Apprendimento Distribuito e un Caso di Studio: TensorFlow

Maxim Gaina e Bartolomeo Lombardi
Lavoro di progetto per Sistemi Peer-to-Peer, Università degli Studi di Bologna

Questo lavoro mira ad apprendere e riassumere le basi della libreria TensorFlow [1], definita come un'interfaccia per esprimere algoritmi in ambito Machine Learning e con la possibilità di implementarli. Successivamente, l'intenzione sarà quella di esplorare le primitive offerte in ambito del calcolo distribuito, e dato un problema facilmente risolvibile tramite reti neurali, fare delle prove pratiche per ottenere una soluzione soddisfacente, usando un sistema di macchine connesse.

Indice

Cosa è TensorFlow

TensorFlow nasce dalla necessità di avere un sistema con le giuste proprietà per *allenare* e adoperare reti neurali in ambienti distribuiti su larga scala. La computazione di un programma, scritto in TensorFlow, è eseguibile su piattaforme multiple ed eterogenee con un minimo o senza alcun cambiamento. Per ogni piattaforma presa in considerazione, è previsto lo sfruttamento delle risorse dei device a disposizione, con capacità di calcolo, come processori centrali (CPU) e acceleratori di grafica (GPU). Le computazioni, vengono espresse da flussi di dati che scorrono all'interno di un grafo, il quale, ogni nodo che lo compone, ha un proprio stato. Ulteriori obiettivi, sono quelli di fornire un linguaggio *flessibile*, che permetta la rapida implementazione di diversi modelli; un linguaggio il più possibile *performante* nonostante la flessibilità appena citata; e forme di parallelismo con requisiti più o meno forti, anche per passare con estrema facilità da ambienti isolati ad ambienti distribuiti.

In questa relazione, verranno quindi riassunte le principali caratteristiche di TensorFlow. In alcuni casi verrà citato direttamente [1] e alcuni schemi o paragrafi al suo interno per aiutarne la comprensione. Il tutto verrà poi combinato con un caso di studio pratico della libreria.

1 Paradigma e Concetti Base

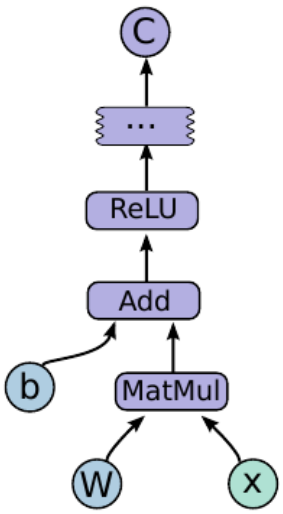

Figura 1: *Esempio di Grafo di Computazione, corrispondente alla Figura 2*

Una computazione TensorFlow è descritta da un *grafo diretto*, composto da un insieme di *nodi*. Il grafo supporta computazioni, ossia flussi di dati che lo attraversano. I nodi mantengono e aggiornano i propri stati persistenti tramite estensioni, che permettono quindi, di creare cicli e altre strutture. Ogni nodo, ha zero o più input/output, descrivibile come l'istanziazione di un'operazione. I valori che *fluiscono* nel grafo, sono detti **tensor**, array di dimensione n nel quale ogni elemento ha un tipo determinato a tempo di costruzione del grafo. Esistono anche particolari tipi di archi, il cui unico scopo è quello di indicare le dipendenze fra nodi, cioè l'ordine di esecuzione. Vediamo ora meglio alcuni concetti.

Operazione Avente nome e attributi, rappresenta l'astrazione di una computazione (es. `add`, `mulmatrix`, etc) su tensors in ingresso.

Kernel Implementazione particolare di un'operazione che le permette di essere eseguito su device particolari come una `GPU CUDA`.

Sessione I grafi dell'utente interagiscono con TensorFlow (vengono lanciati) attraverso sessioni, le quali, avendo un'interfaccia, possono essere usate anche per modificare il grafo a run-time. Le sessioni, hanno quindi un'operazione *Run*, che prendono in input il grafo stesso e i tensori.

Variabile I grafi vengono spesso eseguiti più volte e i tensors devono sopravvivere fra un'esecuzione e l'altra, per questo sono state create le variabili che salvano il loro valore, quando necessario. Le variabili sono anche esse tipi di nodo.

