UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

PROGRAMMAZIONE DI SISTEMI EMBEDDED

MOBILE AI

Sviluppo di un'app per il riconoscimento di frutta e verdura tramite libreria Tensor Flow

Gruppo:
BERLESE LORENZO
PAOLETTO GEREMIA
SCALCO RICCARDO

Anno Accademico 2023/2024

INDICE

INTRODUZIONE

Nel contesto della programmazione di sistemi embedded, l'integrazione di tecnologie di intelligenza artificiale è divenuta cruciale per migliorare l'efficienza e le capacità dei dispositivi mobili. Questa ricerca bibliografica si concentra su tre temi principali: Neural Networks API (NNAPI), Tensor Flow (e Tensor Flow Lite) e PyTorch, evidenziando come ciascuna di queste tecnologie contribuisca allo sviluppo di applicazioni AI su piattaforme mobili.

Le Neural Networks API sono una componente essenziale per l'esecuzione di operazioni di inferenza su dispositivi Android poichè consentendo di sfruttare al massimo l'hardware disponibile, inclusi CPU e GPU. NNAPI permette di definire, compilare ed eseguire modelli di machine learning in modo efficiente, supportando l'implementazione di reti neurali direttamente sul dispositivo senza necessità di connessione a internet. Questo approccio non solo riduce la latenza e migliora la velocità di elaborazione, ma garantisce anche una maggiore privacy dei dati.

TensorFlow Lite rappresenta la versione di TensorFlow per dispositivi mobili e embedded, progettato per eseguire modelli di machine learning leggeri e ottimizzati. Con caratteristiche come il formato FlatBuffers per la rappresentazione dei modelli e il supporto per Android Studio ML Model Binding, TensorFlow Lite facilita lo sviluppo, la conversione e l'implementazione di modelli ML su piattaforme con risorse limitate. Questa tecnologia consente di eseguire inferenze rapide ed efficienti, sfruttando al meglio l'hardware disponibile e riducendo l'impatto sulle risorse del dispositivo.

PyTorch, una libreria sviluppata da Meta, offre un'ampia gamma di strumenti per il training e l'ottimizzazione di modelli di machine learning. Con PyTorch Mobile, è possibile eseguire modelli su dispositivi mobili, sfruttando le caratteristiche di alto livello di PyTorch come la flessibilità e la facilità di utilizzo. La compatibilità con processori CUDA e la capacità di ottimizzazione del calcolo tramite GPU rendono PyTorch una scelta potente per lo sviluppo di applicazioni AI avanzate su piattaforme mobili.

Con questa ricerca esploriamo in dettaglio le caratteristiche, le potenzialità e gli utilizzi pratici di NNAPI, TensorFlow Lite e PyTorch, fornendo una panoramica delle tecnologie attualmente disponibili per la programmazione di sistemi embedded con intelligenza artificiale.

CAPITOLO 1 - RETI NEURALI

1.1 Definizione di reti neurali (neural networks)

"Una rete neurale è un programma di machine learning o un modello che prende decisioni in un modo simile al cervello umano, usando un processo che imita il modo di lavorare insieme dei neuroni per identificare un fenomeni, valutare con un peso (probabilità) le possibili opzioni e arrivare ad una conclusione". Questa è la definizione di rete neurale data da IBM, una delle più grandi aziende tecnologiche i cui obiettivi attuali riguardano anche l'intelligenza artificiale e il machine learning. [1]

Ogni rete neurale consiste in un insieme di nodi. Ogni nodo può connettersi con gli altri e inoltre ha un peso e una soglia associati.

Le reti neurali fanno affidamento all'allenamento per migliorare la loro precisione e la loro affidabilità. Una volta che sono perfezionate, diventano strumenti molto importanti per l'intelligenza artificiale e più in generale per l'informatica. Il tempo di elaborazione di immagini/voce scende drasticamente utilizzando le reti neurali allenate rispetto all'elaborazione umana. Il più grande esempio di rete neurale allenata è l'algoritmo di ricerca di Google.

1.2 Breve storia delle reti neurali

L'idea di base di una rete neurale non è affatto recente. I primi studi affidabili si possono attribuire a Warren McCulloch e Walter Pitts nel 1943. Il loro articolo riguardava il funzionamento dei neuroni nel cervello umano. Per descriverlo hanno modellato una semplice rete neurale con circuiti elettrici. [1][2]

Nel 1949, Donald Hebb dimostrò il fatto che i percorsi neurali si rafforzano ogni volta che vengono utilizzati.

Con l'arrivo di computer più avanzati nel 1959, Bernard Widrow e Marcian Hoff svilupparono due modelli:

- ADALINE: è un modello che leggendo dei bit in streaming da una linea telefonica, può prevedere il bit successivo
- MADALINE: è un sistema basato su reti neurali che tramite un filtro elimina gli echi

sulle linee telefoniche.

Lo sviluppo ha continuato piuttosto lento per diversi anni, anche perché è stato limitato dall'hardware: infatti a causa delle limitazioni dei processori, le reti neurali impiegano molto tempo ad apprendere.

1.3 Operazioni di inferenza

Con il termine "operazioni di inferenza" si intende il processo con cui un modello di machine learning (che può essere una rete neurale) prende come input dei dati e produce un output basato su quanto ha imparato durante la fase di addestramento. Non è detto che l'input sia già stato visto durante l'addestramento, quindi l'output può basarsi su predizioni.

Alcuni esempi di inferenza sono:

- classificazione di immagini
- previsione del comportamento degli utenti
- query di ricerca

Alcuni vantaggi dell'inferenza sono:

- Disponibilità: non è richiesta nessuna connessione internet, quindi l'applicazione può funzionare anche al di fuori della copertura.
- Latenza: è un vantaggio direttamente collegato alla disponibilità. Non è necessario inviare una richiesta tramite una connessione di rete e attendere una risposta.
- Velocità: con lo sviluppo di nuovo hardware e dispositivi più potenti l'elaborazione di reti neurali è molto più rapida di semplici calcoli con CPU generiche.
- Privacy: i dati non escono mai dal dispositivo Android.

Bisogna però considerare anche i possibili fattori negativi:

- Utilizzo della batteria: la valutazione di reti neurali comporta calcoli importanti che possono portare ad un elevato consumo della batteria. Bisogna tenere in considerazione questo fattore soprattutto se i calcoli sono a lunga esecuzione.
- Dimensioni del modello: i modelli possono occupare molto spazio. Bisogna cercare di non utilizzare modelli troppo grandi o in alternativa svolgere alcuni calcoli nel cloud (perdendo alcuni vantaggi).

1.4 API Neural Networks

L'API Android Neural Networks (NNAPI) è un'API Android, scritta in C, che offre

un'interfaccia per le operazioni di machine learning, tra cui appunto le reti neurali. Si può utilizzare NNAPI per eseguire operazioni di inferenza, sfruttando al meglio l'hardware Android.

É disponibile da Android 8.1 (API 27).

In genere le app non utilizzano direttamente NNAPI ma utilizzano framework di livello superiore (come ad esempio Tensor Flow e PyTorch che verranno introdotti in seguito). In questo caso saranno i framework a utilizzare NNAPI per eseguire le operazioni di inferenza.

Il carico di lavoro di una rete neurale a runtime, in Android può essere distribuito in modo efficiente per sfruttare al meglio i processori disponibili. In particolare come si può vedere dalla *figura 1.1*, se disponibili, vengono usati l'hardware dedicato alla rete neurale (se non disponibile si usa semplicemente la CPU), la GPU (Graphics Processing Unit) e il DSP (Digital Signal Processor)

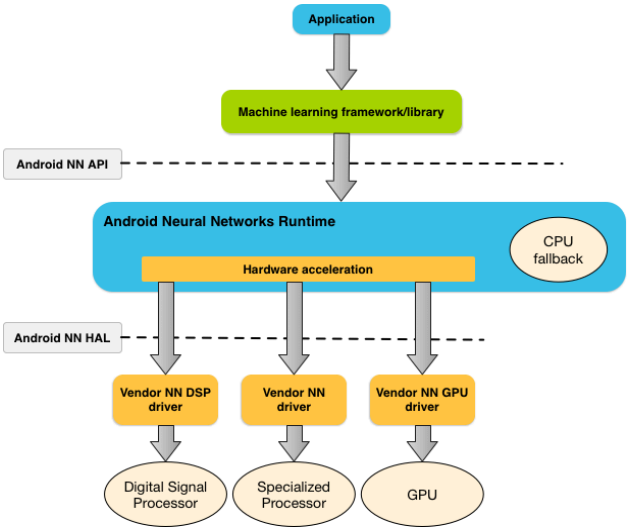

Figura 1.1 Carico di lavoro di una rete neurale [3]

Come prima cosa per utilizzare NNAPI bisogna creare un grafico diretto che definisce i calcoli da eseguire. Questo poi sarà utilizzato insieme ai dati di input per formare il modello per la valutazione del runtime NNAPI.

Vengono utilizzate 4 astrazioni (interfacce):

- Modello: rappresenta il modello della rete neurale da valutare. É un'operazione sincrona. Viene rappresentato con un'istanza di `ANeuralNetworksModel`
- Compilazione: durante la compilazione, il modello viene ottimizzato e tradotto in un formato che può essere interpretato ed eseguito dall'hardware del dispositivo. Anche

questa è un'operazione sincrona ed è rappresentata con un'istanza di `ANeuralNetworksCompilation`.

- Memoria (Buffer): rappresenta la memoria condivisa. In generale un'app crea un buffer di memoria condiviso che contiene tutti i tensori necessari per definire un modello. I tensori possono essere immaginati come degli array multidimensionali (ad esempio un'immagine può essere rappresentata con un tensore tridimensionale in cui le dimensioni corrispondono a altezza, larghezza e colore). Ogni buffer di memoria è rappresentato con un'istanza di `ANeuralNetworksMemory`.
- Esecuzione: rappresenta un'interfaccia per l'applicazione. L'esecuzione può essere sincrona o asincrona ed è rappresentata come un'istanza di `ANeuralNetworksExecution`.

1.5 Fornire accesso ai dati di addestramento

Solitamente i dati allenati sono memorizzati in un file. Bisogna quindi utilizzare `ANeuralNetworksMemory` definito in precedenza. Si usa `ANeuralNetworksMemory_createFromFd()`, a cui bisogna passare come parametro la grandezza del file, il tipo di accesso al file (solitamente sola lettura), un offset che specifica dove la memoria condivisa inizia e infine e ad un puntatore al buffer di memoria creato in precedenza.

Supponiamo ad esempio di avere un modello di rete neurale addestrato per il riconoscimento di immagini. Durante l'addestramento i dati sono stati salvati in un file che chiamiamo "trained_image".

```
ANeuralNetworksMemory* mem1 = NULL;
int fd = open("training_data", O_RDONLY);
ANeuralNetworksMemory_createFromFd(file_size, PROT_READ, fd, 0, &mem1);
```

In questo modo viene fornito un accesso efficiente ai dati.

1.6 Utilizzo dei buffer hardware nativi

I buffer nativi permettono l'accesso diretto alla memoria fisica del sistema, bypassando le gestioni automatiche di memoria dei linguaggi di alto livello, guadagnando così in termini di efficienza. Bisogna ricordarsi, però, di deallocare la memoria una volta finito di utilizzarla. [4]

Possono essere usati per memorizzare dati di input, output o valori costanti del modello di rete neurale. In questo modo alcuni acceleratori NNAPI possono accedere a questi buffer senza la necessità di copiare i dati. Si possono ottenere, in questo modo, prestazioni migliori durante le operazioni di inferenza. Questo non è sempre possibile in quanto i buffer hanno molte configurazioni disponibili e non tutti gli acceleratori NNAPI supportano tutte le configurazioni.

Vediamo un esempio di creazione di questo buffer:

```
#include <android/hardware_buffer.h>
#include <NeuralNetworks.h>

// Configurazione del buffer
AHardwareBuffer_Desc desc;
desc.width = 128;  // Larghezza del buffer in pixel
desc.height = 128; // Altezza del buffer in pixel
//per semplicità viene omessa tutta la configurazione del buffer

// Allocazione del buffer
AHardwareBuffer* ahwb = nullptr;
int result = AHardwareBuffer_allocate(&desc, &ahwb);
if (result != 0) {
    // Gestione dell'errore
}

// Creazione di ANeuralNetworksMemory dal buffer hardware
ANeuralNetworksMemory* mem2 = NULL;
result = ANeuralNetworksMemory_createFromAHardwareBuffer(ahwb, &mem2);
if (result != 0) {
    // Gestione dell'errore
}

// Utilizzo di ANeuralNetworksMemory
// ...

// Pulizia delle risorse
if (ahwb)
    AHardwareBuffer_release(ahwb);

if (mem2)
    ANeuralNetworksMemory_free(mem2);
```

1.7 Creazione, compilazione ed esecuzione del modello

Un modello è l'unità di computazione fondamentale in NNAPI. Ogni modello è definito da una o più operazioni e operandi [3]. Gli operandi sono dati usati nella definizione del grafo (come detto in precedenza il grafo viene definito per la creazione del modello). Ci sono due tipi di operandi che possono essere aggiunti al modello:

- Scalari: rappresenta un valore singolo (che può essere bool, float, int, uint)
- Tensori: sono degli array n-dimensionali. La maggior parte delle operazioni comprende tensori.

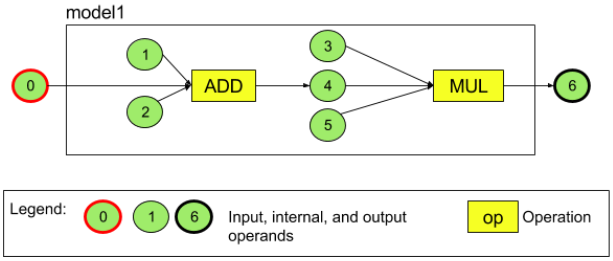

Figura 1.2 Un modello [3]

Ad esempio in *figura 1.2* si può vedere un modello con 2 operazioni (che verranno riprese più avanti), 2 tensori (uno di input e uno di output, rispettivamente i numeri 0 e 6) e 5 scalari.

L'operazione specifica i calcoli da eseguire nel modello di rete neurale. É composta dai seguenti elementi:

- Un tipo di operazione (addizione, moltiplicazione, convoluzione, …)
- Operandi di input
- Operandi di output

Creiamo ora il modello in *figura 1.2*:

```
#include <NeuralNetworks.h>

// Creiamo un modello vuoto
ANeuralNetworksModel* model = NULL;
ANeuralNetworksModel_create(&model);

// Definiamo gli operandi (per semplicità vengono definiti tutti uguali).
Consideriamo i tensori come matrici di dimensione [3][4]
ANeuralNetworksOperandType operandType;
operandType.type = ANEURALNETWORKS_TENSOR_FLOAT32;
operandType.dimensionCount = 2;
operandType.dimensions = (uint32_t[]){3, 4};
```

```
ANeuralNetworksOperandType activationType;
activationType.type = ANEURALNETWORKS_INT32;
activationType.dimensionCount = 0;
activationType.dimensions = NULL;

// Aggiungiamo gli operandi al modello
ANeuralNetworksModel_addOperand(model, &operandType); // 0
ANeuralNetworksModel_addOperand(model, &operandType); // 1
ANeuralNetworksModel_addOperand(model, &activationType); // 2
ANeuralNetworksModel_addOperand(model, &operandType); // 3
ANeuralNetworksModel_addOperand(model, &operandType); // 4
ANeuralNetworksModel_addOperand(model, &activationType); // 5
ANeuralNetworksModel_addOperand(model, &operandType); // 6

// Aggiungiamo un operando per l'output
ANeuralNetworksModel_addOperand(model, &operandType); // output

//Definiamo i valori dei tensori 1 e 3: sono costanti e il loro valore viene
stabilito durante il processo di allenamento
const int sizeOfTensor = 3 * 4 * 4;
ANeuralNetworksModel_setOperandValueFromMemory(model, 1, mem1, 0,
sizeOfTensor);
ANeuralNetworksModel_setOperandValueFromMemory(model, 3, mem1, sizeOfTensor,
sizeOfTensor);

//Impostiamo il valore degli operandi di attivazione
int32_t noneValue = ANEURALNETWORKS_FUSED_NONE;
ANeuralNetworksModel_setOperandValue(model, 2, &noneValue,
sizeof(noneValue));
ANeuralNetworksModel_setOperandValue(model, 5, &noneValue,
sizeof(noneValue));

// Aggiungiamo l'operazione ADD (operandi 0, 1 e 2, risultato in operando 4)
ANeuralNetworksModel_addOperation(model, ANEURALNETWORKS_ADD,
                                  3, (uint32_t[]){0, 1, 2},
                                  1, (uint32_t[]){4});

// Aggiungiamo l'operazione MUL (operandi 3, 4 e 5, risultato in operando 6)
ANeuralNetworksModel_addOperation(model, ANEURALNETWORKS_MUL,
                                  3, (uint32_t[]){3, 4, 5},
                                  1, (uint32_t[]){6});

// Impostiamo input e output del modello
ANeuralNetworksModel_identifyInputsAndOutputs(model,
                                               1, (uint32_t[]){0},
```

```
                                                1, (uint32_t[]){6});

// Finalizziamo il modello
ANeuralNetworksModel_finish(model);

ANeuralNetworksCompilation* compilation;
ANeuralNetworksCompilation_create(model, &compilation);
ANeuralNetworksCompilation_finish(compilation);

ANeuralNetworksExecution* execution;
ANeuralNetworksExecution_create(compilation, &execution);

// Impostiamo i dati di input
float input0 = ...; // Valore dell'input

ANeuralNetworksExecution_setInput(execution, 0, NULL, &input0,
sizeof(input0));

// Impostiamo il buffer per il risultato
float output;
ANeuralNetworksExecution_setOutput(execution, 0, NULL, &output,
sizeof(output));

// Eseguiamo il modello
ANeuralNetworksEvent* event;
ANeuralNetworksExecution_startCompute(execution, &event);
ANeuralNetworksEvent_wait(event);

// Utilizzo del risultato

// Pulizia
ANeuralNetworksEvent_free(event);
ANeuralNetworksExecution_free(execution);
ANeuralNetworksCompilation_free(compilation);
ANeuralNetworksModel_free(model);
```

Gli operandi 3 e 5 vengono definiti come operandi di attivazione: sono funzioni matematiche che vengono applicate all'output per introdurre una non-linearità. In questo modo la rete può apprendere e rappresentare relazioni complesse tra i dati. Un esempio comune di funzione di attivazione è $f(x) = tanh(x)$. Nel nostro esempio usiamo `ANEURALNETWORKS_FUSED_NONE` per dire che non vogliamo nessuna funzione di attivazione.

Durante la compilazione si può includere una preferenza per l'utilizzo della batteria:

```
ANeuralNetworksCompilation_setPreference(compilation,
ANEURALNETWORKS_PREFER_LOW_POWER);
```

Sono disponibili le seguenti preferenze:

- `ANEURALNETWORKS_PREFER_LOW_POWER`: il consumo della batteria è ridotto al minimo.
- `ANEURALNETWORKS_PREFER_FAST_SINGLE_ANSWER`: si vuole ottenere una risposta il più velocemente possibile anche se ciò causa un maggiore consumo (predefinita).
- `ANEURALNETWORKS_PREFER_SUSTAINED_SPEED`: si vuole mantenere una velocità costante su una serie di inferenze continue.

Alla fine della compilazione la funzione `ANeuralNetworksCompilation_finish()` restituisce il codice `ANEURALNETWORKS_NO_ERROR` nel caso non siano presenti errori. Durante l'esecuzione, invece, saranno le funzioni `ANeuralNetworksExecution_startCompute()`, `ANeuralNetworksEvent_wait()` a restituire sempre lo stesso codice di successo.

1.8 Output di dimensioni variabili

Spesso nei modelli non è possibile sapere a priori la dimensione dell'output. Si può quindi assegnare una dimensione che dipende dall'esecuzione del modello.

```
uint32_t myOutputRank = 0;
ANeuralNetworksExecution_getOutputOperandRank(run1, 0, &myOutputRank);

std::vector<uint32_t> myOutputDimensions(myOutputRank);
ANeuralNetworksExecution_getOutputOperandDimensions(run1, 0,
myOutputDimensions.data());
```

In questo esempio otteniamo prima il rango dell'output, ovvero il numero di dimensioni. Ad esempio un tensore [3][4] utilizzato in precedenza ha rango 2. Dopo aver ottenuto il rango determiniamo le dimensioni specifiche di ciascuna dimensione.

