**Gabriele Marconi**
Matricola: **0000999689**

# NuriCheat

## Risoluzione di rompicapi su piattaforme Android tramite Image Processing e Intelligenza Artificiale

### ABSTRACT

L'intenzione del progetto è quella di creare un'applicazione Android che riesca a interpretare e digitalizzare un'immagine contenente la griglia di gioco del rompicapo Nurikabe. Inoltre, dovrà essere in grado di risolvere il puzzle e mostrare a schermo la soluzione.

### OBIETTIVI

1. Digitalizzare una griglia 5x5 che contiene le informazioni del puzzle da un'immagine (anche da un disegno).
2. Utilizzare una rete neurale per riconoscere le cifre contenute in ogni casella.
3. Implementare un algoritmo di risoluzione efficace e performante.

# Indice

## CAPITOLI

### Capitolo I - Image Processing

Nel primo capitolo verranno analizzate le strategie legate all'analisi delle griglie di partenza. Verranno forniti esempi visivi e di codice, grazie ai quali sarà possibile contestualizzare le operazioni ed il loro ordine.

### Capitolo II - Rete Neurale

Il secondo capitolo avrà l'obiettivo di mostrare l'implementazione e l'utilizzo del modello neurale pre-allenato scelto per il progetto, nel formato .tflite.

### Capitolo III - Implementazione dell'algoritmo di risoluzione

Nel terzo capitolo si approfondiranno le scelte di rappresentazione della griglia digitalizzata e le strategie utilizzate per la sua risoluzione.

### Capitolo IV - MainActivity e layout

L'ultimo capitolo tratterà in breve la classe MainActivity di Android Studio per contestualizzare tutti i componenti visti in precedenza.

## STRUMENTI UTILIZZATI NEL PROGETTO

Ovviamente, la creazione di un'applicazione Android è resa possibile da **Android Studio**, al quale sarà integrato un'altro strumento fondamentale:

**OpenCV** è una libreria software multipiattaforma nell'ambito della visione artificiale in tempo reale, qui impiegata nella sua versione 3.4.10 per Java.

L'applicazione è stata testata su device fisico (con Android 7.0 o maggiore) e su emulatore durante lo sviluppo.

# Capitolo I - Image Processing

Nel discutere questo capitolo, ipotizziamo che l'immagine con la griglia sia già stata caricata dalla galleria del cellulare in una variabile di tipo Bitmap**.**

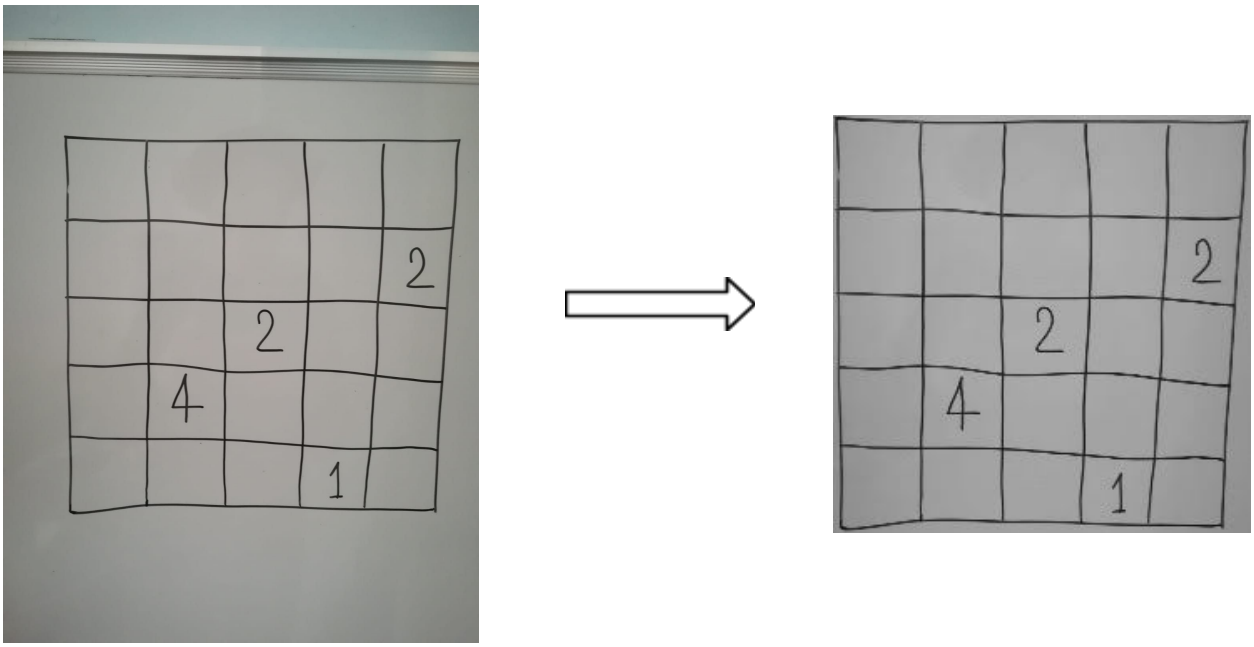
**Bitmap** definisce un'immagine all'interno di Android Studio, ma le operazioni che ci interessano vengono effettuate dalla libreria di Opencv, che utilizza oggetti di tipo **Mat**. Fortunatamente la libreria stessa fornisce metodi di conversione da Bitmap a Mat e viceversa, *BitmapToMat* e *MatToBitmap.*

Nel progetto, ogni metodo (che appartiene ad un'unica pipeline) è definito in una classe Java chiamata **ImageProcessor**. Questa si occupa di elaborare l'immagine originale e dividerla in 25 parti binarie con risoluzione 28x28.

Ogni sezione raffigura una delle 25 caselle che compongono la griglia di gioco. Il passo successivo è la loro valutazione da parte del modello .tflite integrato nell'app.

## focus_and_resize

Le prime trasformazioni avvengono nel metodo chiamato *focus_and_resize*, che ha l'obiettivo di diminuire la risoluzione dell'immagine. Ciò avviene ritagliando l'oggetto Mat cosicché mostri solo la griglia e scalando il risultato finale ad una risoluzione fissa di 250x250 pixel (scelta empirica).



Scalare un'immagine in questo modo migliora di molto le prestazioni della pipeline, è dunque bene farlo il prima possibile. Inoltre ritagliare affinché venga mostrata solo la griglia permette di modificare l'immagine con più sicurezza, evitando problemi di rumore e di distorsione causati dal "resizing".

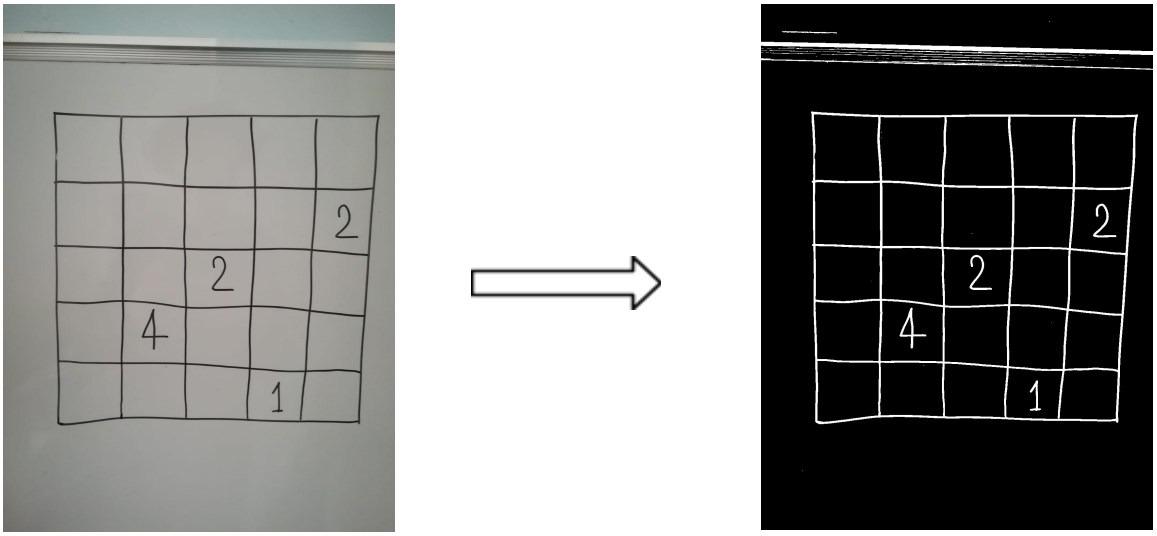
La prima operazione è isolare il canale grayscale dell'immagine, sostituendo l'oggetto Mat passato come argomento alla funzione.

Successivamente viene effettuata una *resize* iniziale che, forzando la larghezza a 500 pixel, mantiene il rapporto con l'altezza. In questo modo si evitano i ritardi causati dall'elaborazione di immagini troppo grandi (immagini provenienti dalla fotocamera del cellulare raggiungono anche i 4k di risoluzione). Inoltre, dopo un'attenta valutazione, immagini più piccole (ad esempio 200x200) non causano problemi dati dall'upscaling.

A questo punto viene applicato un threshold adattivo che, in base a vari parametri, rende binario un Mat precedentemente convertito a grayscale. Il Mat risultante (*thresh*), contiene pixel bianchi o neri (ma nessun valore intermedio).

Visivamente, questa trasformazione risalta il contrasto dell'immagine, fornendo una rappresentazione più chiara della griglia.

```
Imgproc.cvtColor(img, img, Imgproc.COLOR_BGR2GRAY);
double scale = 500.0/img.width();
Imgproc.resize(img, img, new Size(), scale, scale);
Mat thresh = new Mat(img.size(), img.type());
Imgproc.adaptiveThreshold(img, thresh, 255,
Imgproc.ADAPTIVE_THRESH_GAUSSIAN_C, Imgproc.THRESH_BINARY_INV, 157, 20);
```



L'inversione avvenuta durante l'operazione di thresholding (`THRESH_BINARY_INV`) ha portato ad una griglia bianca su sfondo nero. Questo stato è fondamentale per assicurare il successo delle prossime trasformazioni.

Le **trasformazioni morfologiche** sono semplici operazioni che modificano un'immagine binaria in base al kernel passato come argomento e al tipo di effetto.

In questo caso vengono applicate **apertura** e **chiusura.**

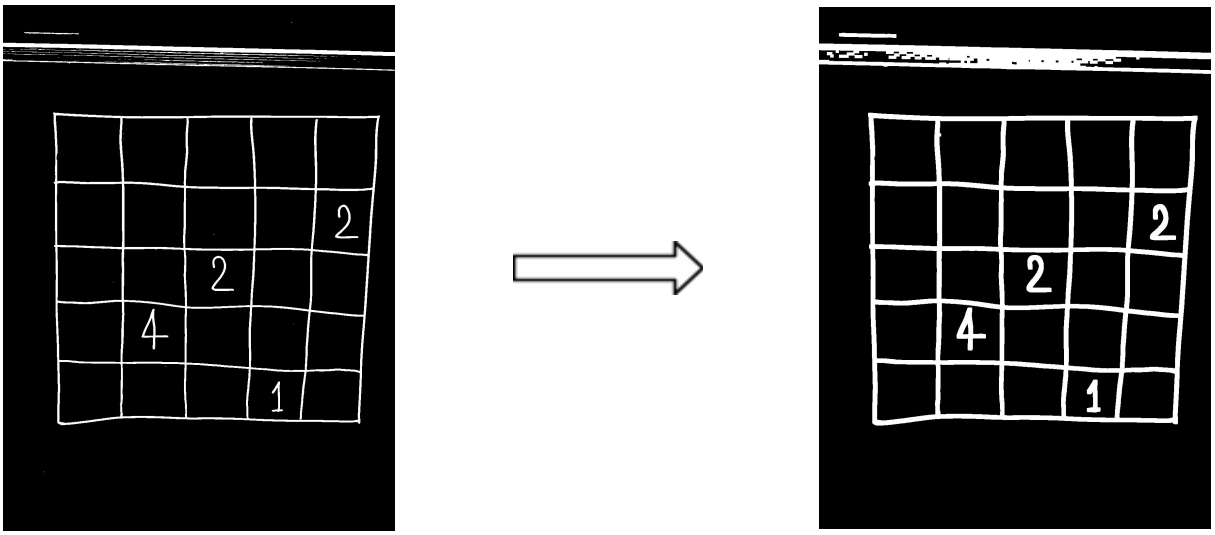
La prima ha il compito di eliminare il rumore, la seconda dilata le figure di colore bianco.

Il kernel d'apertura è il più piccolo possibile e la trasformazione viene eseguita una sola volta, mentre la dilatazione ha un kernel rettangolare 5x5 con 7 iterazioni.

Il risultato sarà una griglia più definita ed uno sfondo con rumore più disperso, per facilitare l'analisi successiva.

```
Mat openKernel = Imgproc.getStructuringElement(Imgproc.MORPH_RECT, new
Size(2,2));
Imgproc.morphologyEx(thresh, thresh, Imgproc.MORPH_OPEN, openKernel, new
Point(-1,-1), 1);

Mat dilateKernel = Imgproc.getStructuringElement(Imgproc.MORPH_RECT, new
Size(5,5));
Imgproc.morphologyEx(thresh, thresh, Imgproc.MORPH_DILATE, dilateKernel, new
Point(-1,-1), 7);
```



Ora è bene introdurre il concetto di **contorno**. In Opencv, un contorno può essere definito come una linea costante di pixel che, chiudendosi, circonda una certa area dell'immagine.

Analizzando l'esempio precedente, è ovvio notare la presenza di molti contorni, tutti collegati alla griglia.

Il prossimo obiettivo è raggiunto trovando il contorno che circonda l'area più grande, il quale è associato inevitabilmente all'intera griglia.

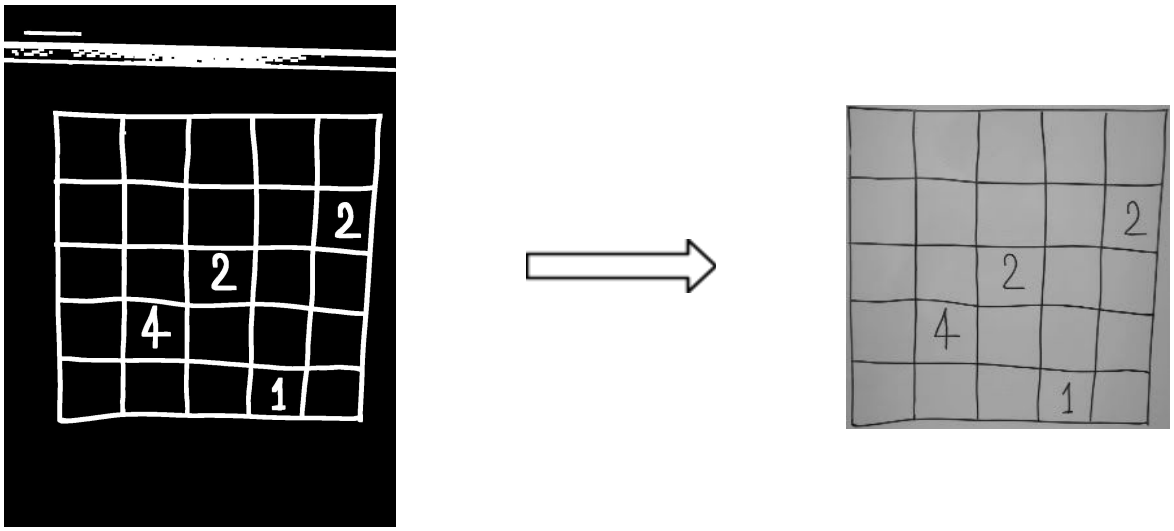
Un contorno è definito con una sottoclasse di Mat, ovvero MatOfPoint, che rappresenta una matrice di posizioni e non di pixel. Il metodo *findContours* seguito da una ricerca basata sugli stream, porta al risultato desiderato.

```
List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
Imgproc.findContours(thresh, contours, new Mat(), Imgproc.RETR_TREE,
Imgproc.CHAIN_APPROX_SIMPLE);
Optional<MatOfPoint> c_max_opt = contours.parallelStream().reduce((c1, c2) ->
Imgproc.contourArea(c1) > Imgproc.contourArea(c2) ? c1 : c2);
MatOfPoint c_max = c_max_opt.get();
```

L’ultimo passo della *focus_and_resize* è usare le coordinate contenute nel contorno per ritagliare l’immagine originale in grayscale.

Infine, il risultato viene ridimensionato alla risoluzione richiesta: 250x250.

```
Rect rect = Imgproc.boundingRect(c_max);
Mat result = img.submat(rect);
Imgproc.resize(result, result, procInput);
```



## getMatrix

*getMatrix* è l'unico metodo pubblico della classe, ed è utilizzato dalla MainActivity di Android Studio per ottenere la griglia digitale.

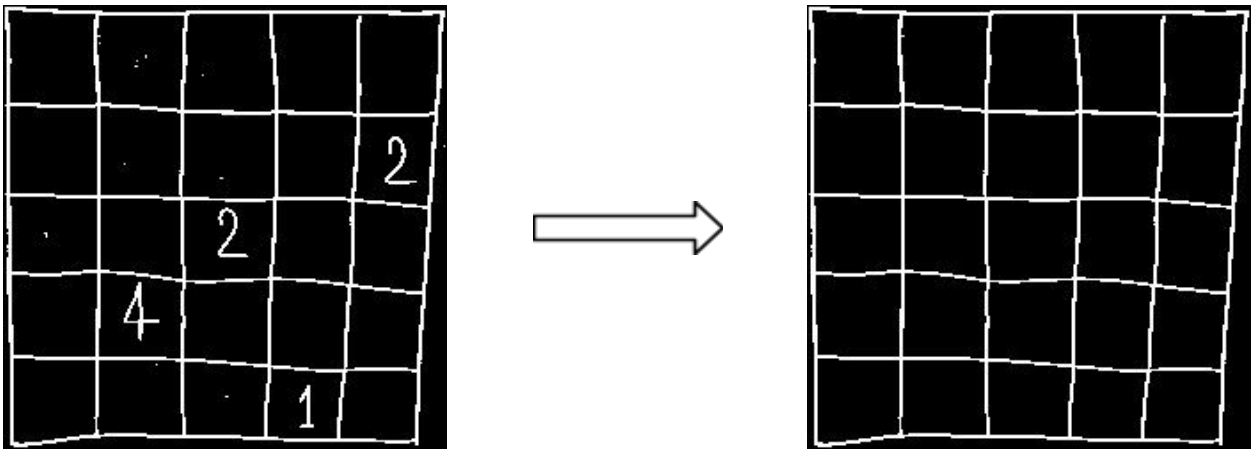
Dopo aver ricavato l'immagine della griglia da *focus_and_resize*, viene applicato un adaptive threshold e trovati tutti i contorni.

La lista dei contorni è utilizzata per colorare il loro interno di nero, cancellando tutti i numeri presenti nelle caselle.

Iterando sulla lista, il metodo *drawContours* riempie le aree con meno di 1000 pixel (valore scelto empiricamente, considerando la risoluzione statica dell'immagine).

*Nota: normalmente drawContours disegna solamente la linea di contorno, tuttavia, se l'argomento thickness è negativo, viene colorata l'area delimitata.*

```
for (MatOfPoint c: contours) {
   area = Imgproc.contourArea(c);
   if (area < 1000) {
       singleC.add(c);
       Imgproc.drawContours(thresh, singleC, -1, new Scalar(0,0,0), -1);
   }
}
```



In questo modo otteniamo una maschera che, similmente a quanto visto nella *focus_and_resize*, potrà essere impiegata per ritagliare l'immagine della griglia esattamente intorno alle caselle.

Infatti, ora la lista risultante da *findContours* è composta solo dai contorni delle caselle e non contiene MatOfPoint parassiti (ovvero contorni di numeri come 6,8,9,4 e 0).

Tuttavia, la lista di contorni non è ordinata. Ciò rende impossibile costruire una griglia digitale coerente.

Il metodo privato *sortContours* risolve questo problema. Grazie al tipo **Rect** è possibile conoscere le coordinate del vertice sinistro-superiore (top-left) del contorno.

Sfortunatamente, ordinare i contorni in base ai valori **x** e **y** nello stesso momento, porta a risultati errati. Ciò probabilmente è causato dalla natura imperfetta del disegno che può essere deformato o ruotato.