2 Ambienti Multi-Device e Distribuiti

Le componenti principali del sistema TensorFlow sono il *client* e il *master*. Il client usa le sessioni per comunicare con il master. Vengono usati uno o più *worker process* per accedere a uno o più *device* con capacità di calcolo. L'interfaccia di TensorFlow è stata implementata sia per ambiente locale che distribuito.

Locale Client, master e worker si trovano sulla stessa macchina nel contesto di un singolo processo di sistema operativo.

Distribuito Client, master e worker si trovano in diversi processi potenzialmente su macchine fisiche diverse.

Ogni worker è responsabile di uno o più *device* (Figura 3), e ogni device ha il suo tipo e un nome. Nello scenario del singolo device, è abbastanza facile eseguire il grafo: le dipendenze fra nodi sono facilmente individuabili e l'esecuzione dei nodi viene gestita tramite una pila, secondo una politica non meglio specificata. Discorso diverso per device multipli, e quindi anche per l'ambiente distribuito.

2.1 Esecuzione su device multipli

Per gestire la presenza di device multipli vengono adottati nuovi accorgimenti, è infatti necessario saper decidere a quale device assegnare la computazione di un dato nodo e saper comunicare fra grafi o sottografi che si trovano in posti diversi.

Posizionamento dei nodi Una delle prime preoccupazioni di TensorFlow, è quella di mappare i nodi all'insieme dei device disponibili, questo viene fatto attraverso un apposito **algoritmo di posizionamento**. In base ad alcune

```
b = tf.Variable(tf.zeros([100]))
W = tf.Variable(tf.random_uniform([784,100],-1,1))
x = tf.placeholder(name="x")
relu = tf.nn.relu(tf.matmul(W, x) + b)
s = tf.Session()
for step in xrange(0, 10):
    input = ...
    result = s.run(C, feed_dict={x: input})
```

Figura 2: *Costruzione di un grafo di computazione e il suo lancio tramite sessione*

```
with tf.Session() as sessione:
        with tf.device("/gpu:1"):
                matrice1 = tf.constant([[3., 3.]])
                matrice2 = tf.constant([[2.],[2.]])
                prodotto = tf.matmul(matrice1, matrice2)
```

Figura 3: *Vincolo di esecuzione tramite la `GPU 1` presente nel sistema*

euristiche, a tempo statico viene approssimato l'input di tale algoritmo, e viene detto **modello di costo**. Per ogni nodo, esso contiene la stima dei suoi tensori in input e in output espressa in byte e la stima del tempo di esecuzione. Ora che possiede l'input, l'algoritmo di posizionamento lancia un'esecuzione simulata del grafo, dove in base a euristiche *greedy* è in grado di scegliere l'opportuno device.

Comunicazione tra Device A questo punto, ogni device all'interno del sistema ha un proprio sottografo da eseguire. Ogni arco che unisce sottografi diversi e che quindi va dal nodo x a y, viene sostituito con un arco che va da x a *send* e un'altro che va da *recv* a y. *send* e *recv* sono nuovi nodi introdotti che gestiscono tutti i messaggi entranti e uscenti da un sottografo (Figura 4).

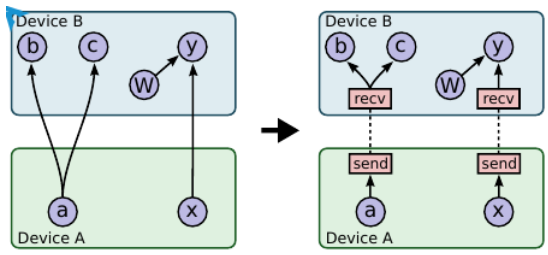

Figura 4: *Esempio di Grafo di Computazione, corrispondente alla Figura 2*

Un pò come il corriere che si occupa di raccogliere tutte le spedizioni di una giornata, le porta in un'altro stato e le distribuisce ai relativi destinatari. Questo approccio, permette di trasmettere dati una sola volta; questo permette di allocare una sola volta la memoria che spetta a un tensor, invece che più volte, e che nonostante il posizionamento dei nodi, l'utente possa esplicitamente assegnare l'esecuzione di un nodo, a un determinato device (Figura 3).

