Andrea Grendene **Matricola: 1184985**

Analisi di OpenKE e pykg2vec

Seconda parte del progetto di Apprendimento Automatico

1. Introduzione

OpenKE e pykg2vec sono due librerie di apprendimento automatico di embeddings per i knowledge graphs, o KG. Essi sono sono dei grafi multi-relazionali composti da entità (nodi) e relazioni (i diversi tipi di archi). Ogni arco è formato da una tripla (head entity, relation, tail entity), chiamata anche fatto (fact), che rappresenta una specifica relazione fra le due entità. Vengono usati per modellare le relazioni fra i termini, e infatti sono stati utilizzati con successo per risolvere problemi di semantic parsing, named entity disambiguation, information extraction, question answering e così via. Data la loro natura prettamente simbolica di queste triple, i KG sono generalmente difficili da manipolare, nonostante siano efficaci per rappresentare dati strutturati. Per risolvere il problema è stato proposto con successo un nuovo ramo di ricerca, chiamato knowledge graphs embedding. L'idea è di incorporare le componenti del KG in spazi vettoriali continui, in modo da semplificarne la manipolazione e preservare la struttura del grafo[1].

1.1. Knowledge graphs e modelli principali

La maggior parte delle tecniche attualmente disponibili eseguono l'embedding basandosi unicamente sui fatti osservati. Dato un KG, esse rappresentano innanzitutto le entità e le relazioni in uno spazio vettoriale continuo, quindi definiscono una funzione di valutazione, che assegna un punteggio ad ogni fatto per misurarne la plausibilità. Gli embeddings delle entità e delle relazioni possono poi essere ottenuti massimizzando la plausibilità totale dei fatti osservati. Durante questa procedura gli embeddings appresi devono soltanto essere compatibili con ogni fatto individuale, e perciò potrebbero non essere sufficientemente predittivi per i downstream tasks. Di conseguenza sempre più ricercatori hanno iniziato a sfruttare altri tipi di informazioni, come i tipi delle entità, i percorsi delle relazioni, le descrizioni testuali e le regole logiche, per apprendere embeddings sempre più predittivi.
Una tipica tecnica di embedding dei KG consiste in 3 passi: rappresentazione delle entità e delle relazioni, definizione della funzione di valutazione e apprendimento di tale rappresentazione. Il primo passo specifica in quale forma le entità e le relazioni vengono rappresentate negli spazi vettoriali continui: solitamente le entità vengono definite come dei vettori, ossia come dei punti deterministici nello spazio vettoriale, mentre le relazioni vengono considerate come delle operazioni in tale spazio, e perciò possono essere rappresentate come dei vettori, delle matrici, dei tensori, delle distribuzioni gaussiane multivariate o un miscuglio di Gaussiane. Nel secondo passo viene definita una funzione di valutazione $f_r(h, t)$ per ogni fatto (h, r, t) per misurarne la plausibilità; i fatti osservati nel KG tenderanno ad avere punteggi maggiori rispetto a quelli non osservati. Infine nell'ultimo passo vengono apprese le rappresentazioni delle entità e delle relazioni, ossia gli embeddings, risolvendo un problema di ottimizzazione che massimizza la plausibilità totale dei fatti osservati.
Le tecniche di embedding possono essere divise in due categorie: i *translational distance models* usano funzioni di valutazione basate sulla distanza fra le due entità, mentre i *semantic matching models* usano funzioni di valutazione basate sulla similarità fra le entità. Per questo progetto sono state analizzate tre tecniche, TransE, TransH e TransR, che appartengono tutte alla prima categoria.
TransE è il translational distance model di base, e rappresenta le entità e le relazioni come vettori nello

stesso spazio. Dato un fatto *(h, r, t)*, la relazione viene interpretata come come un vettore traslazionale **r** tale per cui le entità embedded **h** e **t** possono essere connesse con un errore minimo, ossia **h + r ≈ t** quando si verifica *(h, r, t)*. L'idea è di apprendere le rappresentazioni distribuite delle parole per catturare le regolarità linguistiche come *Psyco – AlfredHitchcock ≈ Avatar – JamesCameron*. Nei dati multi-relazionali tale analogia si verifica per la relazione *DirectorOf*, e tramite essa possiamo ottenere *AlfredHitchcock + DirectorOf ≈ Psycho* e *JamesCameron + DirectorOf ≈ Avatar*. È possibile visualizzare una semplice illustrazione di questa idea nella Fig. 1a. La funzione di valutazione viene perciò definita come le distanza negativa fra **h + r** e **t**, ossia

