

```python
def __init__(self, n_features, hidden_dims, output_dim, init_method='glorot', use_bias=True,
             activation = helpers.ActivationFunctions.gelu, dropout_rate = 0.1, start_lr = 0.1,
             learning_rate = helpers.LearningRate.exp_lr, global_pool = None):
    self.hidden_dims = hidden_dims      dmilosevic252, last month • Refactor
    self.output_dim = output_dim
    self.activation = activation
    self.learning_rate = learning_rate
    self.dropout_rate = dropout_rate
    self.name=f'GCN_{n_features}x'
    # Initialize layers
    self.layers = []
    input_dim = n_features
    for hidden_dim in hidden_dims:
        self.layers.append(layers.GCNLayer(input_dim, hidden_dim, init_method, use_bias, start_lr, learning_rate))
        self.name+=str(hidden_dim)+'x'
        input_dim = hidden_dim

    self.name+=str(output_dim)
    if global_pool is not None:
        self.name+='_globalPool'
        self.layers.append(layers.GlobalPoolLayer(global_pool))
        self.layers.append(layers.DenseLayer(hidden_dim, output_dim, init_method, start_lr, learning_rate))
    else:
        self.layers.append(layers.GCNLayer(hidden_dim, output_dim, init_method, use_bias, start_lr, learning_rate))
```

Nakon toga je potrebno omogućiti trening i inferenciju same mreže. Inferencija je implementirana funkcijom `forward`, pri čemu se propušta signal od prvog do poslednjeg sloja. Poslednji sloj koristi softmax funkciju kako bi dobili verovatnoće za klasifikaciju. Tokom treninga, primenjuje se dropout.

```python
def forward(self, A, X, training = False):
    A_hat = helpers.MatrixHelper.normalize_adjacency_matrix(A)
    H = X
    for layer in self.layers[:-1]:
        H = self.activation(layer.forward(A_hat, H))
        if training:
            H = helpers.Regularization.dropout(H, dropout_rate=self.dropout_rate, training=training)
    H = self.layers[-1].forward(A_hat, H)
    H = helpers.ActivationFunctions.softmax(H)
    return H
```

Potrebno je još implemetirati treniranje mreže - to jest prvo propustiti signal, izmeriti grešku i ažurirati težine. Omogućen je early stopping, odnosno ako performanse mreže se ne poboljšavaju za neku količinu treniranje staje odmah. Takođe, moguće je trenirati mrežu i kada prosledimo više grafova odjednom, tada se treniraju jedan po jedan u svakoj epohi. Na kraju treninga se iscrtava graf zavisnosti vrednosti loss-a od broja epohe.

```python
def train_instance(self, A, X, Y, epoch, mask = None):
    predictions = self.forward(A, X, True)
    if mask is not None:
        loss = self.compute_gradients(predictions*mask, Y*mask, epoch)
        return loss
    loss = self.compute_gradients(predictions, Y, epoch)
    return loss

def train(self, adjecancy, data, targets, epochs=100, early_stopping = None, mask = None):
    prev_loss = 0
    epoch_losses = []

    if len(data.shape) == 3 and len(adjecancy.shape) == 3:
        if data.shape[0] != adjecancy.shape[0]:
            print("Adjecancy and data have to have same number of matrices")
            return

    for epoch in range(epochs):
        if len(data.shape) == 3 and len(adjecancy.shape) == 3: #If we have multiple matrices, multiple graphs - for graph classification
            total_loss = 0
            for i, X in enumerate(data):
                total_loss += self.train_instance(adjecancy[i], X, targets[i], epoch, mask)
        else:
            total_loss = self.train_instance(adjecancy, data, targets, epoch, mask)

        epoch_losses.append((epoch, total_loss))
        print(f"Epoch {epoch + 1}/{epochs}, Total Loss: {total_loss}")

        if early_stopping is not None and abs(total_loss-prev_loss) < early_stopping:
            return
        if np.isnan(total_loss):
            return
        prev_loss = total_loss

    plt.plot(epoch_losses)
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.show()
```

Moguće je specifirati masku za trening ili testiranje, odnosno niz koji označava koje čvorove treba ili ne treba uključiti u računanju vrednosti lossa tokom treninga i testiranja. Ovo ćemo koristiti za procenu performansi mreže.