2.2 Esecuzione Distribuita

L'approccio adottato per l'esecuzione distribuita di un grafo è quasi identico a quello preservato per i device. Qui la comunicazione tra grafi avviene tramite protocollo `TCP` o `RDMA`. Un'altra differenza è che negli ambienti distribuiti urge riflettere a una gestione degli errori di comunicazione. Gli errori, possono avvenire tra i nodi *send* e *recv*, quindi è necessario verificare la salute degli scambi fra questi. Quando un fallimento viene rilevato, l'esecuzione dell'intero grafo viene riavviata, ma questo non implica la perdita dei dati: si ricordi che i nodi variabile, immagazzinano i tensors. Periodicamente, vengono creati dei check-point poiché i nodi variabili, sono collegati a speciali *save node*; quindi è possibile ripristinare stati di esecuzione precedenti.

3 Calcolo Distribuito del Gradiente

Essendo TensorFlow dedicato all'apprendimento automatico, il basilare modello di programmazione comprende alcuni strumenti specializzati. In ambito Machine Learning, molti algoritmi di apprendimento, calcolano il gradiente di una funzione costo; TensorFlow supporta il calcolo automatico di questa funzione. A livello di grafo, si supponga che un tensor C dipenda (tramite un complesso sottografo di operazioni) da un insieme di tensori $\{X_k\}$, allora è disponibile una funziona che ritorna i tensors $\{dC/dX_k\}$. Considerato che trattiamo Machine Learning e l'esecuzione di un grafo su ambiente distribuito, andremo ad approfondire tale estensione.

Supponiamo che la situazione sia quella di dover computare il gradiente di un tensor C, rispetto al gradiente di un tensore I. Ecco quanto accade:

1. viene trovato il cammino fra I e C all'interno del grafo;

2. si va a ritroso da C a I e per ogni nodo operazione viene aggiunto un nuovo nodo al grafo di computazione, che computa la funzione gradiente, per la relativa operazione sul cammino di ritorno;

    a) la funzione gradiente può prendere in input non solo gradienti parziali computati a ritroso ma anche, opzionalmente, gli input e gli output delle relative operazioni.

3. ma un'operazione O può avere un insieme E di archi uscenti (risultati di output) e C può dipendere da un sottoinsieme di E: allora la funzione gradiente di O prenderà in input $dC/de_k = 0$ per ogni arco e_k da quale C risulta *indipendente*;

Il procedimento appena descritto si può intuitivamente osservare nella Figura 5, che riguarda la seguente istruzione:

```
[db,dW,dx] = tf.gradients(C, [b,W,x])
```

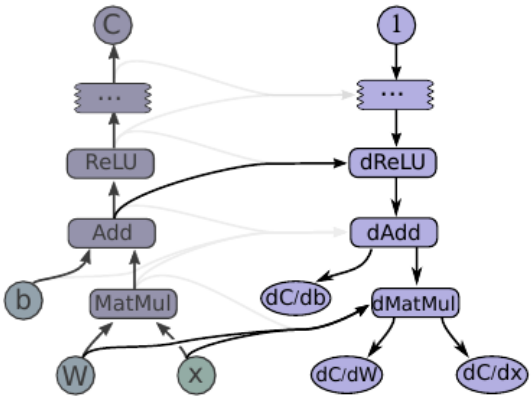

Figura 5: *Esempio di computazione di funzione gradiente*

3.1 Controllo del flusso in ambiente distribuito

Il calcolo del gradiente viene considerato come estensione di TensorFlow. Altre estensioni sono:

- esecuzione parziale del grafo, dove l'utente sceglie quali nodi eseguire;
- vincoli di collocamento di un nodo, tramite i quali l'utente costringe un nodo a essere eseguito su un particolare device;
- controllo del flusso di esecuzione, che è un insieme di primitive che permettono di manipolare l'esecuzione del grafo (costrutti `if`, cicli `while`, `Switch`, `Merge`, etc)
- nodi *feed*, che fungono da fonte di tensors per altri nodi
- `queue` che permettono l'esecuzione asincrona del grafo
- `containers` per gestire gli stati all'interno del grafo, e altre.

Per approfondimenti si può vedere [1].

