UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Edge AI Cluster

Docente:
Prof. Lorenzo Carnevale

Studenti:
Matteo Piccadaci, 514430
Danny De Novi, 517611

ANNO ACCADEMICO 2021/2022

Indice

1 Abstract

Il crescente utilizzo di dispositivi Internet Of Things (IoT) ha accentuato il bisogno di dover spostare l'analisi dei dati direttamente sulla macchina, questo perché spostare ingenti quantità di dati risulta piuttosto oneroso sia in termini di velocità di trasferimento che di computazione sulle macchine server. L'edge computing può di fatti essere utile in tal senso, poiché non necessita di macchine estremamente performanti per effettuare calcoli, in tempi ristretti, che riguardino applicazioni del mondo del *Machine Learning*, come il pattern recognition, ma sono più sufficienti dei dispositivi dalle prestazioni modeste. L'obiettivo dell'esperimento è quello di poter effettuare un deployment *plug and play* di un'architettura composta da *System on chip*, nello specifico Raspberry Pi 4B, sui quali vengono eseguiti algoritmi di Pattern Recognition tramite *Tensorflow* e un server per la computazione di codice Python via browser per semplicità di inserimento e consultazione, *Jupyter*. Per il deployment è stato utilizzato *Kubernetes* come orchestratore di container, in modo tale da poter costituire un *cluster* che, automaticamente, fosse in grado di poter gestire equamente il carico su i vari dispositivi e permettere la miglior fruizione possibile all'utente. Si è potuto notare come l'inferenza venga eseugita correttamente e che la distribuzione del carico venga effettuata trasparentemente. È possibile trovare l'intero progetto sul dedicato repository GitHub.[1].

2 Docker, container e immagini

Prima di poter parlare di orchestrazione di container è bene accennare ciò che essi sono e la loro gestione tramite Docker [1].

2.1 Docker

Docker è una suite di gestione di container Linux, esso fornisce una serie di tool per poter creare, eseguire e pubblicare immagini. Le immagini Docker non sono altro che un insieme di layer che corrispondono ai singoli comandi scritti su un *Dockerfile*. Montando un'immagine si darà vita ad un container che eseguirà in maniera isolata dal sistema operativo i task assegnatogli.

2.2 Container

Un container non è altro che l'istanza dell'esecuzione isolata. Esso condivide con il sistema operativo di base il *kernel*, difatti nelle edizioni Windows e MacOS vi è una macchina virtuale Linux che fa da tramite tra il kernel della macchina host e il container, in modo tale da non perdere le peculiarità di Docker come il binding delle porte e i volumi. È dunque preferibile utilizzare un container rispetto ad una macchina virtuale poiché vengono caricati ed utilizzati solo i componenti software strettamente necessari all'esecuzione del programma, garantendo dunque una velocità di setup e di utilizzo decisamente più elevata [2]. Rispetto all'utilizzo del software direttamente sul sistema operativo, invece, i vantaggi riguardano soprattutto la sfera della sicurezza, in quanto il fatto che ogni container sia isolato dal

[1] GitHub - dannydenovi/ProgettoSistemiDiVirtualizzazione

sistema operativo e dagli altri container fa sì che nel caso vi siano compromissioni esterne, esse rimangano confinate all'interno container stesso senza mettere in pericolo la macchina centrale e dando la possibilità di migrare il servizio su un altro container, permettendo all'utente di continuare trasparentemente ed in sicurezza la sua esecuzione. Una questione non meno importante è lo spazio occupato, in quanto a differenza di un software installato nativamente sul sistema operativo, per eliminare tutti i file necessari al fine dell'utilizzo basterà dare un solo comando per eliminare il container. Le moderne infrastrutture dipendono dal corretto utilizzo dei container: si parla infatti di architetture a microservizi. Ogni tipo di servizio del quale usufruiamo giornalmente nel passato era un processo eseguito in una macchina server, con tutti i rischi legati alla sicurezza e alla fault tolerance. Come detto, la possibilità di migrare rapidamente un container garantisce la scalabilità dell'intera rosa di servizi che un'azienda mette a disposizione, permettendone la fruizione ad un numero enorme di utenti in contemporanea.

