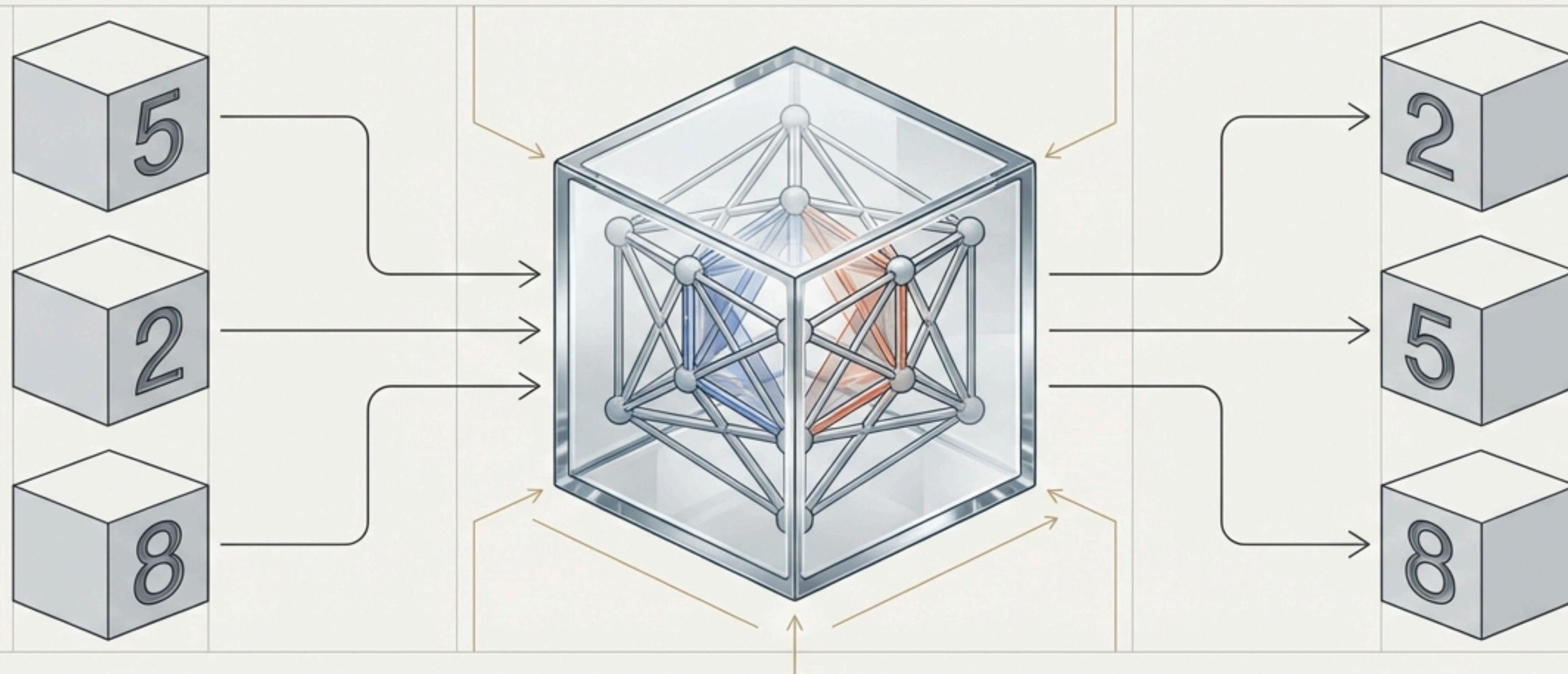


