Sudoku Constraints with Tensors (4×4)

This notebook explains step by step how to build the following loss terms (row / column / block / givens) **using concrete examples**:

- `row_sum = P.sum(dim=1)`
- `col_sum = P.sum(dim=0)`
- block slicing + `reshape`
- givens masking + `nll_loss(log(P))`

We work with a **4×4 Sudoku** (digits 1..4) and **2×2 blocks** to keep the tensor operations transparent.

In [ ]:
```
import torch
import torch.nn.functional as F

torch.set_printoptions(precision=2, sci_mode=False)
```

1) What is `P`?

`P` is a probability tensor with shape `(4, 4, 4)`:

- `P[r, c, k]` = probability that cell `(r, c)` contains digit `(k + 1)`.

For each cell we require:

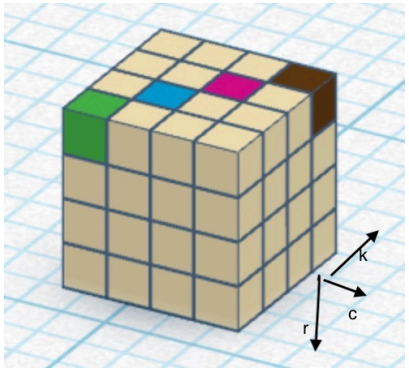
[\sum_k P[r, c, k] = 1]

Example

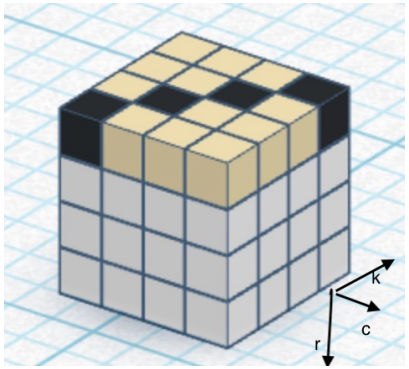
We have a Sudoku grid where `0` means “unknown”:

```
[1, 2, 3, 4]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

To represent digits as probabilities, we extend the 4×4 grid `(r × c)` into a 4×4×4 tensor `(r × c × k)`. You can visualize this tensor as a cube.



If we translate this idea to `P`, we get the following cube:



- Gray cubes represent a probability of `0.25` (uniform uncertainty)
- Beige cubes represent `0.0`
- Black cubes represent given digits with value `1.0`

In the next step we sum over the digit dimension `k`, so we end up with a matrix of shape `(r × c)`. Ideally, the sum over `k` should always be `1`.

In [ ]:
```
# Simple example for P:
# - row 0 is fixed and correct (one-hot per cell)
# - all other cells are initialized uniformly (0.25 per digit)

P = torch.zeros(4,4,4)

# Row 0: [1,2,3,4] as one-hot
P[0,0,0] = 1
P[0,1,1] = 1
P[0,2,2] = 1
P[0,3,3] = 1

# Remaining cells: uniform uncertainty
P[1:,:,:] = 0.25
print("P[0,0,k] =", P[0,0,:], "Sum_P[0,0,:] =", P[0,0,:].sum())
print("P[0,1,k] =", P[0,1,:], "Sum_P[0,1,:] =", P[0,1,:].sum())
print("P[0,2,k] =", P[0,2,:], "Sum_P[0,2,:] =", P[0,2,:].sum())
print("P[0,3,k] =", P[0,3,:], "Sum_P[0,3,:] =", P[0,3,:].sum())

print("P[1,0,k] =", P[1,0,:], "Sum_P[1,0,:] =", P[1,0,:].sum())
print("P[1,1,k] =", P[1,1,:], "Sum_P[1,1,:] =", P[1,1,:].sum())
print("P[1,2,k] =", P[1,2,:], "Sum_P[1,2,:] =", P[1,2,:].sum())
print("P[1,3,k] =", P[1,3,:], "Sum_P[1,3,:] =", P[1,3,:].sum())

print("P[2,0,k] =", P[2,0,:], "Sum_P[2,0,:] =", P[2,0,:].sum())
print("P[2,1,k] =", P[2,1,:], "Sum_P[2,1,:] =", P[2,1,:].sum())
print("P[2,2,k] =", P[2,2,:], "Sum_P[2,2,:] =", P[2,2,:].sum())
print("P[2,3,k] =", P[2,3,:], "Sum_P[2,3,:] =", P[2,3,:].sum())

print("P[3,0,k] =", P[3,0,:], "Sum_P[3,0,:] =", P[3,0,:].sum())
print("P[3,1,k] =", P[3,1,:], "Sum_P[3,1,:] =", P[3,1,:].sum())
print("P[3,2,k] =", P[3,2,:], "Sum_P[3,2,:] =", P[3,2,:].sum())
print("P[3,3,k] =", P[3,3,:], "Sum_P[3,3,:] =", P[3,3,:].sum())