$$f_r(h, t) = -||\boldsymbol{h} + \boldsymbol{r} - \boldsymbol{t}||_{1/2.}$$

Se il fatto *(h, r, t)* si verifica, allora il punteggio dovrebbe essere elevato.
Nonostante la sua semplicità ed efficienza, TransE ha problemi nel gestire relazioni 1 a N, N a 1 e N a N. Ad esempio, se si considerano N film di Alfred Hitchcock, i risultati della funzione di valutazione, e quindi le rispettive rappresentazioni vettoriali, saranno molto simili fra loro, sebbene le entità siano completamente differenti. Una possibile soluzione al problema è utilizzare diverse rappresentazioni della stessa entità quando viene usata con relazioni differenti. In questo modo gli N film manterranno un embedding simile con la relazione *DirectorOf*, ma con altre potrebbero adottare delle rappresentazioni molto distanti fra loro. Tale idea viene implementata da TransH, usando degli iperpiani specifici per ogni relazione. Come è possibile notare nella Fig. 1b, questa tecnica modella ancora le entità come dei vettori, ma ogni relazione viene rappresentata come un vettore **r** in un iperpiano, con $\mathbf{w}_r$ come vettore normale. Dato un fatto *(h, r, t)*, le rappresentazioni delle entità **h** e **t** vengono proiettate nell'iperpiano. Quindi viene assunto che esse siano collegate dalla relazione **r** con un errore minimo se il fatto è verificato, e si ricava di conseguenza la funzione di valutazione.
TransR segue la stessa idea di TransH, ma usa spazi specifici per le relazioni anziché gli iperpiani, come è possibile notare in Fig. 1c. In questo caso ogni relazione viene modellata come un vettore traslazionale dello spazio corrispondente, e di conseguenza, dato un fatto *(h, r, t)*, le rappresentazioni delle entità **h** e **t** vengono proiettate nel medesimo spazio. La funzione di valutazione rimane la stessa di TransH, ma la proiezione delle entità viene rappresentata da una matrice anziché da un vettore, e quindi è possibile modellare più facilmente le relazioni complesse. Di conseguenza però sono necessari *O(dk)* parametri per ogni relazione, dove **d** e **k** sono le dimensioni rispettivamente dello spazio delle entità e di quello della relazione, contro gli *O(d)* parametri richiesti da TransE e TransH. Viene perciò persa la semplicità e l'efficienza dei modelli precedenti[1].

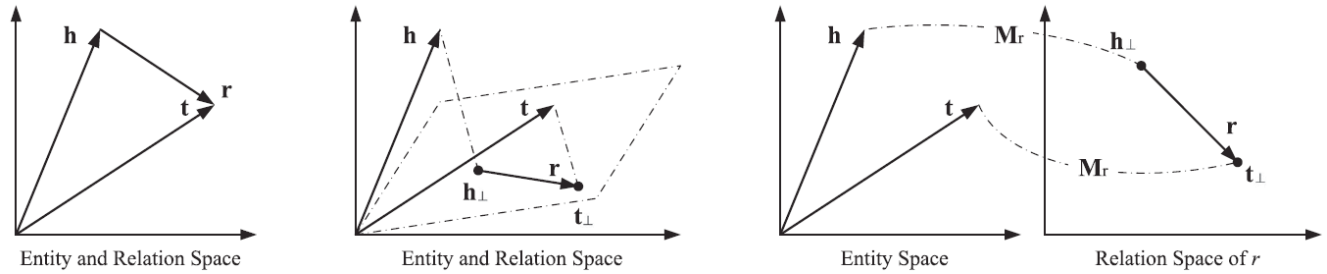

Fig. 1. Semplice illustrazione di TransE, TransH e TransR.

1.2. Uso della GPU per l’apprendimento automatico

Per i processi di apprendimento automatico di solito si preferisce utilizzare la GPU anziché la CPU. Il motivo risiede nella maggiore velocità di memorizzazione e trasferimento dei dati, e in una più elevata esecuzione dei calcoli in aritmetica in virgola mobile[2]. Infatti la GPU contiene migliaia di core per eseguire in parallelo operazioni molti semplici, mentre la CPU ne ha pochi e per effettuare calcoli più complessi[3]. Inoltre nel 2007 NVIDIA ha creato CUDA, una piattaforma di programmazione parallela che permette agli sviluppatori di sfruttare le GPU per elaborazioni a scopo generico. In questo modo è possibile manipolare in modo efficiente grandi blocchi di dati, come avviene nei processi di apprendimento automatico. Due librerie che permettono già di utilizzare le GPU sono PyTorch e TensorFlow, ossia quelle usate nelle due repository analizzate[2].
Come già detto, è possibile usare CUDA per utilizzare la GPU e definire come eseguire le varie operazioni all’interno della scheda, ma soltanto per GPU NVIDIA. In alternativa è possibile usare OpenCL, una libreria open source, funzionante su qualsiasi GPU, ma meno performante perché non può sfruttare le caratteristiche specifiche di ogni scheda[4]. Oppure esiste HIP, un’applicazione che permette di convertire il codice CUDA in C++, in modo da poterlo eseguire sia con GPU AMD, sia con quelle NVIDIA[5].
Nel caso delle librerie analizzate dovrebbe essere possibile ottenere molti benefici con l’utilizzo della GPU, dato che vengono effettuati numerosi calcoli vettoriali e matriciali. È necessario perciò verificare quanto effettivamente la GPU venga usata e se è possibile ottenere ulteriori miglioramenti.

