
Relazione Progetto

Internet of Things based smart systems

**Valutazione dell'impatto della riduzione della precisione
dei pesi in una rete neurale nei confronti
dell'accuratezza di classificazione**

Alessandro Messina, matricola O55000354

Orazio Scavo, matricola O55000414

ANNO ACCADEMICO 2018/2019

Sommario

1	<i>Introduzione</i>	- 3 -
1.1	Ambito applicativo.....	- 3 -
1.2	Obiettivi.....	- 3 -
1.3	Flusso dell'analisi.....	- 4 -
2	<i>Rete originale</i>	- 5 -
2.1	Progettazione della rete neurale	- 5 -
2.2	Scelte operative	- 5 -
2.3	Valutazione dell'accuratezza della rete.....	- 5 -
3	<i>Approximate Computing sulla rete neurale realizzata</i>	- 6 -
3.1	Note sullo Standard IEEE sulla rappresentazione dei float.....	- 6 -
3.1.1	IEEE 754 single-precision binary floating-point formato: binary 32.....	- 6 -
3.1.2	Approssimazione del tipo di dato float	- 6 -
3.2	Scelta delle configurazioni approssimate.....	- 8 -
3.2.1	Layer approssimati	- 8 -
3.2.2	Entità dell'approssimazione	- 8 -
3.3	Flusso dell'approssimazione	- 9 -
3.4	NNAXIM Tool (Neural Network Approximate Computing SIMulator).....	- 10 -
3.4.1	Reti disponibili	- 10 -
3.4.2	Struttura del progetto.....	- 10 -
3.4.3	L'interfaccia e le operazioni disponibili	- 10 -
4	<i>Valutazione dei risultati ottenuti</i>	- 12 -
5	<i>Conclusioni</i>	- 14 -
	Bibliografia	- 15 -

1 Introduzione

Negli ultimi anni l'Approximate Computing è stato riscoperto come uno degli approcci più promettenti per la riduzione del consumo di energia in molte applicazioni caratterizzate da una certa tolleranza agli errori (si dice che queste applicazioni abbiano una *forgiving nature*). Questa tendenza è sicuramente legata alla crescente importanza che i consumi energetici assumono nella maggior parte dei dispositivi in uso al giorno d'oggi (dispositivi mobili, IoT, data center, etc.). La riduzione della quantità di energia e risorse richieste da parte di questi dispositivi comporta una serie di vantaggi non trascurabili, quale ad esempio nel caso di dispositivi IoT o mobili l'incremento della loro autonomia.

Un ambito che si presta bene all'applicazione dell'Approximate Computing è quello delle *reti neurali*, in quanto spesso queste sono caratterizzate da una certa tolleranza agli errori, ma soprattutto sono capaci di auto-correggere gli errori introdotti dalle approssimazioni (*self error-healing*), grazie alla possibilità di eseguire ulteriori training, successivi all'approssimazione, dei parametri della rete. Con il retraining della rete, i pesi approssimati riescono a convergere verso dei valori tali da consentire una minimizzazione dell'effetto dell'errore introdotto dall'approssimazione e una conseguente massimizzazione dei vantaggi della stessa.

Uno dei principali problemi che nascono quando si lavora con delle reti neurali, avendo a disposizione delle risorse limitate, riguarda la quantità di memoria utilizzata per memorizzare i parametri della rete. Ridurre ad esempio il numero di bit per rappresentare tali pesi ha un impatto molto positivo sulla quantità di risorse richieste. La riduzione del numero di bit offre inoltre la possibilità di utilizzare circuiti aritmetici ridotti con una conseguente riduzione dell'area, potenza e possibilmente una riduzione del percorso critico e quindi un aumento della frequenza di clock. Una tale approssimazione ha però delle conseguenze anche sull'accuratezza della rete neurale. Lo scopo di questa trattazione è proprio quello di valutare tale impatto.

1.1 Ambito applicativo

Per rendere più realistica l'analisi, è stata pensata una specifica applicazione in ambito IoT per la gestione di un parcheggio, in cui è richiesta la classificazione degli oggetti immortalati da una fotocamera posta all'entrata dello stesso, in modo da identificare diverse tipologie di veicolo o altri oggetti, la cui presenza in quel punto rappresenta una situazione non prevista. Per questo tipo di applicazione (classificazione) viene solitamente utilizzata una rete neurale di tipo convoluzionale (CNN), che data in input una certa immagine è capace di rilevare oggetti al suo interno e classificarli associandoli ad una tra un certo numero di categorie.

1.2 Obiettivi

L'obiettivo principale di questa trattazione è quello di valutare il *tradeoff* tra memoria risparmiata e perdita di accuratezza in seguito ad una riduzione del numero di bit utilizzati per rappresentare i pesi dei neuroni della rete. A tal fine, sono state scelte 9 configurazioni, variabili sulla base di dove l'approssimazione è applicata e in che misura, e per ognuna di esse è stata effettuata la valutazione di cui sopra.