La soluzione è dividere l'ordinamento in due cicli:

- Nel primo ciclo la lista viene ordinata verticalmente (in base alla coordinata y).
- Nel secondo ciclo, vengono isolate 5 caselle alla volta (che ora appartengono sicuramente ad una stessa riga). Successivamente quelle 5 caselle vengono ordinate orizzontalmente (in base alla coordinata x).

```
List<MatOfPoint> res = new ArrayList<MatOfPoint>();
List<MatOfPoint> row = new ArrayList<MatOfPoint>();

//ordinamento verticale
Collections.sort(contours, (o1, o2) -> {
   Rect rect1 = Imgproc.boundingRect(o1);
   Rect rect2 = Imgproc.boundingRect(o2);
   int result = Double.compare(rect1.tl().y, rect2.tl().y);
   return result;
});
```

```
// ordinamento orizzontale
int idx=1;
for (MatOfPoint c: contours) {
   double area = Imgproc.contourArea(c);
   row.add(c);
   if (idx % 5 == 0) {
       Collections.sort(row, (o1, o2) -> {
           Rect rect1 = Imgproc.boundingRect(o1);
           Rect rect2 = Imgproc.boundingRect(o2);
           int result = Double.compare(rect1.tl().x, rect2.tl().x);
           return result;
       });
       res.addAll(row);
       row.clear();
   }
idx++;
}

return res;
```

Prima di utilizzare le 25 maschere per estrarre i numeri dalla griglia, dobbiamo applicare un altro threshold inverso (diverso dai precedenti, impostato per preservare la forma dei numeri al meglio) poiché la rete neurale richiede un input binario.

Qui possiamo anche definire la lista di **Integer** che rappresenta l'intera griglia e che verrà restituita dal metodo.

```
Mat prepImage = new Mat();
Imgproc.adaptiveThreshold(img, prepImage, 255,
Imgproc.ADAPTIVE_THRESH_GAUSSIAN_C, Imgproc.THRESH_BINARY_INV, 57, 20);

List<Integer> resList = new ArrayList<Integer>();
```

Nell'ultima fase del processo, si itera sulla lista ordinata e, per ogni contorno, si eseguono le seguenti operazioni:

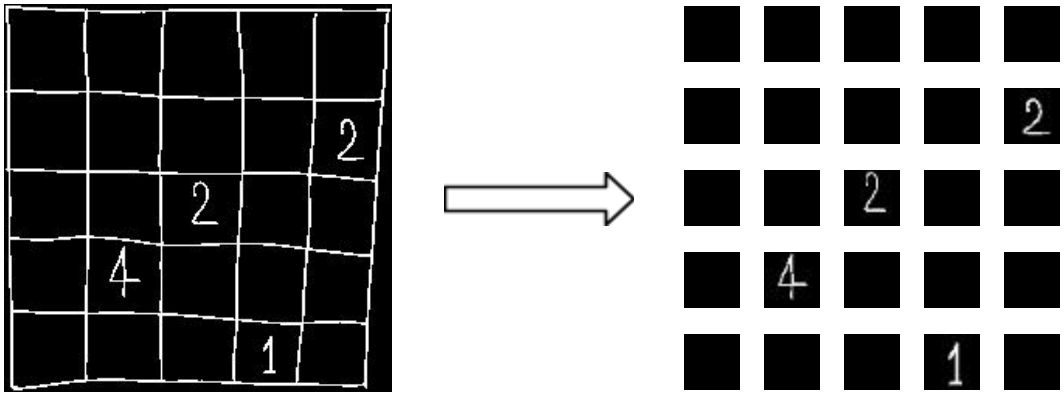
- Grazie al contorno, viene creata la sub-Mat che contiene l'immagine della casella.
- La stessa Mat viene ridimensionata per raggiungere i requisiti di input della rete neurale (28x28).
- Avviene un controllo per verificare che sia presente un numero nella casella. Ciò avviene ottenendo la percentuale di pixel neri nell'immagine (metodo privato *black_px_percentage*). Se la percentuale supera 0.99 (valore empirico) allora la casella è vuota e viene aggiunto uno zero al risultato.
- Se nella casella è presente un numero, l'immagine diventa l'input del classificatore (analizzato nel prossimo capitolo) e l'intero risultante viene aggiunto alla lista.

```
for (MatOfPoint c: contours) {
   Rect rect = Imgproc.boundingRect(c);
   Mat number = prepImage.submat(rect);

   Imgproc.resize(number, number, cnnInput);

   double percentage = black_px_percentage(number);
   if (percentage > 0.99) {
       resList.add(0);
   }
   else {
       resList.add(classifier.classifyMat(number));
   }
}

return resList;
```



# Capitolo II - Rete Neurale

Il processo di scelta del modello adeguato per l'applicazione è stato relativamente breve. Le reti di riconoscimento di cifre sono molto diffuse e supportate.

Il formato richiesto da Android Studio per l'importazione di un modello è .tflite. Fortunatamente, Tensorflow mette a disposizione molti modelli pre-allenati che possono essere facilmente convertiti al formato richiesto.

La rete scelta è stata allenata con il dataset MNIST ed è scaricabile dall'Hub di Tensorflow:

- **Input**: tensore di tipo float con shape [1, 28, 28, 1].
- **Output**: tensore di tipo float con shape [1, 10].

Il processo di importazione è automatico e Android Studio riconosce i tipi in input e output mostrando un esempio di utilizzo in codice Java.

Quest'ultimo viene approfondito ed eseguito nella classe **Classifier** del progetto.

Il costruttore inizializza le variabili di input e di output che sono rispettivamente:

- un oggetto **ByteBuffer** allocato coerentemente.
- un array di float di 10 elementi.

```
imgData = ByteBuffer.allocateDirect(DIM_BATCH_SIZE * DIM_HEIGHT * DIM_WIDTH
* DIM_PIXEL_SIZE * BYTES); // 1 * 28 * 28 * 1 * 4
imgData.order(ByteOrder.nativeOrder());

probArray = new float[10];
```

Tramite l'unico metodo pubblico della classe, *classifyMat*, è possibile convertire un Mat che raffigura una cifra, nell'intero corrispondente.

L'estensione di Tensorflow per Android Studio mette a disposizione una serie di strumenti per semplificare l'elaborazione dell'input:

- Dopo l'importazione è possibile definire ed istanziare un oggetto il cui tipo è il nome del file .tflite (in questo contesto **mnist.tflite**) che contiene il modello.

```
private Mnist mnist;

mnist = Mnist.newInstance(this.activity);
```

- Inoltre, è necessario convertire l'oggetto Mat in ByteBuffer.

```
for (int i = 0; i < DIM_HEIGHT; ++i) {
   for (int j = 0; j < DIM_WIDTH; ++j) {
       imgData.putFloat((float)mat.get(i,j)[0]);
   }
}
```

- Il passo successivo è costruire il tensore di input, eseguire il modello e popolare l'array di uscita.

```
// Creazione input
TensorBuffer inputFeature = TensorBuffer.createFixedSize(new int[]{1, 28, 28,
1}, DataType.FLOAT32);
inputFeature.loadBuffer(imgData);

// Esecuzione modello
Mnist.Outputs outputs = mnist.process(inputFeature);
TensorBuffer outputFeature = outputs.getOutputFeature0AsTensorBuffer();
probArray = outputFeature.getFloatArray();
```

Ora basta restituire al metodo *getMatrix* la posizione dell'unico valore uguale a 1.0 nell'array di output, che equivale alla cifra rilevata.