1.3. Uso del parallelismo

Dato che nei processi di apprendimento automatico vengono eseguiti numerosi calcoli indipendenti fra loro, se si hanno a disposizione più CPU o GPU è possibile effettuarli in parallelo per ottenere un miglioramento delle prestazioni. Non tutte le operazioni possono ottenere benefici con un’esecuzione distribuita, quindi è necessario valutare attentamente i casi in cui ha senso applicarla. Ad esempio la moltiplicazione di matrici e il calcolo della distanza vettoriale sono operazioni da poter eseguire efficacemente in parallelo. Oppure, nel caso fosse necessario applicare la k-fold cross-validation ad un modello sufficientemente piccolo, è possibile eseguire contemporaneamente più apprendimenti con configurazioni di training set e validation set differenti[6].
Nel caso delle deep neural network, esistono due tecniche che possono essere applicate, chiamate *data parallelism* e *model parallelism*. La prima consiste nell’equa suddivisione dei dati di ogni mini-batch per ogni thread: se ad esempio vengono usati mini-batch da 128 esempi e ho 4 GPU, allora ad ognuna verranno assegnati 32 esempi. Di conseguenza la fase forward avviene completamente in parallelo, e soltanto alla fine sarà necessario sincronizzare tutti i risultati per il calcolo del gradiente. Durante la fase backward, invece, bisogna sincronizzare i processi durante il calcolo dei gradienti, perciò, se la matrice dei pesi è grande, è possibile avere dei colli di bottiglia durante la trasmissione del gradiente tra i vari nodi, che possono essere attenuati con alcuni accorgimenti[7].

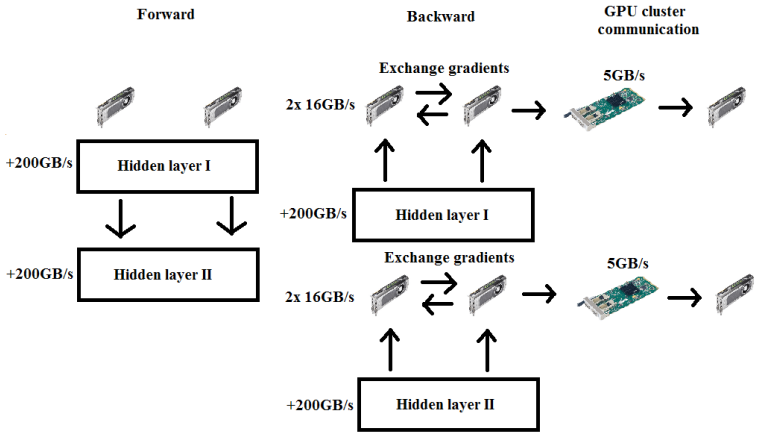

Fig. 2. Diagramma del data parallelism. Non c'è communicazione nella fase forward, e durante la fase backward è necessario sincronizzare i gradienti.

Il model parallelism, invece, prevede la divisione dell'apprendimento del singolo dato in più processi, ossia il modello viene diviso fra i vari thread. Esistono vari approcci che possono essere applicati, uno di questi prevede la suddivisione dei pesi: se ad esempio ho una matrice dei pesi 1000x1000 e 4 GPU, è possibile dividerla in matrici 1000x250. La conseguenza è che durante entrambe le fasi dell'apprendimento è necessario sincronizzare i nodi dopo ogni prodotto scalare con la matrice dei pesi, ma allo stesso tempo è possibile utilizzare modelli molto grandi, che potrebbero non essere contenuti nella memoria di una singola GPU. Infatti il model parallelism è ideale per i modelli grandi, mentre il data parallelism funziona meglio con quelli piccoli[8].