CAPITOLO 2 - TENSOR FLOW INTRODUZIONE

Tensor Flow è un famoso framework open-source creato da Google e pensato per l'implementazione di applicazioni che coinvolgono machine learning e deep learning.
Tensor Flow può allenare ed eseguire deep neural networks per il riconoscimento delle immagini, la classificazione delle cifre scritte a mano, il word embedding, l'elaborazione del linguaggio naturale, il rilevamento video e molto altro. Tensor Flow viene eseguito su più CPU o GPU e anche su sistemi operativi mobile.
La parola Tensor Flow è fatta di due parti:

- **Tensor** è un array multidimensionale
- **Flow** è usato per definire il flusso di dati attraverso delle operazioni

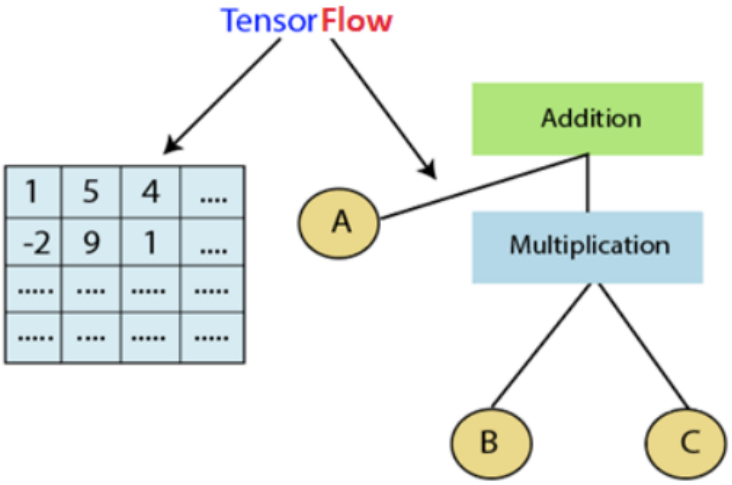

Figura 2.1: definizione di TensorFlow

2.1 Storia ed evoluzione di TensorFlow

Tensor Flow segue le orme di DistBelief, framework closed-source di Google, implementato nel 2012. Basato su reti neurali estese e sul metodo di backpropagation, è stato utilizzato per condurre applicazioni di apprendimento delle funzionalità senza supervisione e deep learning.
Tensor Flow è diverso da DistBelief perché opera in modo indipendente dall'infrastruttura computazionale di Google e ha un architettura di apprendimento automatico più generale che è meno incentrata sulla rete neurale di DistBelief.

Tensor Flow ha raggiunto il livello di rilascio 1.0.0 all'inizio del 2017 sotto licenza Apache Open Source e dall'ottobre del 2019 è disponibile la versione 2.0, che ha ridisegnato il framework in diversi modi per renderlo più semplice ed efficiente.
E' stata aggiunta una nuova interfaccia di programmazione delle API che facilita l'esecuzione di training distribuito e fornisce assistenza per la versione Lite, con cui si implementano modelli su una più ampia gamma di sistemi.

La versione Tensor Flow Lite offre una serie di funzionalità che consentono di eseguire modelli di machine learning Tensor Flow nelle applicazioni Android, andando a limitare le prestazioni e la precisione del modello per ottimizzare il consumo di risorse, l'efficienza energetica e la velocità.

2.2 Come funziona

Tensor Flow consente agli sviluppatori di progettare grafici del flusso di dati, che sono strutture che definiscono come i dati fluiscono tramite un grafico o un insieme di nodi di elaborazione. Ogni nodo nel grafico simboleggia un processo matematico e gli archi tra i nodi rappresentano i dati o i tensori che vengono scambiati tra questi processi.

L'architettura di Tensor Flow consiste di tre componenti:

- Processing dei dati in anticipo
- Creazione del modello
- Sviluppo e valutazione del modello

Il nome, come detto in precedenza deriva dal fatto che accetta input sotto forma di array multidimensionali, noti come Tensori.
Si può creare un diagramma di flusso che rappresenta le azioni che si desiderano condurre sull'input. Quest'ultimo arriva da una parte, passa attraverso il sistema di varie azioni ed esce dalla parte opposta sotto forma di output. Questa modalità di elaborazione è il motivo della parola 'Flow'.

2.3 Grafo

Tensor Flow implementa un'architettura basata sui grafici. Il grafico raccoglie e spiega tutti i calcoli eseguiti durante l'addestramento ed offre diversi vantaggi, come la portabilità, che permette di salvare i calcoli per l'uso corrente o futuro.
Tutti i calcoli nel grafico sono eseguiti collegando i tensori.
Un aspetto chiave dell'architettura di Tensor Flow è la **separazione** tra la definizione del grafico e la sua esecuzione. Questo consente di costruire e modificare il grafico in modo flessibile prima di eseguirlo su una varietà di dispositivi, come CPU, GPU e TPU.
Inoltre, Tensor Flow offre strumenti avanzati per la visualizzazione del grafico, come **Tensor Board**, che permette di monitorare il processo di addestramento, analizzare le prestazioni del modello e diagnosticare problemi. Questa visualizzazione può includere grafici delle metriche di addestramento, immagini del modello, e distribuzioni dei pesi.
Grazie alla sua architettura modulare, Tensor Flow supporta anche l'integrazione con altri framework e librerie di machine learning, facilitando lo sviluppo di soluzioni personalizzate per problemi specifici.

2.4 Tensori.

Tutti i calcoli in TensorFlow usano i tensori, che sono vettori o matrici n-dimensionali che rappresentano tutte le forme di dati. Ogni valore in un tensore ha lo **stesso tipo di dati** e una **forma conosciuta**.
Un tensore può essere generato dai dati di input o dal risultato di una computazione. Tutte le operazioni sono eseguite in un grafo. Il gruppo è un insieme di calcoli che si eseguono in successione. Ogni operazione è chiamata **op node** ed è quindi interconnessa.
Il grafo mostra le operazioni e le relazioni tra i nodi, tuttavia i valori non sono mostrati.

2.5 Features

Tensor Flow offre un'interfaccia di programmazione interattiva e multipiattaforma che è sia scalabile che affidabile, distinguendosi tra le altre librerie di deep learning disponibili.
Di seguito sono elencate le principali proprietà:

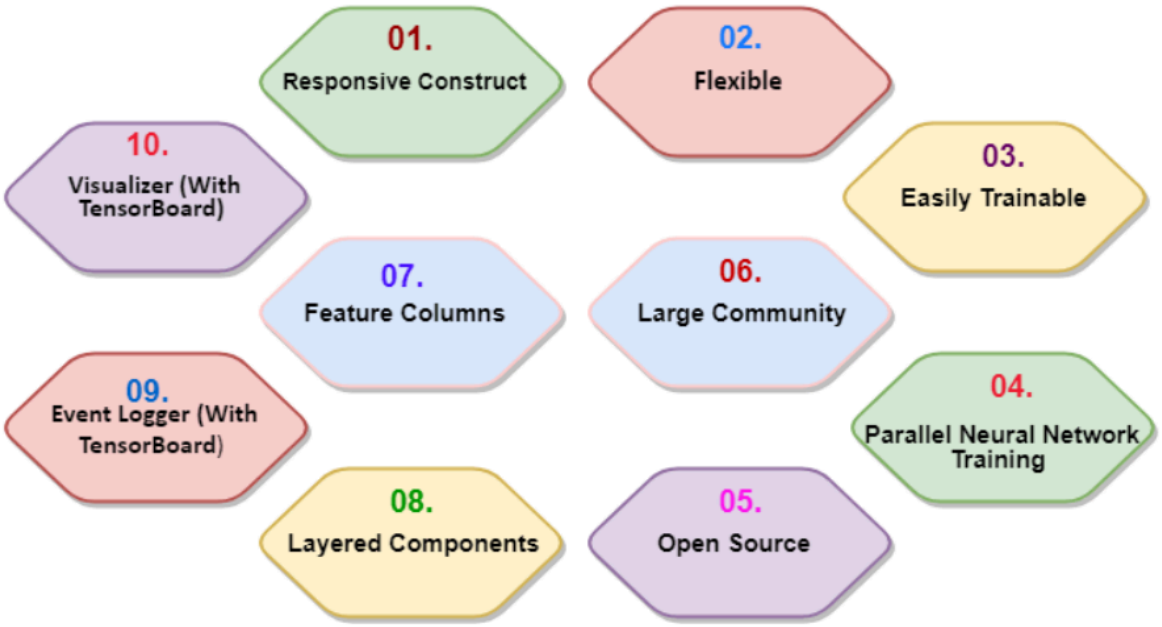

Figura 2.2: Tensor Flow Features

1. **Struttura responsiva**: a differenza di Numpy o SciKit, con Tensor Flow possiamo visualizzare ogni parte del grafico. Per sviluppare un'applicazione di deep learning, sono necessari alcuni componenti fondamentali e un linguaggio di programmazione.
2. **Flessibile**: una delle caratteristiche essenziali di Tensor Flow è la sua flessibilità operativa. È modulare e consente di utilizzare parti del sistema come componenti indipendenti.
3. **Facile da addestrare**: è facilmente addestrabile su CPU e GPU per il calcolo distribuito.
4. **Allenamento parallelo**: Tensor Flow offre un pipeline che consente di addestrare simultaneamente più reti neurali su vari processori grafici (GPU), il che rende i modelli estremamente efficienti su sistemi di grande scala.
5. **Grande comunità**: è stato sviluppato da Google e conta già su un'ampia squadra di ingegneri del software che lavorano continuamente per migliorare la stabilità.
6. **Open source**: la cosa migliore riguardo questo framework è che è open source, quindi chiunque può utilizzarlo purché abbia una connessione internet. Questo permette alle persone di manipolare la libreria e sviluppare una vasta gamma di prodotti utili. Inoltre, ha creato una comunità di appassionati che possono condividere esperienze e aiutarsi reciprocamente tramite un ampio forum.

7. **Feature columns**: possono essere considerate come intermediari tra i dati grezzi e gli stimatori. Fanno in modo di colmare il divario tra i dati di input e il modello.

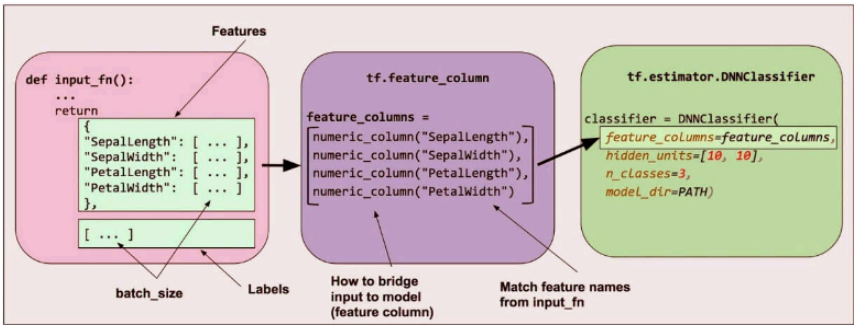

Figura 2.3: Feature columns

8. **Distribuzioni statistiche**: questa libreria fornisce funzioni di distribuzione tra cui Bernoulli, Beta, Chi2, Uniforme, Gamma, che sono fondamentali, specialmente quando si considerano approcci probabilistici come i modelli bayesiani.
9. **Componenti stratificati**: Tensor Flow genera operazioni stratificate di pesi e bias attraverso funzioni come tf.contrib.layers e fornisce anche normalizzazione batch, strato di convoluzione e strato di dropout. Inoltre, tf.contrib.layers.optimizers include ottimizzatori come Adagrad, SGD e Momentum, comunemente usati per risolvere problemi di ottimizzazione nell'analisi numerica.
10. **Tensor Board**: è possibile visualizzare e ispezionare la rappresentazione del modello.

CAPITOLO 3 - PANORAMICA DELLE BASI

In questa sezione forniamo una rapida panoramica delle basi di Tensor Flow. Ogni sezione di questo documento è approfondibile nella documentazione ufficiale.

3.1 Tensori come oggetti tf.Tensor

Tensor Flow opera su array multidimensionali o tensori rappresentati come oggetti `tf.Tensor`.

Vediamo l'esempio di come definire un tensore bidimensionale:

```
import tensorflow as tf
x = tf.constant([[1., 2., 3.],
                 [4., 5., 6.]])

print(x)
print(x.shape)
print(x.dtype)
```

```
>> tf.Tensor(
   [[1. 2. 3.]
   [4. 5. 6.]], shape=(2, 3), dtype=float32)
>> (2, 3)
>> <dtype: 'float32'>
```

Gli attributi principali di un tf.Tensor sono la sua **shape** e **dtype**:

- **`Tensor.shape`:** indica la dimensione del tensore lungo ciascuno dei suoi assi
- **`Tensor.dtype`:** indica il tipo di tutti gli elementi del tensore

TensorFlow implementa operazioni matematiche standard sui tensori, oltre ad una serie di operazioni specializzate per l'apprendimento automatico, vediamo degli esempi partendo dal tensore definito sopra:

1) Somma di due tensori:

```
x + x
```

```
>> <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
   array([[ 2.,  4.,  6.],
          [ 8., 10., 12.]], dtype=float32)>
```

2) Prodotto di un tensore per una costante

```
5 * x
```

```
>> <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
   array([[ 5., 10., 15.],
          [20., 25., 30.]], dtype=float32)>
```

1) Trasposizione

```
x @ tf.transpose(x)
```

```
>> <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
   array([[14., 32.],
          [32., 77.]], dtype=float32)>
```

2) Concatenazione

```
tf.concat([x, x, x], axis=0)
```

```
>> <tf.Tensor: shape=(6, 3), dtype=float32, numpy=
   array([[1., 2., 3.],
          [4., 5., 6.],
          [1., 2., 3.],
          [4., 5., 6.],
          [1., 2., 3.],
          [4., 5., 6.]], dtype=float32)>
```

3) Applicazione della funzione softmax

```
tf.nn.softmax(x, axis=-1)
```

```
>> <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
   array([[0.09003057, 0.24472848, 0.6652409 ],
          [0.09003057, 0.24472848, 0.6652409 ]], dtype=float32)>
```

4) Somma degli elementi del tensore

```
tf.reduce_sum(x)
```

```
>> <tf.Tensor: shape=(), dtype=float32, numpy=21.0>
```

L'esecuzione dei calcoli di grandi dimensioni sulla CPU può essere lenta, per questo è possibile configurare Tensor Flow per l'utilizzo di acceleratori come la GPU per eseguire le operazioni molto più rapidamente.

```
if tf.config.list_physical_devices('GPU'):
  print("TensorFlow **IS** using the GPU")
else:
  print("TensorFlow **IS NOT** using the GPU")
```

```
>> TensorFlow **IS** using the GPU
```

3.2 Variabili

Gli oggetti **`tf.Tensor`** sono immutabili. Per memorizzare qualunque stato mutabile (come i pesi del modello) in Tensor Flow, si utilizza un **`tf.Variabile`**

```
var = tf.Variable([0.0, 0.0, 0.0])
var.assign([1, 2, 3])
```

```
>> <tf.Variable 'UnreadVariable' shape=(3,) dtype=float32, numpy=array([1., 2., 3.],
   dtype=float32)>
```

```
var.assign_add([1, 1, 1])
```

```
>> <tf.Variable 'UnreadVariable' shape=(3,) dtype=float32, numpy=array([2., 3., 4.],
   dtype=float32)>
```

3.3 Differenziazione automatica

Il **gradient descent** [6] è un algoritmo iterativo del primo ordine utilizzato per trovare un minimo locale di una funzione in più variabili differenziabili.
L'idea è quella di effettuare una serie di iterazioni nella direzione opposta del gradiente della funzione rispetto al punto corrente, dato che questa corrisponde alla direzione di discesa più ripida.

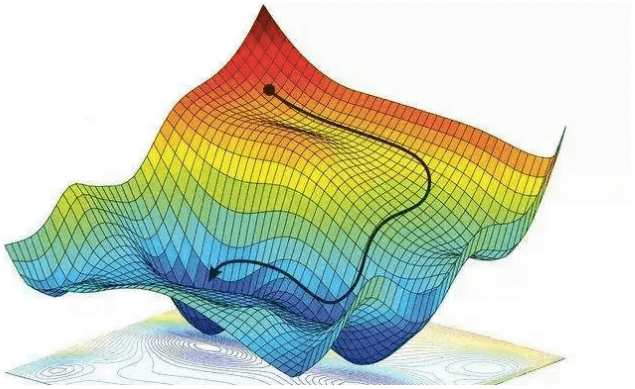

Figura 3.1: gradient Descent

Per consentire ciò, Tensor Flow implementa la differenziazione automatica, che permette di calcolare i gradienti, generalmente la si utilizza per calcolare il gradiente dell'errore o della loss di un modello rispetto ai suoi pesi, per poi effettuare la **backpropagation.**

```
x = tf.Variable(1.0)

def f(x):
  y = x**2 + 2*x - 5
  return y
f(x)
```

```
>> <tf.Tensor: shape=(), dtype=float32, numpy=-2.0>
```

Tensor Flow calcola direttamente la derivata:

```
with tf.GradientTape() as tape:
  y = f(x)

g_x = tape.gradient(y, x)  # g(x) = dy/dx

g_x
```

```
>> <tf.Tensor: shape=(), dtype=float32, numpy=4.0>
```

In questo esempio semplice abbiamo visto la derivata rispetto ad un singolo scalare **x**, ma Tensor Flow può calcolare il gradiente rispetto ad un qualsiasi numero di tensori non scalari contemporaneamente.

3.4 Grafici e funzioni Tensor Flow

Tensor Flow fornisce anche degli strumenti per:

- **Ottimizzare le prestazioni:** per velocizzare allenamento e inferenza
- **Esportazione:** per salvare il modello dopo l'allenamento

Questi strumenti richiedono l'uso di **`tf.function`** per separare il codice Tensor Flow da Python.

```
@tf.function
def my_func(x):
  print('Tracing.\n')
  return tf.reduce_sum(x)
```

La prima volta che si chiama `tf.function`, sebbene venga eseguito in Python, acquisisce un grafico completo e ottimizzato che rappresenta i calcoli Tensor Flow eseguiti all'interno della funzione.

```
x = tf.constant([1, 2, 3])
my_func(x)
```

```
>> Tracing.
```

```
>> <tf.Tensor: shape=(), dtype=int32, numpy=6>
```

Nelle chiamate successive Tensor Flow esegue solo il grafico ottimizzato, possiamo notare che **`my_func`** non stampa la traccia poiché print è una funzione Python, non una funzione TensorFlow

```
x = tf.constant([10, 9, 8])
my_func(x)
```

```
>> <tf.Tensor: shape=(), dtype=int32, numpy=27>
```

Nel caso in cui vengano passati degli input con firma diversa viene generato un nuovo grafico:

```
x = tf.constant([10.0, 9.1, 8.2], dtype=tf.float32)
my_func(x)
```

```
>> Tracing.
>> <tf.Tensor: shape=(), dtype=float32, numpy=27.3>
```

I grafici forniscono una notevole velocità di esecuzione e possono essere esportati utilizzando **`tf.saved_model`**, per eseguirli su altri sistemi come un **server** o un **dispositivo mobile**, senza necessità di installare Python.

3.5 Moduli, livelli e modelli

La classe **`tf.Module`** è utilizzata per gestire oggetti **`tf.Variable`** e **`tf.Function`** che operano su di essi.

Questa classe supporta due caratteristiche significative:

1. Si possono salvare e ripristinare i valori delle variabili utilizzando **`tf.train.Checkpoint`.** Questo torna utile in fase di addestramento per salvare e ripristinare lo stato di un modello
2. E' possibile importare ed esportare i valori **`tf.Variable`** e i grafici **`tf.function`** utilizzando **`tf.saved_model`**.

Questo consente di eseguire il modello in modo indipendente dal programma Python che lo ha creato.

Vediamo un esempio completo di esportazione di un semplice oggetto **`tf.Module`**:

```
class MyModule(tf.Module):
  def __init__(self, value):
    self.weight = tf.Variable(value)

  @tf.function
  def multiply(self, x):
    return x * self.weight
mod = MyModule(3)
mod.multiply(tf.constant([1, 2, 3]))
```

```
>> <tf.Tensor: shape=(3,), dtype=int32, numpy=array([3, 6, 9], dtype=int32)>
```

```
save_path = './saved'
tf.saved_model.save(mod, save_path)
```

Il `SaveModel` risultante è indipendente dal codice che lo ha creato. Si può caricare un `SavedModel` da Python e convertirlo in **Tensor Flow Lite**.

```
reloaded = tf.saved_model.load(save_path)
reloaded.multiply(tf.constant([1, 2, 3]))
```

```
>> <tf.Tensor: shape=(3,), dtype=int32, numpy=array([3, 6, 9], dtype=int32)>
```

Le classi **`tf.keras.layers.Layer`** e **`tf.keras.Model`** si basano su `tf.Module` e forniscono funzionalità aggiuntive e metodi pratici per la creazione, l'addestramento e il salvataggio di modelli.