Per rendere più agevole il processo di training e test delle varie configurazioni è stata inoltre implementata una semplice applicazione in C++, NNAXIM¹, che consente di analizzare il comportamento di una o tutte le configurazioni. Per la realizzazione del programma è stata utilizzata la libreria *TinyDNN*², che consente di implementare (sempre in C++) delle reti neurali a livello software.

¹ NNAXIM (Neural Network Approximate Computing SIMulator) è disponibile su GitHub presso l'indirizzo <https://github.com/Taletex/NNAXIM>. Il nome del progetto è un tributo al simulatore NOXIM, disponibile sempre su GitHub all'indirizzo <https://github.com/davidepatti/noxim>.

² TinyDNN è disponibile gratuitamente su GitHub presso l'indirizzo <https://github.com/tiny-dnn/tiny-dnn>.

1.3 Flusso dell'analisi

I capitoli successivi approfondiscono i vari passi del flusso di lavoro seguito per l'analisi, il quale si articola nei seguenti punti:

- *Capitolo 2.* Progettazione, allenamento e valutazione dell'accuratezza della rete originale.
- *Capitolo 3.* Applicazione dell'Approximate Computing sulla rete realizzata, spiegazione delle scelte relative alle approssimazioni effettuate sulla rete e esposizione dell'applicazione NNAXIM.
- *Capitolo 4.* Confronto dei risultati ottenuti per le diverse configurazioni.
- *Capitolo 5.* Conclusioni sul lavoro svolto.

2 Rete originale

Data l'applicazione, è stato necessario trovare una rete neurale capace di effettuare la classificazione richiesta a partire dalle immagini di input prodotte dalla fotocamera del sistema. Una volta trovata la rete è stato necessario allenarla per ottenere i valori dei pesi per la configurazione base da usare successivamente come punto di partenza per l'applicazione dell'approssimazione nelle varie configurazioni. È stata inoltre valutata l'accuratezza della rete originale in modo da poter effettuare il confronto con le configurazioni approssimate.

2.1 Progettazione della rete neurale

Per la classificazione è stata scelta una *Convolutional Neural Network* utilizzata all'interno dell'esempio *cifar10* di TinyDNN. Questa rete è composta da una serie di 3 blocchi di layer (ognuno dei quali costituito da un layer convoluzionale, un pooling layer e un relu activation layer) seguiti da un fully-connected layer, un relu activation layer e infine un ulteriore fully-connected layer. Per quanto riguarda i dataset di training e di test sono stati usati quelli di CIFAR10³. Questi contengono 60000 immagini (50000 per il training, 10000 per il test) 32x32px raffiguranti degli oggetti riconducibili ad una delle seguenti 10 categorie: aeroplani, automobili, uccelli, gatti, cervi, cani, rane, cavalli, navi e camion. La rete neurale grazie al training impara quindi classificare gli oggetti presenti nelle immagini di input alla rete in una delle suddette categorie.

2.2 Scelte operative

La rete progettata è caratterizzata da un'elevata complessità. Tale complessità si riflette sui tempi di allenamento della stessa e non consente quindi di lavorare agevolmente su di essa durante le fasi dello sviluppo del tool di simulazione (a meno che non si posseggano delle risorse computazionali adeguate). Al fine di disaccoppiare le fasi di sviluppo del tool e di training della rete neurale non approssimata, si è scelto inizialmente di lavorare su una rete estremamente più semplice, il cui obiettivo è quello di predire l'output di una funzione sinusoidale⁴. In questo modo è stato possibile sviluppare e testare velocemente le procedure di training e test sulla rete e la definizione delle varie configurazioni da applicare per l'approssimazione. Dopo aver verificato il corretto funzionamento delle procedure sviluppate è stato effettuato un *porting* dell'applicazione sul modello di rete descritto nel paragrafo precedente. L'applicazione NNAXIM è stata poi sfruttata per eseguire una sola volta il training e il test delle reti approssimate (e non) su un calcolatore più performante, riducendo così le latenze di esecuzione.

2.3 Valutazione dell'accuratezza della rete

L'accuratezza di una rete neurale usata per la classificazione può essere facilmente valutata basandosi sulla percentuale di classificazioni corrette su quelle totali effettuate sul dataset di test.

³ <https://www.cs.toronto.edu/~kriz/cifar.html>.

⁴ Come base di partenza per l'implementazione di questa rete è stato utilizzato l'esempio *sinus_fit* offerto da TinyDNN.

3 Approximate Computing sulla rete neurale realizzata

Come già detto nei capitoli precedenti, l'Approximate Computing applicato alla rete riguarda l'approssimazione dei bit usati per memorizzare i pesi della rete. Nell'approccio ideale sarebbe necessario applicare approssimazioni successive in maniera iterativa (approssimazione, test, training, test) fino a trovare il giusto *tradeoff* tra quantità di memoria risparmiata e accuratezza persa. Fintanto che l'accuratezza rimane sopra il requisito minimo fornito dall'applicazione è possibile continuare ad approssimare. A causa della carenza di risorse computazionali a disposizione si è scelto di utilizzare solamente 9 configurazioni di approssimazione prestabilite.

3.1 Note sullo Standard IEEE sulla rappresentazione dei float

Il tool sviluppato simula l'utilizzo di float, che è il tipo di dato utilizzato dalle reti TinyDNN per la memorizzazione dei pesi della rete, codificati con un numero di bit inferiore rispetto a quello standard (32).

3.1.1 IEEE 754 single-precision binary floating-point format: binary 32

Per effettuare la simulazione è stato necessario esaminare lo standard per la codifica dei float (*IEEE 754 single-precision binary floating-point format: binary32*), esso prevede la rappresentazione del numero come 3 parti fondamentali:

- Segno: 1 bit (0 -> '+' ; 1 -> '-')
- Esponente: 8 bit (intero con segno in complemento a 2)
- Significando o Mantissa (parte frazionaria): 23 bit

Il valore rappresentato dal float viene così valutato:

$$VAL = (-1)^{Segno} \cdot 2^{Esponente-127} \cdot [i, < Significando >]_{base2} \quad (1)$$

È bene notare che nella (1):

- il termine i è un bit implicito che vale '0' se tutti i bit dell'esponente sono '0' (in questo caso $Esponente$ assume valore 1), altrimenti è uguale a '1'.
- Se i bit dell'esponente sono tutti '1', allora il valore rappresentato è infinito.

3.1.2 Approssimazione del tipo di dato float

Dopo aver studiato tale codifica è stato modellato (tramite l'uso di un'apposita funzione in C++) un tipo di dato in virgola mobile codificato in un numero di bit arbitrariamente minore, a costo di un certo arrotondamento (errore) nella mantissa. La procedura di riduzione del dato prevede, stabilito il numero di bit da rimuovere, N , di arrotondare la mantissa al più vicino numero che sia rappresentabile in modo che gli ultimi N bit siano tutti '0'.

Il risultato che si intende ottenere appare evidente nel seguente esempio: si immagini di voler approssimare il numero 1.26, la rappresentazione della parte frazionaria del Significando è la seguente:

01000010100011110101110

Si immagini ora di voler utilizzare 4 bit in meno, si procede all'approssimazione nel seguente modo:

1) Arrotondamento.

```
0100001010001111010 1110 +  
0000000000000000000 1000 =  ← il 4° bit è '1'  
0100001010001111011 0110
```

2) Troncamento. Rimozione degli ultimi quattro bit tramite AND bitwise con una maschera:

$$\begin{array}{r} 0100001010001111011 \ 0110 \ \& \\ 1111111111111111 \ 0000 \ = \quad \leftarrow \text{gli ultimi 4 bit sono '0'} \\ \hline 0100001010001111011 \ 0000 \end{array}$$

In questo modo è stato ottenuto un valore a 32 bit in cui però gli ultimi 4 bit sono certamente 0 e possono quindi essere eliminati dalla rappresentazione del valore in memoria. Il valore codificato da questi bit è 1.2600002, dunque a fronte di un errore trascurabile è stato possibile risparmiare 4 bit.

Il vantaggio è ancora più evidente se durante l'approssimazione ci si basa sulla precisione richiesta piuttosto che sul numero di bit da risparmiare. Ad esempio, considerando sempre 1.26 come valore da approssimare, se si potesse tollerare un errore di 0.01 si potrebbe approssimare il valore a 1.25 in modo da riuscire a rappresentare la mantissa addirittura con solo 2 bit (risparmiandone 21). Infatti, la rappresentazione per la parte frazionaria del Significando sarebbe la seguente:

01 00000000000000000000

All'interno del tool di simulazione NNAXIM la funzione che esegue questa procedura di approssimazione è `roundb(...)` (Fig. 1). Essa viene invocata immediatamente dopo l'aggiornamento dei pesi durante e prima del retraining della rete, dunque i valori salvati nei pesi saranno della dimensione desiderata.

Analizzando più attentamente la tecnica di approssimazione adottata si nota come sia possibile stabilire un valore massimo per l'errore introdotto. Infatti, a seconda dell'ordine di grandezza del numero che si vuole rappresentare e del numero di bit utilizzati è possibile calcolare il massimo scostamento che la riduzione può causare. Si noti che la prima fase dell'operazione di riduzione, ossia quella di arrotondamento, permette di dimezzare tale valore massimo rispetto ad un semplice troncamento.

Tramite dei semplici calcoli è possibile ottenere questi valori per diverse combinazioni di ordini di grandezza e numero di bit in uso. Tali dati assumono particolare importanza nell'applicazione realizzata in quanto, come è possibile osservare dalla Tabella 1, l'errore è molto basso per valori dell'ordine di grandezza unitario, che è proprio il caso dei valori utilizzati per la rete neurale progettata.

Bit di rappresentazione	1.0	10.0	100.0
12 bit	0.0625	0.5	4.
13 bit	0.03125	0.25	2.
14 bit	0.01563	0.125	1.
15 bit	0.00781	0.0625	0.5
16 bit	0.00391	0.03125	0.25
17 bit	0.00195	0.01563	0.125
18 bit	0.00098	0.00781	0.0625
19 bit	0.00049	0.00391	0.03125
20 bit	0.00024	0.00195	0.01563
21 bit	0.00012	0.00098	0.00781
22 bit	0.00006	0.00049	0.00391

Tabella 1. Errore introdotto approssimando i bit di rappresentazione dei float secondo lo standard IEEE 754


```

/* roundb(f, 15) => keep 15 bits in the float, set the other bits to zero */
float roundb(float f, int bits) {
    union {                                     // num.i and num.f are mapped on same bits
        int i;
        float f;
    } num;

    bits = 32 - bits;
    num.f = f;
    num.i = num.i + (1 << (bits - 1)); // round instead of truncate
    num.i = num.i & (-1 << bits);      // AND bitwise between mask and rounded value
    return num.f;
}

```

Figura 1. Funzione di arrotondamento dei float utilizzata in NNAXIM

3.2 Scelta delle configurazioni approssimate

Nella Tabella 2 è mostrata la lista delle configurazioni delle approssimazioni adottate. Si può notare come ognuna delle 9 configurazioni è identificata dai layer coinvolti nell'approssimazione e dalla misura con la quale questa è applicata ai pesi dei neuroni dei layer.

Configurazione	Numero di bit di ogni peso dei neuroni dei layer di input e output	Numero di bit di ogni peso dei neuroni degli hidden layer
Originale	32	32
1	16	16
2	14	14
3	12	12
4	32	16
5	32	14
6	32	12
7	16	14
8	14	12
9	12	11

Tabella 2. Configurazioni scelte per le approssimazioni del numero di bit dei pesi.

A seguire sono spiegate le motivazioni della scelta di queste configurazioni.

3.2.1 Layer approssimati

La prima cosa sulla quale è importante soffermarsi riguarda la tipologia di layer sui quali è stata applicata l'approssimazione. Le 9 configurazioni viste in Tabella 2 possono essere suddivise a tal proposito in 3 gruppi:

- Gruppo 1; approssimazione omogenea sui neuroni di tutti i layer della rete.
- Gruppo 2; approssimazione applicata solo agli hidden layer della rete.
- Gruppo 3; approssimazione differente tra neuroni degli hidden layer e neuroni dei layer di input e output.

Questa classificazione è stata effettuata basandosi sui risultati riportati in [1] e [2], i quali, tramite test più esaustivi, hanno dimostrato che i neuroni degli hidden layer possiedono una maggiore resilienza agli errori.

3.2.2 Entità dell'approssimazione

Ad ognuno dei gruppi di cui sopra appartengono 3 configurazioni che differiscono tra loro in base al numero di bit utilizzati per l'approssimazione. Tale quantità è stata scelta sulla base dell'ambito applicativo (classificazione) e sulla base della Tabella 1, dalla quale risulta evidente come approssimazioni inferiori o uguali ai 16 bit (e che quindi codificano i float con un numero di bit maggiore o uguale a 16) introducono un

errore molto piccolo se le grandezze in gioco sono dell'ordine dell'unità (il nostro caso), mentre approssimazioni maggiori o uguali di 20 bit (e che quindi codificano i float con un numero di bit minore o uguale a 12) introducono un errore molto grande (a prescindere dall'ordine delle grandezze in gioco). Per questo motivo e per altri (relativi soprattutto all'ambito applicativo e alla rete neurale scelta), nelle configurazioni adottate l'approssimazione minima introdotta è di 16 bit⁵, mentre la massima è di 21 bit. Le configurazioni più spinte (approssimazioni di 20/21 bit) sono state utilizzate per mostrare dei limiti oltre i quali è bene non andare se non si vuole degradare eccessivamente l'accuratezza della rete.

3.3 Flusso dell'approssimazione

Dopo aver mostrato le configurazioni scelte per le approssimazioni e le motivazioni dietro di esse è necessario spiegare come ogni approssimazione viene applicata alla rete. Il flusso dell'approssimazione è uguale per ognuna delle configurazioni e prevede, oltre all'approssimazione in sé, anche delle fasi di test e retraining della rete. In particolare, vengono effettuate le seguenti fasi:

1. *Approssimazione dei pesi secondo la configurazione scelta.* Ogni peso della rete viene approssimato (e non troncato, al fine di minimizzare l'errore introdotto) al numero di bit stabilito dalla configurazione. In Fig.2 è mostrato il pezzo di codice (C++) utilizzato all'interno di NNAXIM per applicare tale approssimazione. La configurazione è definita tramite i due parametri `hidden_nlayer_bits` e `extern_nlayer_bits` che indicano rispettivamente il numero di bit da utilizzare per i neuroni degli hidden layer e il numero di bit da utilizzare per i neuroni dei layer di input e output. La funzione che applica effettivamente l'approssimazione sul singolo peso è `roundb(...)`.
2. *Test dopo l'approssimazione.* La rete viene testata in modo da individuare la perdita di accuratezza rispetto alla configurazione originale.
3. *Retraining.* Per mitigare l'effetto negativo sull'accuratezza dovuto all'approssimazione viene sfruttata la capacità di self-healing della rete neurale tramite un opportuno training. Questo training non necessita dello stesso numero di epoche del training della rete originale, poiché i pesi sono già allenati e sono quindi più veloci a convergere. Questa fase è fondamentale e molto delicata poiché il numero effettivo di epoche da eseguire è di cruciale importanza: è stato notato come un numero troppo basso ha effetti negativi sull'accuratezza della rete, mentre un numero troppo alto è superfluo e comporta solo una perdita di tempo.
4. *Test dopo il retraining.* Questa fase ha un duplice scopo: capire se il numero di epoche del retraining è stato sufficiente e calcolare la perdita di accuratezza effettiva dell'approssimazione rispetto alla configurazione originale. Durante questa fase viene raccolto anche il numero di bit risparmiati grazie all'approssimazione.

```
for (size_t k = 0; k < net.depth(); k++) {
    int bits = ((k==0 || k == net.depth() - 1) ? extern_nlayer_bits : hidden_nlayer_bits);

    weights_list = net[k]->weights();

    if (bits < 32) {
        for (size_t i = 0; i < weights_list.size(); i++) {
            for (size_t j = 0; j < weights_list[i]->size(); j++) {
                weights_list[i]->at(j) = roundb(weights_list[i]->at(j), bits);
            }
        }
    }
}
```

Figura 2. Codice per l'approssimazione dei bit pesi per una specifica configurazione.

⁵ Un'approssimazione di 16 bit potrebbe risultare significativa se usata su reti diverse da quella presentata in 2.1 e con pesi diversi da quelli utilizzati per ottenere i risultati presentati in Fig. 5, Fig. 6 e Fig. 7. Le scelte presentate in questa relazione sono state effettuate anche tenendo conto dell'ambito applicativo e delle risorse a disposizione.

3.4 NNAXIM Tool (Neural Network Approximate Computing SIMulator)

NNAXIM (Neural Network Approximate Computing SIMulator) è un tool che consente di simulare l'applicazione dell'Approximate Computing sul numero di bit usati per rappresentare i pesi di una neural network. Il tool, disponibile su GitHub all'indirizzo <https://github.com/Taletex/NNAXIM>, è scritto in C++ e utilizza la libreria TinyDNN per l'implementazione delle reti neurali. Il tool è stato realizzato per facilitare le operazioni di training, test e raccolta delle prestazioni durante l'applicazione dell'AC sulle neural network.

3.4.1 Reti disponibili

Nel tool è possibile utilizzare due reti: una *CNN* per la classificazione (rete di default) e una *MLP* per l'approssimazione di funzioni. Per cambiare rete è necessario cambiare i file sorgente che si trovano all'interno della directory *src/ac_nn* con quelli presenti in *src/sinus_fit_version*.

3.4.2 Struttura del progetto

Le directory del tool sono organizzate così come segue:

- *doc*; contiene la documentazione del progetto (incluso questo file).
- *src*; contiene i file sorgente del progetto. Al suo interno troviamo a sua volta diverse cartelle:
 - o *ac_nn*; contiene i file sorgente di NNAXIM che consistono in un main (*ac_nn.cpp*), le librerie sviluppate per il tool (*ac_nn_lib.cpp* e *ac_nn_lib.hh*), i file con il modello e i pesi della rete (dentro *net_params*) e un file di log (*log.txt*, usato per memorizzare i risultati dei test automatici).
 - o *sinus_fit_version*; contiene gli stessi file presenti in *ac_nn* riadattati per una MLP per l'approssimazione di funzioni. È sufficiente sostituire i file presenti in *ac_nn* con quelli presenti in *sinus_fit_version* per usare la MLP⁶.
 - o *build*; cartella che contiene (una volta creato il progetto tramite cmake) la soluzione Visual Studio di NNAXIM.
 - o *cereal*, *cmake*, *third_party*, *tiny_dnn*; folder provenienti dalla libreria TinyDNN e necessari per il suo utilizzo.

