0.1 OpenLearn documentazione

Questa parte della relazione è dedicata all'illustrazione e al commento della struttura e delle parti principali del componente principale del progetto, la libreria di machine learning sviluppata per la creazione, l'addestramento e l'utilizzo dei modelli neuronali. La libreria è stata sviluppata per un utilizzo general-purpose, quindi non finalizzata alla risoluzione dello specifico problema proposto, il quale è stato utilizzato per testare la risposta di varie strutture neurali alla risoluzione di un problema con soluzione nota ma attraverso l'elaborazione di dati di ingresso fortemente affetti da rumore.
Nella documentazione verranno mostrate soltanto le caratteristiche principali della libreria, le classi, le strutture e i metodi più importanti effettivamente utilizzati benche la libreria ne presenti altri predisposti per future revisioni ed espansioni.

0.1.1 Le librerie importate

Le prime librerie importate sono le librerie per l'utilizzo del linguaggio C/CUDA, proprietario di Nvidia, per la programmazione delle schede della stessa azienda.

```
1 ///////////////CUDA INCLUDES///////////////
2 #include "cuda.h"
3 #include "cuda_runtime.h"
4 #include <device_functions.h>
5 #include "device_launch_parameters.h"
```

Listing 1: librerie cuda

Sono state importate poi diverse librerie standard del linguaggio C++, principalmente:
la classe vector è stata usata per l'incapsulamento e la manipolazione dei dati.
le classi *stream sono state utilizzate per il salvataggio e il caricamento dei modelli su file di testo.
la libreria windows per l'utilizzo delle funzioni di timing ad alta precisione.

```
1
2 #include "stdafx.h"
3 #include <iostream>
4 #include <stdio.h>
5 #include <fstream>
6 #include <sstream>
7 #include <string>
8 #include <windows.h>
9 #include <limits>
10 #include <math.h>
11 #include <vector>
12 #include <list>
13 #include <time.h>
14 #include <algorithm>
15 #include <assert.h>
```

Listing 2: librerie STD

0.1.2 classi di creazione e addestramento modelli su CPU

Di seguito sono illustrate le strutture che formano lo scheletro dei modelli, tali strutture sono state pensate per essere completamente modulari e per permettere la realizzazione di diversi modelli, dalle più semplici feed-forward, alle più complesse ricorrenti. Tale approccio permette la massima maneggevolezza della struttura sacrificando l'efficienza che deriva da una computazione matriciale.

```
1 //prototipo struttura neuron
2 struct Neuron;
3 //dichiarazione puntatore a neuron
4 typedef Neuron *ptNeuron;
5
6 //La struttura arc, presente in array nella struttura Neuron
7 //conserva i dati delle connessioni e il puntatore alla struttura Neuron puntata
8 struct arc {
9   ptNeuron target = nullptr;
10   float weight = 0;
11   float oldDelta = 0;
12   //bool enabled = true;
13 };
14
15 //La struttura interArc è stata creata per l'addestramento MLP supportato da un rete ricorrente
16 //fornisce un arco di interconnessione tra i neuroni di 2 reti
```

```
struct interArc {
  ptNeuron target = nullptr;
  ptNeuron base = nullptr;
};

//La struttura Neuron è l'unità base delle strutture delle reti e ne conserva tutti i parametri
struct Neuron {
  //OutArcs è il vettore di connessioni ai neuroni degli strati successivi
  // to inizialize: = new vector<arc>(10);
  vector<arc> OutArcs;
  // numero di archi in uscita
  u_int numOutArcs = 0;
  //numero di archi in ingresso
  u_int numInArcs = 0;
  //indice riga del neurone
  u_int layer = 0;
  //indice della colonna del neurone
  u_int column = 0;
  //peso del bayes
  float bayes = 0.01f;
  //ultima variazione del peso
  float oldBayesDelta = 0;
  // vettore delle interconnessioni temporali
  //vector<float> timeBayes;
  // vettore contenente la percentuale di influenza relativa ad ogni input
  vector<float> influenceInput;
  // vettore contenente la percentuale di influenza relativa all'errore  retropropagato da ogni
      ↪ output
  vector<float> influenceOutput;
  //potenziale attuale del neurone
  float output = 0;
  //somma in valore assoluto degli input del neurone
  float absOutSum = 0;
  //somma in valore assoluto delle variazioni dei pesi
  float absDeltaSum = 0;
  //errore di retropropagazione
  float BPerr = 0;
  //ogni neurone è contraddistinto da un indice unico che si riferisce alla sua posizione
  int neurIdx = 0;
};

//La struct Layer è stata realizzata per contenere un layer di neuroni
struct Layer {
  vector<Neuron> Neurons;
  u_int numNeurons = 0;
};

//Omap conserva la scala di conversione tra l'output
struct Omap {
  float maxValue = 1;
  float minValue = 0;
};

// struttura necessaria per l'inserimento di un nuovo neurone
struct conMap {
  u_int startLyr;
  u_int startCol;
  u_int arcRef;
  u_int targetLyr;
  u_int targetCol;
};

//conserva una serie temporale prima dell'accomulazione in una struct Dataset
struct timeSeries {
  list<float> evento;
};

//struttura contenente il singolo esempio
struct example {
```

```
  vector<float> input;
  vector<float> Doutput;
};

//Struttura contenente l'intero Dataset di una rete
struct Dataset {
  vector<example> trainingSet; // vettore esempi del training set
  vector<example> validationSet; // vettore esempi del validation set
  float triningErr = 0; //percentuale di errore training set
  float validationErr = 0; //percentuale di errore validation set
};
```

Listing 3: struct strutturali del modello

La classe DatasetCore è stata realizzata per permettere l'estrazione dei dataset da vari formati, riportandoli nello standard di utilizzato dalle classi che li utilizzano per effettuare gli addestramenti.

```
class DatasetCore {
public:
  // lista dei dataset
  list<Dataset> Datasets;

  // costruttore DatasetCore
  DatasetCore() { ... }

  ////////////////////MANIPOLAZIONE DATASET/////////////////////////////////////
  //funzione che consente la lettura di serie temporali da csv e le ristruttura in un dataset
  void readTimeSeriesCsv(string filename, int outStep, int inEx, float trainingPerc) { ... }

  //estrae un dataset dalla lista interna della classe, la parte training se il flag e true
  // o la parte validation se il flag è false
  vector<example> getDataset(int n, bool training = true) { ... }
  ////////////////////////////////////////////////////////////////////////////////

  ////////////////////////ALTRE FUNZIONI//////////////////////////////////////////
  //metodo interno per contare le righe di un file
  int coutnRow(string filename) { ... }
```

Listing 4: class DatasetCore

La classe network fornisce le variabili e i metodi generali che la maggior parte degli oggetti rete utilizza.

```
class Network {

public:
  // vettore di struct layer
  vector<Layer> Layers;
  // vettore di esempi per l'apprendimento
  vector<example> examples;
  // vettore contenente i valori di rimappatura del'output della rete
  vector<Omap> map;

  // nome del file contenente la struttura della rete
  string genoma = "";
  // Layer nella rete compresi strato input e strato output
  u_int nLayers = 0;
  // numero totale neuroni nella rete
  u_int nNeurons = 0;
  //numero totale di tutti gli archi presenti nella rete
  u_int nArc = 0;

  //costruttore
  Network(string filename) {
    genoma = filename;
  }

  ////////////////FUNZIONE COSTRUZIONE RETE DA FILE/////////////////////////////////
  //dato il nome del file contenete la struttura carica l'oggetto corrispondente
  void getNetParams() { ... }
  ////////////////////////////////////////////////////////////////////////////////