CAPITOLO 4 - IMPLEMENTAZIONE DI UN MODELLO DI MACHINE LEARNING

In questa sezione vediamo un'applicazione base in cui si utilizzano i tensori, le variabili, il gradient tape e i moduli per allenare un modello. [7]
In questa implementazione utilizzeremo le classi fondamentali per evidenziare il comportamento e per mettere assieme i concetti esposti nella sezione precedente.
Nel prossimo capitolo verrà introdotta la API **`tf.keras`**, una libreria ad alto livello che fornisce un'interfaccia per risolvere problemi di Machine Learning, e di cui si raccomanda l'utilizzo.

4.1 Setup

Importiamo le librerie necessarie

```
import tensorflow as tf
import matplotlib.pyplot as plt
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

In questa applicazione si implementa un modello lineare, $f(x) = x * W + b$ che ha due variabili **W** (weights) e **b** che sono i bias.
Si tratta di fatto di trovare la pendenza e l'offset di una retta tramite la **regressione lineare**.

4.2 Dati

Implementiamo un approccio di **supervised learning** in cui gli *inputs* e gli *outputs* sono forniti. L'obiettivo è di allenare il modello con queste coppie (input, output) in modo che riesca a predire i valori di futuri input.
Ogni dato di ingresso è quasi sempre rappresentato da un tensore, l'output in questo caso è anch'esso un tensore.
Andiamo a generare dei dati aggiungendo del rumore Gaussiano ai punti lungo una linea:

```
# Definizione dei coefficienti della retta
TRUE_W = 3.0  # Weight
TRUE_B = 2.0  # Bias


# Numero di esempi di dati da generare
NUM_EXAMPLES = 201

# Genera un vettore di 201 valori casuali compresi tra -2 e 2
x = tf.linspace(-2, 2, NUM_EXAMPLES)  # Genera valori da -2 a 2
x = tf.cast(x, tf.float32)  # Converte i valori in virgola mobile

# Definizione della funzione per calcolare i valori y
def f(x):
  return x * TRUE_W + TRUE_B

# Generazione del rumore casuale
noise = tf.random.normal(shape=[NUM_EXAMPLES])

# Calcolo dei valori y applicando la funzione f(x) e aggiungendo il rumore
y = f(x) + noise
# Plot dati
plt.plot(x, y, '.')
plt.show()
```

Tipicamente i tensori sono raggruppati insieme in **batches**, o gruppi di inputs e outputs. Il batching può conferire benefici in fase di training e lavora bene con acceleratori di computazione. In questo caso il dataset è piccolo e viene trattato come un singolo batch.

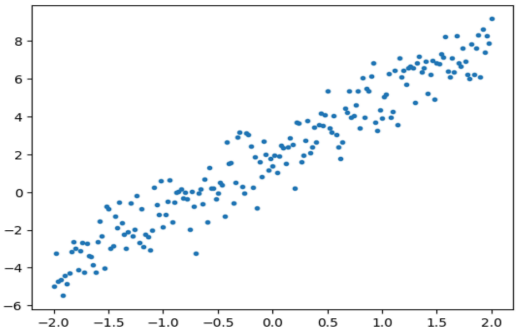

grafico 4.1

4.3 Definizione del modello

Utilizziamo una **`tf.Variable`** per rappresentare tutti i pesi del modello, una variabile memorizza un valore e lo fornisce sotto forma di tensore quando necessario.
La libreria citata precedentemente **`tf.Keras`** utilizza le **`tf.Variable`** per memorizzare i parametri del modello.

Si definiscono quindi W e b come variabili:

```
class MyModel(tf.Module):
  def __init__(self, **kwargs):
    super().__init__(**kwargs)
    # Inizializzazione del peso e del bias
    self.w = tf.Variable(5.0)      self.b = tf.Variable(0.0)

  def __call__(self, x):
    # Calcolo l'output del modello
    return self.w * x + self.b

model = MyModel()

# Stampo le variabili di aggregazione delle variabili incorporate di tf.Module.
print("Variabili:", model.variables)

# Verifico che il modello funzioni utilizzando il metodo __call__
assert model(3.0).numpy() == 15.0  # Verifica se l'output è corretto
```

Le variabili sono inizializzate con dei valori fissati, nelle applicazioni pratiche queste dovrebbero assumere valori di partenza randomici.

4.4 Definizione della loss function

Definiamo la standard **L2 loss function**, conosciuta anche come “mean squared” error:

```
# Funzione che calcola un singolo valore di loss per un batch
def loss(target_y, predicted_y):
  # media dei quadrati tra valori target e predicted
  return tf.reduce_mean(tf.square(target_y - predicted_y))
```

Prima di allenare il modello, possiamo visualizzare il loss value:

```
plt.plot(x, y, '.', label="Data")
plt.plot(x, f(x), label="Ground truth")
plt.plot(x, model(x), label="Predictions")
plt.legend()
plt.show()

print("Current loss: %1.6f" % loss(y, model(x)).numpy())
```

Inizialmente il loss è: `9.949693`

Dovuto al fatto che abbiamo

inizializzato una retta con dei pesi arbitrari.

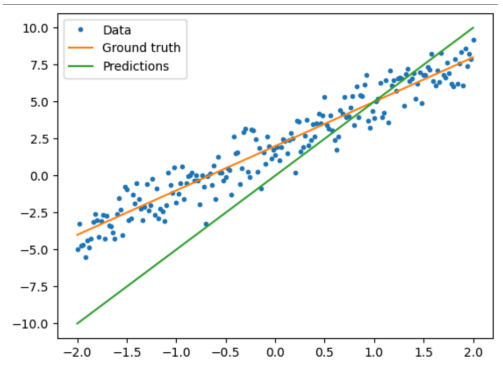

grafico 4.2

4.5 Definizione di un training loop

Il training loop consiste nell'esecuzione ripetuta di quattro tasks in ordine:

1. Mandare un batch di inputs attraverso il modello per generare gli outputs
2. Calcolare un singolo valore di loss comparando gli outputs ottenuti ai valori target
3. Utilizzare il gradient tape per trovare il gradiente
4. Ottimizzare i parametri sottraendo il valore trovato

Per questo esempio utilizziamo il gradient descent, implementando i calcoli matematici con l'aiuto di **`tf.GradientTape`** per la differenziazione automatica (calcolo del gradiente) e **`tf.assing_sub`** per decrementare un valore.

```
def train(model, x, y, learning_rate):

  # Calcolo del gradiente
  with tf.GradientTape() as t:
    # Variabili addestrabili ottenute da GradientTape
    current_loss = loss(y, model(x))

  # Utilizzo GradientTape per calcolare i gradienti rispetto a W e b
  dw, db = t.gradient(current_loss, [model.w, model.b])

  # Sottraggo il gradiente scalato dal tasso di apprendimento
  model.w.assign_sub(learning_rate * dw)
  model.b.assign_sub(learning_rate * db)
```

Possiamo osservare come evolvono i parametri w e b, a seguito di varie iterazioni di training.

```
# Creazione di un'istanza del modello
model = MyModel()
```

```
weights = []
biases = []
epochs = range(10)  # Numero di epoche di addestramento

def report(model, loss):
    return f"W = {model.w.numpy():1.2f}, b = {model.b.numpy():1.2f},
loss={loss:2.5f}"

def training_loop(model, x, y):
    # Ciclo attraverso le epoche di addestramento
    for epoch in epochs:
        # Aggiornamento del modello passando il batch
        train(model, x, y, learning_rate=0.1)

        # Aggiungo i parametri alle liste di output
        weights.append(model.w.numpy())
        biases.append(model.b.numpy())

        # Calcolo della perdita relativa a questa epoch
        current_loss = loss(y, model(x))

        # Stampo il report di quest epoch
        print(f"Epoch {epoch:2d}:")
        print("    ", report(model, current_loss))
```

Effettuiamo il training e visualizziamo l'evoluzione dei parametri:

```
current_loss = loss(y, model(x))

print(f"Starting:")
print("    ", report(model, current_loss))

training_loop(model, x, y)
```

```
>> Starting:
     W = 5.00, b = 0.00, loss=9.94969
>> Epoch  0:
     W = 4.48, b = 0.39, loss=6.17946
>> Epoch  1:
     W = 4.09, b = 0.71, loss=4.01929
>> Epoch  2:
     W = 3.81, b = 0.96, loss=2.77173
>> Epoch  3:
     W = 3.61, b = 1.16, loss=2.04534
```

```
>> Epoch  4:
   W = 3.46, b = 1.32, loss=1.61892
>> Epoch  5:
   W = 3.35, b = 1.45, loss=1.36654
>> Epoch  6:
   W = 3.27, b = 1.55, loss=1.21598
>> Epoch  7:
   W = 3.21, b = 1.63, loss=1.12547
>> Epoch  8:
   W = 3.17, b = 1.70, loss=1.07067
>> Epoch  9:
   W = 3.14, b = 1.75, loss=1.03727
```

Visualizziamo l'evoluzione dei pesi nel tempo e visualizziamo che ipotesi si ottiene dal modello:

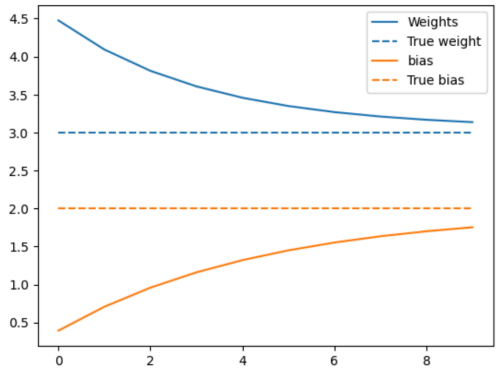

grafico 4.3

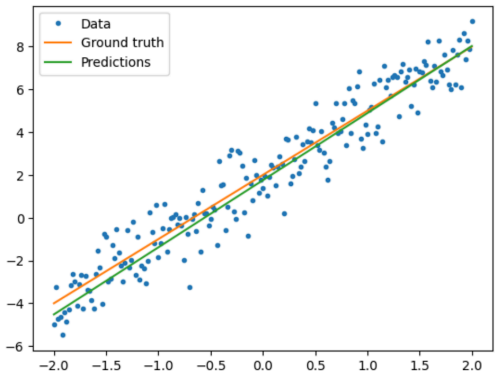

grafico 4.4

4.6 Commento e miglioramenti

In questa sezione abbiamo visto passo passo un'implementazione di un modello di machine learning mettendo in pratica i concetti descritti nel capitolo 3.

Nella prossimo capitolo verrà descritta la libreria **`tf.keras`** che permette di utilizzare features e shortcuts già implementati per semplificare il processo di creazione e training di un modello.

CAPITOLO 5 - Keras

In questa sezione viene descritta l'API Keras, spiegata passo passo sia implementando ed allenando una serie di reti neurali da zero che vedendo come utilizzare delle reti pre-allenate più complesse.
Sono coperte anche nozioni di base su come organizzare e processare i dati di input.

Figura 5.1: Logo di Keras

5.1 Perchè utilizzare Keras

Keras è un API sviluppata originariamente da Francois Chollet. Originariamente era una API di alto livello che stava al di sopra di tre APIs di livello più basso (backend engines) ed agiva come wrapper.
Keras è stata sviluppata con lo scopo di permettere la sperimentazione veloce, passando da idea a implementazione in pochi step.
Negli ultimi anni, Tensor Flow è diventata la backend engine più popolare.

Attualmente Keras è diventato **integrato** nella libreria di Tensor Flow e viene distribuita nello stesso package, quindi scaricando Tensor Flow si ottiene automaticamente anche Keras. Questa integrazione consente di usare funzionalità Keras di alto livello per fare molte cose senza avere la necessità di usare codice Tensor Flow di basso livello.
Per utilizzarla basta semplicemente eseguire il comando:

```
pip install tensorflow
```

Il codice Tensor Flow, viene eseguito su una singola GPU senza necessità di scrivere del codice esplicito, gli unici passaggi necessari sono di installare i drivers Nvidia e il CUDA Toolkit.

5.2 Data processing per il training della rete neurale

5.2.1 Formato dei dati

Quando si allena una qualunque rete neurale con il metodo **supervised learning**, abbiamo bisogno di un insieme di campioni e di labels corrispondenti a tali campioni.

- Campioni: insieme dei dati, dove ogni singolo elemento all'insieme è chiamato campione (sample)
- Etichette: le etichette (labels) sono gli output corrispondenti ad ogni sample, definiti da una funzione chiamata **ground truth**

Tipicamente ci si riferisce ai campioni come dati di input e alle labels come valori di output.
Quando si preparano i dati, il primo passo è quello di capire il formato con cui raccoglierli, in modo da poterli passare al modello di rete neurale che si sta sviluppando.
Nella prima implementazione notevole che verrà proposta si sviluppa un modello **sequenziale**. Il modello sequenziale riceve i dati durante l'addestramento, che avviene quando si chiama la funzione fit() sul modello.
La documentazione della funzione `fit()` [10] richiede che i dati di input debbano essere di uno di questi tre tipi.

- Un `numpy array` o una lista di array
- Un Tensore Flow `Tensor` o una lista di tensori
- Un `dict` che mappa gli input names ai corrispondenti array/tensori
- Un set di dati `tf.data` dovrebbe restituire una tupla di (input, target) o (input, target, sample_weights).

- Un generatore o `keras.utils.Sequence` che restituisce (input, target) o (input, target, sample_weights).

Quindi quando si raggruppano i dati bisogna essere certi che siano contenuti in una delle strutture dati sopra citata.

Come gli input anche le etichette corrispondenti possono essere raggruppate in array Numpy o tensori, si noti che la scelta deve essere coerente con quella utilizzata per i campioni. Non si possono avere campioni Numpy ed etichette tensoriali, o viceversa.

Un altra ragione per cui è importante formattare i dati è quella di trasformarli in modo da rendere più facile, veloce ed efficiente l'apprendimento della rete. Questo avviene tramite tecniche di normalizzazione o standardizzazione dei dati.

Vediamo come effettuare il data processing nell'implementazione del modello di rete sequenziale per effettuare una classificazione:

Si importano le librerie:

```
import numpy as np
from train_labels = []
train_samples = []
```

Si creano due liste vuote per i campioni e per le labels:

```
train_labels = []
train_samples = []
```

5.2.2 Creazione dei dati

In questo caso semplice la creazione dei dati è fatta dal programma:

[9] Come motivazione per questi dati, supponiamo che un farmaco sperimentale sia stato testato su individui di età compresa tra i 13 e i 100 anni in uno studio clinico. Lo studio contava 2100 partecipanti. Metà dei partecipanti aveva meno di 65 anni e l'altra metà aveva 65 anni o più.

Lo studio ha mostrato che circa il 95% dei pazienti di età pari o superiore a 65 anni ha sperimentato effetti collaterali del farmaco, mentre circa il 95% dei pazienti di età inferiore a 65 anni non ha sperimentato effetti collaterali, mostrando in generale che gli individui anziani avevano maggiori probabilità di sperimentare effetti collaterali.

In definitiva, vogliamo costruire un modello che ci dica se un paziente sperimenterà o meno effetti collaterali solo in base alla sua età. Il giudizio del modello si baserà sui dati di addestramento.

Si implementa la generazione dei dati:

```
for i in range(50):
    # Circa il 5% dei giovani che hanno sperimentato effetti collaterali
    random_younger = randint(13,64)
    train_samples.append(random_younger)
    train_labels.append(1)

    # Circa il 5% degli anziani che non hanno sperimentato effetti
collaterali
    random_older = randint(65,100)
    train_samples.append(random_older)
    train_labels.append(0)

for i in range(1000):
    # Circa il 95% dei giovani che non hanno sperimentato effetti
collaterali
    random_younger = randint(13,64)
    train_samples.append(random_younger)
    train_labels.append(0)

    # Circa il 95% degli anziani che hanno sperimentato effetti collaterali
    random_older = randint(65,100)
    train_samples.append(random_older)
    train_labels.append(1)
```

Questo codice crea 2100 campioni e memorizza l'età degli individui nella lista `train_samples` e in `train_labels` se questi hanno avuto delle conseguenze (1 = sì, 0 = no)

5.2.3 Processing dei dati

Si convertono entrambe le liste ottenute in due numpy arrays per renderle compatibili al formato richiesto dalla funzione `fit()`, si effettua successivamente un mescolamento dei dati per rimuovere ogni tipo di ordine e dipendenza importi durante il processo di creazione.

```
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
train_labels, train_samples = shuffle(train_labels, train_samples)
```

Come ultimo passo si utilizza la funzione `MinMaxScaler` di scikit-learn per comprimere tutti i valori nel range da 13 a 100 in un range compreso tra 0 e 1. Si ridimensionano i dati perché la funzine `fit_transform()` non accetta dati unidimensionali.

```
scaler = MinMaxScaler(feature_range=(0,1))
scaled_train_samples= scaler.fit_transform(train_samples.reshape(-1,1))
```

5.2.4 Costruzione del modello

In questa sezione viene spiegata la costruzione del modello di rete sequenziale. Si riportano gli import necessari:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import categorical_crossentropy
```

Come primo passo si crea la variabile **`model`**:

```
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='softmax')
])
```

`model` è un'istanza di un oggetto **`Sequential`.** Un **`tf.keras.Sequential`** model è una pila lineare di livelli. Accetta come parametro una lista in cui ogni elemento è un layer della rete.

Il primo layer (nascosto) è di tipo **Dense**, ovvero il tipo standard di fully-connected (o densely-connected) layer di una rete neurale.

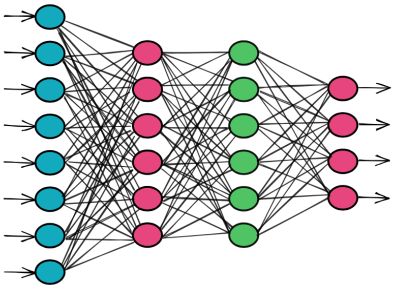

Figura 5.2: Neural network

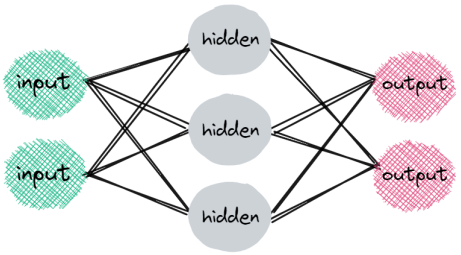

Figura 5.3: Hidden layers

Il primo parametro richiesto dal **Dense** layer è il **numero di neuroni** o **units**, si imposta questo valore arbitrariamente a 16.

Oltre a questo il modello deve conoscere la forma dei dati in ingresso, lo specifichiamo tramite il parametro **`input_shape`** (solo nel primo hidden layer).

I dati di input generati ed elaborati nella sezione precedente sono monodimensionali, il parametro `input_shape` si aspetta una tupla di numeri interi corrispondente alla forma dei dati di ingresso, quindi specifichiamo `(1,)` come shape.

Si può visualizzare il modo con cui si specifica `input_shape` come un ulteriore **layer di input esplicito**.

Infine, si definisce la funzione di attivazione per l'**hidden layer** a **`relu`**, la funzione di attivazione mappa gli input di un nodo nel suo corrispondente output:

$$node\ output\ =\ activation(weighted\ sum\ of\ inputs)$$

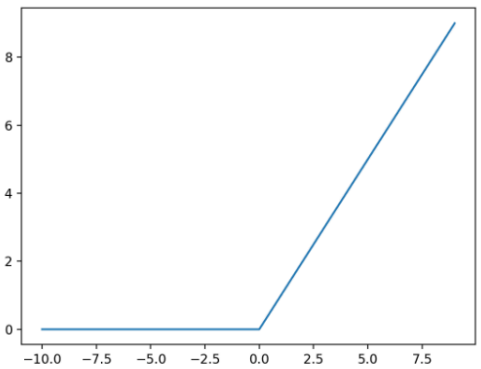

Immagine 5.4: relu activation function

Il prossimo livello è un altro dense layer con 32 neuroni, scelta che anche un questo caso è arbitraria.
Se ci si accorge che questo è insufficiente in fase di testing è possibile risolvere il problema iniziando a sperimentare la modifica dei parametri, come il numero dei livelli, i nodi, ecc.

L'ultimo strato ha 2 neuroni, corrispondenti ai due possibili valori di output: o un paziente ha avuto effetti collaterali, o il paziente non li ha avuti.
Questa volta, la funzione di attivazione che utilizzeremo è **`softmax`**, che restituisce una distribuzione di probabilità tra le possibili uscite.