3.4.3 L'interfaccia e le operazioni disponibili

Al primo avvio dell'applicazione, se non sono stati scaricati anche i pesi e il modello di default, sarà necessario eseguire un primo training della rete che richiederà parecchio tempo (Fig. 3), a seconda del calcolatore utilizzato⁷.



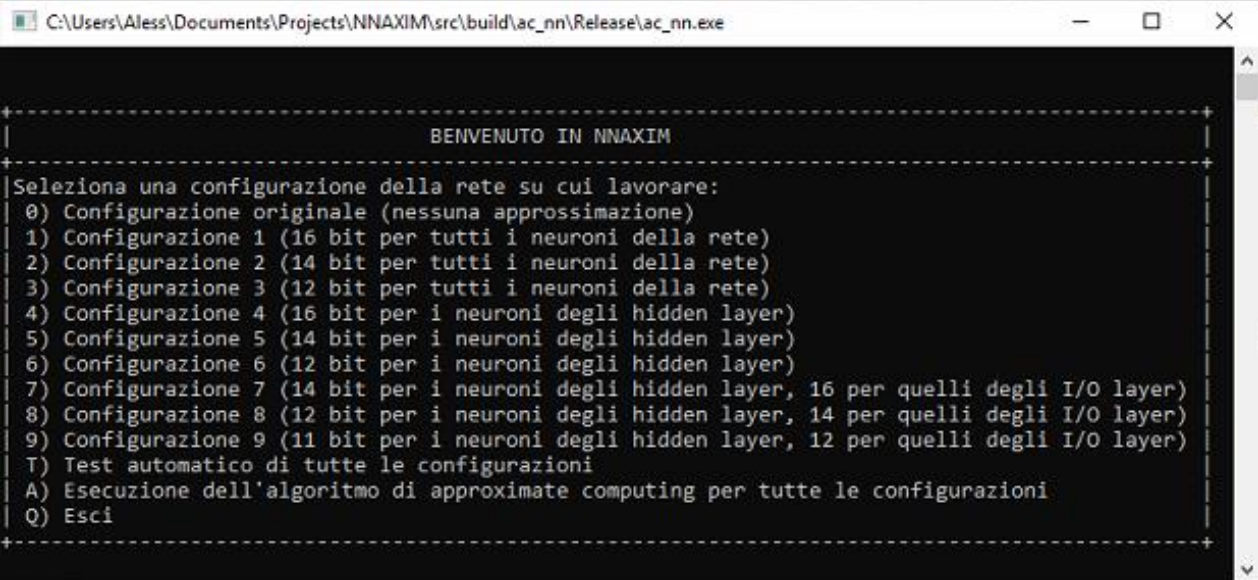
Figura 3. Schermata di avvio nel caso in cui non è presente il file contenente i pesi della rete originale.

⁶ La versione *sinus_fit* presenta comunque delle feature in meno rispetto a quella di base (*ac_nn*).

⁷ Il tool è stato testato solo su Windows ed è necessario utilizzare Visual Studio 2015 o versioni successive e Cmake per poterlo compilare.

Se i pesi della rete originale sono disponibili, all'avvio comparirà l'interfaccia base dell'applicazione (Fig. 4). Da questa è possibile effettuare principalmente 3 azioni:

1. Scegliere una configurazione sulla quale lavorare. In uno step successivo sarà possibile:
 - a. indicare se si vuole effettuare un allenamento della rete con i pesi approssimati e successivo salvataggio degli stessi
 - b. indicare se si vuole solo testare la rete con la configurazione scelta.
2. Eseguire il test automatico di tutte le configurazioni. Il test di una configurazione sarà effettuato sui pesi salvati nel relativo file. Se questo file non esiste verranno presi i pesi originali, verranno troncati caricati nella rete e quindi sarà eseguito il test.
3. Eseguire l'algoritmo di Approximate Computing sulla rete neurale. Questa funzionalità, dopo un primo test della rete nella sua configurazione originale, farà partire 9 iterazioni (una per ogni configurazione) nelle quali saranno svolte le 4 fasi descritte in 3.3. Alla fine delle iterazioni, i risultati dei test saranno mostrati a video e saranno anche salvati su file (log.txt).
4. Uscire dall'applicazione.



```
C:\Users\Aless\Documents\Projects\NNAXIM\src\build\ac_nn\Release\ac_nn.exe

+-----+
|                               |
|          BENVENUTO IN NNAXIM  |
|                               |
+-----+
| Seleziona una configurazione della rete su cui lavorare: |
| 0) Configurazione originale (nessuna approssimazione)  |
| 1) Configurazione 1 (16 bit per tutti i neuroni della rete) |
| 2) Configurazione 2 (14 bit per tutti i neuroni della rete) |
| 3) Configurazione 3 (12 bit per tutti i neuroni della rete) |
| 4) Configurazione 4 (16 bit per i neuroni degli hidden layer) |
| 5) Configurazione 5 (14 bit per i neuroni degli hidden layer) |
| 6) Configurazione 6 (12 bit per i neuroni degli hidden layer) |
| 7) Configurazione 7 (14 bit per i neuroni degli hidden layer, 16 per quelli degli I/O layer) |
| 8) Configurazione 8 (12 bit per i neuroni degli hidden layer, 14 per quelli degli I/O layer) |
| 9) Configurazione 9 (11 bit per i neuroni degli hidden layer, 12 per quelli degli I/O layer) |
| T) Test automatico di tutte le configurazioni          |
| A) Esecuzione dell'algoritmo di approximate computing per tutte le configurazioni |
| Q) Esci                                                |
+-----+
```

Figura 4. Schermata di avvio nel caso in cui è presente il file contenente i pesi della rete originale.