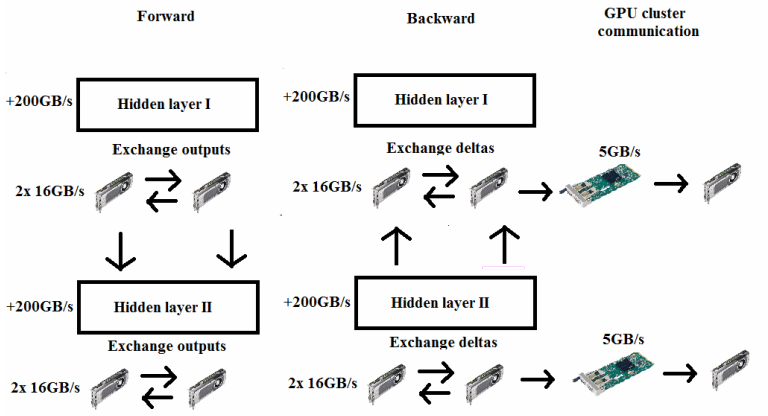

Fig. 3. Diagramma del model parallelism. Risulta necessaria la sincronizzazione dopo ogni prodotto scalare con la matrice dei pesi sia per la fase forward sia per quella backward.

Per quanto riguarda i KG, ed in particolare TransE, TransH e TransR, i modelli risultano essere molto semplici, e quindi non ha senso considerare un adattamento del model parallelism. Gli esempi di base proposti nelle repository delle due librerie utilizzano numerosi dati, divisi in mini-batch. Perciò si potrebbero ottenere dei miglioramenti applicando il data parallelism, in particolare quando viene usata la CPU. Le operazioni effettuate dai modelli analizzati sono principalmente prodotti scalari di vettori e

matrici, e quindi potrebbero essere realizzate in parallelo. In tal caso si otterrebbe un vero beneficio nel caso di spazi vettoriali di dimensioni elevate, in cui l'esecuzione di tali calcoli risulterebbe essere onerosa.

2. OpenKE

La repository della libreria si trova al seguente link: https://github.com/thunlp/OpenKE.

2.1. Informazioni generali

La versione base di OpenKE è scritta con PyTorch, e quindi necessita della versione 3.7 di Python. Non richiede installazione, basta scaricare la copia della repository ed eseguire il file "make.sh", situato dentro a "openke/", con il comando del terminale "bash make.sh". Dopodiché è sufficiente copiare uno dei file dentro a "examples/" nella directory principale ed eseguirlo.
Esiste anche una versione della libreria scritta con TensorFlow anziché PyTorch, di cui ho analizzato velocemente il codice.
Non è stato possibile eseguire il file d'esempio, ossia "train_transe_FB15K237.py", su Windows 10, a causa di un errore di link con una libreria. In teoria su un sistema operativo Linux dovrebbe funzionare correttamente.

2.2. Struttura del codice

All'interno di "train_transe_FB15K237.py" vengono eseguite 6 operazioni principali: caricamento dei dati per il training, caricamento dei dati per il testing, definizione del modello, definizione della loss function, training del modello e testing del modello.
Le prime due operazioni vengono svolte rispettivamente dagli oggetti TrainDataLoader e TestDataLoader, situati nei file "openke/data/TrainDataLoader.py" e "openke/data/TestDataLoader.py". Essi preparano tutti i dati per le fasi successive, utilizzando delle funzioni, scritte in C++, situate nei file all'interno di "openke/base/".
Il modello viene definito dalle corrispettive classi contenute dentro ai file di "openke/module/model/", che contengono i metodi per eseguire le operazioni specifiche della tecnica scelta. I modelli implementati sono "Analogy", "ComplEx", "DistMult", "HolE", "RESCAL", "RotatE", "SimplE", "TransD", "TransE", "TransH" e "TransR".
La loss function viene rappresentata dalla classe NegativeSampling, contenuta nel file "openke/module/strategy/NegativeSampling.py".
Le ultime due operazioni vengono svolte rispettivamente dagli oggetti Trainer e Tester, situati nei file "openke/config/Trainer.py" e "openke/config/Tester.py". Al loro interno sono definiti vari parametri, fra cui "use_gpu", un flag che permette di attivare o disattivare l'utilizzo della GPU. All'interno di Trainer è definito anche un parametro "work_threads", che presumo indichi il numero di thread da avviare ma non viene usato da alcuna parte.
La versione della libreria scritta con Tensorflow è strutturata in modo un po' diverso: il file d'esecuzione per il training è "example_train_transe.py", e quello per il testing è "example_test_transe.py". Dentro alla classe Config, situata all'interno di "config/Config.py", vengono definiti il caricamento dei dati e lo svolgimento del training e del testing. I modelli invece sono definiti all'interno della cartella "models".