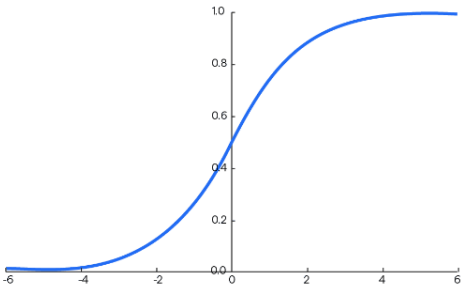

Figura 5.5: softmax activation function

Con la funzione **`summary()`** si ottiene una rapida visualizzazione del modello:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 32)                544
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 66
=================================================================
Total params: 642
Trainable params: 642
Non-trainable params: 0
_________________________________________________________________
```

5.3 Allenamento della rete neurale con Keras

La prima cosa da fare per rendere il modello pronto ad essere allenato è chiamare la funzione **`compile()`**

```
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Questa funzione si occupa di configurare il modello per il training e si aspetta i parametri:

- `optimizer`: specifichiamo l'ottimizzatore Adam che accetta come parametro opzionale un `learning_rate`, che è impostato a 0.0001. Un ottimizzatore è un metodo di **stochastic gradient descent.**
- `loss`: utilizziamo `sparse_categorical_crossentropy`, dato che le labels sono ti tipo intero, l'altra possibile opzione per questo parametro è `binary_crossentropy`, se si configura l'output layer in modo da avere un solo output, invece di due, la scelta è equivalente. Con la scelta di `binary_crossentropy` l'ultimo layer dovrebbe usare la funzione di sigmoid invece di softmax.
- `metrics`: questo parametro accetta una lista di metriche che vengono valutate dal modello durante il training e il testing. In questo caso si specifica solo accuracy.

Ora che il modello è configurato si inizia l'allenamento utilizzando la funzione **`fit()`**:

```
model.fit(x=scaled_train_samples, y=train_labels, batch_size=10, epochs=30,
verbose=2)
```

La funzione fit esegue l'allenamento della rete, i parametri in questo caso sono:

- `x`: corrisponde al training set che è stato creato nelle sezioni precedenti
- `y`: corrisponde alle labels associate al training set
- `batch_size`: specifica il numero di campioni che vengono passati alla rete in una sola volta
- `epochs`: specifica il numero di volte in cui si effettua un singolo passaggio di tutti i dati alla rete
- `verbose`: indica quanto output vogliamo vedere nella console relativo ad ogni epoca del training. Il livello di verbosità varia tra 0 e 2.

Eseguendo il comando si osservano gli output per ognuna delle 30 epoche, e si vede come le metriche di loss e accuracy migliorino costantemente ad ogni iterazione, fino a raggiungere un accuracy di quasi 93% e una loss di 0.27.

Tramite l'esecuzione di questo semplice modello, allenato su un set banale di dati, si può notare comunque come sia possibile ottenere dei buoni risultati in modo semplice e veloce, questa caratteristica di **`tf.keras`** si mantiene anche per modelli più complessi.

5.4 Creazione di un validation set con Keras

Ora che il modello è allenato, l'idea è quella di prendere questo modello, applicarlo a nuovi dati mai visti prima, e fare in modo che il modello preveda accuratamente gli output di questi dati, basandosi su ciò che ha appreso dal set di addestramento.

Prima di iniziare l'addestramento, si può scegliere di rimuovere una parte del training set e di inserirla in un set di validazione. In questo modo, durante l'addestramento, il modello utilizza **solo** il training set per modificare i suoi pesi e bias e sfrutta poi il validation set per capire quanto bene riesce a predire dei dati diversi da quelli che ha utilizzato per imparare.

Durante ogni epoch, si vedranno non solo i risultati di **loss** e **accuracy** per l'insieme di allenamento, ma anche per l'insieme di validazione.

Questo aiuta a capire se il modello è in **overfitting**, ovvero se sta apprendendo troppo in dettaglio le specifiche dei dati di addestramento e non è in grado di predire correttamente i dati su cui non è stato addestrato.

La creazione di un validation set può avvenire manualmente, questo però necessita di creare un set di valutazione **`valid_set = (x_val, y_val)`** formato da array numpy o tensori. Quando si chiama la funzione `model.fit()` si passa questo set come valore del parametro **`validation_data`.**

Keras permette di **creare automaticamente un validation set,** senza doverlo fare manualmente. E' sufficente aggiungere un parametro alla funzione `model.fit()`:

```
model.fit(
      x=scaled_train_samples
    , y=train_labels
    , validation_split=0.1
    , batch_size=10
    , epochs=30
    , verbose=2
)
```

Il parametro `validation_split` si aspetta una frazione compresa tra 0 e 1. Una volta specificato, Keras separa una frazione corrispondente del training set che verrà utilizzata come validation data. Il modello non verrà allenato su questo subset, e valuterà la loss e le metriche su questo set alla fine di ogni epoch.

E' importante notare che la funzione `fit()` di default mescola i dati prima di ogni epoch. Tuttavia quando si specifica il parametro `validation_split`, i dati vengono selezionati dagli ultimi campioni dei set x e y **prima del mescolamento.** In caso sia necessario (come in questo) i dati vanno mescolati prima della chiamata della funzione fit().

Chiamando la funzione `model.fit()` si vedono ora anche i valori di loss e accuracy per il validation set selezionato da Keras in ogni epoca. Si osserva che il modello non impara solamente le caratteristiche del training set ma ottiene valori di loss e accuracy consistenti anche su dati mai visti, che alla fine arrivano a circa 0.25 e 93%.

5.5 Inferenza in una rete neurale con Keras

Si noti che il set di test è l'insieme di dati utilizzati specificamente per la valutazione delle prestazioni della rete dopo la conclusione dell'addestramento.
In generale, il test set deve essere sempre processato nello stesso modo del training set.
Procediamo con la creazione:

```
test_labels =  []
test_samples = []

# * Generazione dei samples come nell'esempio precedente *

test_labels = np.array(test_labels)
test_samples = np.array(test_samples)
test_labels, test_samples = shuffle(test_labels, test_samples)

scaled_test_samples = scaler.fit_transform(test_samples.reshape(-1,1))
```

Per ottenere delle predizioni dal modello, chiamiamo **`model.predict()`**:

```
predictions = model.predict(
      x=scaled_test_samples
    , batch_size=10
    , verbose=0
)
```

I parametri di questa funzione sono il set di test, la batch_size che corrisponde a quella specificata nel training e il livello di verbosity che in questo caso settiamo a 0, dato che l'output delle predizioni non è utile.

Osserviamo le predizioni del modello:

```
for i in predictions:
    print(i)
```

```
>> [ 0.74106830  0.25893170]
>> [ 0.14958295  0.85041702]
>> [ 0.96918124  0.03081879]
>> [ 0.12985019  0.87014979]
>> [ 0.88596725  0.11403273]
>> ...
```

Ogni elemento della lista di predizioni è esso stesso una lista di lunghezza 2. Le due colonne contengono la **distribuzione di probabilità** di ogni possibile evento assumibile dall'output:

- Il soggetto ha sofferto di effetti collaterali (rappresentato da 0)
- Il soggetto non ha sofferto di effetti collaterali (rappresentato da 1)

questo spiega come mai la somma di ogni riga fa 1.

Possiamo visualizzare meglio la predizione più probabile arrotondando i valori:

```
rounded_predictions = np.argmax(predictions, axis=-1)

for i in rounded_predictions:
    print(i)
```

```
>> 0 1 0 1 0 ...
```

Questa visualizzazione consente di osservare le predizioni, ma non permette di valutare quanto queste sono accurate.

Se si hanno le labels corrispondenti per il test set, si possono comparare le labels corrette a quelle ottenute dalla rete grazie alla **confusion matrix**.

Importiamo le librerie necessarie:

```
%matplotlib inline
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
```

Si crea la confusion matrix a cui si passano le labels corrette assieme alle predizioni della rete sul test set arrotondate.

```
cm = confusion_matrix(y_true=test_labels, y_pred=rounded_predictions)
```

Sul sito di scikit-learn [12] è riportata l'implementazione della funzione **`plot_confusion_matrix()`.**

Si definiscono infine le labels per la matrice e si sfrutta la funzione sopra citata per stampare la matrice.

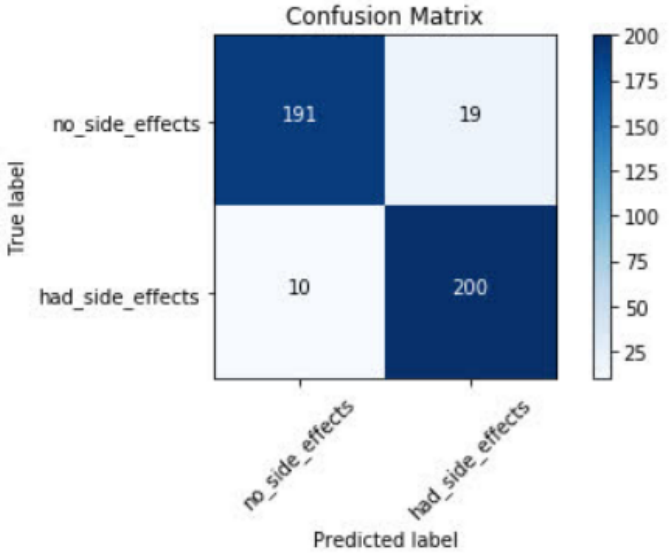

Immagine 5.6: Confusion Matrix

Sull'asse delle x sono rappresentate le predicted labels mentre sull'asse delle y le true labels. Le celle colorate di blu nella diagonale principale contengono il numero di campioni predetti correttamente dal modello. Le celle colorate di bianco contengono il numero di campioni con una predizione errata.

Si nota per esempio che il modello ha predetto accuratamente che i pazienti non avrebbero avuto effetti collaterali 191 volte, mentre ha sbagliato 10 volte.

Come si può vedere, questo è un modo intuitivo per interpretare visivamente ciò che sta facendo il modello nella fase di inferenza e capire dove potrebbe essere necessario apportare delle modifiche.

5.6 Salvare e Importare un modello con Keras

Esistono una serie di modi diversi di salvare un modello Keras. Questi meccanismi servono ognuno a salvare parti diverse, li vediamo uno alla volta.

5.6.1 Salvare e caricare interamente il modello

Se si vuole salvare il modello assieme allo stato in cui si trova dopo la fase di training, in modo da poterlo utilizzare successivamente, è possibile utilizzare la funzione **`save()`**. Questa funzione richiede come parametro il path e il nome del file in cui si vuole salvare il modello, impostando come estensione **h5**.

```
model.save('models/medical_trial_model.h5')
```

La funzione `save()` permette di salvare il modello come un TensorFlow **`SaveModel`**. Questo primo metodo di salvataggio memorizza tutte le informazioni del modello: l'architettura, i pesi, l'ottimizzatore, lo stato dell'ottimizzatore, il learning rate, la loss, ecc. Per caricare il modello salvato si importa la funzione **`load_model()`.** Successivamente si può chiamare la funzione per caricare il modello passando il path del file in cui si trova:

```
from tensorflow.keras.models import load_model
new_model = load_model('models/medical_trial_model.h5')
```

Una volta importato è possibile verificare le varie caratteristiche del modello usando le funzioni `summary()`, `get_weights()` e chiamando `model.optimizer` e `model.loss`, per andare a verificare che corrispondano a quelle del modello precedentemente memorizzato.

5.6.2 Salvare e caricare solo l'architettura del modello

Esiste un altro modo di salvare solo l'architettura del modello, senza memorizzare i pesi e le varie configurazioni.
Questo si ottiene chiamando **`model.to_json()`**, funzione che salva l'architettura del modello sotto forma di stringa JSON.

```
json_string = model.to_json()
```

Ora che il file è salvato se ne può creare uno nuovo partendo da questo. Per prima cosa si importa la funzione **`model_from_json()`** e si carica l'architettura memorizzata:

```
from tensorflow.keras.models import model_from_json
model_architecture = model_from_json(json_string)
```

Chiamando la funzione `model.summary()` si può verificare che l'architettura corrisponde a quella precedentemente salvata:

```
model_architecture.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 32)                544
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 66
=================================================================
Total params: 642
Trainable params: 642
Non-trainable params: 0
_________________________________________________________________
```

5.6.3 Salvare e importare i pesi del modello

Questo ultimo approccio di salvataggio si ottiene chiamando **`model.save_weights()`** e passando il path e il nome del file in cui salvare i pesi, con estensione h5.

```
model.save_weights('models/my_model_weights.h5')
```

Successivamente, si possono importare i pesi ed utilizzarli su un nuovo modello, che deve avere la stessa architettura di quello da cui sono stati salvati.

```
model2 = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='softmax')
])

model2.load_weights('models/my_model_weights.h5')
```

5.7 Preparazione delle immagini per una Convolutional Neural Network con Keras

Vediamo ora un altro esempio il cui scopo è quello di costruire e creare una **CNN** che può identificare accuratamente immagini di gatti e cani.

Il primo passo è quello di ottenere i dati, in questo caso il dataset consiste in un subset dei dati provenienti da Kaggle Dogs Versus Cats competition [13].

Con lo script seguente si organizzano i dati in train, validation, e test set. Per ottenere ogni set si spostano veri sottoinsiemi dei dati in sottocartelle appropriate.

```
os.chdir('data/dogs-vs-cats')
if os.path.isdir('train/dog') is False:
    os.makedirs('train/dog')
    os.makedirs('train/cat')
    os.makedirs('valid/dog')
    os.makedirs('valid/cat')
    os.makedirs('test/dog')
    os.makedirs('test/cat')

    for i in random.sample(glob.glob('cat*'), 500):
        shutil.move(i, 'train/cat')
    for i in random.sample(glob.glob('dog*'), 500):
        shutil.move(i, 'train/dog')
    for i in random.sample(glob.glob('cat*'), 100):
        shutil.move(i, 'valid/cat')
    for i in random.sample(glob.glob('dog*'), 100):
        shutil.move(i, 'valid/dog')
    for i in random.sample(glob.glob('cat*'), 50):
        shutil.move(i, 'test/cat')
    for i in random.sample(glob.glob('dog*'), 50):
        shutil.move(i, 'test/dog')
os.chdir('../../')
```

Questo codice controlla semplicemente se la gerarchia di cartelle è già creata, in caso contrario procede con lo script e crea le sottocartelle, in cui sposta le immagini.

Il dataset completo contiene 25000 immagini, metà di gatti e metà di cani. Lo script sposta 1000 campioni nel training set, 200 campioni nel validation set e 100 nel test set, ognuno ha un numero uguale di gatti e cani.

Le immagini rimanenti non vengono utilizzate, dato che non è necessario avere un dataset così grande per un'implementazione semplice come quella in esame.

5.7.1 Cosa fare se non si hanno le test labels

In molti casi di uso reale non si possiedono le labels corrispondenti al test data. In questo caso la directory **test** dovrebbe essere di questo tipo:

```
test\unknown\
```

Tutti i file senza label vanno inseriti nella sub-directory **unknown** (nome scelto arbitrariamente).

5.7.2 Processare i dati

Come nell'esempio precedente si controlla se Tensor Flow riesce ad identificare una GPU e si abilita il memory growth sulla GPU.

```
physical_devices = tf.config.experimental.list_physical_devices('GPU')
print("Num GPUs Available: ", len(physical_devices))
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

Si creano successivamente le variabili a cui si assegnano i path delle sub-directory create precedentemente in fase di organizzazione:

```
train_path = 'data/dogs-vs-cats/train'
valid_path = 'data/dogs-vs-cats/valid'
test_path = 'data/dogs-vs-cats/test'
```

Fatto ciò si creano delle istanze di **`ImageDataGenerator`** per creare i batches di dati dalle varie directory:

```
train_batches = ImageDataGenerator(
preprocessing_function = tf.keras.applications.vgg16.preprocess_input) \
    .flow_from_directory(directory=train_path, target_size=(224,224),
classes=['cat', 'dog'], batch_size=10)

valid_batches = ImageDataGenerator(
preprocessing_function = tf.keras.applications.vgg16.preprocess_input) \
    .flow_from_directory(directory=valid_path, target_size=(224,224),
classes=['cat', 'dog'], batch_size=10)

test_batches = ImageDataGenerator(
preprocessing_function = tf.keras.applications.vgg16.preprocess_input) \
    .flow_from_directory(directory=test_path, target_size=(224,224),
classes=['cat', 'dog'], batch_size=10, shuffle=False)
```

`ImageDataGenerator.flow_from_directory()` crea un `DirectoryIterator` che genera batch di immagini sotto forma di tensori normalizzati, dalle rispettive directories.
Per ogni `ImageDataGenerator` si specifica `preprocessing_function=tf.keras.applications.vgg16.preprocess_input`.
Questo parametro specifica che viene effettuato un ulteriore processing delle immagini.
Per la funzione `flow_from_directory()` si passano i seguenti parametri:

- Il path dei dati
- `target_size`: che ridimensiona le immagini nella dimensione specificata, in questo caso è determinata dall'input che la rete neurale si aspetta
- `classes`: lista che contiene i nomi delle classi di dati
- `batch_size`: dimensione dei batches

Si specifica inoltre `shuffle=False` solo per il `test_batches`, questo perchè quando si plotterà la confusion matrix sarà necessario che le labels del test set siano in ordine. Di default, i datasets sono mescolati.
Nel caso in cui non si conoscano le labels dei test data va modificato la variabile `test_batches`. Si impostano i parametri `classes=None` e `class_mode=None` in `flow_from_directory()`.

5.7.3 Visualizzare i dati

Per visualizzare un batch di dati e labels dal training set si chiama la funzione `next(train_batches)`.

```
imgs, labels = next(train_batches)
```

Usiamo la funzione di Tensor Flow `plotImages` [14] per vedere le immagini:

```
plotImages(imgs)
print(labels)
```

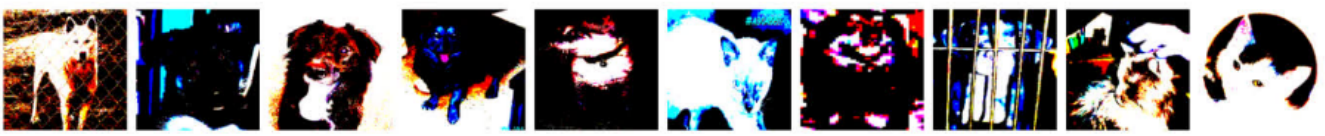

Figura 5.5: Training set batch

```
>> [[0. 1.]
    [0. 1.]
    [0. 1.]
    [0. 1.]
    [1. 0.]
    [1. 0.]
    [0. 1.]
    [0. 1.]
    [1. 0.]
    [1. 0.]]
```

Il colore delle immagini appare distorto perché è stato applicato il processing VGG16.
I cani sono rappresentati da [0,1] mentre i gatti da [1,0] in notazione **one-hot**. La codifica one-hot trasforma le labels in vettori di 0 e 1. La lunghezza di tali vettori è determinata dal numero di classi o categorie che il modello si aspetta di classificare. Nel caso di cani e gatti il vettore codificato in one-hot ha 2 elementi. Se si decidesse di aggiungere un'altra classe, la dimensione del vettore diventerebbe di 3 elementi.

5.8 Creazione e allenamento di una CNN con Keras

In questo esempio dimostrativo si usa il modello **Sequential** di Keras.

```
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3),
           activation='relu', padding = 'same',
           input_shape=(224,224,3)),
    MaxPool2D(pool_size=(2, 2), strides=2),
    Conv2D(filters=64, kernel_size=(3, 3),
           activation='relu', padding = 'same'),
    MaxPool2D(pool_size=(2, 2), strides=2),
    Flatten(),
    Dense(units=2, activation='softmax')
])
```

Il primo layer del modello è convoluzionale 2D, ha 32 filtri, ognuno con un kernel di dimensione 3x3 ed utilizza la funzione di attivazione **relu.**
Da notare che la scelta del numero di filtri di output è arbitraria, e la dimensione 3x3 del kernel è molto comune.
Si abilita lo `zero-padding` specificando `padding='same'`.

Solo nel primo layer si specifica la `input_shape`, che corrisponde alla dimensione dei dati. Le immagini del dataset sono di dimensione `244 x 244` ed hanno tre canali di colore (RGB). Questo implica che il tensore di ingresso avrà la forma: (224,224,3).
Si aggiunge inoltre un layer di `max pooling` che ha lo scopo di ridurre la dimensione dei dati.
A questo segue un altro layer convoluzionale con le stesse specifiche del primo ma con 64 filtri. Generalmente in layer successivi si aumenta il numero di filtri. Segue un altro layer `max pooling`.
Alla fine si utilizza **`Flatten`** per "appiattire" l'output del convolutional layer e lo si passa ad un Dense layer. Questo è l'output layer della rete e quindi ha due nodi, uno per il gatto e uno per il cane. Come detto nell'esempio precedente, avendo più di un output, si sceglie la funzione di attivazione `softmax` in modo da ottenere una distribuzione di probabilità.
Ora che il modello è creato lo si compila specificando gli stessi parametri dell'esempio precedente.