# Capitolo III - Implementazione dell’algoritmo di risoluzione

Prima di analizzare la struttura del risolutore, è bene approfondire le regole del Nurikabe.

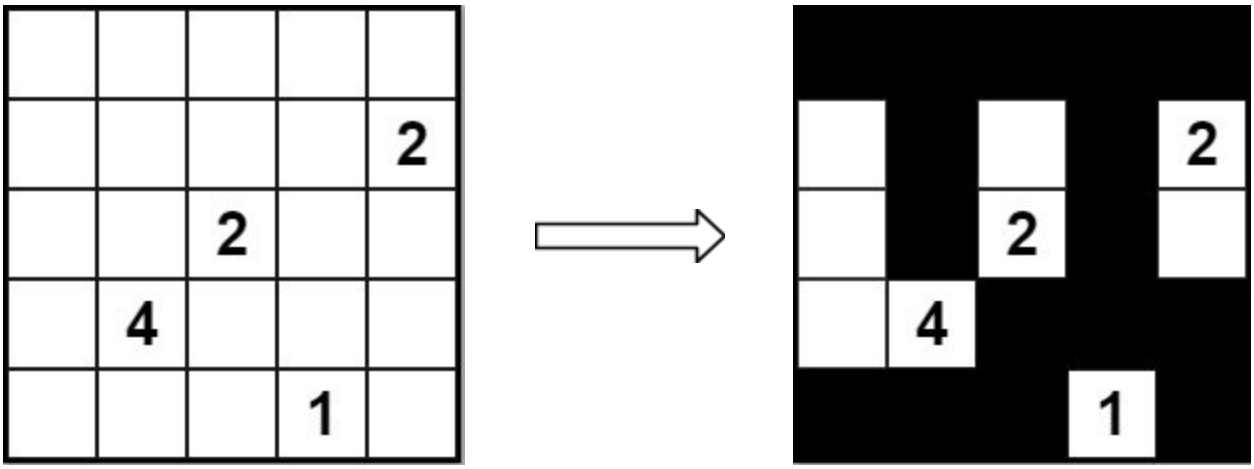
Il puzzle si svolge su una griglia quadrata, alcune celle (pivot) contengono numeri. L’obiettivo è determinare se ognuna delle celle è “nera” o “bianca”.

Le celle nere formano “il nurikabe” (ovvero l’acqua) e:

- devono essere tutte ortogonalmente contigue.
- non devono contenere numeri.
- non possono formare aree 2x2 o più grandi.

Le celle bianche formano le “isole”:

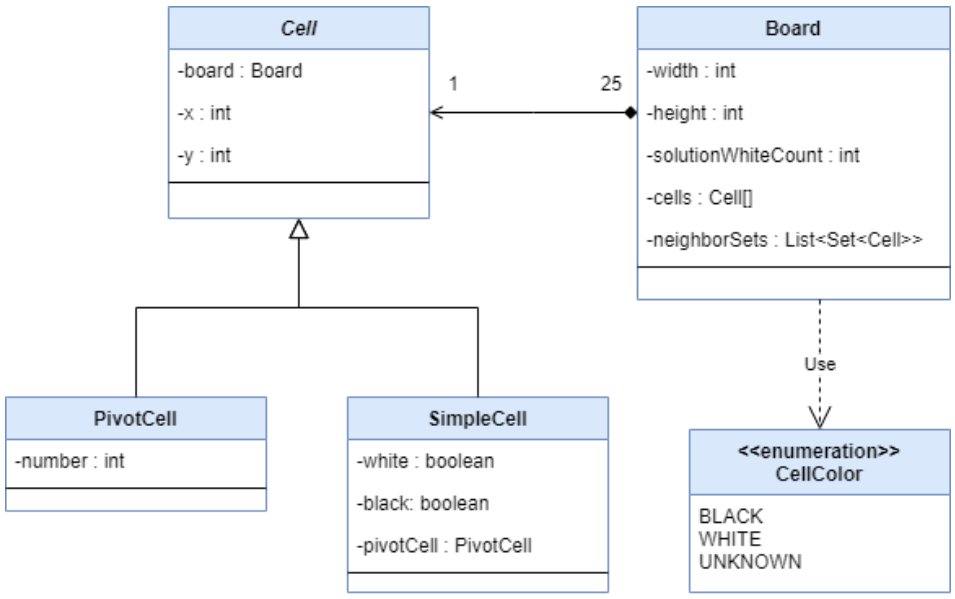
- ciascun numero n deve essere parte di un’isola contenente n celle bianche.
- ogni cella bianca deve appartenere ad una e una sola isola.
- le isole devono avere una sola cella numerata.
- Una cella pivot è sempre bianca.



Lo sviluppo del risolutore è avvenuto in due fasi: nella prima sono state costruite le fondamenta per la rappresentazione della griglia digitalizzata (definizione dei tipi), nella seconda l'attenzione si è spostata sulle strategie di risoluzione.

## STRUTTURA DELLA GRIGLIA

Il seguente diagramma UML riassume i componenti della struttura (contenuti nel package *board*):




Ogni variabile collegata ad un componente della griglia (Board compresa) è dichiarata come **final**, per beneficiare delle proprietà di modifica dei tipi non primitivi.

Infatti, è molto più semplice sviluppare un risolutore avendo un solo oggetto Board che viene modificato indipendentemente dalla posizione del metodo in esecuzione.

**Cell:** è una classe astratta che rappresenta una singola cella. Le proprietà di una cella sono le sue coordinate (x, y), la Board di appartenenza e le informazioni sul colore. Queste ultime vengono approfondite nelle sue sotto-classi.

**SimpleCell:** è una sotto classe di Cell che rappresenta una cella senza cifra. Viene effettuato l'override di tutte le operazioni relative al colore e i metodi *getPivotCell*, *setPivotCell* si riferiscono alla cella pivot collegata, che inizialmente è sempre null perché la SimpleCell non fa parte di nessuna isola (Anche nel caso in cui SimpleCell sia di colore nero, la PivotCell collegata è sempre null).

**PivotCell:** è una sotto-classe di Cell che rappresenta una cella con cifra (considerata bianca). Anche in questo caso si ridefiniscono tutte le operazioni legate al colore ed il metodo *getPivotCell* restituisce **this**.