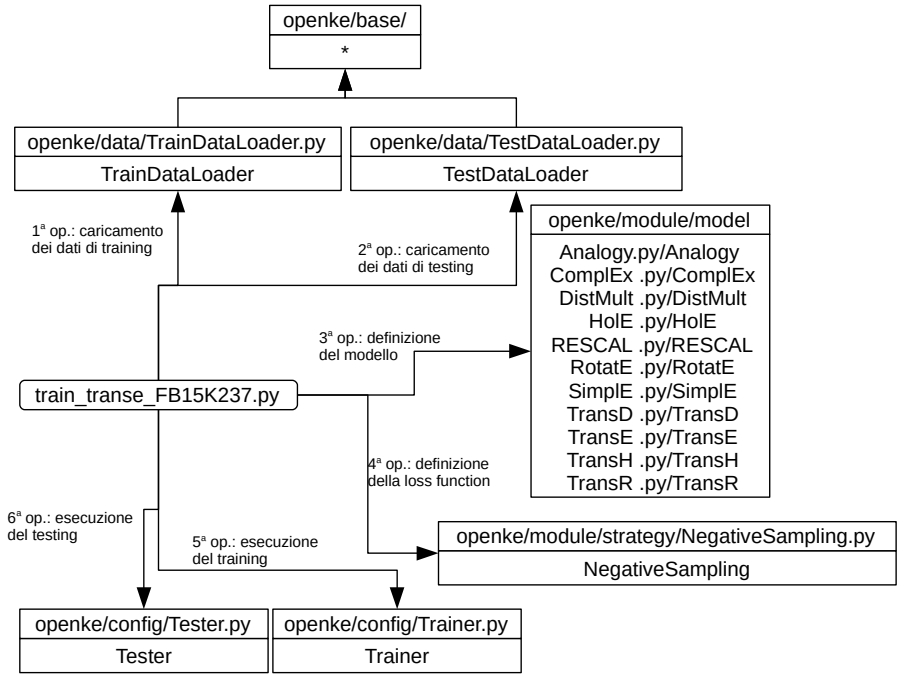

Fig. 4. Schema delle classi principali di OpenKE.

2.3. Possibili miglioramenti

Non avendo provato ad eseguire il codice, non è possibile avere un'idea di quanto vengano usate la CPU e la GPU. Perciò non è possibile sapere se si possono ottenere dei miglioramenti delle prestazioni facendo eseguire dei calcoli dalla scheda grafica.
Per quanto riguarda il parallelismo non c'è nulla che porti a pensare ad una sua implementazione già esistente, ad eccezione della fase di caricamento dei dati, che però costituisce un passo molto veloce da eseguire. Un possibile suggerimento, da capire come applicare, è di usare la classe "torch.nn.DataParallel[9]" per le GPU. In teoria dovrebbe poter essere usata direttamente in Trainer, e quindi non dovrebbe essere necessario modificare ogni file dei modelli. In alternativa si potrebbero implementare direttamente la creazione e la gestione dei thread, anche se ovviamente si tratterebbe di un lavoro molto più lungo. Tale soluzione risulta essere l'unica valida per quanto riguarda le CPU, dato che non è prevista alcuna classe per gestirne automaticamente l'esecuzione in parallelo.
La versione della libreria scritta con TensorFlow dovrebbe sfruttare automaticamente il parallelismo con le GPU, ma bisognerebbe verificarlo per esserne certi. Viene fissato il numero di schede disponibili all'interno dei file "example_train_transe.py" e "example_test_transe.py" tramite l'istruzione "os.environ['CUDA_VISIBLE_DEVICES']", anche se in teoria TensorFlow dovrebbe identificare ed usare tutte le GPU ponendo il valore a 0. È assente un'opzione per utilizzare soltanto le CPU, che potrebbe tornare utile in alcune situazioni.

3. pykg2vec

La repository della libreria si trova al seguente link: https://github.com/Sujit-O/pykg2vec. Invece la repository con la libreria modificata si trova a questo indirizzo: https://github.com/Cenze94/pykg2vec.

3.1. Informazioni generali

La libreria usa TensorFlow 1.13.1, perciò è vivamente consigliato installarla in un ambiente virtuale apposito. Inoltre per installare TensorFlow è consigliato usare Conda, in modo da poter installare e configurare CUDA automaticamente[10]. È possibile installare pykg2vec tramite PiP, oppure scaricando il codice dalla repository e installandolo manualmente. La libreria è stata provata su una macchina con Windows 10, una CPU Intel Core i7-7700HQ da 2.80 GHz (4 core fisici, 8 core virtuali) e una GPU NVIDIA GeForce GTX 1050.
Per installare la versione modificata della libreria dovrebbe essere sufficiente scaricarne una copia dalla repository, spostarla nella directory contenente le altre librerie della versione utilizzata di Python, e rinominare la cartella da "pykg2vec-master" a "pykg2vec". In alternativa è possibile installare la versione normale di pykg2vec, e sostituire i file con quelli della versione modificata.