```

```
////////////////////FUNZIONE COSTRUZIONE DATASET DA FILE//////////////////////////////
//dato il file contenente un dataset prestrutturato correttamente lo carica nel modello
void getDataset(string filename) { ... }
////////////////////////////////////////////////////////////////////////////////////

////////////////////SALVA RETE SU FILE//////////////////////////////////////////////
//salva la rete sul file specificato
void saveNet(string filename = "") { ... }
////////////////////////////////////////////////////////////////////////////////////

////////////////////////SALVA DATASET SU FILE///////////////////////////////////////
//salva il datset sul file specificato
void saveDataset(string filename) { ... }
////////////////////////////////////////////////////////////////////////////////////

////////////////////// FUNZIONI MODIFICA RETE///////////////////////////////////////
//elimina un arco dati i parametri diriferimento
void deleteArc(int Nlayer, int Ncolumn, int targetLayer, int targetColumn) { ... }

//elimina un neurone, compresi tutti gli archi ad esso connesso
void deleteNeuron(int Nlayer, int Ncolumn) { ... }

//aggiunge un arco dati i riferimenti al neurone di partenza e di arrivo
void addArc(int Nlayer, int Ncolumn, int targetLayer, int targetColumn) { ... }
////////////////////////////////////////////////////////////////////////////////////

////////////////////////FUNZIONI VARIE//////////////////////////////////////////////
float DsigOut(int Layer, int Neuron) { ... }
float sigmoid(float x) { return 1 / (1 + exp(-x)); } // Sigmoide
float logit(float x) { return log(x / (1 - x)); } // funzione sigmoide inversa
float gaussian(float x, float mu, float var) { ... }
void WeightsStamp(string mode = "a") { ... }
  //m - stampa le medie dei pesi di ogni layer
  //a - stampa tutti i pesi della rete con alcuni parametri di apprendimento
  //w - stampa tutti i pesi con il riferimento riga colonna al target
  //fc - stampa le medie dei gruppi di pesi tra due layer

//funzione sigmoide
void sigLayer(int lyr) { ... }

//applica il bayes all'output di ogni neurone del dato layer
void bayesLayer(int lyr, bool absSum = false) { ... }

// esegue il reset del potenziale di tutti i neuroni della rete
void resetPotential() { ... }

// esegue il reset della sommatoria di ogni input in valore assoluto di ogni neurone
void resetAbsSumPotenzial() { ... }

// esegue il reset della sommatoria di ogni input in valore assoluto di ogni neurone
void resetAbsSumDelta() { ... }

// esegue il reset dell'errore retropropagato in ogni neurone
void resetBPerr() { ...}

//resetta il valore ID nei neuroni di una rete (necessario per modifiche strutturali)
void resetNeuronsID() { ... }

// salva su vettore i riferimenti numerici delle connesioni verso il layer specificato
vector<conMap> saveConsTowardsLyr(int Layer) { ... }

// ricarica i riferimenti numerici delle connesioni verso il layer specificato
void loadConsTowardsLyr(vector<conMap> con) { ... }

//elimina il contenuto l'oggetto dataset dell'oggetto rete
void ClearDataset() { ... }

//generazione di una serie storica del seno (DEBUG)
void genTestDataset(int nExe, int nIn, int nOut, float step, int type, float offset) { ... }
```

```
99
100   //esegue il settaggio della mappatura di output dell'oggetto
101   void setNetMap(float max, float min) { ... }
102
103   //restituisce il valore normalizzato dell'output della rete
104   float reverseMap(int neur, float val) { ... }
105
106   // crea un vettore di n elementi successivi e li disordina
107   // creazione di una tabella di accesso casuale per un secondo vettore
108   vector<u_int> casualVector(int in, int start = 0) { ... }
109
110   //esegue il mescolamento degli elementi all'interno di un oggetto vector
111   /*vector<T, A>*/
112   template<typename T, typename A>
113   void shackeVector(vector<T, A> const& vec) { ... }
114
115   //ricalcola gli ID dei neuroni dopo il reset (da eseguire dopo delle modifiche strutturali)
116   void refreshNeurIdx() { ... }
117
118   //applica all'intero dataset un valore di offset
119   void datasetOffset(float offset) { ... }
120   ////////////////////////////////////////////////////////////////////////////////
121
122   ////////////////////////ACCESSO A VARIABILI PRIVATE//////////////////////////////
123   int numLayers() { ... }
124   int numNeurons(int Layer) { ... }
125   int numCon(int Layer, int Neuron) { ... }
126   int numInCon(int Layer, int Neuron) { ... }
127   int getConTargetLyr(int Layer, int Neuron, int Arc) { ... }
128   int getConTargetCol(int Layer, int Neuron, int Arc) { ... }
129   float getWeight(int Layer, int Neuron, int Arc) { ... }
130   float getDeltaWeight(int Layer, int Neuron, int Arc) { ... }
131   float getOutput(int Layer, int Neuron) { ... }
132   float getBPerr(int Layer, int Neuron) { ... }
133   ptNeuron getTarget(int Layer, int Neuron) { ... }
134   ptNeuron getConTarget(int Layer, int Neuron, int Conn) { ... }
135   int getOutConTargetID(ptNeuron base, ptNeuron target) { ... }
136   ////////////////////////////////////////////////////////////////////////////////
137
138   ////////////////////////WINDOWS HIGH PRECISION TIMING//////////////////////////
139   BOOL WINAPI QueryPerformanceCounter(_Out_ LARGE_INTEGER *lpPerformanceCount);
140   BOOL WINAPI QueryPerformanceFrequency(_Out_ LARGE_INTEGER *lpFrequency);
141   inline long long PerformanceCounter() noexcept
142   {...}
143   inline long long PerformanceFrequency() noexcept
144   {...}
145   ////////////////////////////////////////////////////////////////////////////////
146 };
```

Listing 5: class Network

La classe MLP (multi-layer perceptron) figlia della classe Network completa il corredo della madre con tutta una serie di strumenti e variabili che supporta l'addestramento e l'utilizzo di questa tipologia di reti.

```
1
2 class MLP : public Network {
3
4 public:
5   // tempo di esecuzione medio in millisecondi
6   float NetPerformance = 0;
7
8   //errore percentuale medio associato alla rete
9   float NetErrPercent = 0;
10
11  //costruttore
12  MLP(string filename) :Network(filename) {};
13
14  ////////////////////FUNZIONI CREAZIONE RETE/////////////////////////////////////
15  //permette la costruzione rapida di una rete MLP quadrata con dimensioni date
```

```
  void qubeNet(int Nlayers, int Ncolumns, int input, int output, bool c, float initValue = 0.01f ) { ... }

  //CREAZIONE RETE QUADRATA COMPLETAMENTE CONNESSA
  //permette la costruzione di una rete MLP quadrata con dimensioni date in cui ogni neurone
  //è connesso a tutti i neuroni di tutti gli strati successivi.
  //(a parità di neuroni tale modello è più pesante ma tipicamente ha un aprrendimento più rapido)
  void qubeNetFC(int Nlayers, int Ncolumns, int input, int output, bool c, float initValue = 0.01f) { ... }

  //CREAZIONE RETE CUSTOM
  //permette la creazione di una rete MLP con la larghezza di ogni layer variabile
  //indicando tali larghezze con un apposito vettore dato come argomento
  void customNet(int Nlayers, vector<int> Ncolumns, float conFill) { ... }
  ////////////////////////////////////////////////////////////////////////////////////

  ////////////////////STIMOLAZIONE RETE/////////////////////////////////////////////
  //procedura di propagazione dell'informazione
  //dati i riferimenti al vettore contenete gli input e ad un vettore di output esegue
  //il processamento degli input e carica l'output derivante sul vettore indicato
  void inputNet(vector<float> &input, vector<float> &output) { ... }

  //esegue una propagazione dell'informazione salvando lo storico di propagazione degli input
  //tale funzione viene utilizzata per l'apprendiment strutturale
  void inputNetProfiler(vector<float> &input, vector<float> &output) { ... }
  ////////////////////////////////////////////////////////////////////////////////////

  ////////////////////////FUNZIONI DI ADDESTRAMENTO MLP/////////////////////////////////
  //Algoritmo di addestramento Back-propagation su tutto il dataset caricato
  void BP(int iter, float eps, float beta, float ErrPercent) { ... }

  //esecuzione del backpropagation per un solo esempio
  //specificando il particolare esempio
  void oneBP(float eps, float beta, example e) { ... }
  ////////////////////////////////////////////////////////////////////////////////////

  ////////////////////////ALTRE FUNZIONI MLP/////////////////////////////////////////////
  //inizializza i vettori di benchmark presenti nei neuroni per l'apprendimento strutturale
  void initVectorsProfiler() { ... }

  //resetta a zero tutti i vettori di profilazione esclusi layer input e output
  void resetVectorsProfiler(bool inInfl, bool outInfl) { ... }

  //stampa a schermo l'influenza degli input per ogni uscita
  void stampInputInfluences(bool all = false) { ... }

  //stampa a schermo l'influenza degli errori degli output per ogni ingresso
  void stampOutputErrorPropagation(bool all = false) { ... }

  //dal riferimento al nodo target e dall'indice del vettore restituisce il puntatore al neurone base
  ptNeuron basePosfromtarget(ptNeuron target, int k) { ... }

  //dato un neurone e l'indice di un suo arco restituisce l'indice di quella connessione all' interno del        //vettore influenceInput all'interno del neurone target
  int idBaseConReftoTargetInfl(ptNeuron base, int arc) { ... }
  ////////////////////////////////////////////////////////////////////////////////////

};
```

Listing 6: class MLP

0.1.3 altre classi non utilizzate nel progetto

Le seguenti classi sono state iniziate e verranno completate in futuro, La classe Hopfield supporta (ancora in parte) la creazione l'addestramento e l'utilizzo delle reti ricorrenti, ovvere reti con connessioni tra i neuroni che si retropropagano, ovvero che portano potenziali di uscita di neuroni nello strato output e hidden al tempo (t), ad altri neuroni nel tempo (t+1).
La classe structural learning è stata realizzata e testa, ma presenta notevoli inefficienze che non permettono di utilizzarla in questa forma su modelli di grandi dimensioni, tale classe comunque è stata realizzata per effettuare un addestramento che mira a modificare la struttura della rete durante l'addestramento eliminando le connessioni che risultano essere mediamente quelle che persistono nell'introduzione di errore rispetto alle altre.

```
class Hopfield : public Network { ... };

class StructuralLearning { ... };
```

Listing 7: altre classi

0.1.4 Classi e kernel per l'interfacciamento con la GPU

In questa sezione sono stati implementati i kernel della GPU, ovvero le funzioni scritte in C/CUDA, che vengono eseguite sulla scheda grafica. tali funzioni sono contraddistinte dalla keyword __global__, e richiedono come argomenti gli indirizzi di memoria corrispondenti ai dati precaricati sulla global memory della GPU. Il richiamo dei kernel dal codice C++ richiedono la sintassi speciale:
CUDAapplyWeightCorrections «<numOfBlocksA, ThxBlock »> (...);
si vede infatti la sequenza di caratteri «<A,B»> dove A è il numero di blocchi su cui deve essere eseguita la funzione mentre B rappresenta il numero di thread per ogni blocco.

```
////////////////////////////////////CUDA Kernels/////////////////////////////////////

//resetta il valore di una variabile all'interno della scheda grafica
__global__ void CUDAresetVar(float *val) {
  *val = 0;
}

//applica ad ogni arco della rete la correzione del peso
__global__ void CUDAapplyWeightCorrections(float eps, float *NeuronOut, float *BPerr, float *
    ↪ weights, int *ArcIn, int *ArcOut, int nArcs) {
  unsigned int i = (blockIdx.x * blockDim.x) + threadIdx.x;
  if (i < nArcs) {
    weights[i] += -eps * BPerr[ArcIn[i]] * NeuronOut[ArcOut[i]];
  }
}

//applica le correzioni hai bayes dei neuroni
__global__ void CUDAapplyBayesCorrections(float eps, float *BPerr, float *Bayes, int startN, int
    ↪  endN) {
  unsigned int i = startN + (blockIdx.x * blockDim.x) + threadIdx.x;
  if (i <= endN) {
    Bayes[i] += -eps * BPerr[i];
  }
}

//retropropaga l'errore nella rete
__global__ void CUDAPropagationErr(float *BPerr, float *weights, float *NeuronOut, int *ArcIn,
    ↪ int *ArcOut, int startA, int endA) {
  unsigned int i = startA + (blockIdx.x * blockDim.x) + threadIdx.x;

  //retropropago l'errore dai neuroni successivi
  if (i <= endA) {
    //BPerr[ArcOut[i]] += BPerr[ArcIn[i]] * weights[i];
    atomicAdd(&BPerr[ArcOut[i]], BPerr[ArcIn[i]] * weights[i]);
  }
}

//moltiplica l'errore delle uscite per la derivata puntuale della sigmoide
__global__ void CUDAoutDiff(float *BPerr, float *NeuronOut, int startN, int endN) {
  int i = startN + (blockIdx.x * blockDim.x) + threadIdx.x;
  if (i <= endN) {
    BPerr[i] *= NeuronOut[i] * (1 - NeuronOut[i]);
  }
}
```

```
42
43 //calcola l'errore dei neuroni dello strato output
44 __global__ void CUDAoutputErr(float *NeuronOut, int OutputRef, int numNeurons, int inputN, float
       ↪  *BPerr, float *examples, int exampleRef, float *mapMaxOut, float *mapMinOut, float *
       ↪ MeanErr) {
45
46   unsigned int i = (OutputRef) + (blockIdx.x * blockDim.x) + threadIdx.x; //indice di
         ↪ scorrimento vettori: NeuronOut, BPerr,
47   unsigned int e = (exampleRef + inputN) + (blockIdx.x * blockDim.x) + threadIdx.x; //indice di
         ↪ scorrimento vettori: examples
48   unsigned int m = (blockIdx.x * blockDim.x) + threadIdx.x; // indice di scorrimento vettori:
         ↪ mapMaxOut, mapMinOut
49   //if (i == 0) *MeanErr = 0;
50   if (i < numNeurons) {
51
52     float delta = mapMaxOut[m] - mapMinOut[m];
53     BPerr[i] = (NeuronOut[i] - ((examples[e] - mapMinOut[m]) / delta)) * NeuronOut[i] * (1 -
         ↪ NeuronOut[i]); // formula valida solo per i neuroni di uscita
54     //atomicAdd(MeanErr, (abs((((NeuronOut[i] * delta) + mapMinOut[m]) - examples[e]) / examples
         ↪ [e]))*100.0f);
55     atomicAdd(MeanErr, abs((((NeuronOut[i] * delta) + mapMinOut[m]) - examples[e])/ examples[e])
         ↪  * 100.0f);
56     // calcolo l'errore percentuale sulla singola uscita e lo sommo
57     //questo rappresenta uno dei punti più inefficienti dell'algoritmo,
58     //la funzione AtomicAdd infatti blocca la parallizzazione dell'algoritmo per tutti
59     //i thread che cercheranno di accedere a tale variabile riportando solo per questi
60     //l'esecuzione ad un modello sequenziale mettendoli in attesa
61   }
62 }
63
64 //resetta un dato vettore
65 __global__ void CUDAresetVector(float *vect, int size) {
66   unsigned int i = (blockIdx.x * blockDim.x) + threadIdx.x;
67   if (i < size) vect[i] = 0.0f;
68 }
69
70 //imposta i valori di output dei neuroni di input al valore dell'esempio del dataset
71 __global__ void CUDAsetInput(float *NeuronOut, int inputN, int exampleRef, float *example) {
72   unsigned int i = (blockIdx.x * blockDim.x) + threadIdx.x;
73   if (i < inputN)NeuronOut[i] = example[exampleRef + i];
74 }
75
76 //imposta i valori di output dei neuroni di input al valore specificato
77 __global__ void CUDAsetSingleInput(float *NeuronOut, int inputN, float *example) {
78   unsigned int i = (blockIdx.x * blockDim.x) + threadIdx.x;
79   if (i < inputN)NeuronOut[i] = example[i];
80 }
81
82 //applica la sigmoide ai potenziali dei neuroni in un dato intervallo (uno strato)
83 __global__ void CUDAsigLayer(float *NeuronOut, int start, int end) {
84   unsigned int i = start + (blockIdx.x * blockDim.x) + threadIdx.x;
85   if (i <= end) {
86     NeuronOut[i] = 1 / (1 + expf(-NeuronOut[i]));
87   }
88 }
89 //aggiunge all'output del neurone il contributo del bayes
90 __global__ void CUDAbayesInput(float *NeuronOut, float *Bayes, int start, int end) {
91   unsigned int i = start + (blockIdx.x * blockDim.x) + threadIdx.x;
92   if (i <= end) {
93     NeuronOut[i] += Bayes[i];
94   }
95 }
96
97 //propaga l'informazione dai neuroni dello strato input a quello di output
98 __global__ void CUDAlayerInput(float *weights, int *ArcIn, int *ArcOut, float *NeuronOut, int
       ↪ start, int end) {
99   unsigned int i = start + (blockIdx.x * blockDim.x) + threadIdx.x;
100  if (i <= end) {
101    atomicAdd(&NeuronOut[ArcIn[i]], NeuronOut[ArcOut[i]] * weights[i]);
```

```
102     //addizione bloccante non permette ad altri thread di sovrascrivere il falore
103     // finche l'operazione non è completata
104   }
105 }
106
107 ////////////////////////////////////////////////////////////////////////////////
```

Listing 8: cuda kernels

Questa classe è l'interfaccia di collegamento con la GPU, la quale conserva i metodi di caricamento dei dati strutturali della rete, e gli esempi di addestramento sulla GPU.

```
1  //api di interfacciamento alla GPU
2  class CUDAcore {
3  public:
4    //Device specs struct (contiene le specifiche della GPU)
5    cudaDeviceProp prop;
6    int GpuID = 0;
7
8    //struttura contenente i puntatori alle aree di memoria conenenti i parametri della rete nella
       ↪  GPU
9    struct devNetParams {
10     float *weights = 0;
11     int *ArcIn = 0;
12     int *ArcOut = 0;
13     float *examples = 0;
14     float *NeuronOut = 0;
15     float *Bayes = 0;
16     float *BPerr = 0;
17     float *mapMaxOut = 0;
18     float *mapMinOut = 0;
19     int *NeurInLyr = 0;
20     int *priority = 0;
21     float *MeanErr = 0;
22     float *InputRT = 0;
23   }gpuNetParams;
24
25   vector<float> weights; //pesi della rete
26   vector<int> ArcIn; //target dell'n-esimo arco
27   vector<int> ArcOut; //base dell'n-esimo arco
28   vector<float> NeuronOut; //vettore contenente l'output dei neuroni
29   vector<float> Bayes; //vettore contenente i bayes dei neuroni
30   vector<float> BPerr; // vettore contenete gli errori retropropagati
31   vector<float> mapMaxOut; //vettore contenente il massimo valore degli output
32   vector<float> mapMinOut; //vettore contenente il minimo valore degli output
33   vector<int> priority; // vettore contenente i punti di sincronizzazione dei thread
34   vector<int> NeurInLyr; //vettore contenente gli indici dell'ultimo neurone di ogni layer
35   vector<float>examples; //vettore degli esempi
36   float MeanErr = 0; //veriabile contenente l'errore medio percentuale della rete
37   int inputN, outputN; //passo di esecuzione elementi del vettore esempi
38
39   CUDAcore(int nGpu) {
40     GpuID = nGpu;
41     checkCuda(cudaGetDeviceProperties(&prop, nGpu)); // carica lo struct cudaDeviceProp prop con
         ↪  le caratteristiche della GPU con indice 0
42   }
```

Listing 9: classe di interfaccia alla GPU

La seguenti funzioni interne della classe di interfaccia alla GPU, rappresentano il metodo di conversione della struttura della rete dalla classe MLP alla classe CudaCore, e viceversa, tale conversione porta il modello da una struttura reticolare a puntatori in una struttura a vettori lineare.
Tale struttura divide i vari parametri della rete in vettori lineari, ogni tipo di elemento viene raggruppato in un unico vettore, mentre le connessioni precedentemente rappresentate da puntatori divengono vettori a loro volta che fungono da tabelle di accesso ordinate, scorrendo dal primo elemento all'ultimo si trovano tutti gli elementi su cui eseguire sequenzialmente le operazioni da svolgere per eseguire la propagazione. La complessità di tale operazione sta proprio nello svolgimento della struttura della rete per poter eseguire con efficienza la computazione sulla scheda grafica, la quale deve essere più "machine-friendly" possibile, data l'enorme quantità di operazioni che deve essere svolta per eseguire l'addestramento. Sono indicate anche altre funzioni di conversione per Hopfield e per il dataset il quale ha bisogno anch'esso di serializzazione per l'utilizzo nelle successive funzioni.

```
//copia del modello da MLP class a CudaCore class
void cudaNetCopyMLP(MLP *pt) {
  cout << "copying the net into CUDAcore.." << endl;
  weights.resize(pt->nArc);
  ArcIn.resize(pt->nArc);
  ArcOut.resize(pt->nArc);
  NeuronOut.resize(pt->nNeurons);
  Bayes.resize(pt->nNeurons);
  BPerr.resize(pt->nNeurons);
  priority.resize(pt->nLayers + 1);
  NeurInLyr.resize(pt->nLayers + 1);
  mapMaxOut.resize(pt->map.size());
  mapMinOut.resize(pt->map.size());
  inputN = pt->numNeurons(0);
  outputN = pt->numNeurons(pt->nLayers - 1);

  int NeuronIdx = 0;
  int ArcIdx = 0;
  vector<int> neurons(pt->nLayers);
  //carico il vettore di mappatura dell'output della rete
  for (int i = 0; i < pt->map.size(); i++) {
    mapMaxOut[i] = pt->map[i].maxValue;
    mapMinOut[i] = pt->map[i].minValue;
  }

  NeurInLyr[0] = -1; // setto il primo valore
  priority[0] = -1; // setto il primo valore

  //carico i parametri della rete
  for (int i = 0; i < pt->nLayers; i++) {

    for (int j = 0; j < pt->numNeurons(i); j++) {

      Bayes[NeuronIdx] = pt->getTarget(i, j)->bayes;

      for (int k = 0; k < pt->numCon(i, j); k++) {

        weights[ArcIdx] = pt->getTarget(i, j)->OutArcs[k].weight;
        ArcIn[ArcIdx] = pt->getTarget(i, j)->OutArcs[k].target->neurIdx;
        ArcOut[ArcIdx] = pt->getTarget(i, j)->neurIdx;
        ArcIdx++;
      }
      NeuronIdx++;
    }
    NeurInLyr[i + 1] = NeuronIdx - 1; // salvo l'indice dell'ultimo neurone del layer corrente
    priority[i + 1] = ArcIdx - 1; // salvo l'indice dell'ultimo arco del layer corrente
  }
}

//copia del modello da CudaCore class a MLP class
void cudaNetPasteMLP(MLP *pt) {
  int idx = 0;
  int Nidx = 0;
  for (int i = 0; i < pt->nLayers; i++) {
    for (int j = 0; j < pt->numNeurons(i); j++) {
      pt->getTarget(i, j)->bayes = Bayes[Nidx++];
      for (int k = 0; k < pt->numCon(i, j); k++) {
        pt->getTarget(i, j)->OutArcs[k].weight = weights[idx++];
      }
    }
  }
}

void cudaNetCopyHopfield(Hopfield* pt) { ... }

void cudaNetCopyExamples(MLP *pt) { ... }
```