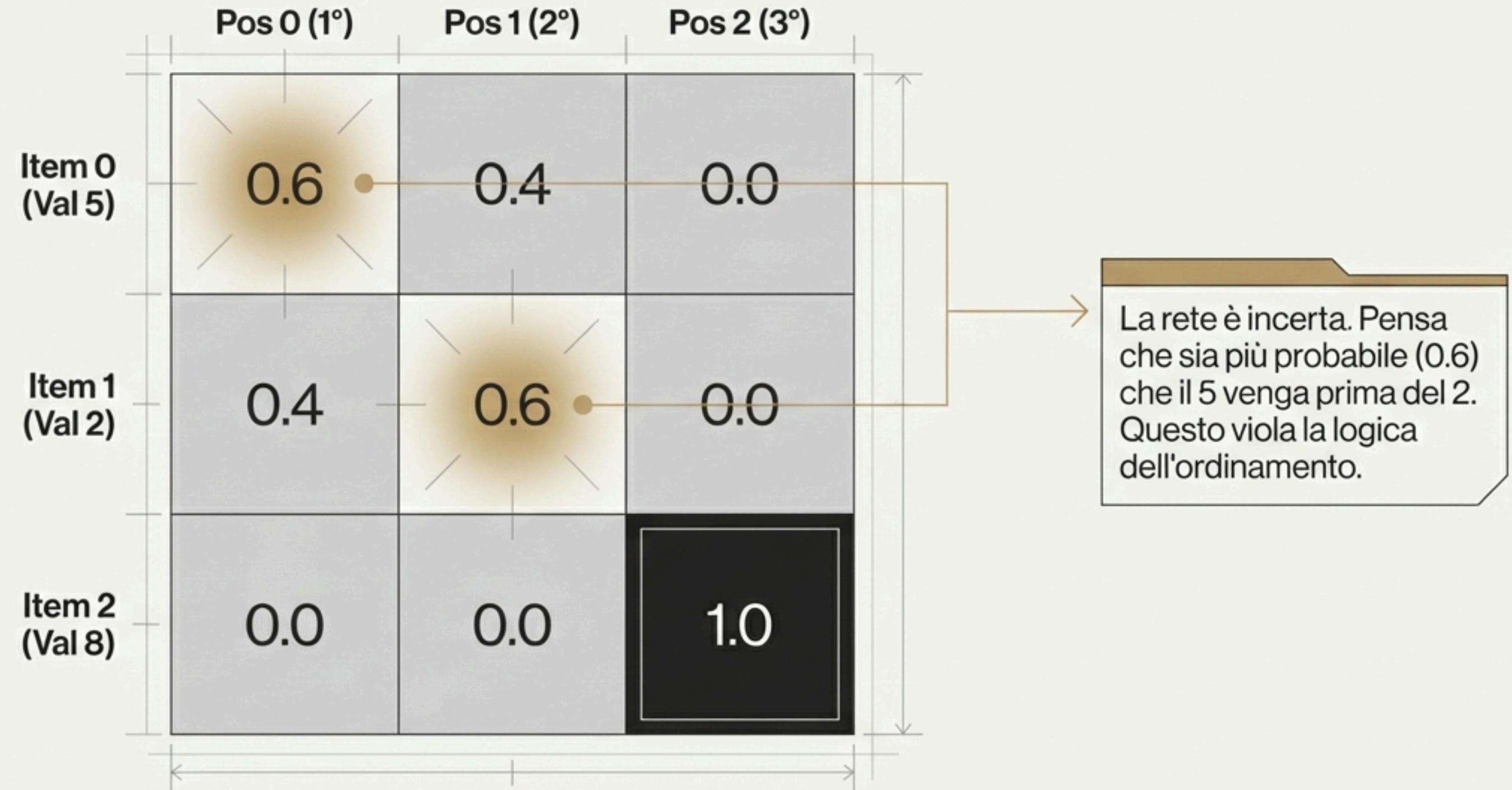
# Insegnare la Logica a una Rete Neurale