# Check: each cell sums over k to 1
```

```
cell_sums = P.sum(dim=2)
cell_sums
```

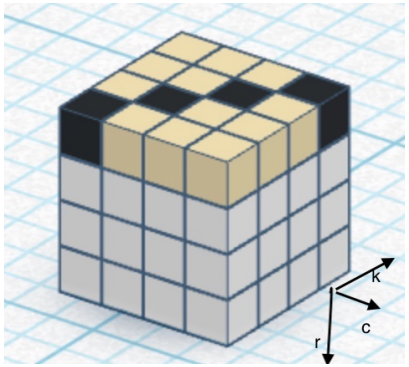
If everything is consistent, `cell_sums` should be `1.0` everywhere.

2) Row Constraint (Row Uniqueness)

For each row `r` and each digit `k`:

[\sum_c P[r, c, k] = 1]

Interpretation: In a row, each digit (1..4) should appear **exactly once**.



- Gray cubes represent a probability of `0.25`
- Beige cubes represent `0.0`
- Black cubes represent given digits with value `1.0`

In the following example we sum over the column dimension `c`, resulting in a matrix of shape `(r × k)`. Ideally, the sum over `c` should be `1` for every `(r, k)`.

In [ ]:
```
print("P[0,j,0] =", P[0,:,0], "Sum_P[0,:,0] =", P[0,:,0].sum())
print("P[0,j,1] =", P[0,:,1], "Sum_P[0,:,1] =", P[0,:,1].sum())
print("P[0,j,2] =", P[0,:,2], "Sum_P[0,:,2] =", P[0,:,2].sum())
print("P[0,j,3] =", P[0,:,3], "Sum_P[0,:,3] =", P[0,:,3].sum())
```

```
print("P[1,j,0] =", P[1,:,0], "Sum_P[1,:,0] =", P[1,:,0].sum())
print("P[1,j,1] =", P[1,:,1], "Sum_P[1,:,1] =", P[1,:,1].sum())
print("P[1,j,2] =", P[1,:,2], "Sum_P[1,:,2] =", P[1,:,2].sum())
print("P[1,j,3] =", P[1,:,3], "Sum_P[1,:,3] =", P[1,:,3].sum())

print("P[2,j,0] =", P[2,:,0], "Sum_P[2,:,0] =", P[2,:,0].sum())
print("P[2,j,1] =", P[2,:,1], "Sum_P[2,:,1] =", P[2,:,1].sum())
print("P[2,j,2] =", P[2,:,2], "Sum_P[2,:,2] =", P[2,:,2].sum())
print("P[2,j,3] =", P[2,:,3], "Sum_P[2,:,3] =", P[2,:,3].sum())

print("P[3,j,0] =", P[3,:,0], "Sum_P[3,:,0] =", P[3,:,0].sum())
print("P[3,j,1] =", P[3,:,1], "Sum_P[3,:,1] =", P[3,:,1].sum())
print("P[3,j,2] =", P[3,:,2], "Sum_P[3,:,2] =", P[3,:,2].sum())
print("P[3,j,3] =", P[3,:,3], "Sum_P[3,:,3] =", P[3,:,3].sum())

row_sum = P.sum(dim=1)  # Summe über Spalten j
row_sum
```

- `P` has shape `(4, 4, 4)`
- `P.sum(dim=1)` sums over the **columns** ⇒ output shape `(4, 4)` = `(rows, digits)`

`row_sum[i, k]` tells you: *How much total probability does row* `i` *assign to digit* `k`*?*

In [ ]:
```
L_row = ((row_sum - 1.0) ** 2).sum()
L_row
```

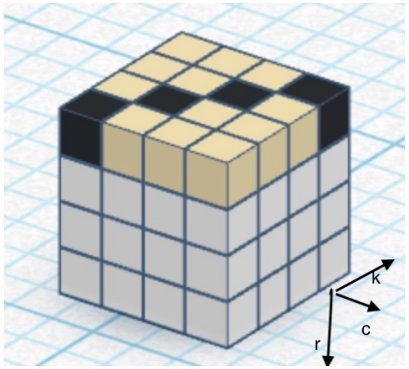
The row loss is `0` if **every** row has sum `1` for **every** digit.

3) Column Constraint (Column Uniqueness)

For each column `c` and each digit `k`:

[\sum_r P[r, c, k] = 1]

Interpretation: In a column, each digit (1..4) should appear **exactly once**.



- Gray cubes represent a probability of `0.25`
- Beige cubes represent `0.0`
- Black cubes represent given digits with value `1.0`

In the following example we sum over the row dimension `r`, resulting in a matrix of shape `(c × k)`. Ideally, the sum over `r` should be `1` for every `(c, k)`.

In [ ]:
```
print("P[i,0,0] =", P[:,0,0], "Sum_P[:,0,0] =", P[:,0,0].sum())
print("P[i,0,1] =", P[:,0,1], "Sum_P[:,0,1] =", P[:,0,1].sum())
print("P[i,0,2] =", P[:,0,2], "Sum_P[:,0,2] =", P[:,0,2].sum())
print("P[i,0,3] =", P[:,0,3], "Sum_P[:,0,3] =", P[:,0,3].sum())