Listing 10: classe di interfaccia alla GPU

0.1.5 metodi che lanciano i kernel sulla GPU

i seguenti metodi interagiscono direttamente con la GPU allocando memoria su di essa, caricando i dati, e lanciando i kernel che eseguono le effettive operazioni sulla GPU.

```
1
2  ////////////////////////////////CUDA Kernel functions/////////////////////////////
3  //esegue le operazioni di allocamento memoria e preparazione al lancio del kernel di
       ↪ propagazione della rete
4  cudaError_t hostCUDAtrainingNet(float eps, int Niter, int ThxBlock) {
5    cout << "learning is started!" << endl;
6    //host variables
7    float *Cweights = &weights[0];
8    int *CArcIn = &ArcIn[0];
9    int *CArcOut = &ArcOut[0];
10   float *CNeuronOut = &NeuronOut[0];
11   float *CBayes = &Bayes[0];
12   float *CBPerr = &BPerr[0];
13   float *CmapMaxOut = &mapMaxOut[0];
14   float *CmapMinOut = &mapMinOut[0];
15   float *Cexamples = &examples[0];
16   int *CNeurInLyr = &NeurInLyr[0];
17   int *Cpriority = &priority[0];
18   float *CMeanErr = &MeanErr;
19
20   //device variables
21   float *dev_weights = 0;
22   int *dev_ArcIn = 0;
23   int *dev_ArcOut = 0;
24   float *dev_examples = 0;
25   float *dev_NeuronOut = 0;
26   float *dev_Bayes = 0;
27   float *dev_BPerr = 0;
28   float *dev_mapMaxOut = 0;
29   float *dev_mapMinOut = 0;
30   int *dev_NeurInLyr = 0;
31   int *dev_priority = 0;
32   float *dev_MeanErr = 0;
33
34   //int ThxBlock = 1024;
35
36   cudaError_t cudaStatus;
37
38   // Choose which GPU to run on, change this on a multi-GPU system.
39   cudaStatus = cudaSetDevice(GpuID);
40   if (cudaCheckStatus(cudaStatus) == true) goto Error;
41
42   // Allocate GPU buffers for vectors
43   cudaStatus = cudaMalloc((void**)&dev_weights, weights.size() * sizeof(float));
44   if (cudaCheckStatus(cudaStatus) == true) goto Error;
45   cudaStatus = cudaMalloc((void**)&dev_ArcIn, ArcIn.size() * sizeof(float));
46   if (cudaCheckStatus(cudaStatus) == true) goto Error;
47   cudaStatus = cudaMalloc((void**)&dev_ArcOut, ArcOut.size() * sizeof(float));
48   if (cudaCheckStatus(cudaStatus) == true) goto Error;
49   cudaStatus = cudaMalloc((void**)&dev_NeuronOut, NeuronOut.size() * sizeof(float));
50   if (cudaCheckStatus(cudaStatus) == true) goto Error;
51   cudaStatus = cudaMalloc((void**)&dev_Bayes, Bayes.size() * sizeof(float));
52   if (cudaCheckStatus(cudaStatus) == true) goto Error;
53   cudaStatus = cudaMalloc((void**)&dev_BPerr, BPerr.size() * sizeof(float));
54   if (cudaCheckStatus(cudaStatus) == true) goto Error;
55   cudaStatus = cudaMalloc((void**)&dev_mapMaxOut, mapMaxOut.size() * sizeof(float));
56   if (cudaCheckStatus(cudaStatus) == true) goto Error;
57   cudaStatus = cudaMalloc((void**)&dev_mapMinOut, mapMinOut.size() * sizeof(float));
58   if (cudaCheckStatus(cudaStatus) == true) goto Error;
59   cudaStatus = cudaMalloc((void**)&dev_examples, examples.size() * sizeof(float));
60   if (cudaCheckStatus(cudaStatus) == true) goto Error;
61   cudaStatus = cudaMalloc((void**)&dev_NeurInLyr, NeurInLyr.size() * sizeof(int));
62   if (cudaCheckStatus(cudaStatus) == true) goto Error;
63   cudaStatus = cudaMalloc((void**)&dev_priority, priority.size() * sizeof(int));
```

```
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&dev_MeanErr, sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;


    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_weights, Cweights, weights.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_ArcIn, CArcIn, ArcIn.size() * sizeof(int),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_ArcOut, CArcOut, ArcOut.size() * sizeof(int),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_NeuronOut, CNeuronOut, NeuronOut.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_Bayes, CBayes, Bayes.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_BPerr, CBPerr, BPerr.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_mapMaxOut, CmapMaxOut, mapMaxOut.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_mapMinOut, CmapMinOut, mapMinOut.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_examples, Cexamples, examples.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_NeurInLyr, CNeurInLyr, NeurInLyr.size() * sizeof(int),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_priority, Cpriority, priority.size() * sizeof(int),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(dev_MeanErr, CMeanErr, sizeof(float), cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;



    ////////////////////lancio dei kernel all'interno della gpu///////////////
    int startA = 0;
    int endA = 0;
    int startN = 0;
    int endN = 0;
    int numLayerArcs = 0;
    int numLayerNeur = 0;
    int numOfBlocksMax = 0;
    int numOfBlocksA = 0;
    int numOfBlocksN = 0;
    int numOfBlocksOut = floorf(outputN / ThxBlock) + 1;
    int exampleRef = 0;
    int outputRef = NeuronOut.size() - outputN;
    long long t0 = 0, t1 = 0;
    long long t0in = 0, t1in = 0;
    double elapsedMilliseconds = 0;
    double elapsedInMilliseconds = 0;

    for (int it = 0; it < Niter; it++) { //scorro le iterazioni

      t0 = PerformanceCounter();

      for (int t = 0; t < (examples.size() / (inputN + outputN)); t++) { //scorro gli esempi
        //imposto il riferimento per l'esempio di input
        exampleRef = t * (inputN + outputN);
```

```
122         t0in = PerformanceCounter();
123         //resetto il vettore contenente lo stato di attivazione dei neuroni
124         numOfBlocksA = (floorf(NeuronOut.size() / ThxBlock) + 1);
125         CUDAresetVector <<<numOfBlocksA, ThxBlock >>> (dev_NeuronOut, NeuronOut.size());
126 
127         //imposto i valori di input ai neuroni dello strato input
128         numOfBlocksA = (floorf(inputN / ThxBlock) + 1);
129         CUDAsetInput <<<numOfBlocksA, ThxBlock >>> (dev_NeuronOut, inputN, exampleRef,
                ↪ dev_examples);
130 
131         //propagazione dell'input nella rete
132 
133         startA = 0; // indice di partenza dei vettori archi
134         endA = 0; // ultimo indice dei vettori archi
135         startN = 0; // indice di partenza dei vettori neuroni
136         endN = 0; // ultimo indice dei vettori neuroni
137 
138         for (int i = 0; i < priority.size() - 1; i++) { //NB non viene applicata la sigmoide
                ↪ allo strato di input eventulmente correggi
139 
140           startA = priority[i] + 1;
141           endA = priority[i + 1];
142 
143           if (i < priority.size() - 2) {
144             startN = NeurInLyr[i + 1] + 1;
145             endN = NeurInLyr[i + 2];
146           }
147 
148           numLayerArcs = endA - startA + 1;
149           numLayerNeur = endN - startN + 1;
150 
151           numOfBlocksA = floorf(numLayerArcs / ThxBlock) + 1;
152           numOfBlocksN = floorf(numLayerNeur / ThxBlock) + 1;
153 
154           if (i < priority.size() - 2) {
155             //propago l'output dei neuroni al prossimo/i layer
156             CUDAlayerInput <<<numOfBlocksA, ThxBlock >>> (dev_weights, dev_ArcIn, dev_ArcOut,
                    ↪ dev_NeuronOut, startA, endA);
157              //applico il contributo dei bayes all output dei neuroni del layer corrente
158             CUDAbayesInput <<< numOfBlocksN, ThxBlock >>> (dev_NeuronOut, dev_Bayes, startN,
                    ↪ endN);
159             //applico la sigmoide allo stato di attivazione dei neuroni
160             CUDAsigLayer <<<numOfBlocksN, ThxBlock >>> (dev_NeuronOut, startN, endN);
161           }
162         }
163 
164         t1in = PerformanceCounter();
165         elapsedInMilliseconds += ((t1in - t0in) * 1000.0) / PerformanceFrequency();
166 
167 
168         //resetto il vettore contenente l'errore  dei neuroni
169         numOfBlocksN = (floorf(BPerr.size() / ThxBlock) + 1);
170         CUDAresetVector <<<numOfBlocksN, ThxBlock >>> (dev_BPerr, BPerr.size());
171 
172         CUDAresetVar <<<1, 1 >>> (dev_MeanErr);
173         CUDAoutputErr <<<numOfBlocksOut, ThxBlock >>> (dev_NeuronOut, outputRef, NeuronOut.size
                ↪ (), inputN, dev_BPerr, dev_examples, exampleRef, dev_mapMaxOut, dev_mapMinOut,
                ↪ dev_MeanErr);
174         cudaMemcpy(CMeanErr, dev_MeanErr, sizeof(float), cudaMemcpyDeviceToHost);
175 
176         MeanErr += *CMeanErr / outputN;
177 
178 
179         //retropropagazione dell'errore
180 
181         for (int i = priority.size() - 2; i > 1; i--) {
182 
183           startA = priority[i - 1] + 1;
184           endA = priority[i];
```

```
185           startN = NeurInLyr[i - 1] + 1;
186           endN = NeurInLyr[i];
187
188           numLayerArcs = endA - startA + 1;
189           numLayerNeur = endN - startN + 1;
190
191           numOfBlocksA = floorf(numLayerArcs / ThxBlock) + 1;
192           numOfBlocksN = floorf(numLayerNeur / ThxBlock) + 1;
193           //numOfBlocksMax = maxOf(numOfBlocksA, numOfBlocksN);
194
195           CUDAPropagationErr <<<numOfBlocksA, ThxBlock >>> (dev_BPerr, dev_weights,
                  ↪ dev_NeuronOut, dev_ArcIn, dev_ArcOut, startA, endA);
196           CUDAoutDiff <<<numOfBlocksN, ThxBlock >>> (dev_BPerr, dev_NeuronOut, startN, endN);
197           cudaStatus = cudaMemcpy(CBPerr, dev_BPerr, BPerr.size() * sizeof(float),
                  ↪ cudaMemcpyDeviceToHost);
198           if (cudaCheckStatus(cudaStatus) == true) goto Error;
199           copy(CBPerr, CBPerr + BPerr.size(), BPerr.begin());
200         }
201
202         //applico a ogni peso la sua correzione
203
204         startN = NeurInLyr[1] + 1; // la correzione dei bais va applicata dal primo layer
                ↪ nascosto in poi
205         endN = NeurInLyr[NeurInLyr.size() - 1];
206
207         numLayerNeur = endN - startN + 1;
208
209         numOfBlocksA = floorf(weights.size() / ThxBlock) + 1;
210         numOfBlocksN = floorf(numLayerNeur / ThxBlock) + 1;
211
212         CUDAapplyWeightCorrections <<<numOfBlocksA, ThxBlock >>> (eps, dev_NeuronOut, dev_BPerr,
                ↪  dev_weights, dev_ArcIn, dev_ArcOut, weights.size());
213         CUDAapplyBayesCorrections <<<numOfBlocksN, ThxBlock >>> (eps, dev_BPerr, dev_Bayes,
                ↪ startN, endN);
214       }
215
216       t1 = PerformanceCounter();
217       elapsedMilliseconds = ((t1 - t0) * 1000.0) / PerformanceFrequency(); // calcolo il tempo
              ↪ di esecuzione di una iterazione di addestramento (tutto il set)
218       MeanErr = MeanErr / (examples.size() / (inputN + outputN)); //calcolo l'errore percentuale
              ↪  medio sul dataset
219       elapsedInMilliseconds = elapsedInMilliseconds / (examples.size() / (inputN + outputN));
220       cout << "Iterazione: " << it << "  " << MeanErr << " %Err  " << "execution time:" <<
              ↪ elapsedMilliseconds << "ms" << endl;
221       cout << "mean InputTime: " << elapsedInMilliseconds << "ms" << endl;
222       printNetSpecs();
223       MeanErr = 0;
224     }
225
226     cudaStatus = cudaMemcpy(Cweights, dev_weights, weights.size() * sizeof(float),
            ↪ cudaMemcpyDeviceToHost);
227     if (cudaCheckStatus(cudaStatus) == true) goto Error;
228     copy(Cweights, Cweights + weights.size(), weights.begin());
229
230     cudaStatus = cudaMemcpy(CBayes, dev_Bayes, Bayes.size() * sizeof(float),
            ↪ cudaMemcpyDeviceToHost);
231     if (cudaCheckStatus(cudaStatus) == true) goto Error;
232     copy(CBayes, CBayes + Bayes.size(), Bayes.begin());
233     //checkpoint di errore (se la GPU richiama un qualunque errore ripare da qui)
234   Error:
235
236     //libero la memoria nella scheda grafica
237     cudaFree(dev_weights);
238     cudaFree(dev_ArcIn);
239     cudaFree(dev_ArcOut);
240     cudaFree(dev_NeuronOut);
241     cudaFree(dev_examples);
242     cudaFree(dev_BPerr);
243     cudaFree(dev_mapMaxOut);
```

```
    cudaFree(dev_mapMinOut);
    cudaFree(dev_priority);
    cudaFree(dev_NeurInLyr);

    //ritorno lo stato della GPU
    return cudaStatus;
}

//esegue il caricamento nella gpu dei parametri della rete
cudaError_t hostCUDAuploadNetParams() {

    cudaError_t cudaStatus;

    cudaStatus = cudaSetDevice(GpuID);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;

    //host variables
    float *Cweights = &weights[0];
    int *CArcIn = &ArcIn[0];
    int *CArcOut = &ArcOut[0];
    float *CNeuronOut = &NeuronOut[0];
    float *CBayes = &Bayes[0];
    float *CBPerr = &BPerr[0];
    float *CmapMaxOut = &mapMaxOut[0];
    float *CmapMinOut = &mapMinOut[0];
    float *Cexamples = &examples[0];
    int *CNeurInLyr = &NeurInLyr[0];
    int *Cpriority = &priority[0];
    float *CMeanErr = &MeanErr;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(GpuID);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;

    // Allocate GPU buffers for vectors
    cudaStatus = cudaMalloc((void**)&gpuNetParams.weights, weights.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.ArcIn, ArcIn.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.ArcOut, ArcOut.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.NeuronOut, NeuronOut.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.Bayes, Bayes.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.BPerr, BPerr.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.mapMaxOut, mapMaxOut.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.mapMinOut, mapMinOut.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.examples, examples.size() * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.NeurInLyr, NeurInLyr.size() * sizeof(int));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.priority, priority.size() * sizeof(int));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.InputRT, inputN * sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMalloc((void**)&gpuNetParams.MeanErr, sizeof(float));
    if (cudaCheckStatus(cudaStatus) == true) goto Error;


    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(gpuNetParams.weights, Cweights, weights.size() * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    if (cudaCheckStatus(cudaStatus) == true) goto Error;
    cudaStatus = cudaMemcpy(gpuNetParams.ArcIn, CArcIn, ArcIn.size() * sizeof(int),
        ↪ cudaMemcpyHostToDevice);
```

```
311     if (cudaCheckStatus(cudaStatus) == true) goto Error;
312     cudaStatus = cudaMemcpy(gpuNetParams.ArcOut, CArcOut, ArcOut.size() * sizeof(int),
            ↪ cudaMemcpyHostToDevice);
313     if (cudaCheckStatus(cudaStatus) == true) goto Error;
314     cudaStatus = cudaMemcpy(gpuNetParams.NeuronOut, CNeuronOut, NeuronOut.size() * sizeof(float)
            ↪ , cudaMemcpyHostToDevice);
315     if (cudaCheckStatus(cudaStatus) == true) goto Error;
316     cudaStatus = cudaMemcpy(gpuNetParams.Bayes, CBayes, Bayes.size() * sizeof(float),
            ↪ cudaMemcpyHostToDevice);
317     if (cudaCheckStatus(cudaStatus) == true) goto Error;
318     cudaStatus = cudaMemcpy(gpuNetParams.BPerr, CBPerr, BPerr.size() * sizeof(float),
            ↪ cudaMemcpyHostToDevice);
319     if (cudaCheckStatus(cudaStatus) == true) goto Error;
320     cudaStatus = cudaMemcpy(gpuNetParams.mapMaxOut, CmapMaxOut, mapMaxOut.size() * sizeof(float)
            ↪ , cudaMemcpyHostToDevice);
321     if (cudaCheckStatus(cudaStatus) == true) goto Error;
322     cudaStatus = cudaMemcpy(gpuNetParams.mapMinOut, CmapMinOut, mapMinOut.size() * sizeof(float)
            ↪ , cudaMemcpyHostToDevice);
323     if (cudaCheckStatus(cudaStatus) == true) goto Error;
324     cudaStatus = cudaMemcpy(gpuNetParams.examples, Cexamples, examples.size() * sizeof(float),
            ↪ cudaMemcpyHostToDevice);
325     if (cudaCheckStatus(cudaStatus) == true) goto Error;
326     cudaStatus = cudaMemcpy(gpuNetParams.NeurInLyr, CNeurInLyr, NeurInLyr.size() * sizeof(int),
            ↪ cudaMemcpyHostToDevice);
327     if (cudaCheckStatus(cudaStatus) == true) goto Error;
328     cudaStatus = cudaMemcpy(gpuNetParams.priority, Cpriority, priority.size() * sizeof(int),
            ↪ cudaMemcpyHostToDevice);
329     if (cudaCheckStatus(cudaStatus) == true) goto Error;
330     cudaStatus = cudaMemcpy(gpuNetParams.MeanErr, CMeanErr, sizeof(float),