2.3 Dockerfile

Per poter creare un'immagine Docker è necessario dare le istruzioni preliminari per l'esecuzione del container. Questo si traduce nella scrittura di un Dockerfile, un file che non è altro che una serie di istruzioni e descrizioni dell'immagine. Nel nostro caso specifico ci serve un'immagine che sia basata su Ubuntu 20.04, contenga tutte le dipendenze necessarie per poter eseguire del codice Python 3, la possibilità di interfacciarsi con Jupyter e la presenza del framework Tensorflow adatto ad un'architettura ARM64.

Vi è anche la possibilità di poter eseguire su Docker delle immagini già create dalla community in modo tale da non dover scrivere di sana pianta un Dockerfile.

Di seguito il Dockerfile utilizzato per il progetto e il repository Docker [2].

[2]DockerHub - tensorflow jupyter rpi

```
FROM ubuntu:20.04

# Packages update:
RUN apt-get update && apt-get -y upgrade

# Environment variable to accept the default answer:
ENV DEBIAN_FRONTEND=noninteractive

# Install dependencies:
RUN apt-get install -y python3 python3-pip libatlas-base-dev wget
gfortran libhdf5-dev
libc-ares-dev libeigen3-dev libopenblas-dev
libblas-dev liblapack-dev
RUN pip3 install --upgrade pip
RUN pip3 install --upgrade setuptools
RUN pip3 install pybind11
RUN pip3 install Cython==0.29.21
RUN pip3 install h5py
RUN pip install gdown
RUN pip install -U matplotlib

# Change working directory:
WORKDIR /tf
RUN gdown https://drive.google.com/u/0/uc
id=1rwUdfuk032GbFydrTmyaAfI9KixTQw3C

# Install TensorFlow:
RUN pip3 install -v tensorflow-2.3.0-cp38-cp38-linux_aarch64.whl

# Install Jupyter:
RUN cd /
WORKDIR /books
RUN pip3 install jupyter
RUN pip3 install pandas matplotlib

# Change init permissions
RUN chmod +x /usr/local/bin/jupyter-notebook
ENTRYPOINT ["/usr/local/bin/jupyter-notebook", "."]

# Exposing Jupyter port:
EXPOSE 8888

# Starting container command:
CMD ["jupyter", "notebook", "--port=8888",
"--no-browser", "--ip=0.0.0.0",
"--NotebookApp.token=''", "--NotebookApp.password=''",
"--allow-root"]
```

3 Kubernetes

La principale difficoltà di un'architettura a microservizi è quella di coordinare l'estremo numero di container: sarebbe inimmaginabile dover tener traccia degli avvenimenti per ogni singolo container manualmente. Per questo motivo sono nati gli orchestratori di container, come Kubernetes [3].

3.1 A cosa serve

Tutte le peculiarità come scalabilità, disponibilità e fault tolerance sono l'obiettivo di software come Kubernetes. In particolare, essa istanzia in ogni nodo di una rete un pod, in maniera tale da non dover comunicare direttamente con il container. Questo rappresenta un passo in avanti per quanto riguarda la sicurezza, ulteriormente enfatizzata dal fatto che gli indirizzi IP dei singoli pod sono mascherati dai *Services*, che rappresentano una vera e propia porta dal quale bisogna passare per poter usufruire del servizio. I volumi sono invece le memorie dove ogni pod conserva i suoi dati: esistono volumi persistenti o effimeri, in quanto di base dopo la distruzione del pod anche i suoi dati verrebbero eliminati. I delpoyment permettono invece funzioni di load balancing, replica e rollback a versioni precedenti. L'archiettura di kubernetes è di tipo Master-Worker, dove è il Master a coordinare eventuali ingressi o uscite di worker dalla rete.

3.2 Deployment

Un deployment Kubernetes è un oggetto risorsa di Kubernetes che fornisce aggiornamenti dichiarativi alle applicazioni. Un deployment consente di descrivere il ciclo di vita dell'applicazione, specificando ad esempio le immagini da utilizzare, il numero di pod necessari e le modalità di aggiornamento relative. Un oggetto Kubernetes serve per indicare al sistema Kubernetes in che modo il carico di lavoro del cluster deve funzionare. Dopo che è stato creato l'oggetto, il cluster agisce per garantirne l'esistenza e per mantenere lo stato desiderato del cluster Kubernetes. L'aggiornamento manuale delle applicazioni containerizzate è esigente in termini di tempo e risorse di lavoro. Per aggiornare la versione di un servizio è necessario avviare la nuova versione del pod, arrestare quella precedente, attendere e verificare che la versione più recente si avvii correttamente; in caso di errore, è a volte necessario reinstallare la versione obsoleta. L'esecuzione manuale di questi passaggi è soggetta a errori e la compilazione corretta dello script può essere laboriosa; entrambi questi fattori possono ostacolare il processo di release, rallentandolo. Un deployment Kubernetes rende il processo automatico e ripetibile. Interamente gestiti dal backend Kubernetes, i deployment eseguono il processo di aggiornamento sul lato server, senza interagire con il client. Un deployment garantisce l'esecuzione e la disponibilità del numero previsto di pod, in ogni momento. Il processo di aggiornamento è completamente registrato e le diverse versioni contengono opzioni per sospendere, continuare o tornare a quelle precedenti.

Di seguito il deployment effettuato:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tensorflow-jupyter-rpi
spec:
  replicas: 2
  selector:
    matchLabels:
      app: tensorflow-jupyter-rpi
  template:
    metadata:
      labels:
        app: tensorflow-jupyter-rpi
    spec:
      containers:
        - name: tensorflow-jupyter-rpi
          image: dannydenovi/tensorflow_jupyter_rpi:1.3
          resources:
            requests:
              memory: "2Gi"
            limits:
              memory: "2Gi"
          ports:
          - containerPort: 8888
          imagePullPolicy: IfNotPresent
          volumeMounts:
          - mountPath: "/books"
            name: nfs
      volumes:
      - name: nfs
        persistentVolumeClaim:
          claimName: nfs
```

3.3 Services utilizzati

La configurazione utilizzata ha imposto un numero di repliche pari a 2, riservando 2 Gb di RAM ad ogni pod, mostrando la porta 8888 per potervici interfacciare ed, infine, montando un volume persistente per ogni nodo. Non è stato in alcun modo possibile aumentare la scalabilità a più pod per macchina in quanto una singola istanza di Tensorflow e Jupyter occupano più della metà delle risorse (nel caso di Raspberry Pi 4B 4GB). La porta d'accesso al servizio Jupyter viene esposta tramite quello che viene definito come *Ingress*. Un Ingress non è altro che un *entrypoint* alla rete del cluster che effettua un binding sulla porta 80 e automaticamente gestisce le politiche di scheduling della risposta. L'utente finale non sarà in grado di sapere su quale macchina risiede la sua istanza corrente, ed ogni volta che viene effettuato

l'accesso dovrà sottostare alle politiche di gestione dell'ingress, dunque potrebbe potenzialmente ricadere su una macchina diversa dalla precedente.

Di seguito il file di configurazione del service istanziato:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: tensorflow-jupyter-rpi
spec:
  ports:
  - port: 8888
    targetPort: 8888
    name: tcp
  selector:
    app: tensorflow-jupyter-rpi
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tensorflow-jupyter-rpi
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: tensorflow-jupyter-rpi
            port:
              number: 8888
```

3.4 Implementazione con training e prediction separati

È possibile separare quelle che sono le macchine atte al training da quelle atte all'attività di predizione istanziando due diversi deployment che differiscono sostanzialmente per nome e per il service a loro legato. In questo tipo di approccio non viene utilizzato un *Ingress* come precedente riportato, ma è fondamentale distinguere con diverse porte quelli che sono i due deployment in esecuzione, e ciò avviene tramite quelle che sono chiamate *NodePort*, nonché dei punti di accesso al cluster mappati su degli specifici deployment. È stato aggiunto un parametro chiamato *nodeSelector* che non fa altro che istanziare il deployment sulle macchine con una specifica label, nel nostro caso detto *role* che può essere di tipo training o prediction.

Di seguito i due deployment completi:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: training-instance
spec:
  replicas: 1
  selector:
    matchLabels:
      app: training-instance
  template:
    metadata:
      labels:
        app: training-instance
    spec:
      #nodeName: "unime-w1"
      nodeSelector:
        role: training
      containers:
        - name: training-instance
          image: dannydenovi/tensorflow_jupyter_rpi:1.3
          resources:
            requests:
              memory: "2Gi"
            limits:
              memory: "2Gi"
          ports:
          - containerPort: 8888
          imagePullPolicy: IfNotPresent
          volumeMounts:
          - mountPath: "/books"
            name: nfs
      volumes:
      - name: nfs
        persistentVolumeClaim:
          claimName: nfs
---
apiVersion: v1
kind: Service
metadata:
  name: training-instance
spec:
  type: NodePort
  selector:
    app: training-instance
  ports:
    - port: 8888
      targetPort: 8888
      nodePort: 30000
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prediction-instance
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prediction-instance
  template:
    metadata:
      labels:
        app: prediction-instance
    spec:
      #nodeName: "unime-master"
      nodeSelector:
        role: prediction
      containers:
        - name: prediction-instance
          image: dannydenovi/tensorflow_jupyter_rpi:1.3
          resources:
            requests:
              memory: "2Gi"
            limits:
              memory: "2Gi"
          ports:
          - containerPort: 8888
          imagePullPolicy: IfNotPresent
          volumeMounts:
          - mountPath: "/books"
            name: nfs
      volumes:
      - name: nfs
        persistentVolumeClaim:
          claimName: nfs
---
apiVersion: v1
kind: Service
metadata:
  name: prediction-instance
spec:
  type: NodePort
  selector:
    app: prediction-instance
  ports:
    - port: 8888
      targetPort: 8888
      nodePort: 30001
```

3.5 Volumi

La doppia possibilità messa a disposizione da Kubernetes, ovvero la costituzione di un volume persistente o meno, si riferisce alla memoria del singolo nodo e strettamente dei suoi pod: per realizzare il nostro scopo, bisogna utilizzare un altro approccio. Esso si è tradotto nel salvare il risultato dell'eleborazione di ogni singolo pod su un server NFS (*Network File System*)[4]. Kubernetes permette infatti l'interfacciamento con gli NFS, nella stessa identica maniera con la quale si impone la scelta fra volume persistente o non, permette anche l'interfacciamento con i filesystem più comuni di servizi cloud, come ad esempio *AWSElasticBlockStore* e *AzureDisk*.

Di seguito il PersistentVolume e il relativo PersistentVolumeClaim istanziato:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  storageClassName: nfs
  nfs:
    path: "/"
    server: dannydenovi.ddns.net

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs
spec:
  storageClassName: nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

4 Edge computing e Kubernetes

Nel nostro studio abbiamo posto enfasi sul fatto che sia necessaria l'elaborazione in locale del maggior quantitativo di dati, in maniera tale da mettere in circolazione quelli strettamente utili, nel nostro caso i pesi del training effettuato, in modo tale da non doverlo ripetere su ogni macchina del cluster. Questo approccio è detto Edge computing, che prende questo nome proprio poiché la filosofia è quella di spostare il calcolo il più vicino possibile alla fonte dei dati. L'utilizzo di Kubernetes in questo ambito è stato quello di avere a disposizione un mezzo per poter realizzare il calcolo non appena un nuovo dispositivo che necessita di elaborazione entra nella rete [5]. L'edge ovviamente dipende dall'utilizzo di piattaforme cloud, nel nostro caso è necessario uno storage nel quale depositare i risultati che ogni nodo ottiene e, successivamente, potervi attingere da qualunque altra macchina nel mondo, ed è proprio per questo scopo che abbiamo introdotto il supporto allo storage di tipo NFS.

4.1 Tensorflow e AI

Il Pattern recognition è solamente una particolare applicazione dell' Intelligenza artificiale che, come ormai noto, ha come obiettivo l'imitazione della mente umana tramite calcolatori. Tensorflow [6] è una libreria open source che mette a dispozione numerosissimi moduli utili a realizzare algoritmi di Machine Learning e di AI in generale. L'aspetto importante di Tensorflow sono le sue API, che rendono facile l'interfacciamento con molti linguaggi di programmazione, fra cui Python. Nel nostro caso si è effettuato un training sul dataset MNIST che consiste in una raccolta di 60'000 campioni di cifre scritte a mano.

4.2 Jupyter

Jupyter[7] mette a disposzione un'interfaccia web per realizzare e consultare rapidamente progetti realizzati in Python. Essa ci ha permesso di rendere disponibile l'algoritmo di AI senza dover accedere in alcuna maniera al volume del POD, ancora una volta a garantire maggiore sicurezza.

5 Implementazione dei Notebook

Si è deciso di suddividere in due Notebook diversi quelli che sono il training dell'AI e quella che è la vera e propria predizione, in modo tale da far evincere maggiormente quello che è il concetto che si vuole dimostrare: effettuare il calcolo preliminare su una macchina ed andare a distribuire la computazione che risulta certamente di dimensione inferiore e di necessità di risorse computazionali esigue, in quanto nel nostro caso corrisponde semplicemente ad un file contenente quelli che sono i pesi ottenuti dalla computazione che successivamente verranno caricati nel caso in cui si volesse effettuare una predizione, senza la necessità di salvare e trasportare l'intero modello o addirittura dover rieffettuare il training che potenzialmente avrà i medesimi risultati a causa della medesima architettura.

Di seguito il notebook che effettua il training e successivamente quello che si occupa della predizione:

```
#       TRAINING        #
import tensorflow as tf
import time

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test = tf.keras.utils.normalize(x_test, axis=1)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax))

start = time.time()
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=3)

with open("weights.txt", "w", encoding = 'utf-8') as f:
    f.write(str(model.get_weights()))

model.save_weights('./model_weights', save_format='tf')
with open("time.txt", "w", encoding = "utf-8") as f:
    text = "Training: "+ str(time.time() - start)
    f.write(text)
```

```
#       PREDICTION        #
import tensorflow as tf
import time
import numpy as np
import matplotlib.pyplot as plt


mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test = tf.keras.utils.normalize(x_test, axis=1)
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax))

start = time.time()
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.train_on_batch(x_train[:1], y_train[:1])
model.load_weights("./model_weights")

with open("time.txt", "a", encoding = "utf-8") as f:
    text = "\nNo training: "+ str(time.time() - start)
    f.write(text)

predictions = model.predict([x_test])
print(np.argmax(predictions[0]))
plt.imshow(x_test[0])
plt.show()
```

6 Conclusioni

L'esperimento ha dimostrato come sia possibile realizzare SOC plug and play per entrare a far parte di un cluster di edge computing e come sia effettivamente conveniente andare a spostare quello che è il risultato della computazione. Un singolo training ha impiegato circa 54 secondi nel caso del dataset completo con 60'000 campioni e meno di 6 secondi importando quelli che sono i pesi ottenuti da esso. Immaginando alcuni ambiti di utilizzo, è facile pensare ad applicazioni riguardo l'ambito della mobilità autonoma o nel mondo delle telecomunicazioni con le cosidette NFV (virtualizzazioni delle funzioni di rete) [8]. Il vincolo di questa archiettura risiede nel dispositivo utilizzato, in quanto la scheda Raspberry non è molto adatta a calcoli intensivi, difatti si è ovviato all'ingente tempo di training con la condivisione sull'NFS dei pesi ottenuti dallo stesso. Il passo successivo è infatti quello di replicare la stessa archiettura su macchine più performanti come la Jetson Nano, dove sarà possbile avere più pod per dispositivo e, di conseguenza, effettuare calcoli ben più complessi in un tempo minore ed eventualmente di un dataset molto più elaborato rispetto ad un MNIST [9].

7 Note e Bibliografia

Riferimenti bibliografici

[1] *Docker Overview* https://docs.docker.com/get-started/overview/

[2] *Performance Analysis of Linux Container and Hypervisor for Application Deployment on Clouds* https://ieeexplore.ieee.org/document/8769146

[3] *Kubernetes Overview* https://kubernetes.io/docs/concepts/overview/

[4] *Network File System (NFS)* https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/ch-nfs

[5] *Distributed analytics in fog computing platforms using tensorflow and kubernetes* https://ieeexplore.ieee.org/document/8094194

[6] *Introduction to Tensorflow* https://www.tensorflow.org/learn

[7] *Introduction to Jupyter* https://realpython.com/jupyter-notebook-introduction/

[8] *Cosa si intende per NFV? - Red Hat* https://www.redhat.com/it/topics/virtualization/what-is-nfv

[9] *Design and Implementation of Kubernetes enabled Federated Learning Platform* https://ieeexplore.ieee.org/document/9620986