print("P[i,1,0] =", P[:,1,0], "Sum_P[:,1,0] =", P[:,1,0].sum())
print("P[i,1,1] =", P[:,1,1], "Sum_P[:,1,1] =", P[:,1,1].sum())
print("P[i,1,2] =", P[:,1,2], "Sum_P[:,1,2] =", P[:,1,2].sum())
print("P[i,1,3] =", P[:,1,3], "Sum_P[:,1,3] =", P[:,1,3].sum())

print("P[i,2,0] =", P[:,2,0], "Sum_P[:,2,0] =", P[:,2,0].sum())
print("P[i,2,1] =", P[:,2,1], "Sum_P[:,2,1] =", P[:,2,1].sum())
print("P[i,2,2] =", P[:,2,2], "Sum_P[:,2,2] =", P[:,2,2].sum())
print("P[i,2,3] =", P[:,2,3], "Sum_P[:,2,3] =", P[:,2,3].sum())

print("P[i,3,0] =", P[:,3,0], "Sum_P[:,3,0] =", P[:,3,0].sum())
print("P[i,3,1] =", P[:,3,1], "Sum_P[:,3,1] =", P[:,3,1].sum())
print("P[i,3,2] =", P[:,3,2], "Sum_P[:,3,2] =", P[:,3,2].sum())
print("P[i,3,3] =", P[:,3,3], "Sum_P[:,3,3] =", P[:,3,3].sum())
```

```
col_sum = P.sum(dim=0)  # Summe über Zeilen i
col_sum
```

- `P.sum(dim=0)` sums over the **rows** ⇒ output shape `(4, 4)` = `(cols, digits)`

`col_sum[j, k]` tells you: *How much total probability does column* `j` *assign to digit* `k` *?*

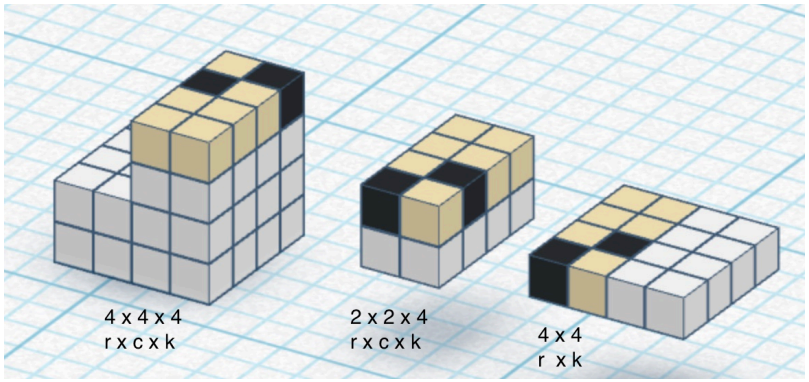
In [ ]:
```
L_col = ((col_sum - 1.0) ** 2).sum()
L_col
```

4) Block Constraint (2×2 Blocks)

For each 2×2 block `b` and each digit `k` :

[\sum_{(r,c) \in b} P[r, c, k] = 1]

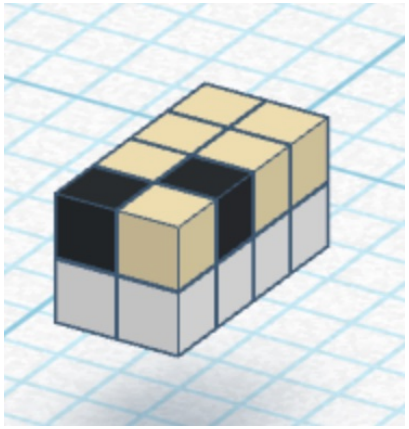
We extract blocks via slicing and sum over the 4 cells of each block.



- Gray cubes represent a probability of `0.25`
- Beige cubes represent `0.0`
- Black cubes represent given digits with value `1.0`

4.1) Inspect a block (top-left)

The top-left block covers rows `0..1` and columns `0..1`.

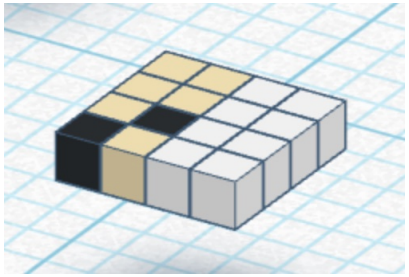


In [ ]:
```
blk = P[0:2, 0:2, :]   # (2,2,4)