```
model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy',
metrics=['accuracy'])
```

5.8.1 Allenamento della CNN

Si utilizza la funzione `model.fit()` introdotta nell'esempio precedente a cui si passano i `DirectoryIterators` ottenuti precedentemente.

```
model.fit(x=train_batches,
    steps_per_epoch=len(train_batches),
    validation_data=valid_batches,
    validation_steps=len(valid_batches),
    epochs=10,
    verbose=2
)
```

Le uniche differenze sono **`steps_per_epoch`** che si ottengono dal numero dei `train_batches` e **`validation_steps`** che si ottiene dal numero dei `valid_batches`.
Eseguendo il codice si possono osservare gli output del modello per le 10 epochs.
E' importante notare che la performance del modello sul training set raggiunge un accuracy del 100% e una loss che si avvicina a 0, però, confrontando questi risultati alle metriche di

validazione si vede che il modello implementato soffre di **overfitting**, dato che l'accuracy in questo caso è inferiore al 70%.

5.9 Testing della CNN creata con Keras

In questa sezione si utilizza la CNN creata per valutare le sue prestazioni sul test set, nonostante le validation performance evidenziassero **overfitting**.

Si chiama la funzione `model.predict()` per ottenere le predizioni sui dati del test set:

```
predictions = model.predict(x=test_batches, steps=len(test_batches),
verbose=0)
```

Si passano come parametri i `test_batches` e si impostano gli **steps** alla lunghezza di `test_batches` ovvero al numero di batches da processare prima di concludere un epoch.

E' possibile stampare le predizioni arrotondate nello stesso modo del precedente esempio.

Otteniamo una visualizzazione delle prestazioni del modello con la **confusion matrix**:

```
cm = confusion_matrix(y_true=test_batches.classes,
y_pred=np.argmax(predictions, axis=-1))
```

Da notare che si accede alle true labels chiamando **`test_batches.classes`**. Si rendono le labels predette codificate in one-hot compatibili alle true labels usando `np.argmax(predictions, axes = -1)`.

```
cm_plot_labels = ['cat','dog']
plot_confusion_matrix(cm=cm, classes=cm_plot_labels, title='Confusion
Matrix')
```

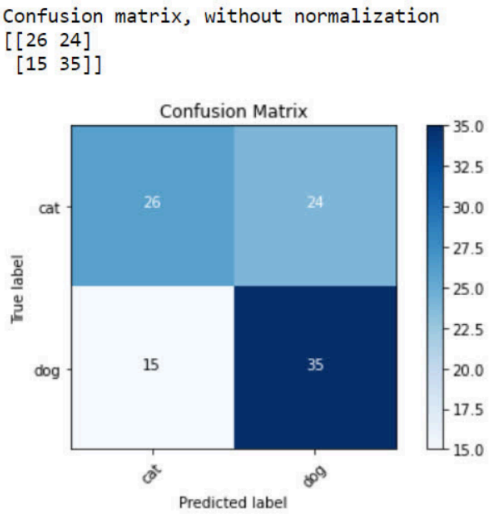

Figura 5.6: Training set batch

Possiamo vedere che il modello ha predetto correttamente che un'immagine è un gatto 26 volte correttamente ma ben 15 volte ha sbagliato la predizione.
Questi risultati rispecchiano ciò che ci si aspettava dalle validation metrics.

5.10 Costruzione di una rete neurale ottimizzata con Keras

In questa sezione si introduce un modello **pre-allenato** per classificare immagini di gatti e cani, chiamato VGG16, che è il vincitore della competizione ImageNet [15] del 2014.
Il modello VGG16 è allenato per classificare immagini appartenenti a 1000 categorie, tra cui cani e gatti, su cui il modello è già allenato. Lo scopo di questa sezione è quello di scaricare il modello e capire come modificarlo perchè sia ottimizzato a riconoscere solo immagini appartenenti alle 2 classi cane e gatto.
Dalla sezione 2.1 Architecture [16] del paper di VGG16 si capisce che l'unico preprocessing effettuato è la sottrazione del valore medio di RGB, calcolato sul training set, per ogni pixel.
Questo spiega il motivo del colore distorto che si vede quando si stampa un batch di dati.
Il primo passo per la costruzione del modello è quello di importare la rete VGG16 da Keras:

```
vgg16_model = tf.keras.applications.vgg16.VGG16()
```

I modelli importati da Keras vengono scaricati la prima volta e salvati nella macchina, in modo da consentire di accedervi nei successivi utilizzi.
Stampando il `summary()` del modello si nota immediatamente la sua complessità:

```
vgg16_model.summary()
```

```
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

E' importante notare che l'ultimo layer ha 1000 outputs, che corrispondono alle 1000 categorie della libreria ImageNet [15].

Dato che andremo a classificare solo due categorie, è necessario modificare il modello in modo che classifichi **solo** gatti e cani.
Prima di modificare il modello lo si converte a `Sequential()` replicando tutti i layer tranne l'ultimo (output).

```
model = Sequential()
for layer in vgg16_model.layers[:-1]:
    model.add(layer)
```

Si rendono poi tutti i layer **non allenabili**, dato che il modello è già stato allenato sulle due classi che ci interessa classificare.
Aggiungiamo un nuovo output layer, che consiste solo di 2 nodi corrispondenti a **cat** e **dog**. L'output sarà l'unico **layer allenabile** della rete.

```
model.add(Dense(units=2, activation='softmax'))
```

Chiamando la funzione `model.summary()` si può verificare che l'aggiornamento del modello è avvenuto correttamente.

5.11 Allenamento di una rete neurale ottimizzata con Keras

In modo simile a come è stato fatto per il modello creato da zero, per compilare la rete si utilizzano l'ottimizzatore Adam, loss di tipo `categorical_crossentropy` e `accuracy` come metrica:

```
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='categorical_crossentropy', metrics=['accuracy'])
```

Ora si effettua l'allenamento , con l'unica differenza che in questo caso si eseguono 5 epochs:

```
model.fit(x=train_batches,
          steps_per_epoch=len(train_batches),
          validation_data=valid_batches,
          validation_steps=len(valid_batches),
          epochs=5,
          verbose=2
)
```

Dagli output del training si vede che dopo solo 5 epochs i risultati sono sorprendenti, specialmente se comparati con quello che si ottiene con il modello precedente.

La accuracy parte dall' 88% e raggiunge il 99% al termine delle 5 epochs. In modo simile anche l'accuracy della validation passa da 95% a 99%.
Questo risultato è dovuto al fatto che la rete è ottimizzata per il riconoscimento di immagini, e i parametri migliorano velocemente dato che gli unici da allenare sono quelli relativi all output layer.
Notiamo inoltre che il modello generalizza molto bene e non si ha più overfitting.

5.12 Inferenza in una rete neurale ottimizzata con Keras

Per prima cosa si ottengono le predizioni del modello sul test set come già visto nei precedenti esempi:

```
predictions = model.predict(x=test_batches, steps=len(test_batches),
verbose=0)
```

Si crea la confusion matrix:

```
cm = confusion_matrix(y_true=test_batches.classes,
y_pred=np.argmax(predictions, axis=-1))
cm_plot_labels = ['cat','dog']
plot_confusion_matrix(cm=cm, classes=cm_plot_labels, title='Confusion
Matrix')
```

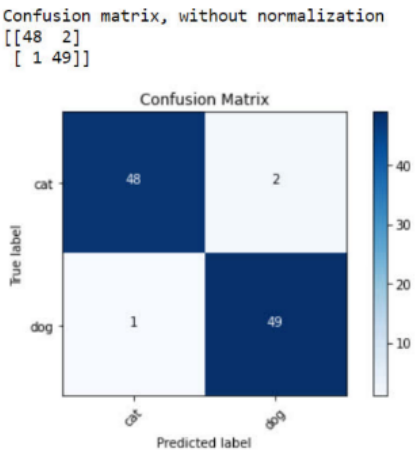

Figura 5.7: Confusion matrix

Si vede che il modello predice erroneamente solo 3 campioni su 100, ed ha una accuracy del 97%.

CAPITOLO 6 - TENSOR FLOW LITE

Tensor Flow Lite è un insieme di strumenti che rendono possibile l'implementazione di modelli di machine learning da eseguire su dispositivi mobili, embedded ed edge.

Figura 6.1: Tensor Flow Lite

6.1 Caratteristiche principali

Le principali caratteristiche di questo framework sono:

- Ottimizzazione per il machine learning on-device rispetto a 5 caratteristiche chiave:
  - Latenza: non c'è uno scambio di informazioni con un server
  - Privacy: nessun dato personale lascia il dispositivo
  - Connettività: non è richiesta la connessione a internet
  - Dimensioni: il modello ha dimensioni ridotte rispetto a quelle dei modelli Tensor Flow standard
  - Consumo energetico: l'inferenza è efficiente e non è necessaria la connessione ad internet
- Supportato da più piattaforme, tra cui Java, Swift, Python, Objective-C e C++.
- Prestazioni elevate, con accelerazione hardware e ottimizzazione dei modelli
- Presenta di esempi end to end per implementazioni comuni

6.2 Sviluppo di un modello

Un modello Tensor Flow Lite è rappresentato in un formato speciale, efficiente e portabile chiamato `FlatBuffers`.

Questo formato fornisce molti vantaggi rispetto al formato a `ProtocolBuffer` di Tensor Flow: dimensioni ridotte e inferenza più rapida consentono di ottenere ottime prestazioni eseguendo modelli Tensor Flow Lite in dispositivi con risorse di calcolo e memoria limitate.
Un modello Tensor FLow Lite può includere opzionalmente dei metadati.

6.2.1 Generazione di un modello Tensor Flow Lite

Ci sono tre modi per poter generare un modello Tensor Flow Lite:

1. **Usare un modello Tensor Flow Lite già esistente:** esistono molti modelli pre-allenati, nella documentazione sono disponibili guide end to end di come implementare alcuni casi notevoli in un applicazione Android.
2. **Creare un modello Tensor Flow Lite:** è disponibile una libreria all'interno dell'ecosistema chiamata **Tensor Flow Lite Model Maker**. Questa libreria è progettata per semplificare la creazione e la conversione di modelli di machine learning ottimizzati per dispositivi mobili e sistemi embedded. La libreria offre un'interfaccia di alto livello che consente agli utenti di addestrare, ottimizzare e convertire modelli con facilità, riducendo il bisogno di scrivere codice complesso.

6.2.2 Eseguire inferenza

Con inferenza si intende il processo di eseguire un modello Tensor Flow Lite in un dispositivo per fare predizioni basate su dati di input. L'inferenza può essere effettuata nei due seguenti modi:

- **Modello senza metadati:** utilizzare l'API Tensor Flow Lite Interpreter.
- **Modello con metadati:** utilizzare le API pre configurate usando la Tensor Flow Lite Task Library o costruire pipeline di inferenza personalizzate con la Tensor Flow Lite Support Library.

6.3 Il formato FlatBuffers

FlatBuffers è un'efficiente libreria di serializzazione multipiattaforma per C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust e Swift. È stata originariamente creata da Google per lo sviluppo di giochi e altre applicazioni critiche dal punto di vista delle prestazioni ed è disponibile in modo open source sotto la licenza Apache v2.

6.3.1 Perché usare i FlatBuffers

Vengono elencati una serie di motivi per cui è conveniente l'utilizzo dei FlatBuffers:

- **Accesso ai dati serializzati senza parsing/unpacking**: ciò che differenzia FlatBuffers è che rappresenta i dati gerarchici in un buffer binario in modo tale da potervi accedere direttamente senza parsing/unpacking, pur continuando a supportare l'evoluzione della struttura dei dati (compatibilità avanti/indietro).
- **Efficienza e velocità della memoria**: l'unica memoria necessaria ad accedere ai dati è quella occupata dal buffer, non viene richiesta nessuna memoria aggiuntiva. Il tempo di accesso è vicino a quello per accedere ad una struct grezza, con solo un incremento extra per consentire l'evoluzione del formato e i campi opzionali.
- **Flessibilità**: i campi opzionali garantiscono un'ottima compatibilità in avanti e all'indietro (importante per i giochi). Oltre a questo è possibile decidere cosa scrivere e cosa no.
- **Quantità di codice ridotta**: viene generata una piccola quantità di codice e solo un header come dipendenza minima, molto semplice da integrare.
- **Fortemente tipizzato**: gli errori si verificano in fase di compilazione, non è necessario dover scrivere controlli error-prone a tempo di esecuzione.
- **Comodo da usare**: Il codice C ++ generato consente di avere metodi di accesso e di costruzione molto semplici.
- **Codice multipiattaforma senza dipendenze**: il codice C++ funziona con qualsiasi gcc/clang recente e VS2010. Viene fornito con file di compilazione per i test e gli esempi

6.3.2 Perché non usare i Protocol Buffers

I Protocol Buffers sono relativamente simili ai FlatBuffers, con la differenza principale che sta nel modo in cui questi vengono **memorizzati** e **deserializzati**.

I Protocol Buffers vengono serializzati in un formato binario compatto (come i FlatBuffers) ma in fase di ricezione, per essere interpretati, necessitano una conversione.

I FlatBuffers invece possono essere letti e manipolati direttamente dal formato binario senza necessità di deserializzazione poiché la struttura binaria è progettata in modo tale da essere **facilmente navigabile** senza bisogno di conversioni.

Il codice nei Protocol Buffer è di un ordine di grandezza più grande e questi non hanno un opzione di import/export di testo.

6.4 Creazione di un modello TensorFlowLite

La libreria TensorFlow Lite Model Maker semplifica il processo di allenamento di un modello usando un dataset custom. Utilizza il **transfer learning** per ridurre il quantitativo di training data necessario e per diminuire il tempo di allenamento.

La libreria supporta attualmente le seguenti attività di Machine Learning: classificazione di immagini, riconoscimento di oggetti, classificazione di testo, BERT risposta a domande, classificazione di audio, suggerimento, ricerca.

Se la funzionalità che si vuole implementare non è supportata, prima si implementa un modello tramite Tensor Flow e poi lo si **converte** in un modello Tensor Flow Lite.

Model Maker permette di allenare un modello in pochissime righe di codice, sono riportati di seguito i passi per creare un classificatore di immagini:

```
from tflite_model_maker import image_classifier
from tflite_model_maker.image_classifier import DataLoader

# Load input data specific to an on-device ML app.
data = DataLoader.from_folder('flower_photos/')
train_data, test_data = data.split(0.9)

# Customize the TensorFlow model.
model = image_classifier.create(train_data)

# Evaluate the model.
loss, accuracy = model.evaluate(test_data)

# Export to Tensorflow Lite model and label file in `export_dir`.
model.export(export_dir='/tmp/')
```

Ci sono due modi per installare Tensor Flow Lite Model Maker:

1. Il primo è di installare un pacchetto **`pip`**:
    a. pip install tflite-model-maker
    b. pip install tflite-model-maker-nightly
2. Il secondo modo consiste nel clonare la repository GitHub di TensorFlow.

La versione "nightly" indica una build che viene rilasciata ogni notte, contenente gli aggiornamenti più recenti. Queste versioni sono utili per ottenere le ultime funzionalità e correzioni di bug, ma potrebbero essere meno stabili rispetto alle versioni ufficiali.

6.5 Conversione di un modello Tensor Flow

La conversione dei modelli Tensor Flow in formato Lite può variare a seconda del contenuto. Come primo passo di questo processo, è necessario valutare il modello per determinare se può essere convertito **direttamente**. Questa valutazione determina se il contenuto del modello è supportato dagli ambienti di runtime standard di Tensor Flow Lite, in base alle **operazioni utilizzate**. Se il modello utilizza operazioni al di fuori dell'insieme supportato, si può scegliere di modificarlo o di utilizzare tecniche di conversione avanzate.
Ad alto livello i passi di conversione sono i seguenti:

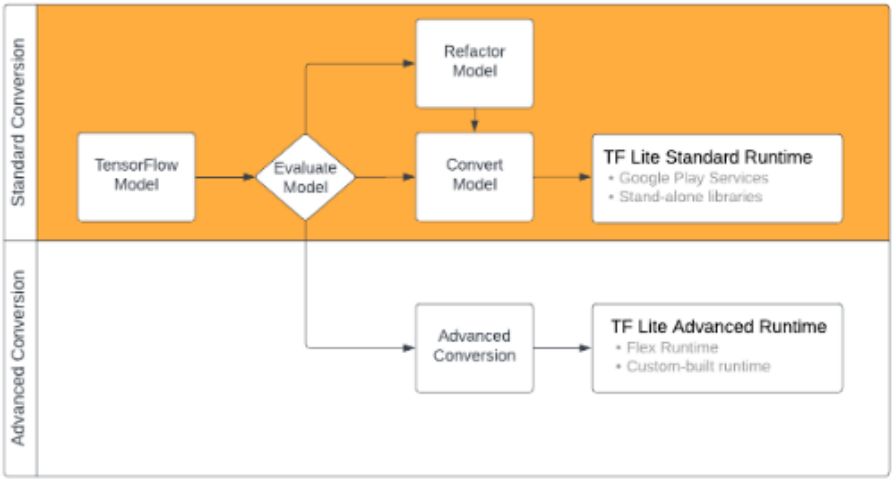

Figura 6.2: Conversione di un modello TensorFlow

Il convertitore può essere usato con i seguenti formati di input:

- `SavedModel`: un modello Tensor Flow salvato sotto forma di un insieme di files sul disco
- `Modello Keras`: un modello creato usando la API di alto livello Keras
- `Formato Keras H5`: un'alternativa più leggera al formato SavedModel supportata dalla API Kearas
- Modello creato da `concrete functions`: un modello creato utilizzando l'API TensorFlow di basso livello

Una volta importato il modello è necessario valutare se il suo contenuto è compatibile con il formato Tensor Flow Lite. È inoltre necessario determinare se il modello è adatto all'uso su dispositivi mobili ed edge rispetto alle dimensioni dei dati utilizzati, ai requisiti di elaborazione hardware e alle dimensioni e complessità complessive del modello.
Per la maggior parte dei modelli il convertitore dovrebbe funzionare subito. Tuttavia la libreria degli operatori integrati in Tensor Flow Lite supporta un sottoinsieme degli operatori

principali di Tensor Flow, ciò significa che alcuni modelli potrebbero richiedere ulteriori passaggi prima di essere convertiti. Oltre a questo alcune operazioni supportate da Tensor Flow Lite hanno requisiti di utilizzo limitati per motivi di prestazioni. E' possibile consultare la guida degli operatori per determinare se il proprio modello necessita delle modifiche per la conversione.

6.5.1 Conversione

Il convertitore Tensor Flow Lite prende in ingresso un modello Tensor Flow e genera un modello Tensor Flow Lite in formato **FlatBuffer** identificato dall'estensione del file **.tflite**.
È possibile caricare un modello salvato o convertire direttamente un modello creato nel codice.
Il convertitore accetta 3 flag principali (opzioni) che consentono di personalizzare la conversione:

1. **Flag di compatibilità:** consentono di specificare se la conversione deve consentire operatori personalizzati.
2. **Flag di ottimizzazione:** consentono di specificare il tipo di ottimizzazione da applicare durante la conversione. La tecnica di ottimizzazione più comunemente usata è la quantizzazione post-training.

   La **quantizzazione post-training** è una tecnica di ottimizzazione che migliora le prestazioni e riduce la dimensione del modello Tensor Flow Lite. Questa tecnica converte i pesi e, in alcuni casi, le attivazioni del modello da una rappresentazione a 32 bit a una rappresentazione a 8 bit. La perdita di precisione nella maggior parte delle applicazioni è trascurabile.
3. **Flag dei metadati:** consentono di aggiungere metadati al modello convertito, facilitando la creazione di codice wrapper specifico per la piattaforma quando si distribuiscono i modelli sui dispositivi.

È possibile convertire il modello utilizzando l'API Python o tramite riga di comando.
Nel caso in cui insorgano degli errori durante la conversione del modello è molto probabile che ci siano degli operatori che hanno problemi di compatibilità. Per risolvere questo problema è consigliato modificare il modello o usare opzioni di conversione avanzata che permettono di creare un formato Tensor Flow Lite custom e un ambiente runtime adatto a quel modello.