3.12 Testiranje implementacije

Potrebno je još testirati mrežu na nekom prostom primeru kako bi proverili da sve radi. Kako je mreža pravljena bez global pooling layer-a koristićemo u ovom, kao i u daljim primerima, za zadatke na nivou čvorova, tj previđaćemo labele čvorova.

Za potrebe testiranja je dodata i pomoćna klasa `Metrics` sa statičnom metodom `accuracy` kojom merimo preciznost predviđanja modela. Ova metoda nalazi indeks sa maksimalnom verovatnoćom u svakoj labeli i bira taj indeks kao klasu koja se zatim upoređuje sa pravom labelom.

```python
class Metrics:
    @staticmethod
    def accuracy(y_true, y_pred_probs):
        y_pred = np.argmax(y_pred_probs, axis=1)

        if len(y_true.shape) > 1:
            y_true = np.argmax(y_true, axis=1)

        correct_predictions = np.sum(y_true == y_pred)
        total_predictions = y_true.shape[0]

        accuracy = correct_predictions / total_predictions
        return accuracy
```

Zatim ćemo pokrenuti trening nad nasumično generisanim grafovima pri čemu će se broj čvorova u grafu povećavati od 5 do 10000 čvorova u 6 koraka, broj feature-a svakog čvora će biti 20. Svaki put kada mreža bude trenirana štampaćemo accuracy kao i vreme treniranja. Model ima 2 skrivena sloja sa po 3 čvora u svakom sloju. Vrši se klasifikacija sa 5 klasa, tako da očekujem accuracy oko 20%, s obzirom da su podaci nasumični.

```
_____________
Training with 5 nodes
_____________
Using 20 features
Epoch 1/100, Loss: 26.795784178771083
Epoch 2/100, Loss: 2.9433914601285786e-08
Accuracy: 0.4
Train time: 0.0003080368041992187 5 s


_____________
Training with 2004 nodes
_____________
Using 20 features
Epoch 1/100, Loss: 37.89716981096274
Epoch 2/100, Loss: 46.13999771896742
Epoch 3/100, Loss: 2.5047933252226777e-08
Accuracy: 0.21407185628742514
Train time: 0.7109838379406084 s


_____________
Training with 4003 nodes
_____________
Using 20 features
Epoch 1/100, Loss: 20.866026437927637
Epoch 2/100, Loss: 2.4982993471273912e-08
Accuracy: 0.20634524010911981
Train time: 4.152876959609985 s


_____________
Training with 6002 nodes
_____________
Using 20 features
Epoch 1/100, Loss: 20.616811165793688
Epoch 2/100, Loss: 2.5104257359120236e-08
Accuracy: 0.20459846717760746
Train time: 12.937769889831543 s


_____________
Training with 8001 nodes
_____________
Using 20 features
Epoch 1/100, Loss: 21.2828078073853
Epoch 2/100, Loss: 2.5045612850917106e-08
Accuracy: 0.20109986251718534
Train time: 31.96138906478882 s


_____________
Training with 10000 nodes
_____________
Using 20 features
Epoch 1/100, Loss: 37.62337169122144
Epoch 2/100, Loss: 2.4928157415924221e-08
Accuracy: 0.2078
Train time: 53.428889751434326 s
```

Kao što možemo videti preciznost je oko 20% što smo i očekivali, izuzev kada imamo mali broj čvorova kada postoji blagi overfitting.

Pored toga, moguće je dodati i GlobalPool layer i vršiti predikciju labele celog grafa. Ispod je dat jedan primer gde smo ručno specifirali 2 grafa. Takođe, dodajemo funkciju measure_accuracy koja vrši predviđanje i merenje tačnosti mreže.

```python
def measure_accuracy(self, adjecancy, data, targets):
    predicted = []
    if len(data.shape) == 3 and len(adjecancy.shape) == 3:
        #If we have multiple matrices, multiple graphs - for graph classification
        if data.shape[0] != adjecancy.shape[0]:
            print("Adjecancy and data have to have same number of matrices")
            return
        for i, X in enumerate(data):
            predicted.append(self.forward(adjecancy[i], X))
    else:
        predicted = self.forward(adjecancy, data)
    return helpers.Metrics.accuracy(targets, predicted)
```

```python
import numpy as np
import models
A = np.array([
    [[0, 1, 0, 0],
     [1, 0, 1, 1],
     [0, 1, 0, 1],
     [0, 1, 1, 0]
    ],
    [[0, 1, 0, 0],
     [1, 0, 1, 1],
     [0, 1, 0, 1],
     [0, 1, 1, 0]
    ]])

X = np.array([
    [
    [1, 0, 1],
    [0, 1, 0],
    [1, 1, 0],
    [0, 0, 1]
    ],
    [
    [1, 0, 0],
    [0, 1, 0],
    [1, 1, 0],
    [0, 0, 1]
    ]])

targets = np.array([
    [1, 0],
    [0, 1]
])

gcn = models.GCN(3, hidden_dims=[16, 16], output_dim=targets.shape[1], global_pool='max')
gcn.train(A, X, targets=targets, early_stopping=0.001)
print(gcn.measure_accuracy(A, X, targets))
```

```
Epoch 1/100, Total Loss: 0.7471398520925184
Epoch 1/100, Total Loss: 0.7471398520925184
Epoch 2/100, Total Loss: 0.7415001263781025
Epoch 3/100, Total Loss: 0.7365955836010705
Epoch 4/100, Total Loss: 0.7322297930589681
Epoch 5/100, Total Loss: 0.728306528403156
Epoch 6/100, Total Loss: 0.7247681851005179
Epoch 1/100, Total Loss: 0.7471398520925184
Epoch 2/100, Total Loss: 0.7415001263781025
Epoch 3/100, Total Loss: 0.7365955836010705
Epoch 4/100, Total Loss: 0.7322297930589681
Epoch 1/100, Total Loss: 0.7471398520925184
Epoch 2/100, Total Loss: 0.7415001263781025
Epoch 2/100, Total Loss: 0.7415001263781025
Epoch 3/100, Total Loss: 0.7365955836010705
Epoch 4/100, Total Loss: 0.7322297930589681
Epoch 5/100, Total Loss: 0.728306528403156
Epoch 6/100, Total Loss: 0.7247681851005179
Epoch 7/100, Total Loss: 0.7215731792650971
Epoch 8/100, Total Loss: 0.7186874563853748
Epoch 9/100, Total Loss: 0.7160812486326305
Epoch 10/100, Total Loss: 0.7137278046165281
Epoch 11/100, Total Loss: 0.7116028538601262
Epoch 12/100, Total Loss: 0.7096843359406338
Epoch 13/100, Total Loss: 0.7079522187569539
Epoch 14/100, Total Loss: 0.7063883437832714
Epoch 15/100, Total Loss: 0.7049762790742633
Epoch 16/100, Total Loss: 0.7037011761631781
Epoch 17/100, Total Loss: 0.7025496317826739
Epoch 18/100, Total Loss: 0.7015095562480235
Epoch 19/100, Total Loss: 0.7005700499953087
1.0
```

Mreža se uspešno trenira pa sada možemo pogledati neke realne primene.

4. Primena graf neuronskih mreža

Koristićemo graf neuronsku mrežu na primeru Cora skupa podataka koji predstavlja skup naučnih radova i njihove međusobne veze, odnosno koji rad citira koji drugi rad. Cilj je predvideti temu naučnog rada na osnovu atributa rada kao i veza između rada. Pokazaćemo kako pripremiti skup podataka, atribute neuronske mreže koje dovode do dobrih rezultata kao i same rezultate. Takođe ćemo uporediti sa klasičnim tehnikama (random forest classifier) kako bi pokazali prednost graf neuronskih mreža kod skupova podataka koji imaju grafovsku strukturu.

4.1 Pregled skupa podataka

Skup podatka čine 2708 čvorova, odnosno naučnih radova kao i 5429 veza između datih radova. Svaki rad je opisan 1433 dimenzionalnim vektorom, koji predstavljaju atribute izvučene iz svakog naučnog rada. Na kraju je potrebno predvideti temu naučnog rada - jednu od 7 mogućih.

```
'Case_Based': [1, 0, 0, 0, 0, 0, 0],
'Genetic_Algorithms': [0, 1, 0, 0, 0, 0, 0],
'Neural_Networks': [0, 0, 1, 0, 0, 0, 0],
'Probabilistic_Methods': [0, 0, 0, 1, 0, 0, 0],
'Reinforcement_Learning': [0, 0, 0, 0, 1, 0, 0],
'Rule_Learning': [0, 0, 0, 0, 0, 1, 0],
'Theory': [0, 0, 0, 0, 0, 0, 1]}
```

Podatke pre svega pročitamo iz fajla a zatim se obrada vrši na sledeći način

- Podaci o čvorovima su dati u zapisu gde prvo ide identifikator čvora, zatim 1433 atributa i na kraju ciljnu labelu. Ovde kreiramo
    - Mapu identifikatora koja preslikava identifikator čvora na indeks čvora u matrici feature-a
    - Matricu featura-a
    - Niz ciljnih labela, koje kasnije enkodiramo korišćenjem one-hot labela
- Podaci o granama su dati kao niz parova indeksa. Svaki par označava vezu između čvorova sa datim identifikatorima. Na osnovu toga kreiramo matricu susedstva

Podatke delimo na 3 dela

- Ime tj id čvora
- Feature-e svakog čvora
- Ciljni atribut, tj temu naučnog rada

```
# Split - 1. element je id cvora, od 2. do zadnjeg su featuri, zadnji je klasa
ids = []
features = []
targets = []

for row in data:
    split_row = row.split('\t')
    ids.append(split_row[0])
    feature_row = list(map(int, split_row[1:-1]))
    features.append(feature_row)
    targets.append(split_row[-1])
```

✓ 0.2s

Odredimo i preslikavanje, tj kako se kodira svaka tema (koristimo one hot encoding)

```
def get_onehot(num, max):
    res = np.zeros(max)
    res[num] = 1
    return list(map(int,res))
unique = np.unique(targets)
target_map = {str(x): get_onehot(y, len(unique)) for y,x in enumerate(unique)}

target_map
```

✓ 0.0s

```
A = np.zeros((len(features),len(features)))
A
```

✓ 0.0s

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
for edge in edges:
    start_id, end_id = list(map(int, edge.split('\t')))
    start_index = id_map[start_id]
    end_index = id_map[end_id]
    A[start_index,end_index] = 1
    A[end_index,start_index] = 1
A
```

✓ 0.0s

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

4.2 Treniranje Random Forest Classifier-a

Nad datim skupom podataka možemo trenirati Random Forest klasifikator kako bi dobili rezultate koje ćemo kasnije porediti sa graf neuronskim mrežama. Podelićemo skup podataka tako da 80% se koristi za trening i 20% za testiranje. Treniranjem Random Forest klasifikatora sa 300 estimatora dobijamo preciznost od oko 72%, što su solidni rezultati za algoritam koji ne uzima u obzir veze između čvorova, odnosno naučnih radova.

```
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.metrics import accuracy_score
```

✓ 0.1s

```
label_encoded_target = np.array([int(np.argmax(x)) for x in y])
```

✓ 0.0s

```
train_indices = [i for i,x in enumerate(train_mask) if x == [1]]
test_indices  = [i for i,x in enumerate(test_mask) if x == [1]]
```

✓ 0.0s

```
X_train = X[train_indices]
y_train = label_encoded_target[train_indices]

X_test = X[test_indices]
y_test = label_encoded_target[test_indices]
```

✓ 0.0s

```
rfc = RFC(n_estimators=300)
rfc.fit(X_train, y_train)
print(f'Random forest, 300 estimators - Accuracy: {accuracy_score(y_test, rfc.predict(X_test))}')
```

✓ 1.6s

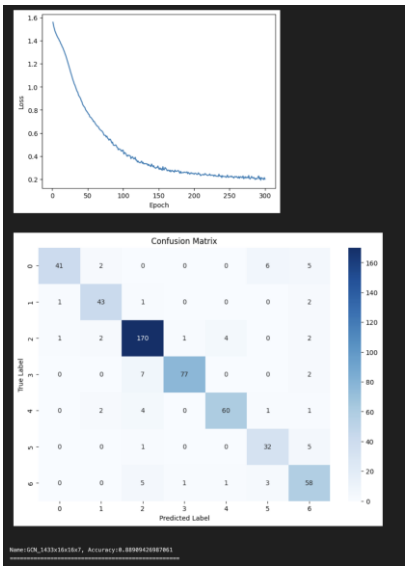
```
Random forest, 300 estimators - Accuracy: 0.711645101663586
```

4.3 Treniranje graf neuronske mreže

Sada definišemo arhitekturu graf neuronsku mreže. Isprobavanjem različitih parametara nađena su dobra podešavanja

- Koristimo 2 graf konvoluciona sloja sa po 16 neurona
- Aktivaciona funkcija je swish. Gelu je takođe dobra opcija ali swish daje bolje rezultate
- Dropout rate je 0.2, odnosno svaki neuron ima 20% šanse da bude isključen
- Learning rate je relativno visok - 0.5 i ne menja se u toku treninga. Zbog toga mreža relativno brzo konvergira dobrom rešenju
- Mreža trenira tokom 300 epoha
- 80% skupa se koristi za trening, 20% za testiranje
- Koristi se glorot inicijalizacija

Nakon 300 epoha, mreža dostiže loss od oko 0.20 i preciznost nad trening skupom od oko 89%, što je znatno bolje od Random Forest klasifikatora. Povećanjem broja slojeva ne dolazi do poboljšanja rezultata. To je posledica jednostavnosti skupa podataka, daleke veze nisu toliko bitne za dati problem.



I kod relativno prostog problema kao što je Cora dataset, graf neuronske mreže postižu znatno bolje rezultate kod grafovskih podataka nego klasične metode. Pored toga, ovde je realizovana relativno prosta arhitektura, dok složenije arhitekture koriste i skip veze između slojeva i attention mehanizme pa time postižu još bolje rezultate nego ovde obrađen primer.

5. Literatura

[1] **A Gentle Introduction to Graph Neural Networks** - https://distill.pub/2021/gnn-intro/

[2] **Graph neural network** - https://en.wikipedia.org/wiki/Graph_neural_network

[3] **Graph Neural Network and Some of GNN Applications** - https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications

[4] **What Are Graph Neural Networks** - https://blogs.nvidia.com/blog/what-are-graph-neural-networks/

[5] **Graph neural networks in data mining** - https://cse.msu.edu/~mayao4/dlg_book/chapters/chapter12.pdf

[6] **Understanding Convolutions on Graphs** - https://distill.pub/2021/understanding-gnns/

[7] **Graph Convolutional Networks made simple** - https://www.youtube.com/watch?v=2KRAOZIULzw&list=PLSgGvve8UweGx4_6hhrF3n4wpHf_RV76_&index=2

[8] **The Cora dataset** - https://graphsandnetworks.com/the-cora-dataset/

[9] **Hands-on Graph Neural Networks with PyTorch Geometric (1): Cora Dataset** - https://medium.com/@koki_noda/ultimate-guide-to-graph-neural-networks-1-cora-dataset-37338c04fe6f

[10] **Graph neural networks: A review of methods and applications** - https://arxiv.org/pdf/1812.08434