**Board:** contiene lo stato corrente della griglia e mette a disposizione metodi utili alla gestione delle risorse, ad esempio:

- *connectWhiteCell*: collega una cella bianca senza pivot di riferimento al pivot di una cella bianca adiacente (se presente).
- getWhiteGroupsWithPivotCell: restituisce una mappa in cui una entry è composta da una cella pivot (chiave) e il set di celle bianche ad essa collegate (valore).
- *getGroups*: restituisce tutti i gruppi di celle di un certo CellColor.
- *isSolution*: verifica se lo stato della board corrente sia la soluzione della board stessa:
    - controlla se il numero di celle bianche corrisponde a quello richiesto (ovvero la somma delle cifre nei pivot).
    - controlla che non esista una cella bianca non collegata ad un pivot.
    - controlla che ogni gruppo bianco sia completo (ovvero che l'isola sia composta da un numero di caselle uguale al numero nel pivot).
    - controlla che tutte le celle bianche adiacenti ad un pivot siano collegate solo a quel pivot.
    - controlla che esista un solo gruppo di celle nere.
    - controlla che non esista un'area 2x2 di celle nere.

Nel costruttore, inoltre, ogni gruppo di celle adiacenti ad ognuna delle 25 celle viene inserito in una lista ordinata chiamata *neighborSets*, utile per i processi di ricerca.

**CellColor:** classe enumerativa che definisce i tre possibili stati di una cella: `BLACK`, `WHITE` e `UNKNOWN`.

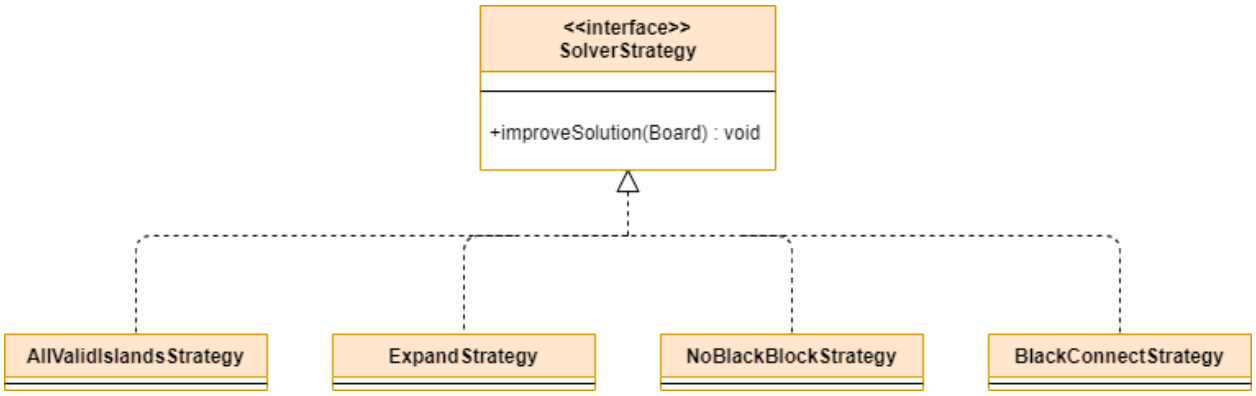
## RISOLVERE IL PUZZLE

Esistono varie strategie che permettono di risolvere Nurikabe, ma nessuna di queste è abbastanza efficace da completare tutti i puzzle generabili.

Ogni strategia, infatti, mira ad una specifica certezza che nasce dalle regole di gioco. Ad esempio, poiché non può esistere un'area 2x2 di celle nere, se nella griglia è presente una sezione con 3 celle nere ed una sconosciuta, questa è sicuramente bianca.

Ciò vuol dire che il modo migliore per assicurare un processo affidabile di risoluzione è quello di impiegare più strategie diverse che vengano applicate alla stessa Board.

Il pattern Strategy è adeguato per implementare questa parte del progetto.




Una proprietà importante di tutte le strategie è la loro indipendenza: non esiste un ordine di esecuzione che rende impossibile risolvere un puzzle.

Come possiamo osservare ogni strategia utilizza il metodo *improveSolution* per modificare lo stato della Board corrente.

## AllValidIslandStrategy

AllValidIslandStrategy è la strategia più complessa perché analizza in profondità lo stato della griglia, basandosi sulle seguenti certezze:

- Le celle che non sono raggiungibili partendo da tutti i pivot, sono sempre nere.
- Se tutte le isole possibili a partire da un pivot condividono una cella, quella è sicuramente bianca.
- Le celle bianche non collegate che sono raggiungibili da un solo pivot, devono essere collegate a quel pivot.

Inizialmente viene costruita una mappa per contenere le celle e tutti i pivot che possono raggiungere la cella (in base alla cifra e posizione del pivot). All'inizio sono presenti solo le chiavi (tutte le celle non nere), mentre i valori (set di celle pivot) sono solo inizializzati.

```
final Map<Cell, Set<PivotCell>> reachableMap = new HashMap<>();
for (final Cell cell : board) {
   if (!cell.isBlack()) {
       reachableMap.put(cell, new HashSet<PivotCell>());
   }
}
```

Il punto più importante dell'algoritmo è la generazione delle isole "valide". Per isole valide si intende l'insieme di tutte le combinazioni di celle che possono formare un'isola completa a partire da una cella pivot.

Per fare questo viene utilizzato il metodo statico *generateValidIslands* proveniente da una classe statica ausiliaria chiamata **BoardOps.**

```
final Set<Set<Cell>> validIslands = BoardOps.generateValidIslands(pivotCell,
whiteGroup);
if (validIslands.isEmpty()) {
   // non dovrebbe mai accadere per uno stato legittimo della griglia
   throw new IllegalStateException();
}
```

Per ogni isola valida, vengono controllate il numero di presenze di una certa cella:

- inAllIslands è un set composto dalle celle che compaiono in tutte le isole valide (il pivot appartiene sempre a questo set).
- inSomeIslands è un set composto da celle che compaiono in una o più isole, ma non in tutte.

```
for (final Set<Cell> validIsland : validIslands) {
   if (inAllIslands == null) {
       inAllIslands = new CellSet(validIsland);
   }
   else {
       inAllIslands.retainAll(validIsland);
   }
   inSomeIslands.addAll(validIsland);
}
```

Ogni cella di inAllIslands è sicuramente bianca e appartiene alla cella pivot considerata, mentre quelle contenute in inSomeIslands sono usate per popolare reachableMap.

```
for (final Cell cell : inAllIslands) {
   // belongs to the current fixed cell
   cell.setWhite();
   cell.setPivotCell(pivotCell);
}

for (final Cell cell : inSomeIslands) {
   // mark it as reachable
   reachableMap.get(cell).add(pivotCell);
}
```

Infine viene analizzata la mappa, per ogni entry:

- se nessun pivot può raggiungere la cella, questa diventa nera.
- se la cella è bianca e raggiungibile da un solo pivot, la cella viene collegata a quel pivot, entrando a far parte della sua isola.

```
for (final Entry<Cell, Set<PivotCell>> entry : reachableMap.entrySet()) {
    final Cell cell = entry.getKey();
    final Set<PivotCell> reachableFrom = entry.getValue();
    if (reachableFrom.isEmpty()) {
        cell.setBlack();
    }
    else if (cell.isWhite() && reachableFrom.size() == 1) {
        cell.setPivotCell(reachableFrom.iterator().next());
    }
}
```

## ExpandStrategy

Questa strategia si basa su una semplice regola: se un gruppo dello stesso colore (nero o bianco) non è completo e può espandersi in una sola cella sconosciuta, quella cella deve appartenere al gruppo. Ciò accade quando un'isola non è composta da abbastanza celle bianche, oppure se il numero di celle nere non ha raggiunto l'obiettivo (celle totali - somma delle cifre nei pivot).