In questo paragrafo, ci soffermeremo con più attenzione, al controllo del flusso in quanto **cruciale in ambiente distribuito**. Poiché, viene usato un meccanismo importante di coordinazione distribuita. Questo nasce dalla necessità di gestire l'esecuzione *loop* che riguardano nodi assegnati a differenti device o macchine, il loro stato va infatti in qualche modo mantenuto. La soluzione che propone TensorFlow è quella della *riscrittura del grafo*: durante il

partizionamento dei nodi (Sottosezione 2.1), un nodo di controllo viene aggiunto per ogni sottografo. I nodi di controllo implementano una *macchina a stati* che orchestra lo start e la fine di ogni iterazione, oltre alla terminazione del loop stesso. Per ogni iterazione, il device che possiede il predicato della terminazione del loop, spedisce un messaggio di controllo verso tutti gli altri device coinvolti. Questi accorgimenti sono necessari anche per computare il gradiente precedentemente visto, in quanto è necessario sapere, per esempio, quale ramo di un *if-then-else* è stato imboccato per arrivare a C; oppure quante iterazioni sono state fatte da un particolare loop o quali erano i valori intermedi.

4 Forme di Parallelismo

5 Ottimizzazioni e Strumenti

Per completezza, verranno citate, ma non approfondite, alcune caratteristiche di TensorFlow che non risultano di primaria importanza in questo lavoro.

Il sistema presenta alcuni accorgimenti per incrementare le performance e la gestione delle risorse. TensorFlow è in grado di individuare pezzi di grafo ridondanti che fanno "la stessa cosa" ed eliminarli, preservando la semantica del programma; può effettuare un controllo diretto dell'utilizzo della memoria e l'attivazione dei nodi di tipo *recv* (Sottosezione 2.1), secondo politiche `as-soon-as-possible` oppure `as-late-as-possible`; implementa kernel non bloccanti e introduce librerie ottimizzate per lo sviluppo di kernel.

Viene offerto un insieme di strumenti chiamato `TensorFlowBoard`. Esso permette agli utenti di esplorare graficamente la struttura del grafo di computazione e l'analisi del suo comportamento; aiuta la visualizzazione degli stati interni e traccia le performance.

6 Apprendimento distribuito: test empirici

7 Appunti Bart

Dopo svariate prove con la libreria, ci siamo posti come obiettivo, di parallelizzare la fase di training in modo asincrono, mettendo a disposizione le risorse di tre macchine, due impostate come worker e una per gestire il tutto, settata come parameter server. Facendo uso di una rete sigmoidea con una bassa frequenza di apprendimento, abbiamo misurato le diverse performance su MNIST. L'obiettivo non è stato quello di ottenere un'alta accuratezza dopo la fase di training, ma apprendere le funzionalità della libreria in ambiente distribuito. Il primo test valutativo, è stato quello di addestrare la rete per 20 epoche, usando un solo worker, con un risultato del 29% di accuratezza. L'accuratezza è incrementata quando abbiamo aggiunto alla rete altre due macchine settate come worker.

7.1 Addestramento distribuito

Esistono differenti modi per addestrare una rete in ambienti distribuiti. L'approccio più semplice è quello di condividere tutto il modello dei parametri con tutti i worker mentre i dati vengono parallelizzati e il gradiente viene aggiornato. In un modello **sincrono**, alcuni gruppi vengono processati nello stesso lasso di tempo. Una volta che tutti i worker all'intero della rete hanno processato e terminato la fase di training, viene effettuata la media del tempo impiegato e vengono applicati gli aggiornamenti dei parametri, una sola volta. Invece, in un modello **asincrono**, ogni worker aggiornerà il modello dei parametri una volta che ha terminato la propria fase, senza aspettare la fine dell'esecuzione degli altri. Durante la fase di prova, abbiamo usato un modello sincrono, poiché la regolazione dei parametri per il modello asincrono risultava molto complessa.