4 Valutazione dei risultati ottenuti

Per ognuna delle configurazioni approssimate è stata valutata l'accuratezza, in termini percentuali di classificazioni corrette su quelle totali effettuate sul dataset di test, prima e dopo l'esecuzione di un retraining della rete neurale. Tali risultati sono stati confrontati con l'accuratezza della rete nella sua configurazione originale (non approssimata) in modo da poter definire la perdita di accuratezza percentuale (Fig. 5). Questo dato, insieme alla quantità di bit risparmiati, ha consentito di valutare accuratamente il *tradeoff* memoria risparmiata – accuratezza persa per ogni configurazione (Fig. 6 e Fig. 7).

Come si può notare dai grafici di seguito riportati, i risultati ottenuti rispecchiano il comportamento atteso dalla rete, ossia un impatto sull'accuratezza crescente all'aumentare dell'entità dell'approssimazione (ovvero all'aumentare dei bit risparmiati). Inoltre, tale impatto è maggiore nel momento in cui vengono approssimati anche i layer di input e output della rete a conferma dei risultati esposti in [1] e [2] riguardo alla maggiore resilienza degli hidden layer.

Un'ulteriore conferma delle aspettative arriva da risultati ottenuti dopo il retraining: con una giusta quantità di epoche di training effettuato dopo l'approssimazione, l'accuratezza della rete cresce (nella maggior parte dei casi), consentendo così di massimizzare il *tradeoff* memoria risparmiata – accuratezza persa. È bene notare come in alcuni casi il retraining consente di raggiungere o addirittura superare le performance della configurazione originale, mentre in altri il retraining non porta nessun miglioramento. Questo risultato in realtà è dovuto all'uso di una rete neurale non sufficientemente allenata e performante (il dataset di training e di test sono relativamente piccoli, inoltre non è stato possibile allenare la rete in maniera ottimale a causa delle limitate risorse computazionali).