Il metodo privato *tryExpand* viene utilizzato per questo scopo.

```
private boolean tryExpand(Board board, Set<Cell> group, CellColor cellColor) {
    final Set<Cell> unknownNeighbors = BoardOps.unknownNeighbors(board,
group);

    if (unknownNeighbors.size() == 1) {
        unknownNeighbors.iterator().next().setColor(cellColor);
        return true;
    }
    else {
        return false;
    }
}
```

## NoBlackBlockStrategy

Il metodo *improveSolution* di questa strategia cerca nella board aree 2x2 formate da tre celle nere e una sconosciuta.

Se ne trova, cambia il colore della cella sconosciuta a bianco, perché non possono esistere blocchi 2x2 di celle nere.

## BlackConnectStrategy

L'ultima strategia può essere applicata solamente nel caso in cui ci siano 2 o più gruppi di celle nere. Se la condizione è verificata, si cerca di unire quei gruppi.

Per ogni cella sconosciuta vicina di un gruppo nero, si cerca un percorso differente che raggiunga un altro gruppo nero. Se non è presente, vuol dire che la cella considerata è nera.

La ricerca del percorso alternativo avviene grazie al metodo privato *pathToOtherGroupExists*.

```
final Set<Set<Cell>> blackGroups = board.getGroups(CellColor.BLACK);
if (blackGroups.size() < 2) {
   // almeno 2 gruppi neri
   return;
}
for (final Set<Cell> blackGroup : blackGroups) {
   for (final Cell unknownNeighbor : BoardOps.unknownNeighbors(board,
blackGroup)) {
       if (!pathToOtherGroupExists(board, blackGroup, unknownNeighbor)) {
           unknownNeighbor.setBlack();
           return;
       }
   }
}
```

## Solver

La classe che si occupa di gestire la risoluzione del gioco è **Solver**.

Il costruttore ha come argomento un array di strategie che saranno poi utilizzate dal metodo pubblico *tryToSolve*.

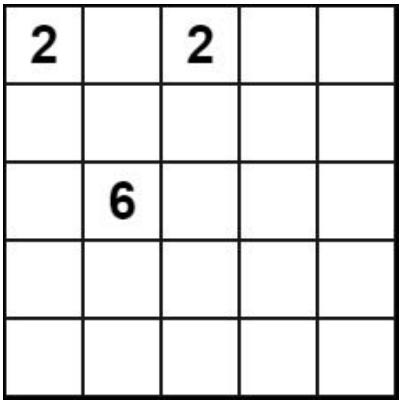
Inizialmente, *tryToSolve* avrebbe dovuto semplicemente applicare tutte le strategie un numero controllato di volte: se da un ciclo di esecuzione di strategie all'altro lo stato della board (riassunto in una stringa) non fosse cambiato e la griglia non avesse ancora raggiunto la soluzione, il puzzle era impossibile da risolvere.

```
for (final SolverStrategy strategy : strategies) {
   try {
       strategy.improveSolution(board);
   }
   catch(IllegalStateException e) {
       return false;
   }
}
String newStringState = board.toString();
```

Tuttavia, sebbene la maggior parte delle griglie venisse risolta senza problemi, una piccola percentuale di configurazioni (ovviamente risolvibili) risultava impossibile da completare.

Queste configurazioni avevano delle proprietà in comune:

- Pochi pivot.
- Un pivot più grande degli altri.



L'area di celle vuote non popolata da pivot creava uno stallo nella progressione dell'algoritmo poiché nessuna delle strategie riusciva a modificare quella porzione di griglia.

Sfortunatamente, dopo un'attenta ricerca, non è stata trovata nessuna strategia diversa che riuscisse a sbloccare la risoluzione.

Dunque, per questi casi è stato scelto un approccio ibrido.

```
if (board.isSolution()) {
   return true;
}
else {
   if(newStringState.contains(CellColor.UNKNOWN.ordinal()+"") &&
oldStringState.equals(newStringState)) {
       BoardState boardState = new BoardState(board);
       if (inferIsland(board, boardState)) {
           return true;
       }
       else break;
   }
   if(!newStringState.contains(CellColor.UNKNOWN.ordinal()+"")) {
       break;
   }
}
```

Nel caso in cui, dopo aver applicato le quattro strategie, lo stato precedente della griglia non sia cambiato e ci siano ancora celle di tipo sconosciuto, viene salvato lo stato corrente e chiamato il metodo privato *inferIsland*.

Quest'ultimo rileva il pivot che sta creando il problema e genera tutte le sue isole valide usando il metodo *generateValidIslands*, già visto in precedenza.

Dopodichè, si tenta di risolvere la griglia forzando la creazione di tutte le isole (ovvero rende tutte le celle bianche e le collega al pivot corrispondente) chiamando ricorsivamente il metodo *tryToSolve*.

Solo una tra tutte le possibili isole può completare il puzzle. Dunque, finché non viene trovata, si eseguono le seguenti operazioni:

- si tenta di risolvere la griglia dopo aver definito l'isola.
- se l'isola non porta alla soluzione, *tryToSolve* restituirà sicuramente false. Ciò può accadere a causa del lancio di un'eccezione (in AllValidIslandsStrategy) oppure perchè la griglia è stata riempita (non esistono più celle sconosciute) però non risolta (*isSolution* restituisce false).
- A questo punto, viene restaurato lo stato della griglia precedente alla chiamata di *inferIsland* e lo stesso algoritmo è applicato all'isola valida successiva.

Se l'isola adatta è trovata la ricorsione restituisce true, altrimenti la griglia è sicuramente irrisolvibile.

```
// Per ogni isola valida del pivot problematico
for (final Set<Cell> validIsland: validIslands) {
   if(tryWithNewIsland(bigPivot, validIsland)) {
       return true;
   }
   else {
       boardState.restoreState();
   }
}
return false;
```

```
private boolean tryWithNewIsland(PivotCell pivotCell, Set<Cell> validIsland) {
   for (final Cell cell: validIsland) {
       cell.setColor(CellColor.WHITE);
       cell.setPivotCell(pivotCell);
   }
   return tryToSolve(pivotCell.getBoard());
}
```

# Capitolo IV - MainActivity e layout

La classe **MainActivity** è presente in tutti i progetti creati in Android Studio e racchiude la logica dell'applicazione, collegando il resto delle classi ai componenti della User Interface definiti nei file .xml dedicati.