3.2. Struttura del codice

Non c'è un file per eseguire il testing, mentre il file di esecuzione del training è "train.py", che in sostanza esegue 4 operazioni. La prima è l'analisi delle opzioni ricevute in input, dato che è possibile specificare vari parametri nel comando di esecuzione via terminale; ad esempio è possibile decidere di utilizzare TransH scrivendo "python train.py -mn TransH". La seconda è la preparazione del dataset per il training e il testing. La terza è l'importazione e il salvataggio dei vari dati di configurazione. L'ultima infine è l'inizializzazione del trainer, a cui segue la costruzione del modello e l'avvio del training.
Per l'analisi delle opzioni ricevute viene creato un oggetto KGEArgParser, definito nel file "pykg2vec/config/config.py". In quest'oggetto vengono definiti tutti i comandi, specificabili nell'istruzione di esecuzione del terminale, per impostare vari parametri, come il tipo di modello da usare, il percorso del dataset, la dimensione del batch in fase di training, e così via. Per rendere effettivamente attive queste opzioni viene utilizzato un oggetto di tipo ArgumentParser, appartenente alla libreria "argparse".
La preparazione del dataset per il training e il testing avviene attraverso un oggetto di tipo KnowledgeGraph, definito nel file "pykg2vec/utils/kgcontroller.py". Innanzitutto è possibile utilizzare un proprio dataset, fornendo il percorso e il nome della cartella, oppure uno fra i seguenti: "FreebaseFB15k", "DeepLearning50a", "WordNet18", "WordNet18_RR", "YAGO3_10", "FreebaseFB15k_237", "Kinship", "Nations", "UMLS". Se il dataset non è predefinito viene verificata la validità del percorso; altrimenti viene controllato se è già presente in locale, ed eventualmente viene scaricato ed estratto. In entrambi i casi vengono salvati i metadati e i percorsi dei dati di training, testing e validation, che infine verranno analizzati e salvati a loro volta nei parametri di KnowledgeGraph.
L'importazione dei dati di configurazione avviene tramite un oggetto Importer, sempre definito nel file "pykg2vec/config/config.py". Esso restituisce due oggetti, config_def e model_def, che rappresentano i dati di configurazione e la classe del modello scelto; essi vengono ricavati rispettivamente dalle altre classi contenute in "pykg2vec/config/config.py" e dai file in "pykg2vec/core". Ad esempio, nel caso di TransE, vengono restituiti oggetti di tipo TransEConfig e TransE. Successivamente l'oggetto con i dati di configurazione viene inizializzato con gli argomenti della prima operazione, e verrà usato a sua volta per inizializzare il modello.
L'inizializzazione del trainer inizia con la creazione di una variabile Trainer, definita nel file "pykg2vec/utils/trainer.py". Nel costruttore vengono semplicemente preparati i parametri della classe, mentre la costruzione del modello e l'avvio del training avvengono nei metodi "build_model()" e "train_model()", invocati nel file d'esecuzione.

In build_model() viene costruito il modello, definendo i tipi degli input, i parametri, la loss function, l'ottimizzatore e altri oggetti da usare durante il training. Inoltre viene avviata la sessione di TensorFlow. Tutte queste definizioni fanno riferimento ai metodi delle classi dei modelli contenuti nei file di "pykg2vec/core/", derivate dalla classe astratta TrainerMeta. Più precisamente i modelli già implementati, secondo il Readme di GitHub, sono Complex, ConvE, DistMult, KG2E, NTN, ProjE, RESCAL, RotatE, SLM, SME, TransD, TransE, TransH, TransM, TransR e TuckER, mentre come file, quindi probabilmente previsti ma non testati, sono presenti anche ConvKB, HoLE e TransG.
In train_model() viene innanzitutto caricato il salvataggio del modello, se ne esiste già uno, oppure viene definito un nuovo generatore. Successivamente viene stampato il riassunto dei risultati finali del training, mentre gli embeddings ottenuti vengono salvati in un file di formato ".tsv" e la sessione viene chiusa. Per quanto riguarda il generatore, innanzitutto viene inizializzato un oggetto GeneratorConfig, definito in "pykg2vec/config/global_config.py", contenente i dati di configurazione del generatore. Dopodiché viene inizializzato un oggetto Generator, definito nel file "pykg2vec/utils/generator.py", passando le configurazioni del generatore e del modello. Esso avvia dei thread per preparare le triple di dati per il training; ogni thread avrà un proprio buffer, in cui verranno salvate le triple già pronte. Tramite la configurazione del generatore è possibile definire la dimensione del buffer e il numero di processi da avviare. Dopo il generatore viene inizializzato un oggetto Evaluation, definito nel file "pykg2vec/utils/evaluation.py", per poter effettuare la validazione del modello dopo ogni epoca. In esso viene avviato un thread per coordinare tutte le operazioni di validazione e salvare i risultati ottenuti. L'ultimo passo prevede l'esecuzione del training vero e proprio, per il numero di epoche indicato nei dati di configurazione del modello.

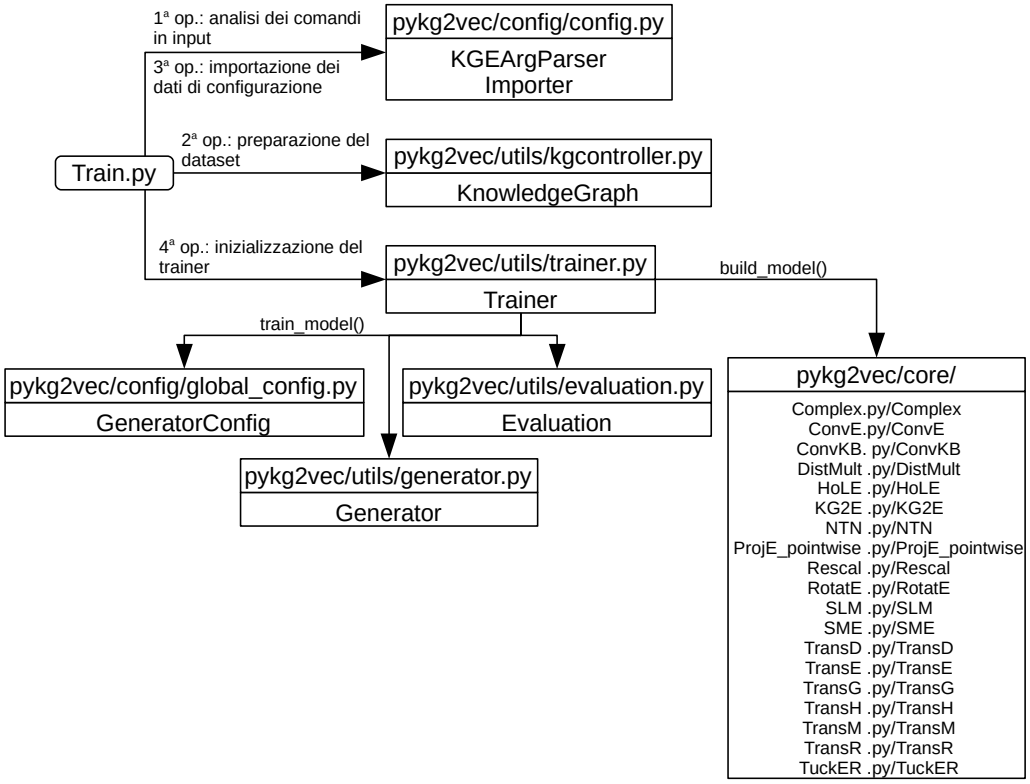

Fig. 5. Schema delle classi principali di pykg2vec.

3.3. Modifiche eseguite

Dato che inizialmente non riuscivo ad utilizzare la GPU durante l'esecuzione, ho provato ad aggiornare la libreria per poter usare TensorFlow 2.0. Successivamente ho scoperto che il problema riguardava un mio errore nell'installazione di TensorFlow, ma comunque ho mantenuto la modifica per poter usufruire delle ultime funzionalità del framework. Inoltre in questo modo è possibile usare la documentazione più recente e gli aiuti relativi alle versioni più nuove di TensorFlow. Per poter eseguire l'aggiornamento ho usato uno script apposito del framework, chiamato "tf_upgrade_v2.py". Quindi ho aggiunto l'istruzione "tf.compat.v1.disable_eager_execution()" al costruttore della classe Trainer, dato che nell'ultima versione di TensorFlow l'eager execution è attiva di default, e di conseguenza entra in conflitto con il metodo "tf.compat.v1.placeholder()".
Nelle classi KGEArgParser e BasicConfig è stata aggiunta l'opzione "use_gpu": si tratta di un flag per indicare se usare anche le GPU o soltanto le CPU, utile ad esempio per provare l'applicazione nel caso una GPU valida non sia disponibile. Al momento questa configurazione viene usata solo per indicare se rendere le GPU visibili a CUDA, ma potrebbe tornare utile anche in altri punti dove è possibile usare la libreria di NVIDIA, perché l'assenza di GPU potrebbe portare ad errori a runtime.

3.4. Risultati ottenuti

È stato eseguito soltanto l'esempio proposto nella descrizione della libreria, che usa il dataset "Freebase15k" e il modello TransE. Utilizzando soltanto la CPU si ottiene un tempo di training di ogni epoca che varia fra i 70 e i 90 secondi, mentre con la GPU i tempi calano ad un intervallo compreso fra i 20 e i 25 secondi. Perciò l'utilizzo della scheda grafica permette già di ottenere un elevato incremento delle prestazioni, probabilmente dovuto alle istruzioni di Tensorflow, dato che nell'applicazione la libreria CUDA non viene utilizzata. Analizzando la percentuale di utilizzo delle due schede prima e durante il training, usando il Task Manager di Windows, si nota però un fatto curioso: la CPU passa dal 25-40% circa di utilizzo al 75-95%, mentre la GPU passa dallo 0% al 4-10%. Di conseguenza si potrebbero ottenere notevoli aumenti delle prestazioni se si riuscisse a far svolgere alla scheda grafica parte del lavoro della CPU. Infatti se si riuscisse a riequilibrare la percentuale di utilizzo delle due schede si potrebbero aumentare i batch di training e testing, velocizzando l'apprendimento del modello. Un'altra prova che ho fatto usando soltanto la CPU è variare il numero di processi avviati dal generatore e la dimensione dei loro buffer. L'intenzione era di capire se si potevano ottenere degli aumenti di prestazioni con queste modifiche, ma i tempi di esecuzione non sono migliorati. Facendo qualche controllo in più ho potuto constatare che durante il training i buffer risultavano essere già pieni, perciò l'apprendimento del modello rappresentava il collo di bottiglia dell'intera esecuzione.

3.5. Possibili miglioramenti

Il primo miglioramento riguarda un aggiornamento della libreria per poter riabilitare l'eager execution, dato che dovrebbe migliorare le prestazioni dell'applicazione. Probabilmente si tratterebbe di un lavoro abbastanza lungo, dato che bisognerebbe correggere ogni istruzione che entra in conflitto con questa opzione.
In questa libreria viene applicato il parallelismo in alcuni casi, ed è da verificare se negli altri venga applicato automaticamente da TensorFlow con più GPU disponibili. In caso contrario potrebbe essere sfruttato molto di più: ad esempio si potrebbe applicare il data parallelism, eseguendo l'aggiornamento delle variabili del modello su più dati del batch in contemporanea. Di conseguenza bisognerebbe modificare il codice all'interno della classe Trainer, in cui viene definito come avviene l'esecuzione

dell’apprendimento. Inoltre si potrebbe valutare l’applicazione del parallelismo dentro a Evaluation, per cercare di eseguire più velocemente la validazione.
Come già fatto notare nella sottosezione precedente, la GPU viene usata molto poco, quindi è probabile che si possano migliorare le prestazioni dell’esecuzione massimizzando il numero di operazioni eseguite dalla scheda grafica. Bisognerebbe perciò verificare cosa viene calcolato ancora dalla CPU, e se è trasferibile alla GPU attraverso CUDA e TensorFlow.

Bibliografia

1. Q. Wang, Z. Mao, B. Wang e L. Guo, “Knowledge Graph Embedding: A Survey of Approaches and Applications”, in “*IEEE Transactions on Knowledge and Data Engineering*”, Vol. 29, No. 12, December 2017, pp. 2724-2740.
2. https://towardsdatascience.com/how-to-use-gpus-for-machine-learning-with-the-new-nvidia-data-science-workstation-64ef37460fa0
3. https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/
4. https://streamhpc.com/blog/2011-06-22/opencl-vs-cuda-misconceptions/
5. https://github.com/ROCm-Developer-Tools/HIP
6. https://www.kdnuggets.com/2016/11/parallelism-machine-learning-gpu-cuda-threading.html/2
7. https://timdettmers.com/2014/10/09/deep-learning-data-parallelism/
8. https://timdettmers.com/2014/11/09/model-parallelism-deep-learning/
9. https://pytorch.org/tutorials/beginner/former_torchies/parallelism_tutorial.html
10. https://towardsdatascience.com/tensorflow-gpu-installation-made-easy-use-conda-instead-of-pip-52e5249374bc