6.5 Metadati

I metadati forniti da Tensor Flow rappresentano uno standard per le descrizioni dei modelli, sono una fonte importante di informazioni su ciò che il modello fa e sui formati di input e output. I metadati comprendono:

1. **Parti leggibili dall'uomo** che indicano le best practice da rispettare quando si usa il modello.
2. **Parti leggibili dalla macchina** che possono essere utilizzate dai **generatori di codice** (per esempio in Android).

6.5.1 Formato di un modello con metadati

I metadati del modello sono definiti nel file FlatBuffer metadata_schema.fbs. Come si vede nella *Figura 6.5*, sono memorizzati nel campo metadati dello schema del modello sotto il nome di "TFLITE_METADATA".

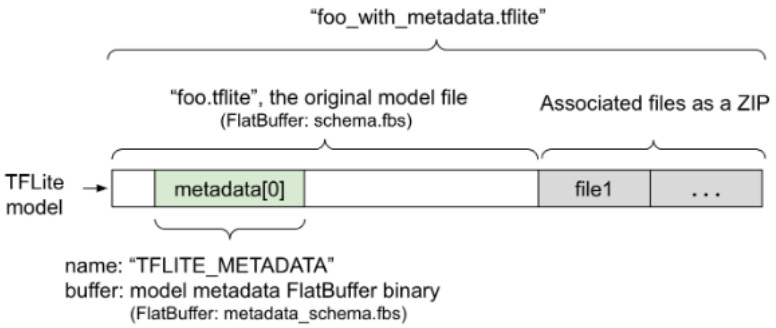

Figura 6.5:Modello TFLite con associati metadati

È Possibile aggiungere dei dati al modello, come file di labels e di classificazione. Questi file vengono concatenati alla fine del file come stotto forma di ZIP. L'interprete TFLite riesce ad utilizzare il nuovo formato del file allo stesso modo di prima.
La libreria Python necessaria per aggiungere metadati è ottenibile eseguendo il comando:

```
pip install tflite-support
```

6.5.2 Tipi di I/O supportati, file associati, normalizzazione

Per quanto riguarda i metadati è indifferente cosa faccia il modello, purché i tipi di input e output siano composti dai seguenti o da una combinazione dei seguenti tipi, il modello è supportato dai metadati di TensorFlow Lite:

- Feature: numeri interi o float32 senza segno.
- Immagine: immagini RGB e in scala di grigi.
- Bounding box: rettangoli delimitatori.

I modelli Tensor Flow Lite possono essere forniti con diversi file associati, senza questi (se ce ne sono), un modello non funzionerà correttamente.
I file associati possono ora essere raggruppati con il modello tramite la libreria metadata di Tensor Flow Lite. Il nuovo modello Tensor Flow Lite diventa un file zip che contiene sia il modello che i file associati. Questo nuovo formato di modello continua a utilizzare la stessa estensione .tflite, ed è compatibile con il framework e l'interprete esistenti.
Memorizzando le informazioni sui file associati si permette al Tensor Flow Lite Android code generator di poter applicare automaticamente il pre e post processing agli oggetti.

6.6 Android Studio ML Model Binding

I metadati di Tensor Flow Lite rendono disponibile la generazione di codice wrapper per consentire l'integrazione di modelli in Android. E' disponibile un'interfaccia grafica che è di facile utilizzo, nel caso sia necessaria maggiore personalizzazione è possibile agire tramite riga di comando.
Se si possiede una versione Android Studio 4.1 o superiore è possibile usare Android Studio ML Model Binding per configurare automaticamente le impostazioni del progetto e generare classi wrapper basate sui metadati del modello. Il codice wrapper elimina la necessità di interagire con ByteBuffer e permette invece di sfruttare gli oggetti tipizzati **Bitmap** e **Rect.**

6.6.1 Importazione di un modello TFLite

Il modello può essere importato facendo click con il tasto destro sul modulo desiderato, oppure cliccando su File> Nuovo > Altro > Modello Tensor Flow Lite.

Una volta arrivati a questo punto si seleziona il modello TFLite, Android Studio inserirà automaticamente le dipendenze necessarie nel file .gradle del modulo.

E' possibile selezionare l'importazione di Tensor Flow GPU, prima di cliccare Fine.

CAPITOLO 7 - LA LIBRERIA PYTORCH DI META

7.1 Origini di PyTorch

figura 7.1: Logo PyTorch

PyTorch nasce principalmente dall'esigenza di sveltire il processo che va dalla fase di progettazione a quella di sperimentazione di moduli di machine learning.
È scritto sulla base di Torch, una libreria scritta in Lua su uno scheletro scritto in C, dalla quale recupera le strutture fondamentali, come i package torch e torch.nn, aggiungendogli un'interfaccia in Python (È disponibile anche un front end in C++, seppur meno usato e aggiornato).
Il motivo principale dell'evoluzione da Torch a PyTorch sta nelle ottimizzazioni e nell'efficienza di alcuni calcoli, ad esempio quello dei gradienti tramite torch.autograd.
Inoltre, vivendo nell'ecosistema di Python, PyTorch si integra con librerie esistenti come scikit-learn, NumPy, SciPy, rendendo lo sviluppo di moduli di machine learning più semplice e diretto. [18]

Lo sviluppo di PyTorch è iniziato nel 2016 da parte di Meta, come progetto di tirocinio di un ricercatore dell'azienda, Adam Paszke, sotto la supervisione di uno degli sviluppatori di Torch, Soumith Chintala [19]. La prima beta è stata rilasciata nel gennaio 2017, un anno dopo è stato reso pubblico PyTorch 1.0. Qualche anno dopo, nel 2022 PyTorch cambia nome in PyTorch Foundation, rientrando nel panorama della Linux Foundation e rendendosi disponibile sotto licenza BSD modificata.

Il board del progetto è costituito dalle maggiori aziende attive nell'ambito dell'intelligenza artificiale (e della tecnologia in generale), come AMD, AWS, Google Cloud, Meta, Microsoft Azure, e NVIDIA.

7.2 Caratteristiche di alto livello

PyTorch si può considerare molto simile a Tensor Flow, poiché come esso permette di gestire problemi di regressione, classificazione e di costruire reti neurali, nonché di fare eseguire i calcoli sia su CPU, che su architetture che supportano la tecnologia CUDA.
Permette di creare reti neurali tramite l'astrazione fornita della classe Module, dal concetto di layer e di Tensor, che non è altro che un array n-dimensionale.
Inoltre fornisce una vasta gamma di funzioni di attivazione per i nodi della rete, nonché di funzioni di loss e ottimizzatori.
Per il calcolo dei gradienti, PyTorch offre torch.autograd, che implementa la backpropagation tramite gradient tape e grafi di computazione.
È inoltre disponibile la funzionalità di scripting dei modelli creati, che permette di creare del codice in C++ eseguibile su altri ambienti non-Python e ottimizzabile dal compilatore del sistema ospite. Ciò è ottenuto tramite l'uso di TorchScript, un sottoinsieme del linguaggio Python da cui è mappabile tramite i metodi script e trace del pacchetto torch.jit.

In generale l'architettura di PyTorch è in continua definizione e cambia con un ritmo abbastanza sostenuto, con pacchetti e funzionalità che si impongono come vincenti e facilmente usufruibili (che quindi vengono mantenuti e aggiornati), e altri pacchetti che nella durante i mesi vengono deprecati. Il team di PyTorch considera molto rilevante il feedback dei programmatori e di chi usa questo framework dato ad esempio sui forum come https://discuss.pytorch.org/ , e su tale feedback si basa per eventuali future modifiche alle funzionalità.

7.3 Pacchetti fondamentali

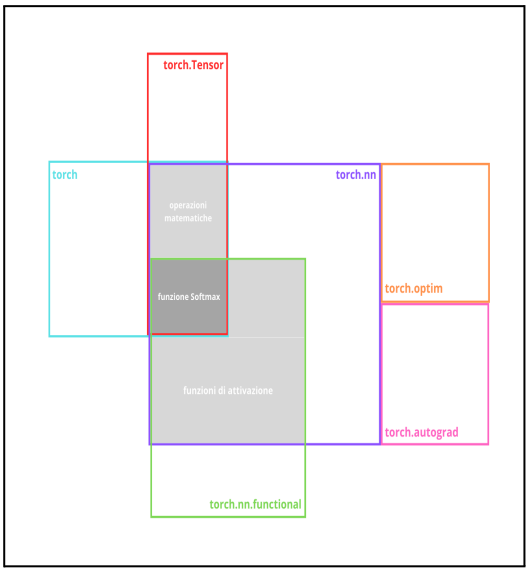

Figura 7.2: pacchetti fondamentali

In *figura 7.2* si possono vedere i pacchetti fondamentali di PyTorch e come interagiscono tra di loro. Ora i principali verranno analizzati uno alla volta:

7.3.1 Pacchetto torch

Il pacchetto torch contiene metodi per costruire oggetti di tipo Tensor (in particolare sono presenti metodi factory per ottenere tale risultato), ma non la classe Tensor. Inoltre sono presenti dei metodi di supporto per:

- Controllare lo stato dei Tensor, come ad esempio:
  - `torch.is_tensor(obj)`, che ritorna True se obj è di tipo Tensor;
  - `torch.get_default_type/set_default_type(d)`, per farsi ritornare/cambiare il valore di default dell'attributo dtype di un membro della classe Tensor;
  - `get/set_default_device`, per farsi ritornare/cambiare il valore di default dell'attributo device nella costruzione di oggetti Tensor;
  - `torch.numel(obj)`, che ritorna il numero di elementi dentro un oggetto Tensor.
- Creare Tensor, come:
  - `torch.tensor(data)`, che costruisce un oggetto Tensor copiando i dati dentro la variabile data passata come argomento. Il tipo di dato è dedotto

automaticamente. Inoltre per default tale Tensor non ha memoria autograd (verrà visto in seguito come questo parametro serva per il calcolo efficiente del gradiente della funzione di loss);

- `torch.as_tensor(data)`, che preserva la memoria autograd e dove possibile evita copie;
- `torch.from_numpy(ndarray)`, che crea un tensore che condivide la memoria con un NumPy array;
- `torch.zeros(size)`, `torch.zeros(input)`, dove il primo metodo crea un Tensor della size specificata, riempito con zeri, mentre il secondo crea un Tensor riempito di zeri della stessa size del Tensor input. Tali metodi esistono anche nella variante `torch.ones(size)` e `torch.ones_like(input)`: il funzionamento è analogo ma il tensore viene riempito di uni.

- Creare Tensor **inizializzati con dati random**, ad esempio:
  - `torch.rand(dim1, dim2, …)`, che ritorna un Tensor della size specificata con i valori inizializzati a valori random da 0 a 1 secondo una distribuzione uniforme;
  - `torch.randn(dim1, dim2, …)`, che ritorna un Tensor della size specificata con i valori inizializzati a valori random da 0 a 1 secondo una distribuzione normale;
  - `torch.randint(low, high, dim1, dim2, …)`, che ritorna un Tensor della size specificata con i valori inizializzati a valori random da low a high secondo una distribuzione uniforme;
  - `torch.randperm(n)`, che ritorna un Tensor a una dimensione che ha come valori delle celle una permutazione casuale degli interi da 0 a n-1.
- Creare altre varianti di Tensor, usando:
  - `torch.eye(dim)`, che ritorna un Tensor a due assi, di rango pari a dim, con tutti gli elementi a zero tranne quelli sulla diagonale, che sono inizializzati a 1 (matrice identità).
  - `torch.empty(dim1, dim2, …)`, che restituisce un Tensor con valori non inizializzati con le dimensioni specificate dagli interi passati come argomenti;
  - `torch.range(min, max, step)`, che restituisce un Tensor 1-D con una distribuzione spaziata uniformemente nell'intervallo (min, max), con step come intervallo tra un valore e quello successivo, come elementi;

- `torch.linspace(min, max, steps)`, che restituisce un Tensor 1-D con un numero di elementi pari a steps, i cui valori sono quelli di una distribuzione spaziata uniformemente nell'intervallo (min, max);
- `torch.logspace(min, max, steps, base)`, come sopra, ma l'intervallo è ($base^{min}$, $base^{max}$).

- Salvare e caricare oggetti su file con formato .pt, con:
  - `torch.save(obj, path)`
  - `torch.load(path)`

  Tali metodi esistono anche in torch.jit per salvare e caricare file TorchScript con formato .pt o .ptl (file che verranno eseguiti sull'interprete di dispositivi mobili).
- Compiere operazioni matematiche sui Tensori:

  Operazioni su ciascuna cella
  - `torch.abs(obj)`, che calcola il valore assoluto di ciascun elemento del Tensor obj;
  - `torch.add(input, other)`, che calcola la somma tra il Tensor input e other, che può essere un Tensor o uno scalare. Lo stesso vale per `torch.mul(input, other)`, `torch.sub(input, other)`, `torch.div(input, other)` con la moltiplicazione, la sottrazione e la divisione;
  - funzioni trigonometriche: `torch.sin()`, `torch.sinh()`, `torch.asin()`, con le relative versioni per il coseno, tangente, eccetera;
  - altre funzioni: `torch.log(input)`, `torch.sqrt(input)`, `torch.reciprocal(input)`, `torch.sigmoid(input)`, che applicano il logaritmo, la radice quadrata, la funzione reciproco e la funzione sigmoidea a tutti gli elementi di input;
  - `torch.ceil(obj)/torch.floor(obj)/torch.clamp(obj, min, max)`, `torch.trunc(obj)`: i primi due calcolano il ceiling e il floor di ciascun elemento, il terzo metodo "chiude" i valori del Tensor obj all'interno dell'intervallo delimitato da min e max, trasformando i valori che non fanno parte di questo intervallo nell'intero più vicino tra min e max. torch.trunc(obj) restituisce un Tensor che ha come valori tutti i valori di obj, ma troncati.

  Operazioni di riduzione: ritornano un unico scalare/un Tensor ridotto a partire da un Tensor

- `torch.mean(input)/torch.median(input)/torch.std(input)/torch.var(input)`, che restituisce la media (mediana/deviazione standard/varianza) degli elementi di input;
- `torch.max(input)/torch.min(input)`, che restituisce l'elemento maggiore (minore) di input;
- `torch.argmax(input)/torch.argmin(input)`, che ritornano l'indice dell'elemento maggiore (minore);
- `torch.sum(input)/torch.prod(input)`, che restituisce la somma (prodotto) di tutti gli elementi di input;
- `torch.unique(input, sorted)`: ritorna il Tensore di partenza, dopo aver cancellato gli elementi "doppi". È possibile ordinare tali elementi in ordine crescente tramite l'uso del boolean sorted presente tra i parametri, il quale è di default impostato a True.

Operazioni di comparazione

- `torch.eq(input, other)`, ritorna un tensor con dtype = boolean in base al confronto degli elementi di input con i rispettivi elementi di other, il quale può essere un Tensor delle stesse dimensioni di input, oppure uno scalare;
- `torch.ge(input, other)/torch.gt(input, other)`, che verifica, elemento per elemento, se il valore in input è maggiore (maggiore uguale nella versione torch.ge()) del valore in output, che può essere un Tensor delle stesse dimensioni di input, oppure uno scalare;
- `torch.le(input, other)/torch.lt(input, other)`, come sopra ma con minore (e minore uguale);
- `torch.maximum(input, other)/torch.minimum(input, other)`, che inserisce in un nuovo Tensor con le stesse dimensioni di input il valore massimo (minimo per torch.minimum()) tra quello in input e il corrispettivo in other, che può essere un Tensor delle stesse dimensioni di input, oppure uno scalare;
- `torch.isin(elements, test_elements)`, che ritorna un Tensor di boolean delle stesse dimensioni di input, che riporta True se il valore in elements compare in test_elements. Sia elements che test_elements possono essere Tensor o scalari, ma non entrambi.

Altre operazioni

- `torch.flatten(input, start_dim)`, che restituisce un Tensor 1-D con tutti i valori di input posizionati nella stessa dimensione. Tramite l'argomento start_dim si può specificare la dimensione da cui iniziare tale operazione. In tal caso le dimensioni minori di start_dim rimangono strutturate come sono;
- `torch.roll(input, shift, dim)`, che esegue uno shifting degli elementi di shift, lungo la direzione specificata da dim. Se dim = None (come di default), al Tensor viene prima applicato torch.flatten(), poi lo shift, e poi fatto ritornare alla dimensione originale. Shift può essere anche una tupla, e dim deve essere necessariamente della stessa size;
- `torch.flip(input, dims)`, che restituisce il Tensor di partenza, dopo aver invertito gli elementi lungo le dimensioni specificate da dims (che può essere un int o una tupla).

- Indexing, Slicing, Joining, Mutating Ops:
  - `torch.cat(inputs, dim)`, che concatena i Tensor specificati nella sequenza di tensori in inputs lungo la dimensione dim. È necessario che i vari Tensor abbiano la stessa shape, a parte per quella su cui si concatena;
  - `torch.split(input, split_size_or_sections, dim = 0)`, che ritorna una lista di Tensor, ottenuti dividendo il tensore input in tensori di grandezza split_size_or sections. La grandezza è considerata lungo la dimensione dim, che di default è uguale a 0;
  - `torch.reshape(input, (dim1, dim2, …))`, che ritorna un tensore con gli stessi dati di input, ma organizzati in una shape diversa. Se le dimensioni sono compatibili, non vengono copiati i dati ma viene creata una view. Se invece la shape specificata non è compatibile, viene creato un tensore con dati copiati.

Vediamo ora degli esempi per chiarire il codice spiegato sopra. Partiamo con delle soluzioni su come creare tensori:

1) Tensore identità del rango specificato

```
torch.eye(3)
```

```
>> tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
```

```
        [ 0.,  0.,  1.]])
```

2) Tensore di zeri di dimensione 2*3

```
torch.zeros(2, 3)
```

```
>> tensor([[ 0.,  0.,  0.],
          [ 0.,  0.,  0.]])
```

3) Tensore di 1 unidimensionale grande 5

```
torch.ones(5)
```

```
>> tensor([ 1.,  1.,  1.,  1.,  1.])
```

4) Tensore riempito con numeri random nell'intervallo [0,1) delle dimensioni specificate

```
torch.rand(2, 3)
```

```
>> tensor([[ 0.8237,  0.5781,  0.6879],
          [ 0.3816,  0.7249,  0.0998]])
```

5) Tensore di una dimensione riempito con numeri dal primo al secondo parametro e con il passo specificato dal terzo

```
torch.range(1, 4, 0.5)
```

```
>> tensor([1.0000,  1.5000,  2.0000,  2.5000,  3.0000,  3.5000,  4.0000])
```

6) Tensore unidimensionale riempito con numeri dal primo al secondo parametro e con il numero di elementi specificato dal terzo parametro

```
torch.linspace(3, 10, steps=5)
```

```
>> tensor([3.0000,   4.7500,   6.5000,   8.2500,  10.0000])
```

Passiamo ora ad analizzare degli esempi riguardo ad operazioni sulle dimensioni. Prima di tutto definiamo i seguenti tensori:

```
t1 = torch.tensor([
 [1,2],
 [3,4]
])
t2 = torch.tensor([
 [5,6],
 [7,8]
])
```

1) Concatenazione dei due tensori …

```
torch.cat((t1, t2), dim=0)
```

```
>> tensor([[1,2],
           [3,4],
           [5,6],
           [7,8]])
```

2) Concatenazione dei due tensori ma con …

```
torch.cat((t1, t2), dim=1)
```

```
>> tensor([[1, 2, 5, 6],
           [3, 4, 7, 8]])
```

3) Ora definiamo un nuovo tensore di dimensioni 2*3 con numeri random estratti secondo una distribuzione normale con media 0 e varianza 1. Successivamente facciamo la trasposta di tale vettore

```
x = torch.randn(2, 3)
torch.transpose(x, 0, 1)
```

```
>> tensor([[ 1.0028, -0.9893,  0.5809],
           [-0.1669,  0.7299,  0.4942]])

>> tensor([[ 1.0028, -0.1669],
           [-0.9893,  0.7299],
```

```
            [ 0.5809,  0.4942]])
```

4) Tramite la funzione stack si uniscono più tensori, ma impilandoli in una nuova dimensione

```
t1 = torch.tensor([
 [1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 1]
])
t2 = torch.tensor([
 [2, 2, 2, 2],
 [2, 2, 2, 2],
 [2, 2, 2, 2],
 [2, 2, 2, 2]
])
t3 = torch.tensor([
 [3, 3, 3, 3],
 [3, 3, 3, 3],
 [3, 3, 3, 3],
 [3, 3, 3, 3]
])

t = torch.stack((t1, t2, t3))
t.shape
torch.Size([3, 4, 4])

torch.flatten(t)
t = torch.tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3])

torch.flatten(t, 1)
t = torch.tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3]])
```

5) Definiamo il seguente tensore per poi fare la somma prima di ogni colonna (dim=0) e poi di ogni riga (dim=1)

```
t = torch.tensor([
    [1,0,0,2],
    [0,3,3,0],
```

```
    [4,0,0,5]
], dtype=torch.float32)

t.sum(dim=0)

t.sum(dim=1)
```

```
>> tensor([5., 3., 3., 7.])

>> tensor([ 3.,  6., 9.])
```

7.3.1.1 Classe torch.Tensor

La classe Tensor (facente parte di torch.Tensor) rappresenta un tensore, cioè una array n-dimensionale (rappresenta cioè oggetti che vanno dagli scalari a matrici pluridimensionali). Nell'ambiente di una rete neurale un Tensor raffigura un nodo, il quale è associato ad un Layer e quindi ad un Module in fase di costruzione di quest'ultimo.

Gli attributi di un oggetto Tensor sono:

1. torch.dtype
2. torch.device
3. torch.layout
4. torch.requires_grad
5. tensor.grad

Per quanto riguarda **dtype**, esso può essere del tipo indicato in *figura 7.3*:

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	torch.float32	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.float64	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.float16	torch.HalfTensor	torch.cuda.HalfTensor
8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16-bit integer (signed)	torch.int16	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.int32	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.int64	torch.LongTensor	torch.cuda.LongTensor

figura 7.3: torch.dtype

Per quanto riguarda l'attributo **device**, esso indica il luogo fisico dell'architettura dove i calcoli su un dato tensore verranno eseguiti. Inoltre è importante notare come le operazioni tra più Tensor debbano avvenire sullo stesso dispositivo fisico. In PyTorch è possibile specificare i seguenti tipi di device/interfacce per i calcoli:

1. CPU
2. GPU che supportano CUDA di NVIDIA
3. GPU che supportano XPU di Intel
4. IPU
5. MPS, XLA, META

L'attributo **layout** indica invece com'è salvato in memoria un Tensor. Il valore di default è "strided", cioè salvato tramite una struttura di supporto che registra i "salti" da applicare in memoria per spostarsi da un elemento ad un altro in una delle n dimensioni del Tensore.

L'attributo **requires_grad**, di tipo boolean, indica se il tensore debba essere incluso o meno nel calcolo del gradiente nel momento in cui si applica la backpropagation (che sfrutta la differenziazione automatica, implementata in PyTorch nel package `torch.autograd` tramite il gradient tape) per calcolare il gradiente della loss function (implementata in `torch.nn.functional`).

L'attributo **grad** è impostato a None al momento della creazione del Tensor, ma diventa un Tensor esso stesso a partire dalla prima volta in cui viene invocata la funzione `torch.Tensor.backward()`, la quale popola questo attributo con il gradiente della loss function in quel nodo (Tensor) e lo aggiorna man mano che essa viene invocata.

L'importanza dei primi due attributi di un tensore è che tutte le operazioni tra tensori devono essere eseguite tra tensori con stesso dtype e che risiedono sullo stesso device.
È possibile inoltre modificare i primi tre attributi tramite il metodo `Tensor.to()`. Vediamo degli esempi su come fare:

```
# Di default si ha dtype=float32, device=cpu, modifichiamo questi
# parametri tramite il metodo tensor.to
tensor = torch.randn(2, 2) #creazione di un tensore 2*2 con valori random
tensor.to(torch.float64)
```

```
>> tensor([[-0.5044,  0.0005],
           [ 0.3310, -0.0584]], dtype=torch.float64)

cuda0 = torch.device('cuda:0')
tensor.to(cuda0)
>> tensor([[-0.5044,  0.0005],
           [ 0.3310, -0.0584]], device='cuda:0')

tensor.to(cuda0, dtype=torch.float64)
>> tensor([[-0.5044,  0.0005],
           [ 0.3310, -0.0584]], dtype=torch.float64, device='cuda:0')
```

La maggior parte dei metodi di torch.Tensor sono degli alias di quelli già presentati per il pacchetto torch. Alcuni metodi invece sono esclusivi della classe torch.Tensor, ad esempio:

1. `torch.Tensor.view()`, per referenziare una parte di memoria occupata da un Tensor, ma con una struttura dati con shape diversa. Le shape del Tensor originale e della view devono essere compatibili;
2. `torch.Tensor.item()`, applicabile quando il tensore ha un solo elemento. Ritorna tale elemento;
3. `torch.Tensor.tolist()` e `torch.Tensor.numpy()`, che ritornano i valori all'interno di un tensore sotto forma di lista o numpy array;
4. le versioni in-place di alcune funzioni di torch, caratterizzate dal suffisso "_", le quali permettono un risparmio di memoria;
5. `torch.Tensor.size()/torch.Tensor.shape()`, che ritorna le dimensioni lungo i vari assi del tensore.

```
t = torch.tensor([
    [1,2,3],
    [4,5,6],
    [7,8,9]], dtype=torch.float32)

t.mean() #stampa la media
```

```
>> tensor(5.)
```

```
t.mean().item()
```

```
>> 5.0
```

```
t.mean(dim=0).tolist() #stampa la media lungo le colonne
```

```
>> [4.0, 5.0, 6.0]
```

```
t.mean(dim=0).numpy() #stampa la media lungo le colonne (Numpy array)
```

```
>> array([4., 5., 6.], dtype=float32)
```

Poiché sono presenti metodi che si sovrappongono per la creazione di tensori e per applicare loro varie operazioni, di seguito si propone un piccolo riassunto delle differenze tra i diversi modi di costruire un elemento della classe Tensor.

```
data = np.array([1, 2, 3])
type(data)
```

```
>> numpy.ndarray
```

```
o1 = torch.Tensor(data)
o2 = torch.tensor(data)
o3 = torch.as_tensor(data)
o4 = torch.from_numpy(data)
print(o1)
print(o2)
print(o3)
print(o4)
```

```
>> tensor([1., 2., 3.]) #o1
>> tensor([1, 2, 3], dtype = torch.int32) #o2
>> tensor([1, 2, 3], dtype = torch.int32) #o3
>> tensor([1, 2, 3], dtype = torch.int32) #o4
```

In questi esempi il primo metodo invoca il costruttore della classe Tensor, il secondo metodo è in realtà una factory per gli oggetti di tipo Tensor.

Il costruttore della classe Tensor usa come dtype i float32, che sono il tipo di default, mentre gli altri tre metodi per la creazione di Tensor hanno dedotto il tipo dei dati a partire dal numpy.array di interi che gli è stato passato.

Un'altra differenza è che i primi due modi di inizializzazione di un tensore compiono una copia dei dati su cui sono stati creati, mentre le funzioni `from_numpy()` e `as_tensor()` si riferiscono ai dati originari, per cui se i dati originari vengono cambiati, i tensori creati con le funzioni `from_numpy()` e `as_tensor()` cambiano allo stesso modo (mentre i tensori creati con `torch.Tensor()` e `torch.tensor()` sono oggetti "indipendenti" in un certo senso). Come ulteriore esempio delle differenze di costruzione tra la classe Tensor e i metodi factory di torch, si presenta il seguente, dove vengono creati due tensori con dati non inizializzati (e quindi il loro valore è uguale a ciò che era scritto precedentemente in memoria). La differenza consiste nel fatto che con il primo metodo, il tipo dei dati è automaticamente impostato a float a 32 bit, mentre nel secondo caso sarebbe stato possibile deciderlo con il parametro dtype all'interno dei parametri della funzione. In questo caso specifico, anche `torch.empty()` imposta l'argomento dtype a float a 32 bit.

```
t1 = torch.Tensor() #Tensor vuoto e con dati non inizializzati
t2 = torch.empty() #stesso risultato di sopra
```

Ricapitolando, dovendo scegliere tra il costruttore della classe Tensor e il factory method, è meglio usare `torch.tensor()`, poiché deduce automaticamente il tipo di dati. Inoltre, se bisogna creare tensori di una size specifica shape, si può utilizzare `torch.*`, in una delle sue varianti. Se invece si desidera costruire un tensore con la stessa shape di un tensore esistente, si può ricorrere a `torch.*_like`, mentre se si cerca di creare un tensore con il dtype simile ad un tensore esistente, ma di diversa dimensione, è disponibile la funzione `tensor.new_*`.

7.3.2 Pacchetto torch.nn

7.3.2.1 - Module

Un `Module` è la rappresentazione a grafo di una rete neurale. È una classe che fa parte del package `torch.nn` e nel momento in cui si crea il proprio modello di rete neurale, tale classe andrebbe estesa da una sottoclasse personalizzata in cui si riscrive:

1. Il costruttore, dove vengono specificati il numero di layers della rete e da quanti nodi ciascuno di questi è composto;
2. Il metodo `forward(self, params)`, cioè il codice che implementa il forward pass.

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()

    def forward(self, x, h):
        new_h = torch.tanh(x + h)
        return new_h, new_h
my_cell = MyCell()
x = torch.rand(3, 4)
h = torch.rand(3, 4)
print(my_cell(x, h))
```

```
>> (tensor([[0.8219, 0.8990, 0.6670, 0.8277],
            [0.5176, 0.4017, 0.8545, 0.7336],
            [0.6013, 0.6992, 0.2618, 0.6668]]),
            tensor([[0.8219, 0.8990, 0.6670, 0.8277],
            [0.5176, 0.4017, 0.8545, 0.7336],
            [0.6013, 0.6992, 0.2618, 0.6668]]))
```

Nell'esempio riportato sopra, è stata creata un'istanza della classe MyCell, chiamando il costruttore della superclasse Module, poi sono stati creati due tensori 3x4 con valori da una funzione random distribuita uniformemente da 0 a 1 e poi è stato applicato il forward pass a tali tensori, usando come sintassi il nome dell'oggetto di tipo MyCell creato, con i due tensori tra parentesi. In questa maniera viene invocato direttamente il metodo forward() della classe stessa.

Ciascun Module possiede la variabile **training**, un bool che rappresenta se il modulo è in modalità allenamento o valutazione. Questa flag è usata ad esempio quando si ottimizza un modello prima di usarlo su dispositivi mobile, perché condiziona il tipo di operazioni eseguite per l'ottimizzazione stessa. Tale argomento è modificabile tramite la funzione `torch.Module.eval()`, che imposta training a False oppure torch.Module.train(), che ottiene l'effetto opposto.

7.3.2.2 Sequential

`nn.Sequential` è una sottoclasse di Module che permette di salvare nel suo costruttore una sequenza ordinata di Modules (che saranno i Layer della rete neurale). Un elemento della classe Sequential può essere creato direttamente dal costruttore:

```
model = nn.Sequential(
          nn.Conv2d(1, 20, 5),
          nn.ReLU(),
          nn.Conv2d(20, 64, 5),
          nn.ReLU()
         )
```

oppure può essere creato per aggiunte successive di Layer tramite il metodo `nn.Sequential.append(layer)`:

```
model = nn.Sequential()
model.append(nn.Conv2d(1, 20, 5))
model.append(nn.ReLU())
model.append(nn.Conv2d(20, 64, 5))
model.append(nn.ReLU())
```

In questo caso si ha che quando un input viene passato a model, esso in realtà viene passato ad un layer alla volta e viene processato tramite le funzioni `forward()` di ciascun layer.

7.3.2.3 Layer

I layer sono sottoclassi di Module che implementano specifiche funzioni `forward()` al loro interno. Possono essere aggiunti ad un Module nel suo costruttore per impostare la topologia della rete neurale specificata. Inoltre permettono di specificare come parametri il numero di nodi in input e in output, nonché altre variabili specifiche per ciascun tipo di layer. Ciò permette in fase di costruzione della rete neurale di tenere sott'occhio in maniera condensata le caratteristiche strutturali e funzionali del modello.

I tipi di layer supportati da PyTorch sono:

1. **Layer convoluzionali**, che includono `nn.Conv1d, nn.Conv2d, nn.Conv3d`, il metodo `forward()` dei quali permette di eseguire una convoluzione 1d, 2d, 3d rispettivamente su uno (o più) input. I principali parametri per la loro costruzione sono:
    a. canali_ingresso → numero di input alla volta che il layer processa (se si tratta del primo layer e si processano immagini rgb, i canali di ingresso sono 3);
    b. canali_uscita → numero di output

c. dimensioni_kernel → una tupla che indica le dimensioni del kernel per performare la convoluzione;
d. stride → salto tra una convoluzione e un'altra. Di default a uno;
e. padding → spessore del padding per mantenere le dimensioni dell'immagine inalterate. Di default 0.

2. **Layer di Pooling**, come ad esempio `nn.MaxPool1d, nn.AvgPool1d` (dei quali esistono anche le versioni 2d e 3d), che, a partire da un input di una certa dimensione, selezionano un valore "rilevante" zona per zona, per ridurre le dimensioni degli elementi che attraversano la rete, e per provare a ridurre l'overfitting, tentando di riconoscere pattern negli input. I principali parametri per la loro costruzione sono:
   a. kernel_size → grandezza della finestra da cui viene eletto il valore "rilevante". Può essere un rettangolo o anche un quadrato;
   b. stride → come sopra;
   c. padding → come sopra.

In questa categoria cade anche `nn.MaxUnpool1d` (e le relative versioni a più dimensioni), che prova a ristabilire le dimensioni originali di un elemento precedentemente sottoposto a MaxPool, posizionando "chiazze" di zeri (grandi quanto il kernel con cui è stato eseguito il pooling) attorno al valore di partenza.

3. **Layer di padding**, che comprendono `nn.ReflectionPad1d, nn.ReplicationPad1d, nn.ZeroPad1d` (e le loro versioni a più dimensioni), applicano un padding al vettore passato. Il parametro per la costruzione di questo layer è:
   a. padding → un int o una tupla, che descrive lo spessore del padding. Può essere anche specificato con 'valid' (nessun padding) oppure 'same' (mantiene le dimensioni dell'input).
4. **Layer di normalizzazione**, che normalizzano gli output del layer precedente (portandoli ad avere media 0 e deviazione standard 1) in modo da evitare il problema del vanishing e exploding gradient, permettendo una convergenza più veloce. Alcuni tra i metodi implementati sono `nn.BatchNorm1d, nn.InstanceNorm1d, nn.LayerNorm`. I primi due metodi comprendono anche le versioni in 2d e 3d, così come una versione con lazy initialization.
5. **Layer ricorrenti**
6. **Layer transformer**

7. **Layer lineari**, dei quali il più usato è `nn.Linear()`. La funzione di forward di tale layer applica una funzione lineare ai tensori di input. Possiede due attributi, weight e bias, che sono i parametri allenabili dalla rete tramite gradient descent. I parametri in fase di costruzione sono:
    a. canali_ingresso;
    b. canali_uscita;
    c. bias → se impostato a False, il layer non allena l'attributo bias.
8. **Layer di dropout**, i quali rendono casualmente nulli alcuni degli elementi (oppure canali interi) dei tensori passati come input, in modo tale da ridurre l'overfitting. Un esempio di questa classe di layers è `nn.Dropout`, che ha due parametri:
    a. p → probabilità con cui azzerare un elemento del tensore. Di default è 0.5;
    b. inplace → se impostato a True, la funzione lavora in-place.
9. **Attivazioni non lineari**, delle quali riportiamo le più importanti con le loro firme:
    a. `nn.Sigmoid(args)` → applica la funzione Sigmoidea su tutti gli elementi di args;
    b. `nn.Softmax(dim=None)` → scala tutti i valori del tensore di input in modo tale che sommino a 1 e rientrino nel range [0, 1] tramite la funzione Softmax. Il parametro dim serve per specificare un eventuale asse su cui eseguire questo calcolo;
    c. `nn.Softmin(dim=None)` → come sopra, scalando con la funzione Softmin;
    d. `nn.Threshold(threshold, value, inplace=False)` → mette una soglia a tutti i valori del tensore passato come argomento. Se il valore è maggiore di threshold, il suo valore rimane lo stesso, altrimenti diventa pari a value;
    e. `nn.ReLU(inplace=False)` → per ogni elemento del tensore passato, se il suo valore è maggiore di 0, tale valore non cambia, altrimenti diventa 0.

7.3.2.4 Loss functions

La loss function in una rete neurale ha il compito di quantificare quanto bene sta predicendo il risultato di un dato input (per esempio collegare un nome a partire dalla foto di un oggetto). In generale tale funzione eseguirà un qualche tipo di sottrazione tra valore atteso e valore reale ottenuto. Esistono molti tipi di loss functions in base a come questa sottrazione viene "condita" con ulteriori operazioni matematiche. Ad esempio nel package torch.nn ci

sono molti esempi di loss functions che sono classi che derivano da `torch._Loss` e che possono essere applicate a due tensori x, y allo stesso modo in cui si applica un Module ad un tensore (inizializzando la classe con i suoi parametri e poi "chiamare" la classe, mettendo tra parentesi i vettori su cui calcolare la loss). Alcune tra le loss functions presenti in `torch.nn` sono:

1. `nn.MSELoss(reduction='mean')` → calcola l'errore quadratico medio di due vettori. Il parametro reduction definisce se il valore ritornato è la somma dei quadrati delle differenze ('sum') oppure la media ('mean'). Di default tale parametro è impostato a 'mean';
2. `nn.L1Loss(reduction='mean')` → calcola il valore assoluto delle differenze tra gli elementi di due vettori. Tramite il parametro reduction si può decidere se farsi ritornare la somma ('sum') di tali valori assoluti o la media ('mean'). Di default tale parametro è impostato a 'mean';
3. `nn.CrossEntropyLoss()` → utile nei problemi di classificazione con C classi.

Esempio di codice per costruire un criterio che misuri la loss, e sua applicazione su due vettori random:

```
loss = nn.MSELoos()
input = torch.randn(3, 5, requires_grad = True)
target = torch.randn(3, 5)
output = loss(input, target)
output.backward()
```

La loss function durante il training di un modello deve essere minimizzata (a zero) tramite il gradient descent, implementato tramite backpropagation (che in PyTorch è stata scritta per sfruttare il gradient tape). Ciò è stato fatto nell'esempio sopra esposto tramite lo specificare il parametro "`requires_grad=True`" nella creazione del Tensor input, e tramite la chiamata a `output.backward()`, che invoca la backpropagation su tutto il grafo a partire dal valore di loss ottenuto.

7.3.3 Pacchetto torch.nn.functional

In tale pacchetto sono definite le funzioni che i vari layer descritti sopra invocano a cascata come funzione di attivazione.
Queste funzioni in generale accettano come parametri:

1. un tensore su cui calcolare la funzione;
2. un tensore che rappresenta i pesi con cui considerare ciascun nodo di input;
3. un tensore che rappresenta i bias da applicare alle funzioni di attivazione;
4. i propri parametri specifici.

Quando un layer invoca tali funzioni, passa come primo parametro l'output del layer precedente, come secondo parametro i pesi che ha aggiornato/sta aggiornando tramite l'allenamento e come terzo i bias ottenuti sempre tramite l'allenamento. Per quanto riguarda i parametri specifici, passa i parametri passati a loro volta al costruttore del layer.

7.3.4 Pacchetto torch.autograd

In questo pacchetto sono presenti funzioni per il calcolo automatico del gradiente. A partire da un modello dove sono presenti dei tensori con l'attributo `requires_grad = True`, è possibile calcolare il gradiente di una funzione (nel nostro caso la loss function) in relazione ai pesi (e i bias) di ciascuno di tali nodi. In particolare, esistono due funzioni che sono dedicate a questo all'interno di `torch.autograd`, e sono:

1. `torch.autograd.backward(tensors)` → funzione che aggiorna l'attributo grad (che memorizza il gradiente) dei vettori su cui calcola il gradiente della loss function. Il parametro tensors corrisponde ai tensori su cui si calcola il gradiente;
2. `torch.autograd.grad(output, inputs)` → funzione che restituisce i valori del gradiente degli outputs rispetto agli inputs passati.

La funzione `torch.autograd.backward()` è presente anche nella classe torch.Tensor e compie lo stesso calcolo, calcolando il gradiente rispetto alle foglie del grafo computazionale (grafo dove i nodi sono tutti i tensor con `requires_grad = True`, mentre gli archi sono i layer, i quali applicano funzioni ai nodi).

Gradient tape è una strategia per implementare il calcolo del gradiente in maniera più veloce. Consiste nel registrare tutte le operazioni che vengono eseguite sui vari nodi di un Model, creando dinamicamente un grafo, per poter poi ripercorrere a ritroso tutti questi passaggi e calcolare con la regola della catena il gradiente della loss function in maniera efficiente.

Questo metodo è quello implementato in `torch.autograd`.

7.3.5 Pacchetto torch.optim

Dopo aver calcolato i gradienti per tutti i nodi del grafo computazionale, è necessario aggiornare i valori dei pesi e dei bias di tali nodi. Questo compito è svolto dall'optimizer, il

quale decrementa il valore dei pesi e dei bias dei vari tensori di una quantità pari al gradiente in quel punto moltiplicato per il learning factor, un coefficiente che misura l'"aggressività" con cui si imbocca la direzione suggerita dal gradiente (direzione di massima discesa della loss function). Normalmente i valori del learning rate variano dentro il range 0.1 - 0.0001, ma è modificabile in base ai risultati che si ottengono e alla velocità con cui il modello impara.

I principali optimizer in PyTorch sono:

1. `torch.optim.Adam`;
2. `torch.optim.ASGD` → averaged stochastic gradient descent;
3. `torch.optim.Rprop` → resilient backpropagation algorithm.

In generale gli optimizer chiedono come primo parametro la lista di tutti i nodi che devono essere aggiornati, e tale lista è referenziabile tramite il metodo `torch.nn.Model.parameters()` che ritorna un iteratore su di essi. Come secondo parametro c'è il learning rate e per i tre optimizer citati sopra, il valore di default è 0.01.

Un esempio di costruzione di un optimizer:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

Un esempio di utilizzo di un optimizer:

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

dove optimizer.step() è la funzione che aggiorna i pesi e i bias dei vari nodi di model. Inoltre è da notare come all'inizio di ogni ciclo sia invocato il metodo optimizer.zero_grad(), che azzera tutti i valori presenti nel parametro grad dei vari nodi, poiché il metodo backward() non riscrive ogni volta i nuovi gradienti, ma li memorizza in un buffer, cosa che potrebbe diminuire le prestazioni della rete.

7.4 - Esempio di training di un modello

In questo paragrafo viene esposta la costruzione di una rete semplice per fare la regressione di una funzione ignota (si è scelta la funzione sen(x)) andando ad ottimizzare i tre coefficienti di un polinomio di terzo grado.

Come prima cosa, si importa torch per avere tutti i pacchetti di cui si è parlato in questo report e math per avere l'implementazione della funzione seno.

```
import torch
import math
```

Poi si creano gli assi della funzione seno vera tramite due Tensor

```
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)
```

A questo punto si inizializza xx prendendo i 2000 valori di x, elevandoli alla prima, alla seconda e alla terza potenza, ottenendo un tensor di size (2000, 3)

```
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)
```

Si costruisce il modello della rete tramite un layer lineare con 3 nodi in ingresso e 1 in output e un layer che appiattisce il tensor del layer precedente dalla dimensione 0 alla dimensione 1 (ottenendo tutte le dimensioni in fila). Come loss si è scelta MSELoss, con somma dei quadrati degli errori.

```
# tramite torch.nn si può creare il modello e inizializzare la loss function
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
loss_fn = torch.nn.MSELoss(reduction='sum')
```

Poi si crea l'optimizer, scegliendo RMSProp con learning rate di 0.001 e come parametri si prendono tutti quelli del modello

```
learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
```

Ora si può far allenare la rete per 2000 iterazioni, predicendo prima un valore per i 3 coefficienti facendo passare ripetutamente l'input attraverso il modello creato, poi andando a misurare la loss tra il risultato ottenuto e la funzione seno calcolata all'inizio, successivamente invocando la backpropagation che calcola i gradienti, i quali servono all'optimizer per modificare i pesi e i bias dei nodi del modello (funzione step())

```
for t in range(2000):
    # Forward pass: calcolo del valore di y a partire da xx
    y_pred = model(xx)

    # calcolo della loss function
    loss = loss_fn(y_pred, y)

    # azzerati i campi grad dei nodi
    optimizer.zero_grad()

    # Backward pass: calcolati i gradienti per ciascun nodo
    loss.backward()

    # Chiamando la funzione step dell'optimizer, i pesi e i bias vengono
    # modificati
    optimizer.step()
```

7.5 - Ottimizzazione con processori CUDA

CUDA è un software che permette di interfacciare le GPU di un computer a codice scritto in linguaggi ad alto livello, come la libreria PyTorch. PyTorch supporta tutte le operazioni descritte sopra anche per esecuzioni su GPU, specificando su quale dispositivo della propria architettura condurre i calcoli tramite 3 sintassi in particolare, le quali si possono chiamare sia sui tensori, che su modelli (per portare tutti i tensori istanziati al loro interno sulla GPU).
Quando si compiono calcoli anche attraverso dispositivi CUDA, bisogna avere cura di non eseguire calcoli tra tensori che risiedono in processori diversi, poiché tali calcoli non verrebbero eseguiti.

```
cuda = torch.device('cuda')     #dispositivo CUDA di default
cuda0 = torch.device('cuda:0')
cuda2 = torch.device('cuda:2')   #GPU 2
```

```
d = torch.randn(2, device=cuda2)
e = torch.randn(2).to(cuda2)
f = torch.randn(2).cuda(cuda2)
```

Si possono chiamare i metodi to(), cuda() e specificare il parametro device anche senza referenziare in anticipo i dispositivi CUDA desiderati, scrivendo il nome della GPU scelta tra apostrofi:

```
model = Generate()
model.to('cuda')    #intero modello spostato sulla GPU
```

Per poter far riconoscere a PyTorch la GPU del proprio computer è necessario scaricare i driver NVIDIA del proprio dispositivo, il toolkit NVIDIA e la versione di PyTorch che supporta l'uso di GPU.

7.6 PyTorch Mobile

PyTorch Mobile fa riferimento ad una serie di funzioni che permettono di sfruttare i modelli creati con PyTorch su dispositivi mobili che supportano ad esempio Android e iOS. In particolare i pacchetti torch.jit e torch.utils sono quelli più rilevanti in tal senso.
Il tipico workflow per poter usare un modello allenato su PyTorch anche su dispositivi Android e iOS è riassunto nell'immagine seguente:

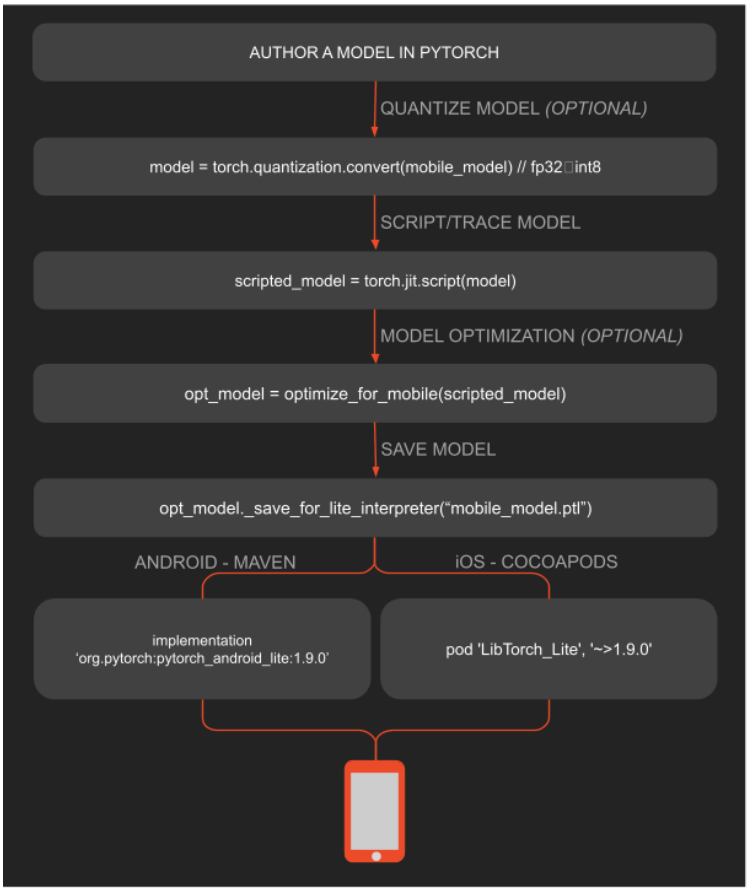

figura 7.4: workflow per eseguire un modello PyTorch su Android o iOS

Ora vediamo i passaggi uno alla volta.

7.6.1 Fondere i layer

Come prima cosa può essere utile unire layer per ottenere un modello più snello ma allo stesso tempo qualitativamente ottimo. Non è possibile fondere tutti i tipi di layer, ma solo un sottoinsieme abbastanza piccolo formato da:

1. layer convoluzionali e batchNorm;
2. layer convoluzionali, batchNorm e ReLU;
3. layer convoluzionali e ReLU;

4. layer linear e Relu;
5. layer batchNorm e ReLU.

La fusione di layer (che sono dei Module) viene effettuata tramite la funzione:

```
torch.quantization.fuse_modules(nomeModule, [['siglaLayer1',
'siglaLayer2']], inplace=False).
```

7.6.2 Quantizzare il modello

La quantizzazione di un modello consiste nel cast del dtype di tutti i Tensor del modello da float di 32 bit a int di 8 bit (qint8, uno dei tipi disponibili nella classe Tensor). Quantizzare è un'opzione da perseguire per rendere il proprio modello ancora più leggero (e quindi veloce), sempre senza rinunciare alla sua attendibilità. È consigliabile farlo a prescindere, tranne quando ciò non sia già stato ottenuto tramite l'uso di altre librerie (ad esempio torchvision permette di avere direttamente la versione quantizzata di alcuni modelli)
Per quantizzare esistono 3 approcci:

1. post training dynamic quantization;
2. post training static quantization → degrada di più il modello, ma quantizza di più;
3. quantization aware training → allena modello con pesi e attivazioni quantizzate a differenza degli altri due metodi che quantizzano dopo aver allenato il modello;

7.6.3 Generare script e ottimizzare per dispositivi mobili il modello

Dopo aver allenato il progetto, aver fuso eventuali layer e aver quantizzato il proprio modello, è necessario trasformare il proprio script Python in un TorchScript, il quale può essere eseguito in un ambiente C++ (disponibilie in iOS e Android). Si può inoltre ottimizzare il modello convertito per renderlo ancora più adatto e veloce per i dispositivi mobili.
I due comandi principali per creare un TorchScript sono:

- `torch.jit.trace`
- `torch.jit.script`

facenti parte entrambi del package torch.jit.

Il metodo trace consiste nel passare attraverso l'intero modello un input generico, chiamando il metodo forward per arrivare fino all'output layer. Ciò che si ottiene alla fine è la traccia del

modello. Per poter utilizzare questa modalità di scripting, è necessario che il modello non contenga blocchi if o for all'interno della sua definizione.

```
dummy_input = torch.rand(1, 3, 224, 224)
torchscript_model = torch.jit.trace(model_quantized, dummy_input)
```

Il metodo script invece consente di tenere in conto eventuali blocchi di controllo di flusso (if e for) nello scripting del modello.

```
torchscript_model = torch.jit.script(model_quantized)
```

Il metodo script sembra risolvere tutti i casi d'uso per lo scripting di un modello, però non è possibile usarlo indistintamente in tutti i frangenti, poiché, dato che TorchScript è un sottoinsieme di Python, se il codice da scriptare contiene righe scritte in linguaggio che non è TorchScript, il metodo torch.jit.script non può essere eseguito. Il metodo trace in questi casi riesce a convertire il codice Python al posto di trascriverlo solamente.

7.6.4 Ottimizzare il modello

Per ottimizzare il modello appena scriptato, si usa la funzione:

```
optimize_for_mobile(nomeModelloScriptato)
```

che fa parte di `torch.utils.mobile_optimizer`. Tale funzione fonde alcuni layer che su mobile sarebbero ridondanti, così come rimuove i Layer di dropout (se la variabile training del Module è false).

7.6.5 Salvare il modello

Come ultimo passaggio c'è il salvataggio del modello come file .ptl o .pt, il quale può essere successivamente posizionato nella cartella asset di un'eventuale applicazione Android.
Tale operazione viene eseguita tramite il comando:

```
torch.jit.save(torchscript_model_optimized, "mobilenetv2_quantized.pt")
```

CAPITOLO 8 - FRUITIFY

Per comprendere al meglio Tensor Flow abbiamo scelto di sviluppare un'app che sfrutta la classificazione di immagini. In particolare abbiamo allenato un modello per il riconoscimento di alcuni tipi di frutta e verdura. L'applicazione funziona solo se si garantiscono i permessi alla fotocamera.

8.1 Creazione del modello

Il modello è creato da noi e per svilupparlo abbiamo seguito questi passaggi:

- Per prima cosa abbiamo individuato un dataset di immagini di frutta, [18] da cui abbiamo scaricato le immagini grazie all'API di Kaggle:

```
os.environ['KAGGLE_CONFIG_DIR'] = '/content'
!kaggle datasets download -d
       kritikseth/fruit-and-vegetable-image-recognition
!unzip /content/fruit-and-vegetable-image-recognition.zip -d
       /content/dataset
```

- Il passo successivo è stato quello di creare uno script in python per processare i dati e allenare il modello. Abbiamo implementato il modello con la libreria Keras di Tensor Flow, utilizzando il transfer learning per adattare il modello MobileNetV2 al nostro caso.

```
base_model = MobileNetV2(weights='imagenet', include_top=False,
input_shape=(img_height,img_width,3))
base_model.trainable = False # Blocco la possibilità di allenare tutti i
                             # layers
inputs = base_model.input

x = GlobalAveragePooling2D()(base_model.output)

x = Dense(128, activation='relu')(x)
x = Dense(128, activation='relu')(x)

outputs = tf.keras.layers.Dense(36, activation='softmax')(x)

model = Model(inputs=inputs, outputs=outputs)

model.compile(
```

```
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

- Abbiamo eseguito lo script su Google Colab sfruttando la GPU messa a disposizione dal runtime.

```
model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples //train_generator.batch_size,
    validation_data=val_generator,
    validation_steps=val_generator.samples //val_generator.batch_size,
    epochs=20)
```

Il modello a volte può sbagliare in quanto è allenato su un dataset limitato e il numero di epoch di allenamento è limitato dal tempo di utilizzo gratuito di Colab. Per ottenere, quindi, un modello più performante sarebbe necessario migliorare il dataset o modificare i parametri della rete ed eseguirla per un numero maggiore di epoch.

8.2 Struttura dell'applicazione

L'applicazione ha tre funzioni principali (*figura 8.1)*.

1. La prima funzione permette di visualizzare e gestire liste di frutta e verdura.
2. La seconda ha lo scopo di visualizzare i principali valori nutrizionali dei prodotti.
3. Infine c'è la possibilità di riconoscere un frutto tramite la fotocamera per vederne i valori nutrizionali o per aggiungerlo alla lista corrente.

Figura 8.1

Figura 8.2

8.3 Visualizzazione e modifica di liste e dei rispettivi elementi

In questa sezione è possibile creare nuove liste di frutta e verdura (*figura 8.2)*. Dopo aver aggiunto una lista è possibile rinominarla (tramite il componente Dialog di Android X) o eliminarla. Due o più liste inoltre possono avere lo stesso nome in quanto nel database Room sono univocamente determinate da un id numerico.
Cliccando su una lista si possono vedere i prodotti al suo interno (*figura 8.3)*.

In questa schermata è possibile aggiungere elementi alla lista. Per farlo bisogna cliccare sul bottone “più” e successivamente sul simbolo “cerca”. Si apre una schermata in cui è possibile scegliere un elemento e la rispettiva quantità da aggiungere. Nel caso si scelga di aggiungere un frutto già presente la quantità viene sommata a quella che già c’era. Tramite “Toast” viene segnalato se il frutto è stato aggiunto correttamente.

E’ possibile aggiungere un frutto alla lista anche cliccando su “fotocamera”. Nella sezione fotocamera (verrà analizzata in seguito) che si aprirà basterà inquadrare un frutto e dopo il riconoscimento da parte del modello cliccare su “aggiungi”. In questo verrà aggiunto 1 solo prodotto, la quantità sarà modificabile una volta tornati nella schermata della lista.

Figura 8.3

Dopo aver aggiunto un frutto (o una verdura) è possibile modificarne la quantità cliccando sull'elemento della lista e interagendo con il Dialog che compare (*figura 8.4)*.

Figura 8.4

E' infine possibile eliminare un elemento dalla lista cliccando sul "cestino" che compare cliccando il "più".

8.4 Visualizzazione di valori nutrizionali

Se nella home page si seleziona "cerca" compare una schermata per ricercare un frutto o una verdura. Una volta individuato il prodotto che si sta cercando è possibile cliccarlo per ottenere informazioni sui suoi valori (*figura 8.5)*.

Figura 8.5

8.5 Fotocamera

Si accede a questa schermata cliccando su “foto” dalla home page o, come descritto in precedenza, nella fase di aggiunta di un frutto ad una lista. Nel caso ci si acceda dalla home, dopo aver individuato un frutto viene richiamata l’activity corrispondente al frutto individuato.

Si tratta della parte principale della nostra app in quanto implementa il modello da noi creato con tensor flow.

Dopo aver aperto la fotocamera il modello tenterà di riconoscere un frutto o una verdura. Tramite un buffer contenente bitmap (che sono le immagini in tempo reale della fotocamera) viene chiesto periodicamente al modello di restituire una predizione. Dalla distribuzione di probabilità ritornata dal modello viene scelto l’elemento con il valore maggiore. Questo elemento viene preso in considerazione solo se supera il valore di threshold di 90%. In caso affermativo viene mostrato il risultato e un bottone per compiere l’azione descritta qualche riga sopra.

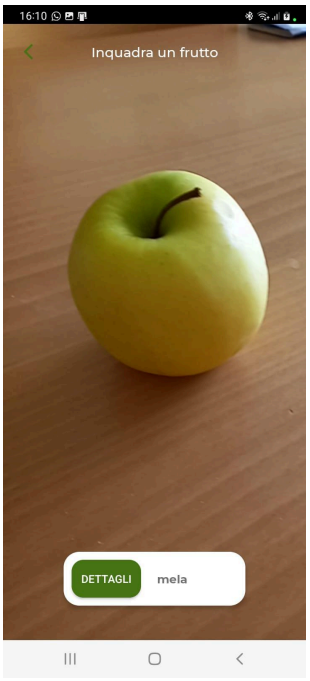

figura 8.6

Le icone dell'app di Fruitify sono state prese da Flat Icon [19], Freepik [20], Pexels[21].

BIBLIOGRAFIA

[1] IBM - What is a neural networks
https://www.ibm.com/topics/neural-networks

[2] Stanford - Neural Networks
https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

[3] Android Developers - API Neural Networks
https://developer.android.com/ndk/guides/neuralnetworks

[4] Android Developers - Native Hardware Buffer
https://developer.android.com/ndk/reference/group/a-hardware-buffer

[5] What is TensorFlow? Meaning, Working, and importance
https://www.spiceworks.com/tech/devops/articles/what-is-tensorflow/

[6] Gradient Descent
https://en.wikipedia.org/wiki/Gradient_descent

[7] Basic Training Loops
https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/basic_training_loops.ipynb#scrollTo=gFzH64Jn9PIm

[8] Keras: l'API di alto livello per TensorFlow
https://www.tensorflow.org/guide/keras?hl=it

[9] TensorFlow - Python Deep Learning Neural Network API
https://deeplizard.com/learn/video/RznKVRTFkBY

[10] fit() function documentation
https://www.tensorflow.org/api_docs/python/tf/keras/Sequential#fit

[11] Deep Learning Fundamentals - Classic Edition
https://deeplizard.com/learn/video/hfK_dvC-avg

[12] Confusion matrix
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py

[13] Dogs vs. Cats
https://www.kaggle.com/c/dogs-vs-cats/data

[14] Images classification
https://www.tensorflow.org/tutorials/images/classification?hl=it#visualize_training_images

[15] Imagenet
https://www.image-net.org/

[16] VGG16 paper
https://arxiv.org/pdf/1409.1556

[17] Tensorflow introduction
https://www.javatpoint.com/tensorflow-introduction

[18] Stackshare
https://stackshare.io/stackups/pytorch-vs-torch

[19] PyTorch - Golden
https://golden.com/wiki/PyTorch-NMGD4Y4

[19] Kaggle
https://www.kaggle.com/code/nimapourmoradi/fruits-and-vegetables-image-mobilenetv2/input

[20] Flat Icon
https://www.flaticon.com/

[21] Freepik
https://it.freepik.com/

[22] Pexels
https://www.pexels.com/

[23] PyTorch documentation
https://pytorch.org/docs/stable/index.html

[24] Stackshare
https://stackshare.io/stackups/pytorch-vs-torch

[25] PyTorch
https://en.wikipedia.org/wiki/PyTorch

[26] Torch
https://en.wikipedia.org/wiki/Torch_(machine_learning)

[27] IBM - Cos’è PyTorch
https://www.ibm.com/it-it/topics/pytorch