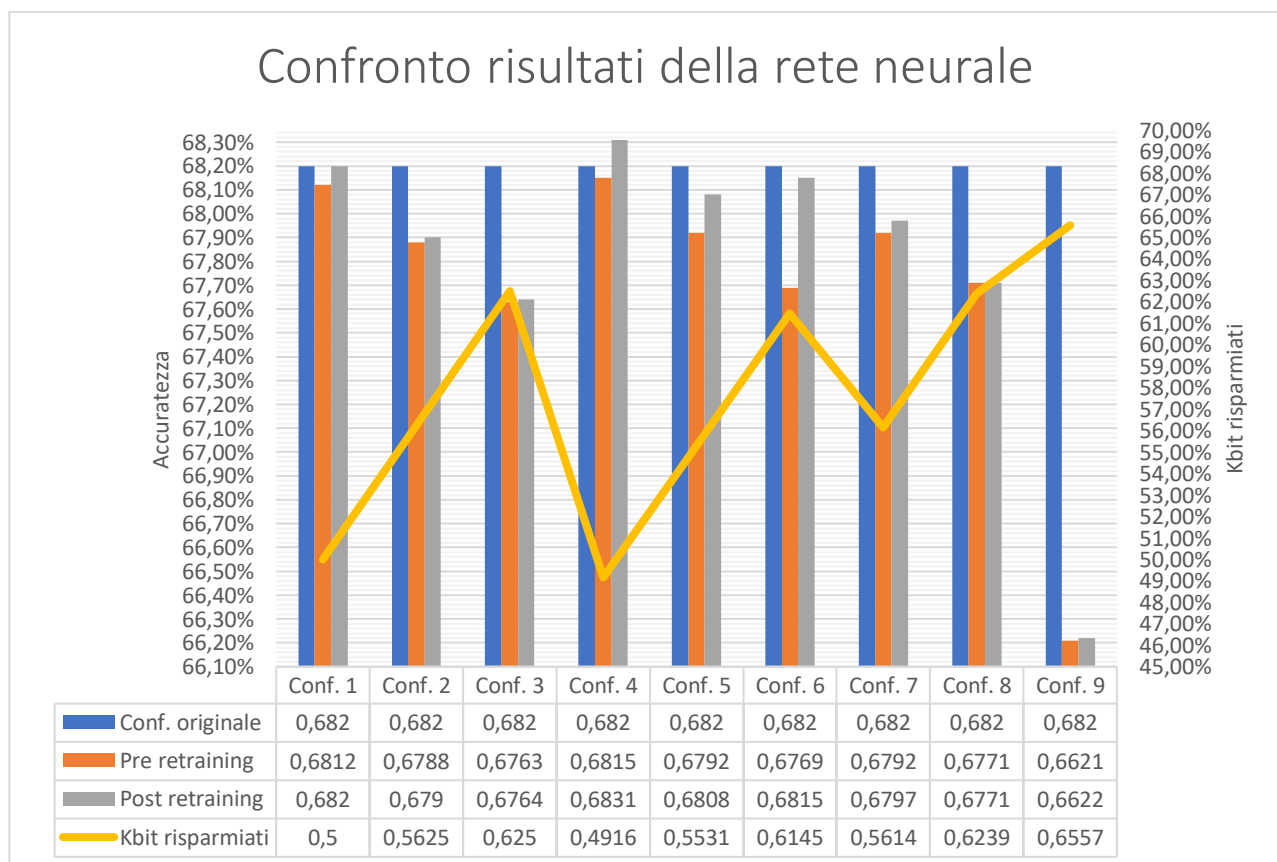


Figura 5. Grafico del confronto dei risultati ottenuti dalla rete (valori normalizzati all'unità nella legenda). Si può notare come le configurazioni che presentano il migliore *tradeoff* memoria risparmiata - accuratezza persa dopo il retraining siano quelle in cui l'approssimazione è applicata solo ai pesi dei neuroni degli hidden layer, a conferma di quanto esposto in [1] e [2]

+ ===== TEST RESULTS ===== +			
CONFIGURATION	ACCURACY LOSS BEFORE RETRAIN	ACCURACY LOSS AFTER RETRAIN	SAVED BITS
1	0.08%	0.00%	2329248
2	0.32%	0.30%	2620404
3	0.57%	0.56%	2911560
4	0.05%	-0.11%	2290336
5	0.28%	0.12%	2576628
6	0.51%	0.05%	2862920
7	0.28%	0.23%	2615540
8	0.49%	0.49%	2906696
9	1.99%	1.98%	3054706
+ ===== +			

Figura 6. Tabella dei risultati stampata alla fine dell'esecuzione dell'algoritmo di NNAXIM

	Classifica prima del retraining	Classifica dopo il retraining
1°	Configurazione 4	Configurazione 6
2°	Configurazione 1	Configurazione 4
3°	Configurazione 7	Configurazione 1
4°	Configurazione 5	Configurazione 5
5°	Configurazione 2	Configurazione 7
6°	Configurazione 8	Configurazione 2
7°	Configurazione 6	Configurazione 8
8°	Configurazione 3	Configurazione 3
9°	Configurazione 9	Configurazione 9

Figura 7. Classifica dei tradoff delle configurazioni (Kbit risparmiati rispetto all'accuratezza persa) di approssimazione. Il retraining altera notevolmente la classifica, in particolare per quanto riguarda le configurazioni in cui l'approssimazione è applicata solo ai neuroni degli hidden layer: tramite il self-healing la perdita di accuratezza dovuta all'approssimazione dei pesi dei neuroni più resilienti riesce a sanarsi.

5 Conclusioni

I risultati ottenuti confermano le grandi potenzialità delle tecniche di Approximate Computing quando applicate alle reti neurali. La tecnica utilizzata ha consentito, nelle approssimazioni migliori, di risparmiare un grande quantitativo di bit per la memorizzazione dei pesi della rete, a fronte di una accettabile perdita di accuratezza, che consente ancora di utilizzare la rete all'interno dell'applicazione per cui questa è stata pensata.

Adattando il codice sorgente del tool NNAXIM è possibile simulare l'effetto dell'utilizzo della tecnica di Approximate Computing descritta su qualsiasi rete neurale, ottenendo informazioni che possono tornare molto utili in fase di progettazione per sistemi di questo tipo. È infatti possibile definire semplicemente una diversa struttura per la propria rete, i parametri di training e valutazione della rete, e un qualsiasi numero di configurazioni per l'approssimazione. Infine, sarebbe possibile ampliare le potenzialità del tool aggiungendo nuove feature per la simulazione di altre tecniche di Approximate Computing.

Bibliografia

- [1] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna Palem, Olivier Temam, Chengyong Wu, "*Leveraging the Error Resilience of Machine-Learning Applications for Designing Highly Energy Efficient Accelerators*", 2014.
- [2] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy and Anand Raghunathan, "*AxNN: Energy-Efficient Neuromorphic Systems using Approximate Computing*", 2014.
- [3] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan and Qiang Xu, "*ApproxANN: An Approximate Computing Framework for Artificial Neural Network*", 2015.