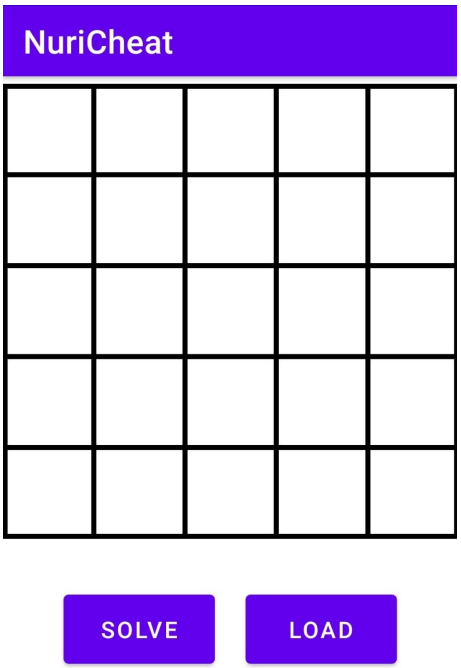
Il layout è molto semplice:

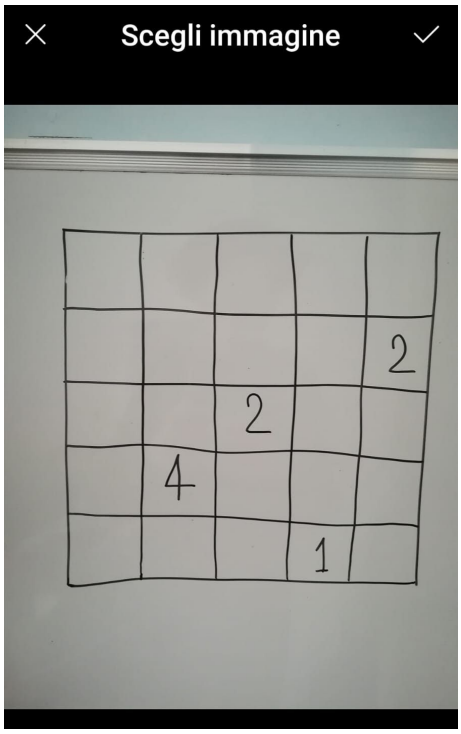
- la griglia è composta da 25 **TextArea** contenute e organizzate in un **GridLayout**.
- loadButton è un bottone che inizializza il processo di caricamento dell'immagine attraverso un oggetto **Intent**. In caso di successo, la griglia viene popolata in una funzione di callback.

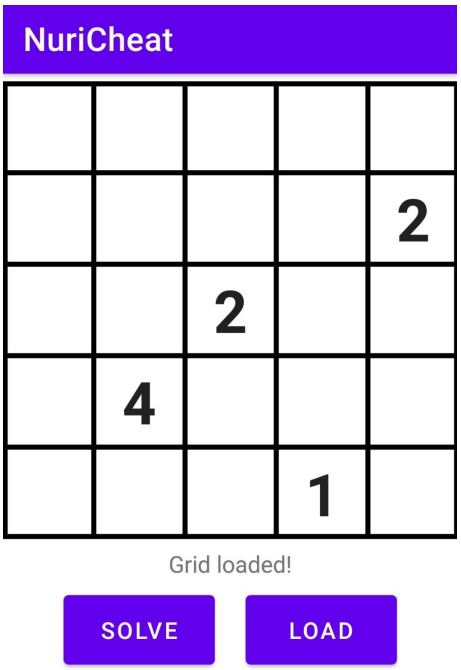
```
loadButton.setOnClickListener(v -> {
   Intent photoPickerIntent = new Intent(Intent.ACTION_PICK);
   photoPickerIntent.setType("image/*");
   startActivityForResult(photoPickerIntent, GALLERY_REQUEST);
});
```

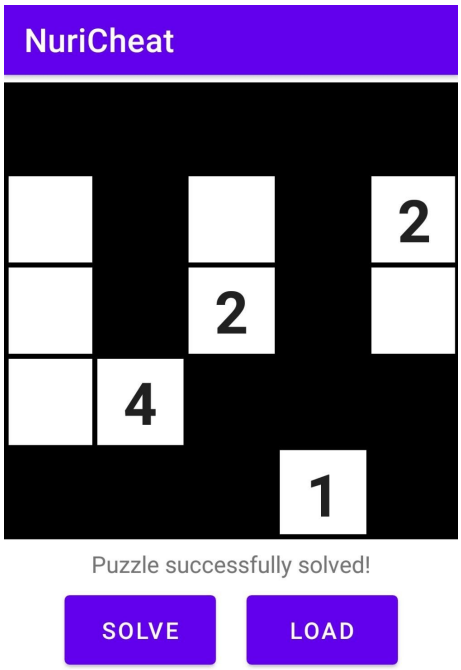
- solveButton è un bottone che chiama *tryToSolve*. In caso di successo, la griglia viene colorata.

```
solveButton.setOnClickListener(v -> {
   final Board board = new Board(grid);
   final boolean success = solver.tryToSolve(board);
   if(success) {
       for (Cell cell: board) {
           if (cell.isBlack()) {
               single_cell = (EditText)
gridLayout.getChildAt(BoardOps.coordsToIndex(cell.getX(), cell.getY()));
               single_cell.setBackgroundColor(Color.BLACK);
           }
       }
       alertView.setText("Puzzle successfully solved!");
   } else {
       alertView.setText("The grid could not be solved...");
   }
});
```


NuriCheat
SOLVE
LOAD


Scegli immagine
2
2
4
1


NuriCheat
2
2
4
1
Grid loaded!
SOLVE
LOAD


NuriCheat
2
2
4
1
Puzzle successfully solved!
SOLVE
LOAD

# Conclusioni e prospettive future

Sebbene l'applicazione funzioni a dovere e le performance siano soddisfacenti, un punto critico che potrebbe essere affrontato in eventuali estensioni è quello della dimensione della griglia.

Se per un puzzle 5x5 l'algoritmo di risoluzione non incontra alcun problema, lo stesso potrebbe non essere vero per griglie più grandi.

La fase di image processing è totalmente scalabile e non dovrebbe presentare gravi differenze, tuttavia l'algoritmo di forza bruta implementato in *inferIsland* porterebbe ad un aumento di peso esponenziale per griglie di maggiori dimensioni.

Detto questo, per le regole di creazione di configurazioni risolvibili, le griglie che richiedono l'utilizzo di *inferIsland* sono la minoranza; il resto andrebbe incontro ad un semplice aumento lineare dei tempi di risoluzione.

# Bibliografia

Studio delle strategie di risoluzione:
https://www.conceptispuzzles.com/index.aspx?uri=puzzle/nurikabe/techniques
http://www.huttar.net/lars-kathy/puzzles/nurikabe.html

Documentazione opencv:
https://docs.opencv.org/3.4/javadoc/index.html

Tensorflow Hub per la ricerca di modelli:
https://tfhub.dev/s?subtype=module,placeholder