            ↪ cudaMemcpyHostToDevice);
331     if (cudaCheckStatus(cudaStatus) == true) goto Error;
332
333     if (false) {
334       Error:
335       //libero la memoria nella scheda grafica
336       cudaFree(gpuNetParams.weights);
337       cudaFree(gpuNetParams.ArcIn);
338       cudaFree(gpuNetParams.ArcOut);
339       cudaFree(gpuNetParams.NeuronOut);
340       cudaFree(gpuNetParams.examples);
341       cudaFree(gpuNetParams.BPerr);
342       cudaFree(gpuNetParams.mapMaxOut);
343       cudaFree(gpuNetParams.mapMinOut);
344       cudaFree(gpuNetParams.priority);
345       cudaFree(gpuNetParams.NeurInLyr);
346       cout << "ERRORE: libero la memoria della gpu. " << endl;
347     }
348
349     return cudaStatus;
350   }
351
352   //esegue il download dalla gpu dei parametri della rete
353   cudaError_t hostCUDAdownloadNetParams() {
354
355     cout << "downloading net params from gpu.." << endl;
356
357     float *Cweights = &weights[0];
358     float *CBayes = &Bayes[0];
359
360     cudaError_t cudaStatus;
361
362     cudaStatus = cudaSetDevice(GpuID);
363     if (cudaCheckStatus(cudaStatus) == true) goto Error;
364
365     cudaStatus = cudaMemcpy(Cweights, gpuNetParams.weights, weights.size() * sizeof(float),
            ↪ cudaMemcpyDeviceToHost);
366     if (cudaCheckStatus(cudaStatus) == true) goto Error;
367
368     cudaStatus = cudaMemcpy(CBayes, gpuNetParams.Bayes, Bayes.size() * sizeof(float),
```

```
        ↪ cudaMemcpyDeviceToHost);
369     if (cudaCheckStatus(cudaStatus) == true) goto Error;
370
371
372     if (false) {
373     Error:
374       //libero la memoria nella scheda grafica
375       cudaFree(gpuNetParams.weights);
376       cudaFree(gpuNetParams.ArcIn);
377       cudaFree(gpuNetParams.ArcOut);
378       cudaFree(gpuNetParams.NeuronOut);
379       cudaFree(gpuNetParams.examples);
380       cudaFree(gpuNetParams.BPerr);
381       cudaFree(gpuNetParams.mapMaxOut);
382       cudaFree(gpuNetParams.mapMinOut);
383       cudaFree(gpuNetParams.priority);
384       cudaFree(gpuNetParams.NeurInLyr);
385       cout << "ERRORE: libero la memoria della gpu. " << endl;
386     }
387
388     return cudaStatus;
389   }
390
391   //esegue l'input della rete gia addestrata prendendo in input l'esempio dato
392   cudaError_t hostCUDAInputNet(float *input, int ThxBlock) {
393     //inportante verificare che l'input abbia la stessa dimansione dell'input della rete
394
395     cudaError cudaStatus;
396
397     cudaStatus = cudaSetDevice(GpuID);
398     if (cudaCheckStatus(cudaStatus) == true) goto Error;
399
400     ////////////////////lancio dei kernel all'interno della gpu///////////////
401     //int ThxBlock = 1024;
402     int startA = 0;
403     int endA = 0;
404     int startN = 0;
405     int endN = 0;
406     int numLayerArcs = 0;
407     int numLayerNeur = 0;
408     int numOfBlocksMax = 0;
409     int numOfBlocksA = 0;
410     int numOfBlocksN = 0;
411     int numOfBlocksOut = floorf(outputN / ThxBlock) + 1;
412     int outputRef = NeuronOut.size() - outputN;
413     long long t0in = 0, t1in = 0;
414     double elapsedInMilliseconds = 0;
415
416     t0in = PerformanceCounter();
417     //resetto il vettore contenente lo stato di attivazione dei neuroni
418     numOfBlocksA = (floorf(NeuronOut.size() / ThxBlock) + 1);
419     CUDAresetVector <<<numOfBlocksA, ThxBlock >>> (gpuNetParams.NeuronOut, NeuronOut.size());
420
421     cudaStatus = cudaMemcpy(gpuNetParams.InputRT, input, inputN * sizeof(float),
        ↪ cudaMemcpyHostToDevice);
422     if (cudaCheckStatus(cudaStatus) == true) goto Error;
423
424     //imposto i valori di input ai neuroni dello strato input
425     numOfBlocksA = (floorf(inputN / ThxBlock) + 1);
426     CUDAsetSingleInput <<<numOfBlocksA, ThxBlock>>> (gpuNetParams.NeuronOut, inputN,
        ↪ gpuNetParams.InputRT);
427
428     //propagazione dell'input nella rete
429
430     startA = 0; // indice di partenza dei vettori archi
431     endA = 0; // ultimo indice dei vettori archi
432     startN = 0; // indice di partenza dei vettori neuroni
433     endN = 0; // ultimo indice dei vettori neuroni
434
```

```
435     for (int i = 0; i < priority.size() - 1; i++) { //NB non viene applicata la sigmoide allo
            ↪ strato di input eventulmente correggi
436
437       startA = priority[i] + 1;
438       endA = priority[i + 1];
439
440       if (i < priority.size() - 2) {
441         startN = NeurInLyr[i + 1] + 1;
442         endN = NeurInLyr[i + 2];
443       }
444
445       numLayerArcs = endA - startA + 1;
446       numLayerNeur = endN - startN + 1;
447
448       numOfBlocksA = floorf(numLayerArcs / ThxBlock) + 1;
449       numOfBlocksN = floorf(numLayerNeur / ThxBlock) + 1;
450
451       if (i < priority.size() - 2) {
452         CUDAlayerInput <<<numOfBlocksA, ThxBlock >>> (gpuNetParams.weights, gpuNetParams.ArcIn,
                ↪ gpuNetParams.ArcOut, gpuNetParams.NeuronOut, startA, endA); //propago l'output dei
                ↪  neuroni al prossimo/i layer
453         CUDAbayesInput <<<numOfBlocksN, ThxBlock >>> (gpuNetParams.NeuronOut, gpuNetParams.Bayes
                ↪ , startN, endN); //applico il contributo dei bayes all output dei neuroni del
                ↪ layer corrente
454         CUDAsigLayer <<<numOfBlocksN, ThxBlock >>> (gpuNetParams.NeuronOut, startN, endN); //
                ↪ applico la sigmoide allo stato di attivazione dei neuroni
455
456       }
457     }
458
459     //copio l'output dei neuroni dello strato output nella memoria della cpu
460     cudaStatus = cudaMemcpy(&NeuronOut[0] + outputRef, gpuNetParams.NeuronOut + outputRef,
            ↪ outputN * sizeof(float), cudaMemcpyDeviceToHost); //TODO da errore e non carica il
            ↪ vettore trovare il BUG
461     if (cudaCheckStatus(cudaStatus) == true) goto Error;
462
463     t1in = PerformanceCounter();
464     elapsedInMilliseconds = ((t1in - t0in) * 1000.0) / PerformanceFrequency();
465
466     ////////////////////////////visualizzazione dell'esempio//////////////////////////////
467
468     float delta;
469     cout << "input time: " << elapsedInMilliseconds << " ms" << endl;
470
471     for (int on = 0; on < outputN; on++) {
472       delta = mapMaxOut[on] - mapMinOut[on];
473       cout << "Y" << on << ": " << (NeuronOut[NeuronOut.size() - outputN + on] * delta) +
              ↪ mapMinOut[on] << endl;
474     }
475     cout << endl;
476
477     ////////////////////////////////////////////////////////////////////////////////////
478
479     if (false) {
480     Error:
481       //libero la memoria nella scheda grafica
482       cudaFree(gpuNetParams.weights);
483       cudaFree(gpuNetParams.ArcIn);
484       cudaFree(gpuNetParams.ArcOut);
485       cudaFree(gpuNetParams.NeuronOut);
486       cudaFree(gpuNetParams.examples);
487       cudaFree(gpuNetParams.BPerr);
488       cudaFree(gpuNetParams.mapMaxOut);
489       cudaFree(gpuNetParams.mapMinOut);
490       cudaFree(gpuNetParams.priority);
491       cudaFree(gpuNetParams.NeurInLyr);
492     }
493
494     return cudaStatus;
```

```
  }
  ////////////////////////////////////////////////////////////////////////////////
  
  ////////////////////////////////CUDA UTILITY////////////////////////////////////
  //verifica la corretta esecuzione di un operazione
  inline cudaError_t checkCuda(cudaError_t result){ ... }

  //verifica la corretta esecuzione di un operazione restituendo un bool
  bool cudaCheckStatus(cudaError_t cudaStatus) { ... }

  //stampa a schermo le principali proprietà della scheda
  void printDeviceSpecs() { ... }

  //stampa i parametri della rete che vengono passati alla scheda
  void printNetSpecs() { ... }

  //calcola il peso del modello
  float sizeOfModel(string mesureUnit = "B") { ... }

  template<typename T, typename A>
  float sizeOfVector(vector<T, A> const& vect,string mesureUnit = "B") { ... }
  ////////////////////////////////////////////////////////////////////////////////
};
```

Listing 11: classe di interfaccia alla GPU