Oltre il Pattern Matching: Infondere Vincoli Simbolici nel Deep Learning



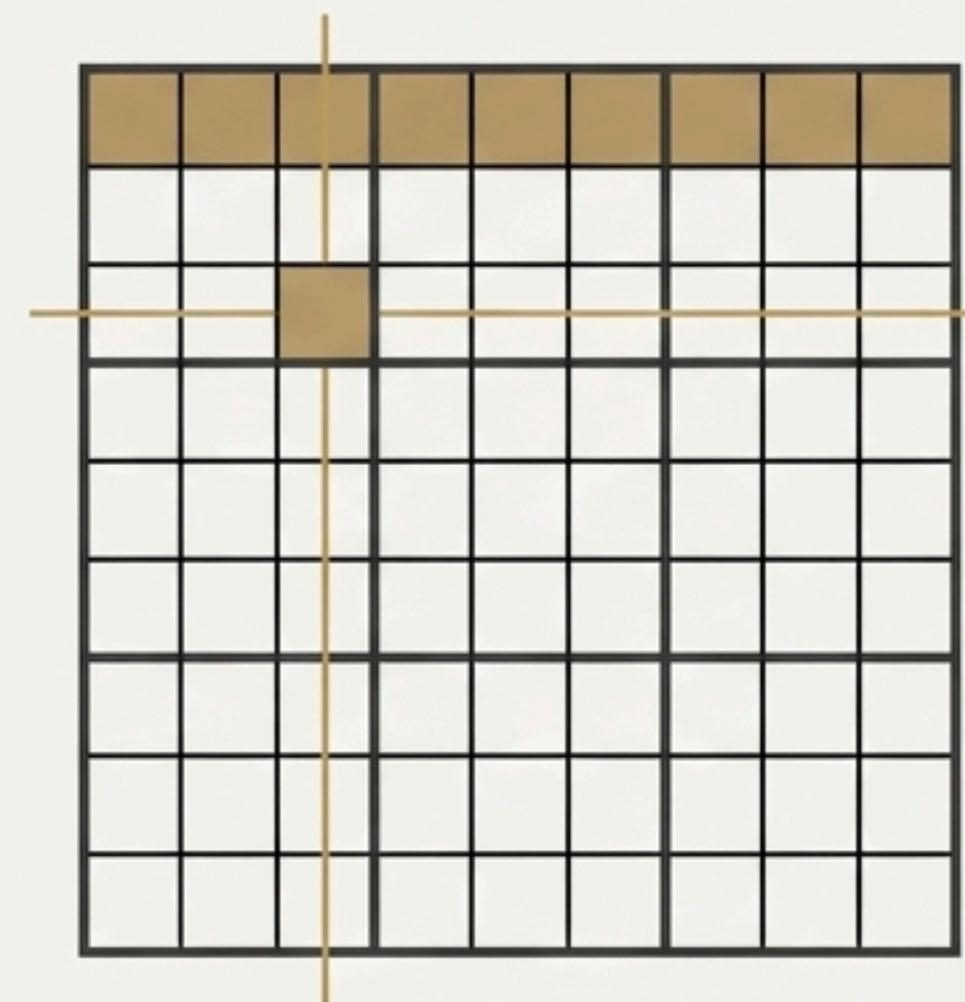
Come possiamo garantire che l'output non solo *sembri* corretto, ma rispetti *sempre* le regole?

# La Confusione della Rete

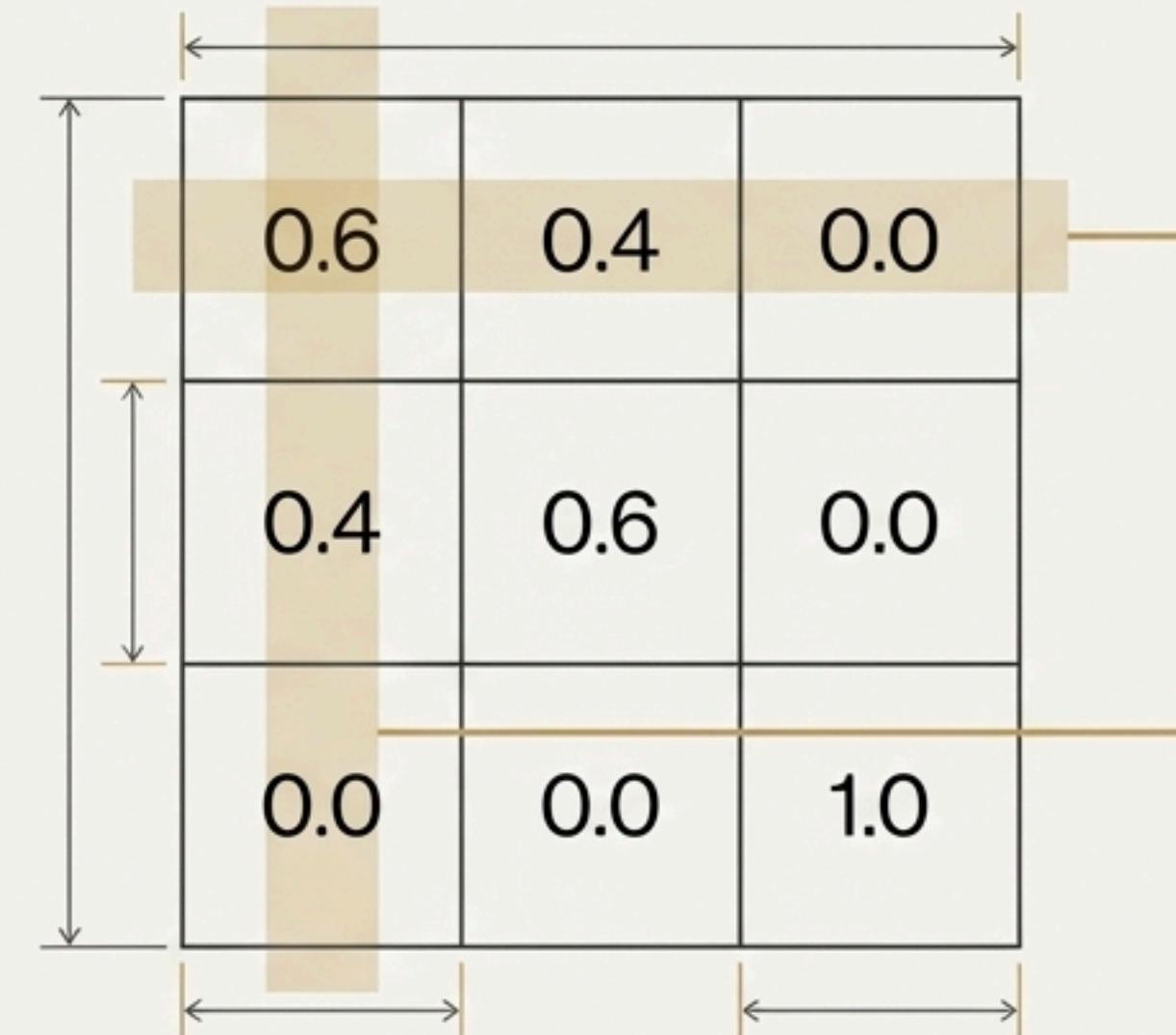


# Le Regole del Gioco: Vietato Barare

## Il Vincolo Strutturale “Exactly-One”



Il Controllo Sudoku



### Regola Righe

$$0.6 + 0.4 + 0.0 = 1.0$$

\*Spiegazione\*: Ogni numero deve finire in una sola posizione. Non può sparire o sdoppiarsi.

### Regola Colonne

$$0.6 + 0.4 + 0.0 = 1.0$$

\*Spiegazione\*: Ogni posizione deve contenere un solo numero.

Questo vincolo forza la rete a generare una Matrice di Permutazione valida.

## Approccio I:



Costruzione esplicita di alberi logici proposizionali (via SymPy).

$$(P_{i,j} \wedge P_{k,j+1}) \Rightarrow Val(i) \leq Val(k)$$

## (CNF Simbolica)

### Come Funziona (Il Divieto)

1.

Il sistema vede che  $Val(5) > Val(2)$ .

2.

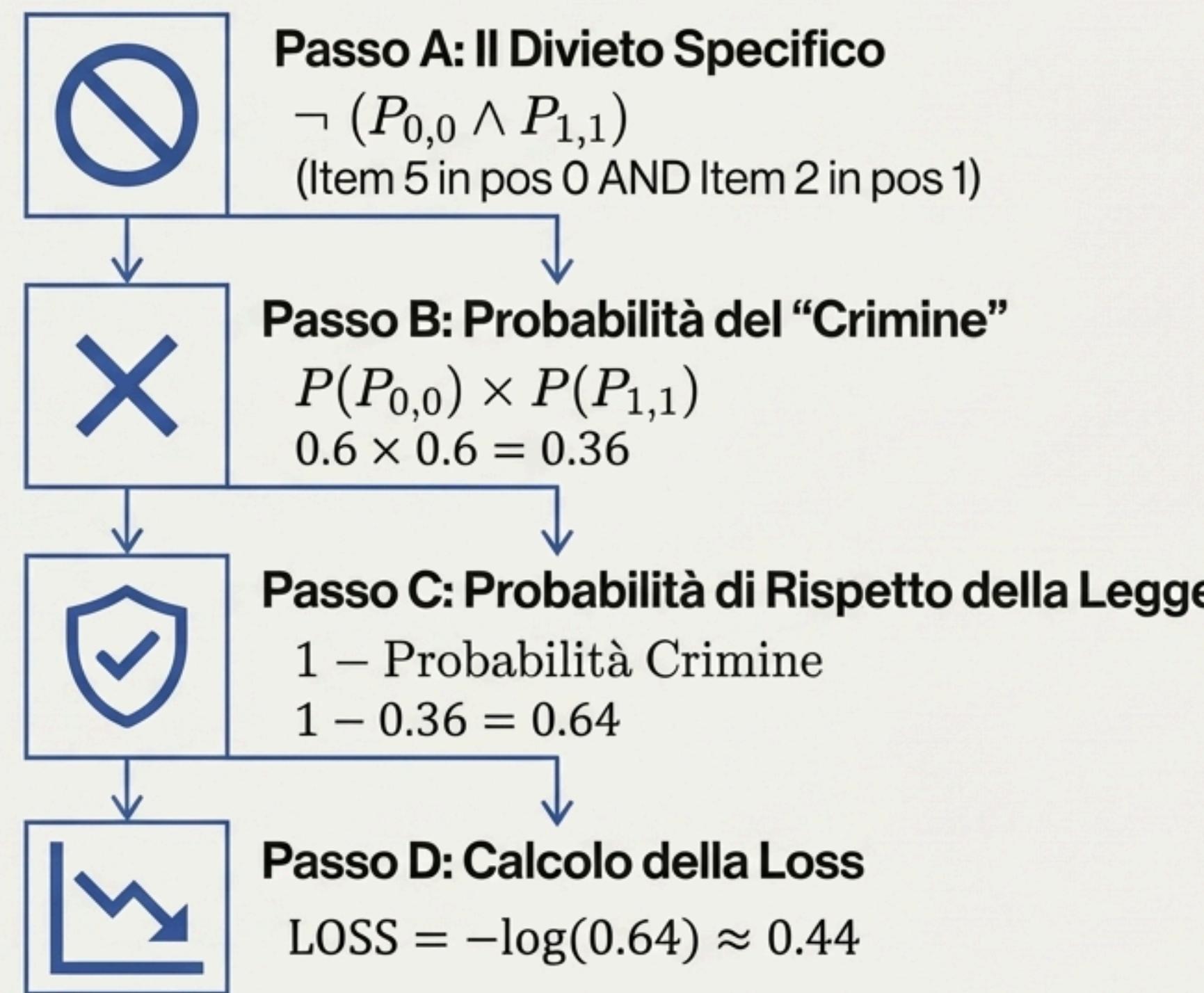
Scrive una clausola di conflitto:  
“È VIETATO che l’Item 5 sia in posizione 0  
E l’Item 2 sia in posizione 1.”

3. Formula CNF:

$$\neg P_{i,j} \vee \neg P_{k,j+1}$$

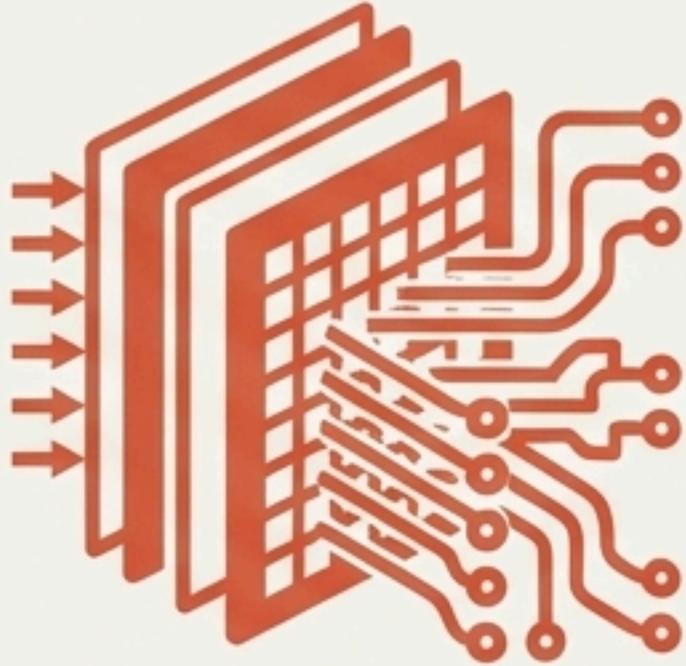
**Limitazione Chiave:** Richiede una valutazione ricorsiva dell’albero su CPU. È preciso, ma lento e non scalabile.

# La CNF in Azione: Calcolare la Probabilità del “Reato”



La loss punisce la rete specificamente per la violazione logica identificata.

# Approccio II: (Pairwise Ottimizzata)



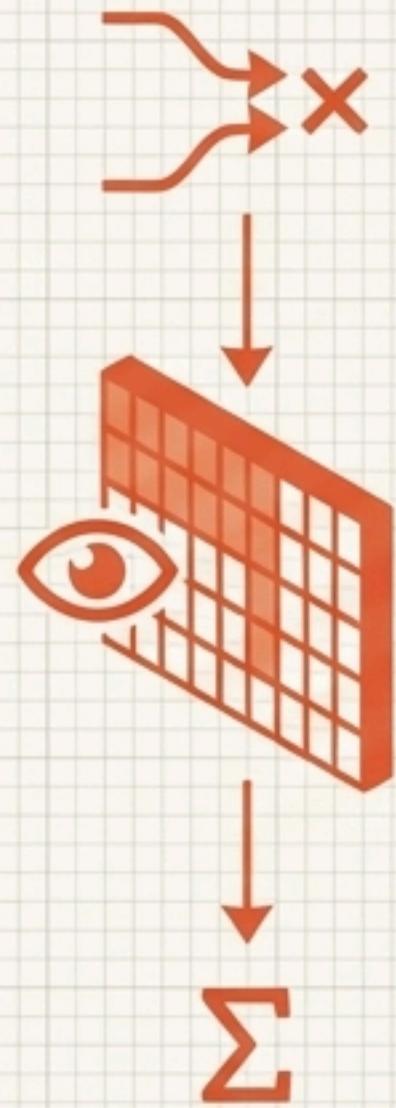
**Method:** Operazioni matriciali su GPU (basate su Tensori).

**The Mechanism:** Calcolare la probabilità congiunta per tutte le coppie simultaneamente usando la Moltiplicazione di Matrici (Batch Matrix Multiplication).

$$\mathcal{L} \propto -\log \sum_{i,k} \left( P(i_{@j}) \cdot P(k_{@j+1}) \cdot \mathbf{M}_{i,k} \right)$$

**Vantaggio Decisivo:** Parallelizza la valutazione logica, rendendo il training scalabile e mantenendo le garanzie simboliche.

# La Pairwise in Azione: Il Filtro delle Coppie Valide



## Passo A: Generare le “Storie” Possibili (Probabilità Congiunta)

- Storia Sbagliata:  $P(5 \text{ in pos 0 E } 2 \text{ in pos 1}) = 0.6 \times 0.6 = 0.36$
- Storia Giusta:  $P(2 \text{ in pos 0 E } 5 \text{ in pos 1}) = 0.4 \times 0.4 = 0.16$

## Passo B: Applicare la Maschera di Validità ( $M_{i,k}$ )

0.36	0	X
0.16	1	✓

Coppia (5, 2)  $\rightarrow 5 \leq 2$  è Falso  $\rightarrow$  Maschera = 0. La probabilità 0.36 viene azzerata.  
Coppia (2, 5)  $\rightarrow 2 \leq 5$  è Vero  $\rightarrow$  Maschera = 1. La probabilità 0.16 sopravvive.

## Passo C: Calcolo della Loss

Probabilità Totale di essere Ordinati = 0 + 0.16 = 0.16  
Calculation: LOSS =  $-\log(0.16) \approx 1.83$

► La loss si basa sulla probabilità totale di tutte le configurazioni \*valide\*.

# Due Filosofie a Confronto



## CNF (L'Avvocato)

### Logica

“C'è il 36% di probabilità che tu stia mettendo specificamente il 5 prima del 2. Ti punisco per questo errore specifico.”

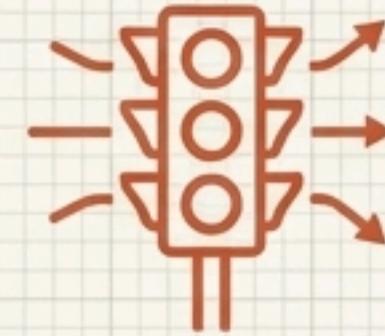
### Azione sulla Rete

Spinge la rete ad **abbassare** la probabilità della configurazione errata.



### Caratteristica

Precisa e mirata.



## Pairwise (Il Vigile del Traffico)

### Logica

“Solo il 16% di tutte le tue strade possibili porta a una lista ordinata. Sei bocciato.”

### Azione sulla Rete

Spinge la rete ad **aumentare** la probabilità delle configurazioni corrette.



### Caratteristica

Globale e ottimizzata.

# Dal Concetto al Codice

## semantic\_loss\_ordering\_cnf

```
# ...
# Costruisce clausole con SymPy
clause = Or(Not(p_i_j), Not(p_k_j1))
cnf_formula = And(*active_clauses)

# Valutazione ricorsiva
prob_cnf_satisfied = eval_formula_prob(...)
# ...
```

Ottimizzato con caching per costruire la struttura logica una sola volta.

## semantic\_loss\_pairwise\_ordering

```
# ...
# Calcolo parallelo della prob. congiunta
joint_prob = torch.bmm(prob_at_j.unsqueeze(2),
                      prob_at_j_plus_1.unsqueeze(1))

# Maschera booleana basata sui valori
valid_pair_mask = (labels_i <= labels_k).float()

# Filtro e somma
prob_satisfied = torch.sum(joint_prob *
                            valid_pair_mask)

# ...
```

Due implementazioni dello stesso principio logico, ottimizzate per due paradigmi di calcolo diversi: sequenziale (CPU) e parallelo (GPU).