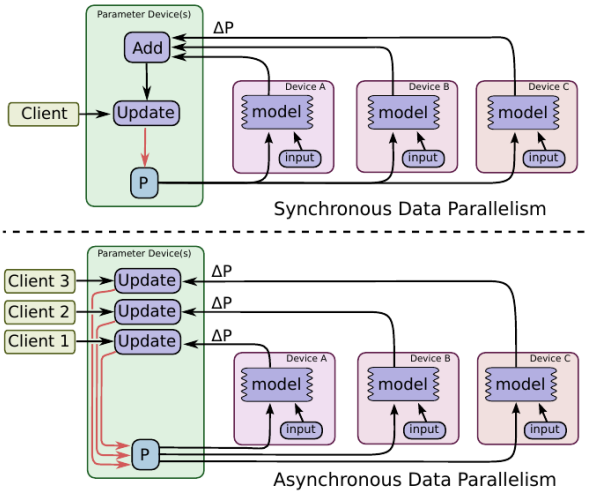

Figura 6: *Fase di training parallelizzata: sincrono e asincrono.*

7.2 Implementazione

E' di fondamentale importanza, che ogni macchina all'interno del cluster conosca se stessa e le altre che ne fanno parte. Lo script è stato implementato in modo tale che, la fase di training non parta finché tutte le macchine all'interno del cluster siano online. Le specifiche del cluster sono le stesse per ogni macchina: costituite da parameter server e worker. Mentre il parameter server mantiene solamente i parametri condivisi, i worker effettuano i calcoli per il grafo. ps e worker sono chiamati **jobs**, i quali, fondamentalmente, sono contenitori per uno o più **task**. Il task è il processo che il worker elaborerà durante l'esecuzione, volendo, è possibile aggiungere più task sulla stessa macchina. Questo potrebbe aver senso, in una macchina con più acceleratori di grafica (GPU); in questa ottica, tutti i task avranno un grafo completo del modello. Se volessimo parallelizzare il modello anziché i dati, dovremmo cambiare il comportamento di ogni task, più nello specifico le operazioni che deve eseguire ogni task. Nella parte di codice 8, andiamo a inizializzare le macchine per l'esecuzione. Per assicurarsi, che venga eseguito il task corretto, al primo avvio dello script (settato come worker), *pc-02* sarà di default il worker con il $task\_index = 0$. Nella parte di codice 8 è mostrata la configurazione dei parametri e il caricamento dei dati MNIST. Se lo script viene avviato come parameters server, viene eseguita la funzione *server.join()*, in modo che si unisca al cluster. Quello che segue sono le configurazioni delle variabili, e la definizione della computazione che ogni worker deve eseguire. Dal momento in cui, si è scelto di computare l'intero modello su tutti i dispositivi, abbiamo aggiunto allo scope tutti i worker. Quello che seguirà è l'implementazione del modello; Attraverso l'inclusione di una variabile *global_step* (incrementata di uno ogni volta che il modello si aggiorna) riusciamo a migliorare la tracciabilità, durante la fase di training. Inoltre, a causa della gestione interna dei threads da parte di TensorFlow, questo non rende lo script deterministico, ovvero, non ci si può aspettare gli stessi risultati dopo ogni esecuzione. Detto questo, abbiamo bisogno di ottenere la sessione al fine di eseguire i nostri cicli di training in un sistema distribuito; quindi risulta obbligatorio settare una macchina che prenda la posizione di **chief** (capo). Il *chief* è un worker (nel nostro caso task 0) il quale gestisce e monitora il resto del cluster; la sessione quindi viene gestita dal *chief*, attraverso il metodo Supervisor della classe train, 10. Abbiamo ottenuto la sessione con il comando seguente:

```
with sv.prepare_or_wait_for_session(server.target) as sess:
```

Riferimenti bibliografici

[1] Google Research, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, Preliminary White Paper, 2015.

```
parameter_servers = ["pc-01:2222"]
workers = [ "pc-02:2222",
            "pc-03:2222",
            "pc-04:2222"]
cluster = tf.train.ClusterSpec({"ps":parameter_servers,
                                "worker":workers})
```

Figura 7: *Specifiche per il cluster.*

```
tf.app.flags.DEFINE_string("job_name", "",
                           "Either 'ps' or 'worker'")
tf.app.flags.DEFINE_integer("task_index", 0,
                           "Index of task within the job")
FLAGS = tf.app.flags.FLAGS

# start a server for a specific task
server = tf.train.Server(cluster,
                         job_name=FLAGS.job_name,
                         task_index=FLAGS.task_index)
```

Figura 8: *Specifiche e impostazione dei parametri per il cluster.*

```
  with tf.device(tf.train.replica_device_setter(
    worker_device="/job:worker/task:%d" % FLAGS.task_index,
    cluster=cluster)):
```

Figura 9: *Replica del modello e aggiunta allo scope dei worker.*

```
  sv = tf.train.Supervisor(is_chief=(FLAGS.task_index == 0),
                           global_step=global_step,
                           init_op=init_op)
```

Figura 10: *Instanza dell'oggetto supervisor.*