print("blk[0,0,:] =", blk[0,0,:])
print("blk[0,1,:] =", blk[0,1,:])
print("blk[1,0,:] =", blk[1,0,:])
print("blk[1,1,:] =", blk[1,1,:])

blk.shape, blk
```

4.2) Flatten the block + sum per digit

We want to treat the four cells of the block as a list (`N = 4`):

- `blk.reshape(-1, 4)` converts `(2, 2, 4) → (4, 4)`
- then we sum over the 4 cells (`dim=0`) ⇒ output shape `(4,)` (one sum per digit)



In [ ]:
```
blk_flat = blk.reshape(-1, 4)     # (4,4)

blk_sum  = blk_flat.sum(dim=0)    # (4,)
blk_flat.shape, blk_flat, blk_sum
```

In [ ]:
```
blk_loss = ((blk_sum - 1.0) ** 2).sum()
blk_loss
```

4.3) Iterate over all 2×2 blocks

For a 4×4 Sudoku with 2×2 blocks, block rows start at `br = 0, 2` and block columns at `bc = 0, 2`.

In [ ]:
```
L_blk = 0.0
for br in range(0, 4, 2):
    for bc in range(0, 4, 2):
        print(br, bc)
        blk = P[br:br+2, bc:bc+2, :]       # (2,2,4)
        blk_sum = blk.reshape(-1, 4).sum(dim=0)  # (4,)
        L_blk = L_blk + ((blk_sum - 1.0) ** 2).sum()

L_blk
```

5) Givens Constraint (Hard Clues)

`puzzle` contains:

- `0` = empty

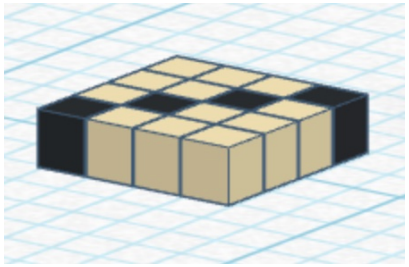
- `1..4` = given digit (clue)

We build:

- `givens_mask = puzzle > 0` (boolean mask)
- `givens_target = puzzle - 1` (0-based classes: `0..3`)

Then:

- collect the probabilities of the given cells: `given_P = P[givens_mask]` → shape `(Ngivens, 4)`
- collect the correct class indices: `targets = givens_target[givens_mask]` → shape `(Ngivens,)`



Loss: `nll_loss(log(P), targets)`

In [ ]:
```
puzzle = torch.tensor([
    [1, 2, 3, 4],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
])

givens_mask = puzzle > 0
givens_target = puzzle.clamp(min=1) - 1  # 1..4 -> 0..3; 0 wird durch clamp sicher gemacht

puzzle, givens_mask, givens_target
```

In [ ]:
```
given_P = P[givens_mask]                # (Ngivens, 4)
targets = givens_target[givens_mask]    # (Ngivens,)
given_P.shape, given_P, targets
```

Why `clamp(min=1)`?

Because empty cells are `0`, and `0 - 1` would become `-1` (invalid index).

Important: empty cells are excluded by `givens_mask` anyway, so their target value does not matter for the loss.

In [ ]:
```
eps = 1e-9
if givens_mask.any():
    print("log.given_P[0,0]+eps =", (given_P[0,0]+eps).log())
    print("log.given_P[1,1]+eps =", (given_P[1,1]+eps).log())
    print("log.given_P[2,2]+eps =", (given_P[2,2]+eps).log())
    print("log.given_P[3,3]+eps =", (given_P[3,3]+eps).log())
    L_giv = F.nll_loss((given_P + eps).log(), targets, reduction="sum")
else:
    L_giv = P.new_tensor(0.0)

L_giv
```

7) Entropy Loss (Encouraging Confident Predictions)

In addition to the structural constraints, we introduce an **entropy loss** that controls the uncertainty of the probability tensor `P`.

For a single cell `(r, c)`, the entropy of the probability distribution over digits is defined as:

[H(P[r,c,:]) = - \sum_k P[r,c,k] \log(P[r,c,k])]

In code, this is computed as:

```
ent = -(P * (P + eps).log()).sum(dim=2)  # shape: (rows, columns)
L_ent = ent.sum()
```

In [ ]:
```
ent = -(P * (P + eps).log()).sum(dim=2)  # (9,9)
L_ent = ent.sum()

-(P * (P + eps).log()), ent, L_ent
```

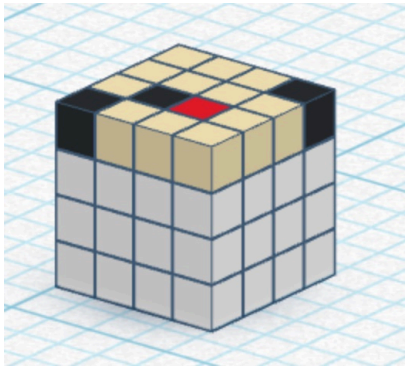
8) Total Loss (Example)

You can weight the individual parts and sum them into a single scalar loss.

In [ ]:
```
w_row, w_col, w_blk, w_giv, w_ent = 1.0, 1.0, 1.0, 2.0, 0.01
L_total = w_row*L_row + w_col*L_col + w_blk*L_blk + w_giv*L_giv + w_ent*L_ent
L_row, L_col, L_blk, L_giv, L_total
```

7) Mini Experiment: Create a Row Error on Purpose

We break row 0 by assigning the same digit to two cells.



In [ ]:
```
P_bad = P.clone()
# Set (0,2) also to digit 2 (index 1) instead of 3 (index 2)
P_bad[0,2,:] = 0
P_bad[0,2,1] = 1

row_sum_bad = P_bad.sum(dim=1)
L_row_bad = ((row_sum_bad - 1.0) ** 2).sum()
row_sum_bad, L_row, L_row_bad
```

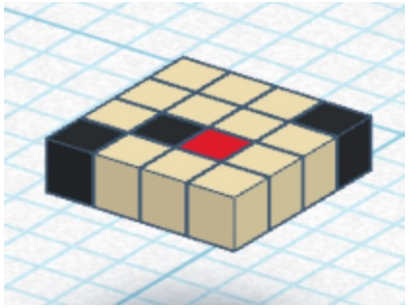
You can see: as soon as a digit gets *too much* probability within a row, the row loss increases.

8) Effect of Violating a Given Cell

In this example, we deliberately violate one of the given cells by assigning it a very low probability for the correct digit. This allows us to observe how the **givens loss reacts to hard constraint violations**.

First, we extract only the probabilities and targets corresponding to the given cells:

- `given_P_bad` contains the predicted probabilities for the given cells



- `targets` contains the correct digit indices for those cells

In [ ]:
```
given_P_bad = P_bad[givens_mask]                  # (Ngivens, 4)
targets = givens_target[givens_mask]     # (Ngivens,)
given_P.shape, given_P_bad, targets

eps = 1e-9
if givens_mask.any():
    print("log.given_P_bad[0,0]+eps =", (given_P_bad[0,0]+eps).log())
    print("log.given_P_bad[1,1]+eps =", (given_P_bad[1,1]+eps).log())
    print("log.given_P_bad[2,2]+eps =", (given_P_bad[2,2]+eps).log())
    print("log.given_P_bad[3,3]+eps =", (given_P_bad[3,3]+eps).log())
    L_giv = F.nll_loss((given_P_bad+ eps).log(), targets, reduction="sum")
else:
    L_giv = P.new_tensor(0.0)

L_giv
```

You can see: the loss of P_bad[2,2] is very high because its value is 0. We would expect a value of 1.0 for a correct solution.

Conclusion: Why the Givens Loss Dominates

Observation:
In the examples above, the *givens loss* increases much more strongly than the row, column, or block losses when a given cell is violated.

This is **intentional**.

Given digits (“givens”) are **hard constraints**: they must never be violated. Row, column, and block constraints mainly enforce *structural consistency*, while givens restrict the solution space.

Mathematically, this difference comes from the loss formulation:

- **Row / Column / Block losses** are quadratic penalties and grow *gradually* with the constraint violation.
- **Givens loss** is a **negative log-likelihood**. As soon as the correct digit gets very small probability in a given cell, the loss grows rapidly (logarithmically towards ∞).

Intuition:

- Constraint losses say: *“This solution is structurally inconsistent.”*
- Givens loss says: *“This solution is wrong.”*

At this point we have defined what a valid Sudoku solution means — but not yet **how** to minimize these losses. That is the topic of the next chapter, where we introduce the optimization algorithm.

In [ ]: