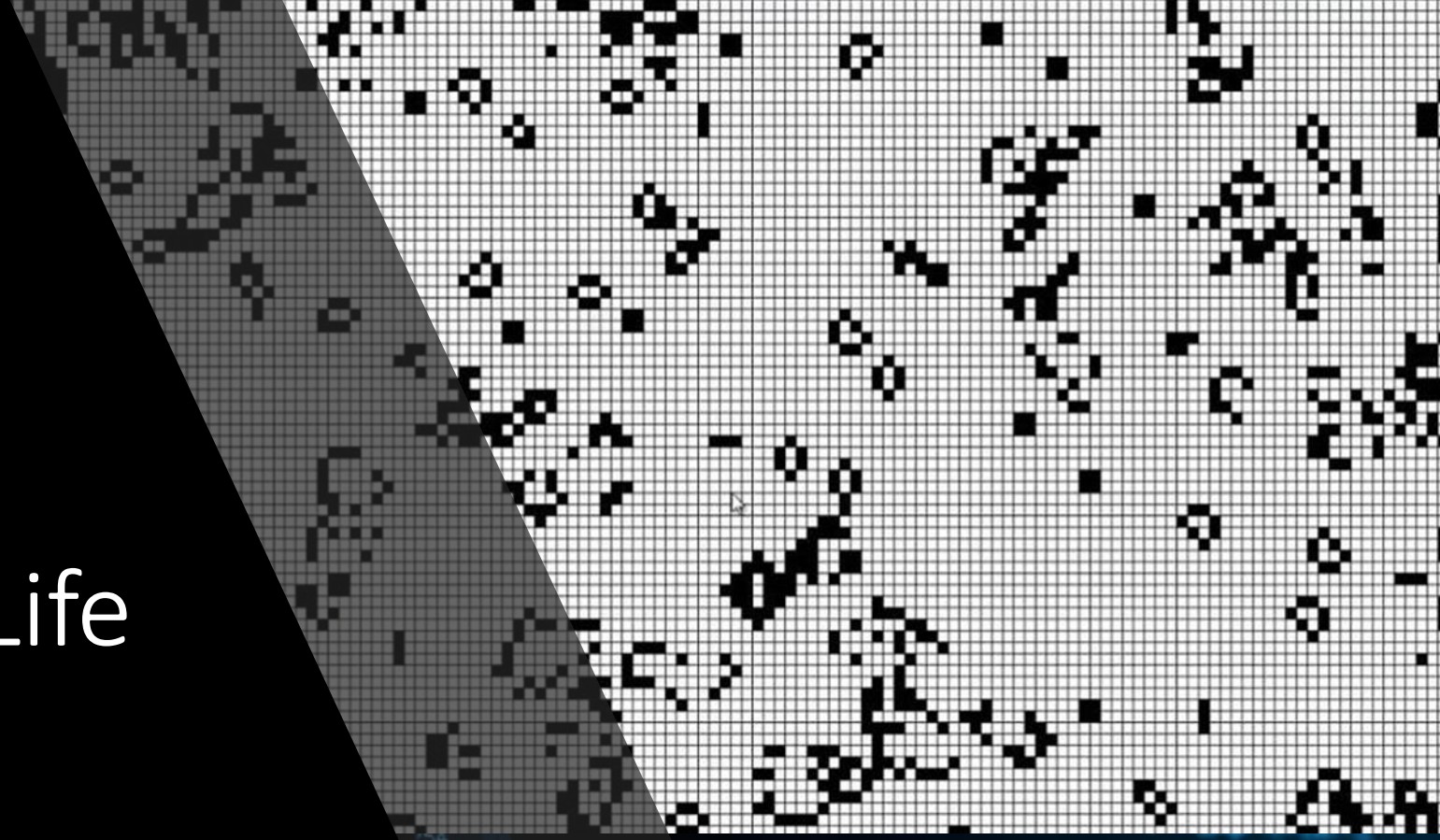
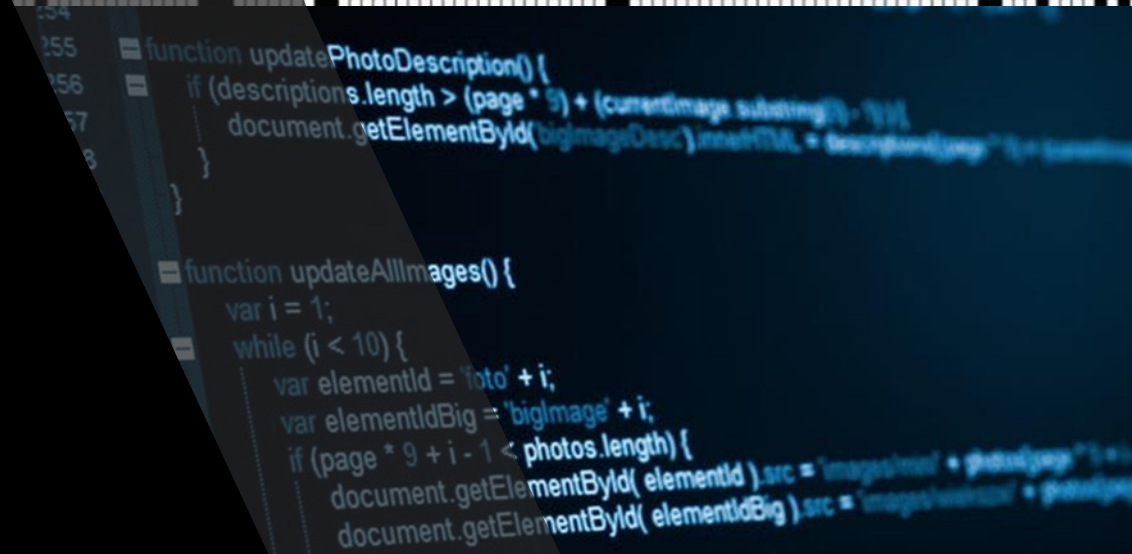
Coding with Neural Networks & Conway's Game of Life





Luca Scimeca &
Andrzej Banburski

ls769@cam.ac.uk

Where it all started….

RNN Adventures

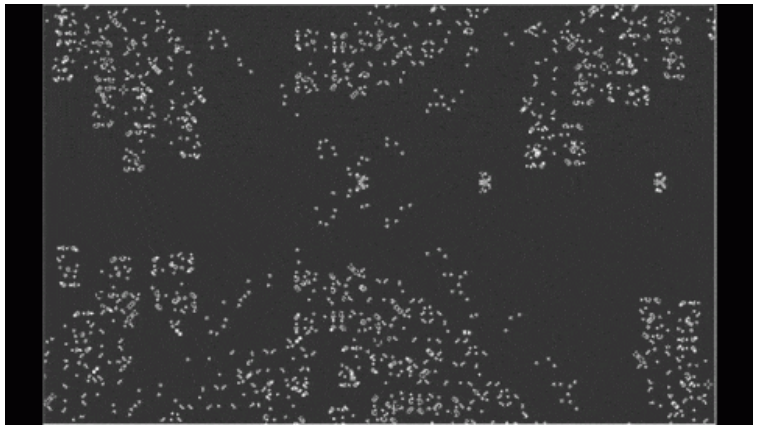
- Can an RNN learn Game of Life?
- Can an RNN simulate basic program structures (e.g. for loops)?

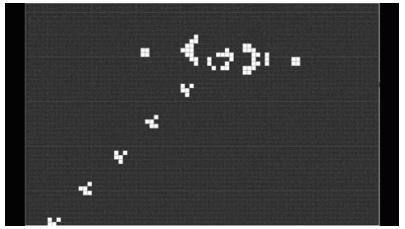
From Andrzej Banburski's presentation

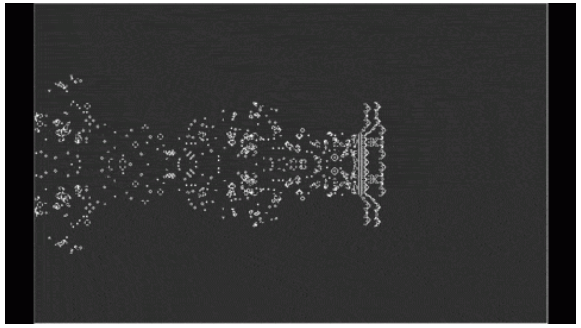
John Conway’s Game of Life

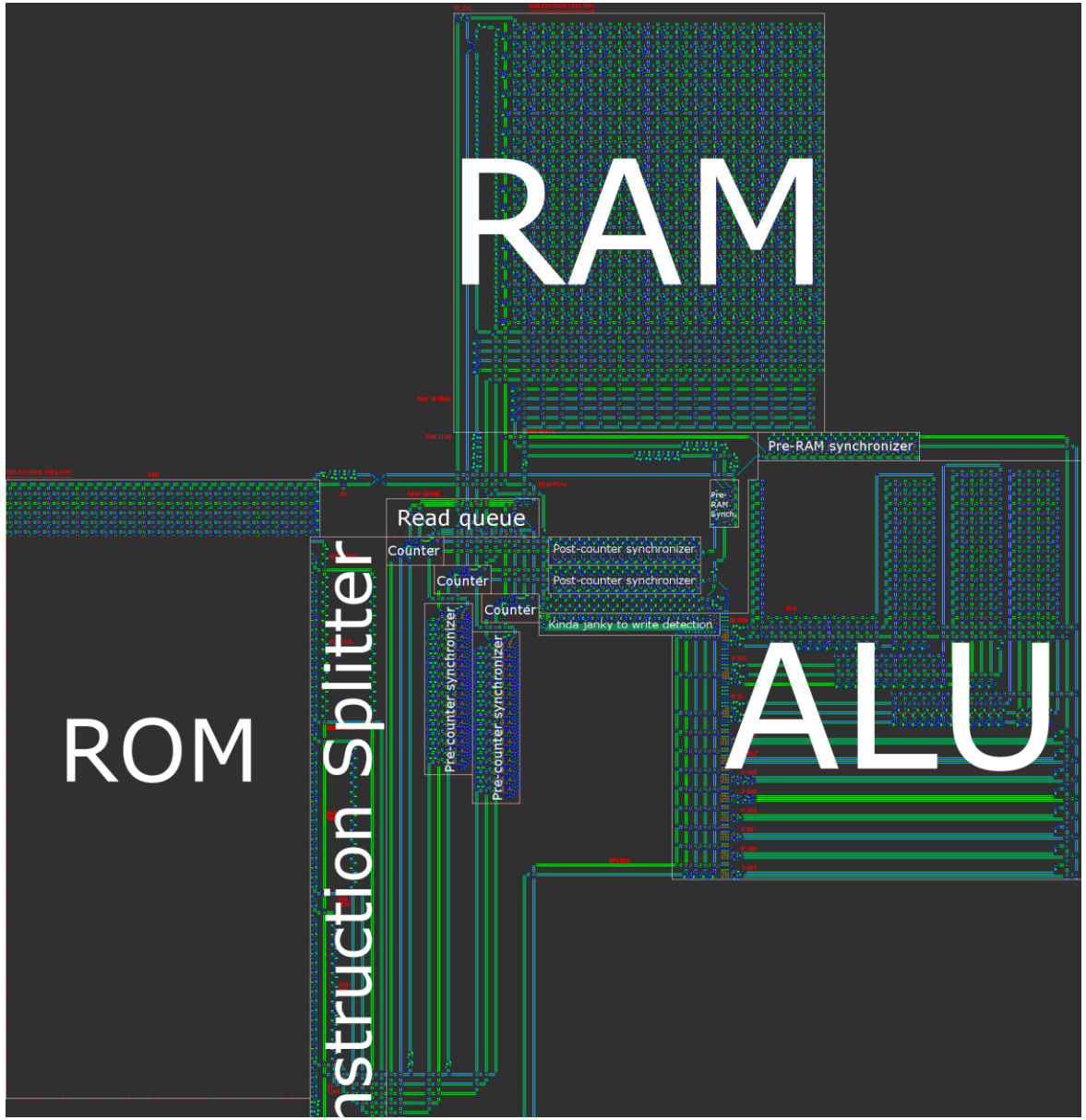
- Rules:

  - Each cell with one or no neighbors dies, as if by solitude.
  - Each cell with four or more neighbors dies, as if by overpopulation.
  - Each cell with two or three neighbors survives.
  - Each cell with three neighbors becomes populated.

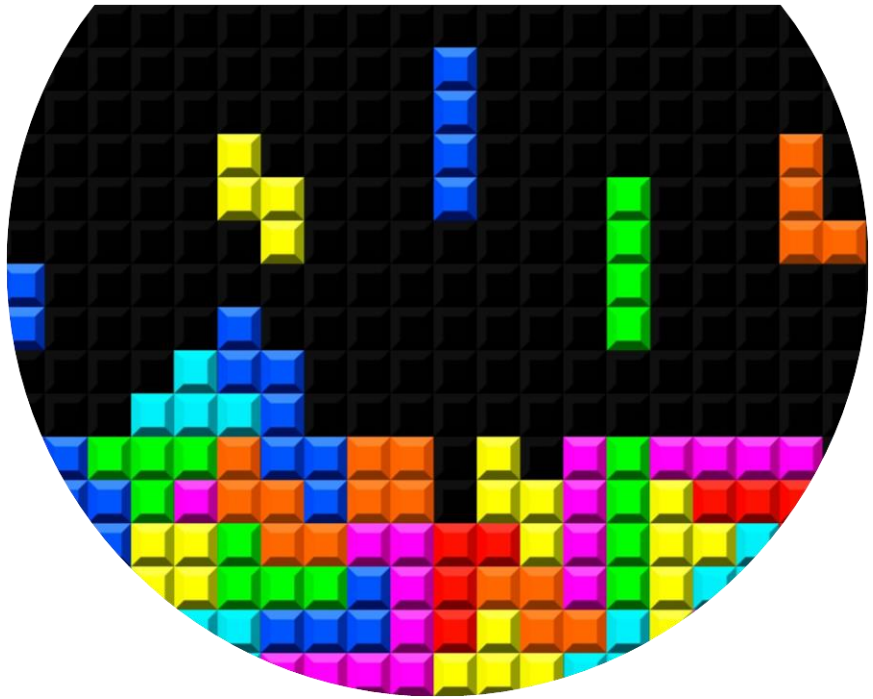




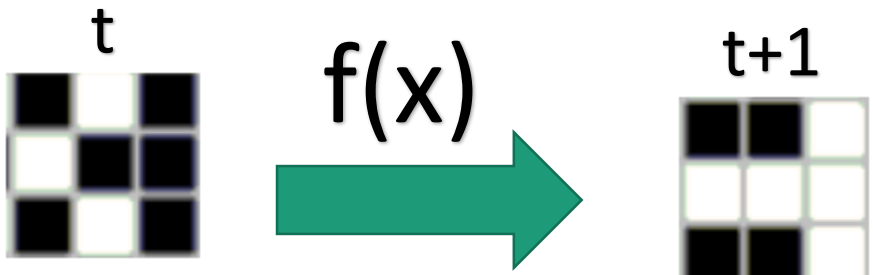




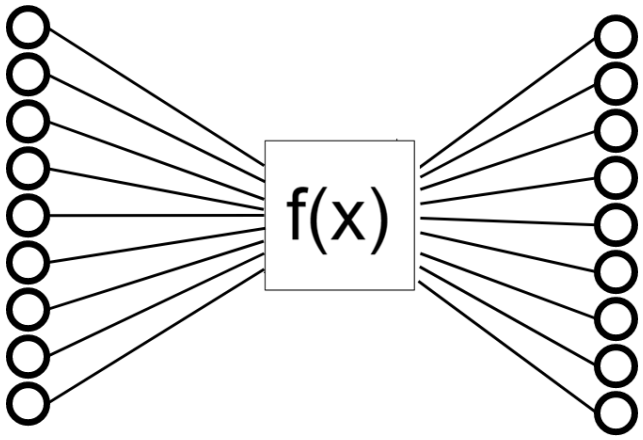
Game of Life is Turin complete!



Game of Life Learning

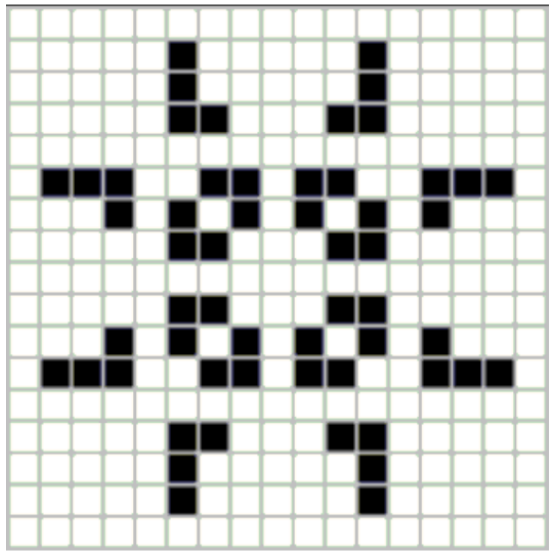



```
determinant_vas = input[:4] + input[5:]
if sum(determinant_vas) < 2:
    output[4] = 0.
elif sum(determinant_vas) == 3:
    output[4] = 1.
elif sum(determinant_vas) > 3:
    output[4] = 0.
else:
    output[4] = output[4]
return output
```



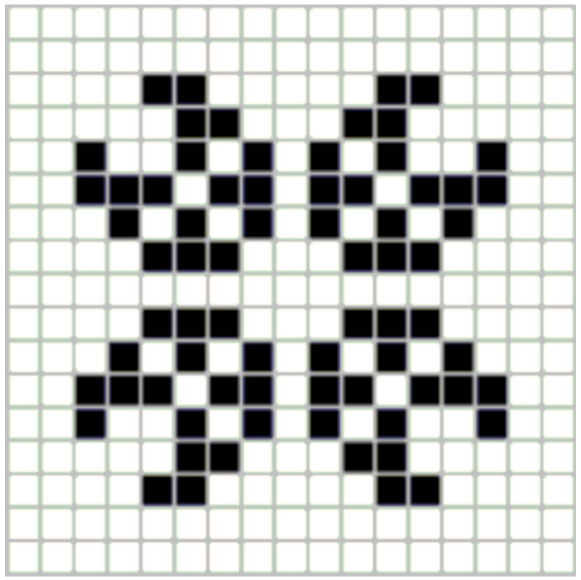
Game of Life Learning

t

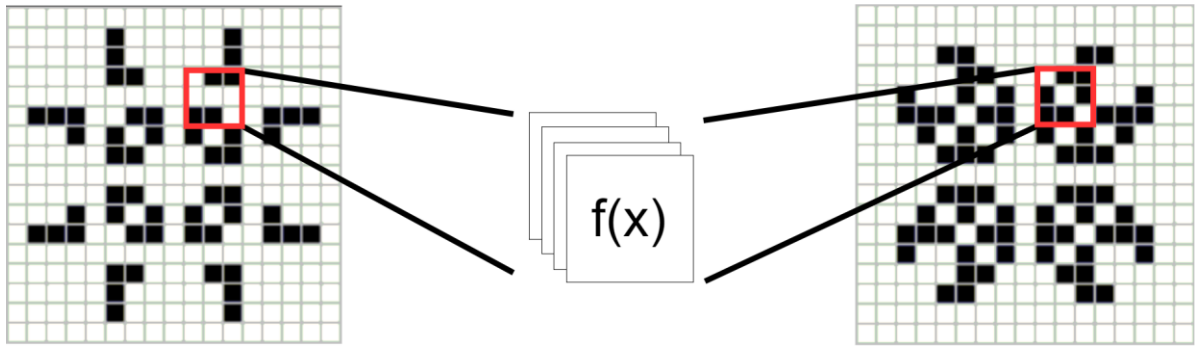


f(x)

t+1

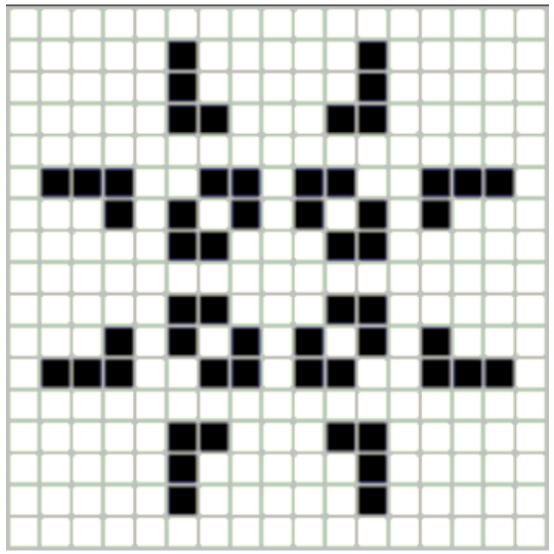


```
board = inputs[elem_idx, :].reshape(dim, -1)
next_board = copy.deepcopy(board)
for i in range(dim-2):
    for j in range(dim-2):
        board_patch = board[i+side_pad-1:i+side_pad+2,
                            j+side_pad-1:j+side_pad+2]
        predictions = self.gof_next_fun(board_patch.reshape(1, -1))
        next_board[i+side_pad, j+side_pad] = predictions[0, 4]
outputs[elem_idx, :] = next_board.reshape(1, -1)
```



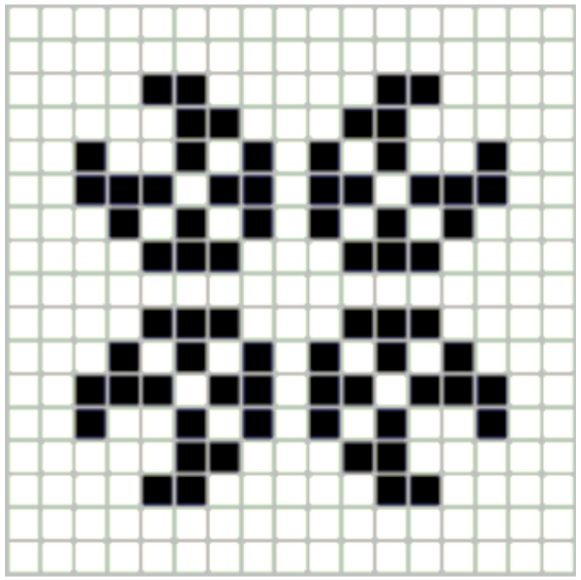
Game of Life Learning

t



for i=0; i<n-1; n++:
do: f(x)

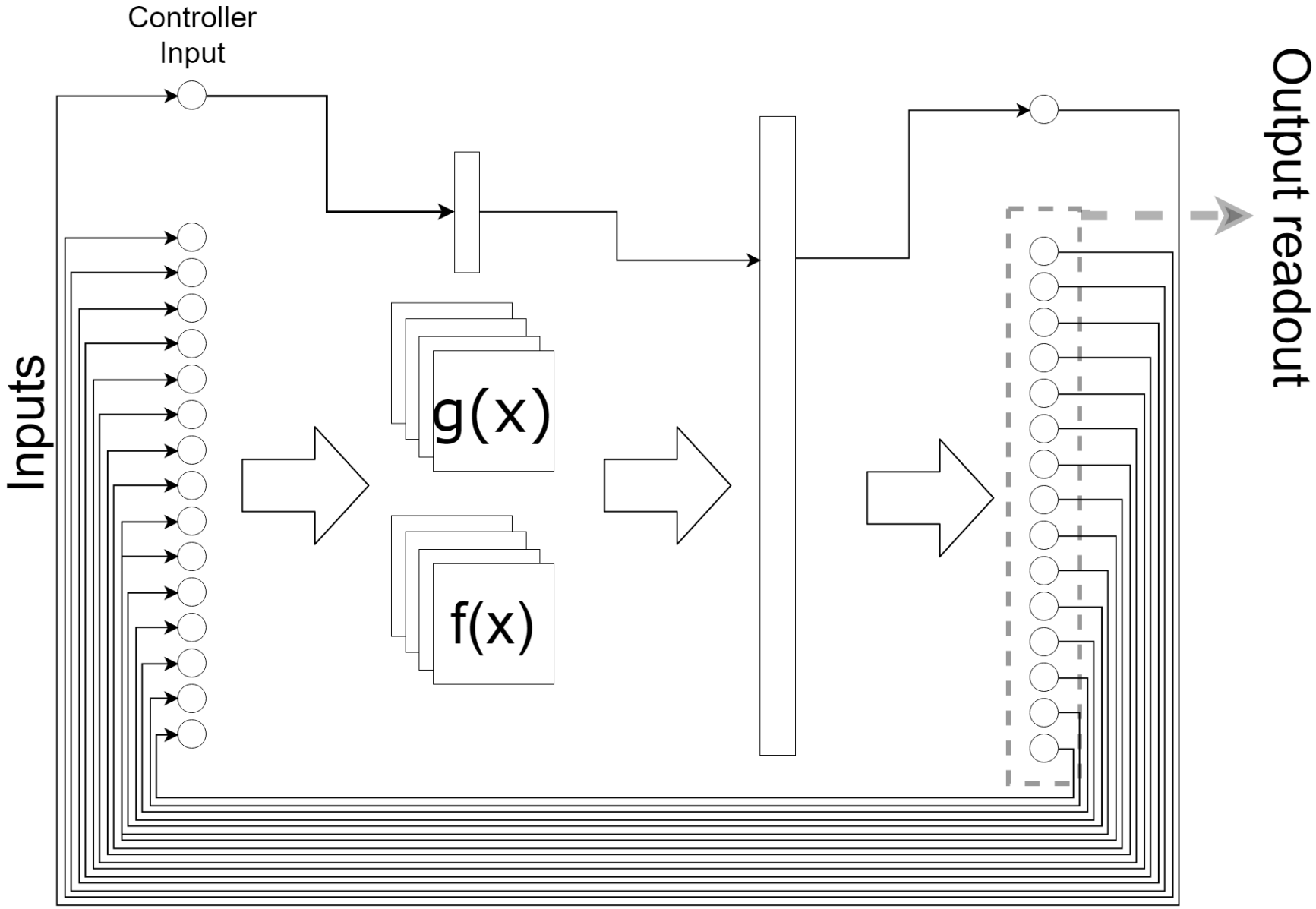
t+n



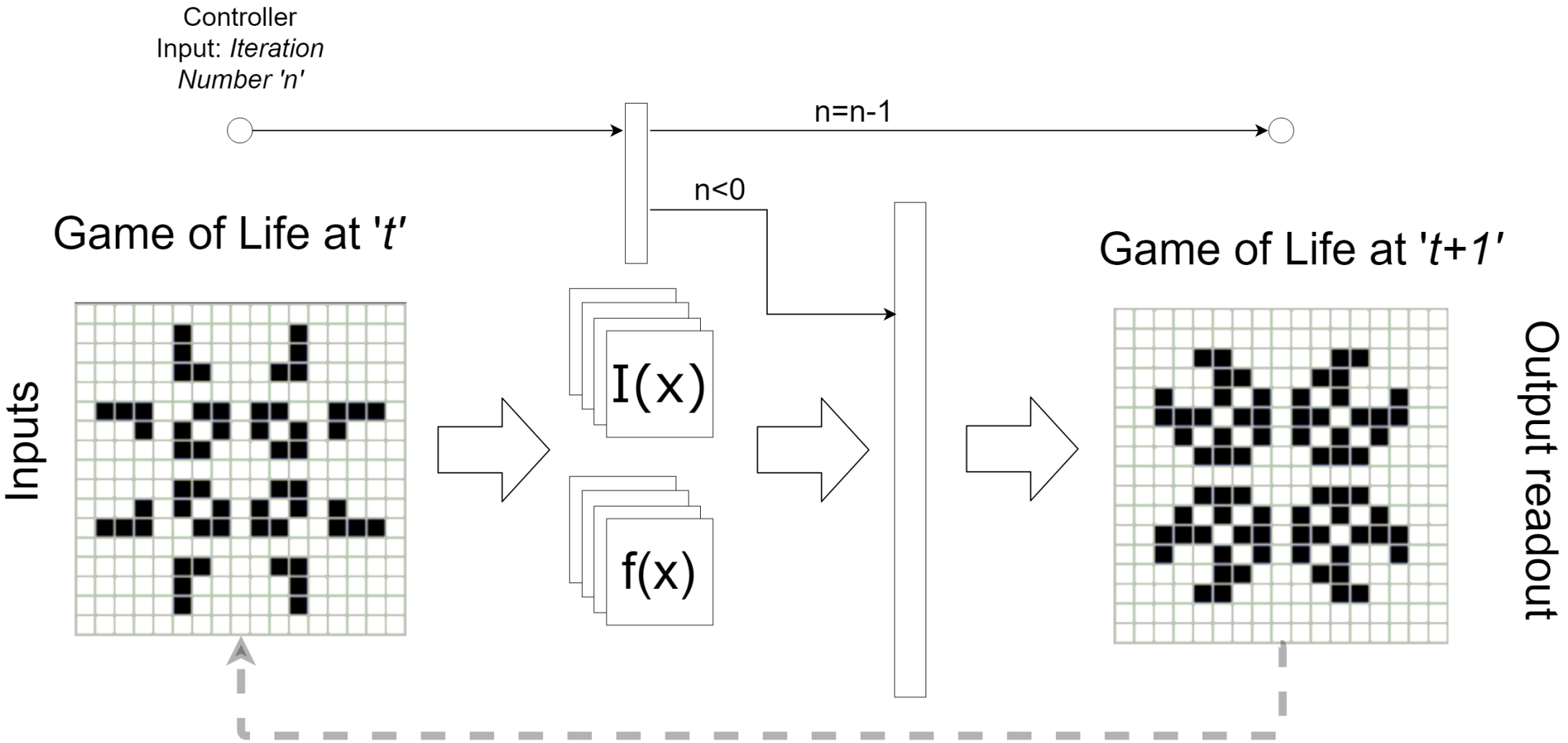
```
outputs = copy.deepcopy(inputs)
dim = np.sqrt(inputs.shape[1]).astype(np.int32)
for elem_idx in inputs.shape[0]:
    board = inputs[elem_idx, :].reshape(dim, -1)
    next_board = copy.deepcopy(board)
    for i in range(dim-2):
        for j in range(dim-2):
            board_patch = board[i+side_pad-1:i+side_pad+2,
                                j+side_pad-1:j+side_pad+2]
            predictions = self.gof_next_fun(board_patch.reshape(1, -1))
            next_board[i+side_pad, j+side_pad] = predictions[0, 4]
    outputs[elem_idx, :] = next_board.reshape(1, -1)
```

?

Modular Transfer Learning for Code Duplication



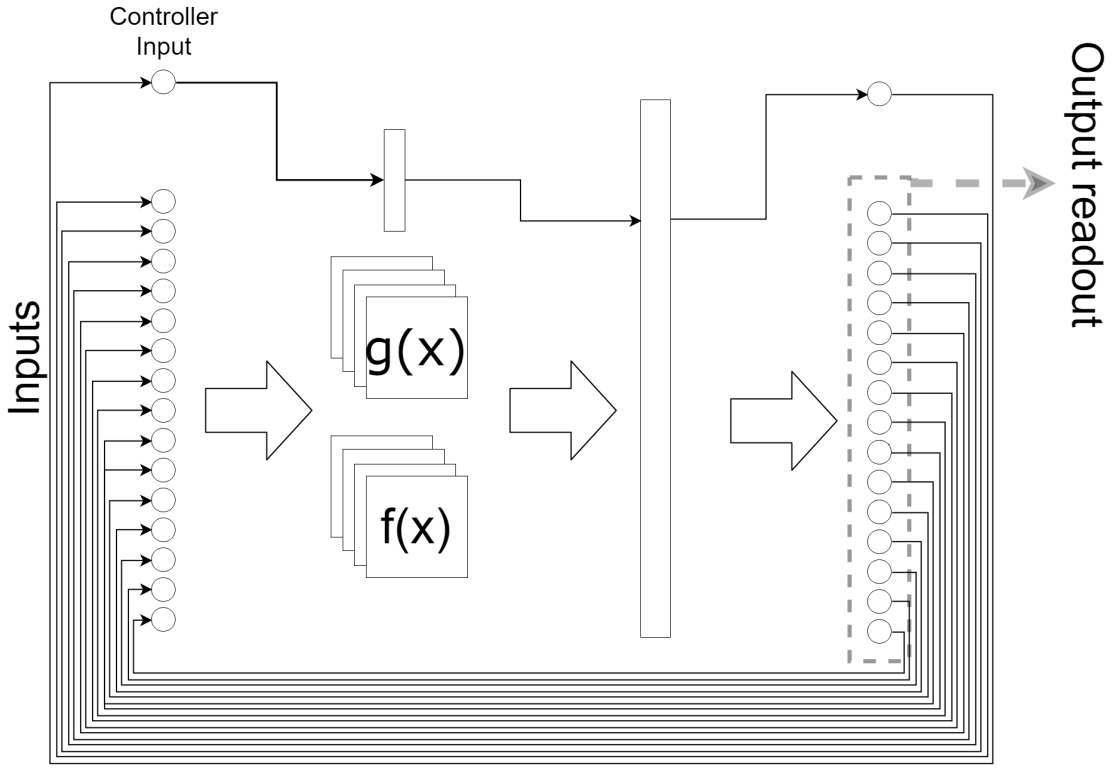
Modular Transfer Learning for Code duplication – Game of Life



Modular Transfer Learning for Code duplication

Pros

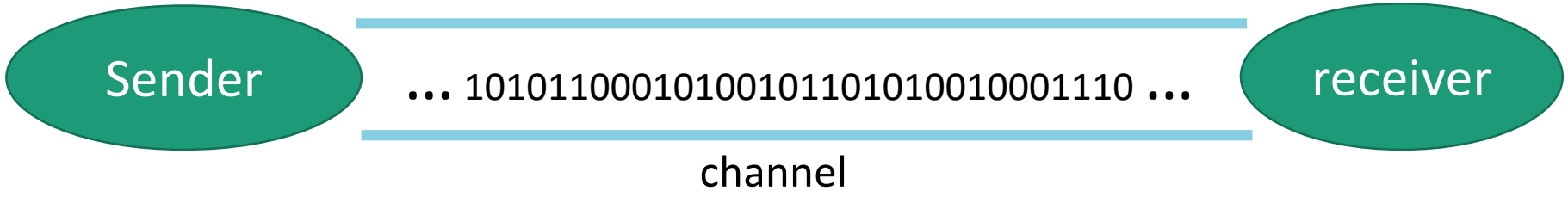
- Modularity
- Transparency
- Composability
- Unit-testable!
- Output-to-input -> akin to recursion!



Cons

- Sparse training (though it can be automated)
- f(x) can be arbitrary complex

More fun…. Parity bit checking

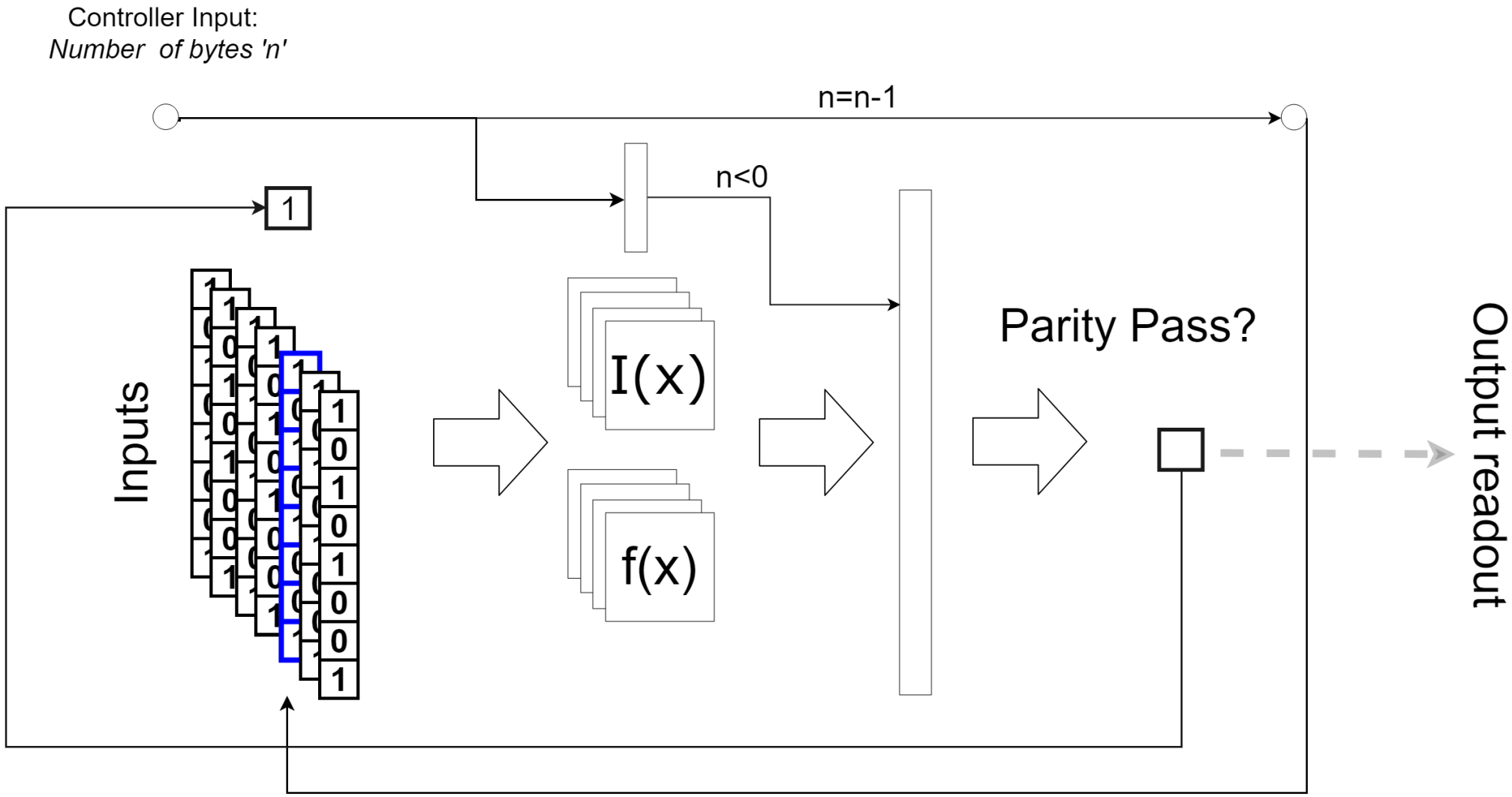


Parity bit

Information bits

byte 1	1	1	1	1	0	1	1	0
byte 2	1	0	0	1	0	1	0	1
byte 3	0	1	1	1	1	1	1	0
byte 4	1	0	0	0	0	0	1	0
byte 5	0	1	1	0	1	0	0	1
byte 6	1	0	0	0	1	0	0	0
byte 7	1	0	1	0	1	1	1	1
byte 8	0	0	0	1	1	0	1	0
byte 9	0	0	0	1	0	0	1	0

Modular Transfer Learning for Code duplication – Parity Bit Checking



Some repercussions of this…

- Depth in NN is overrated!
  Selective training can obviate (some) of the need for depth! i.e. some seemingly large or infinite problems can be solved trivially with appropriate (shallow) architectures)

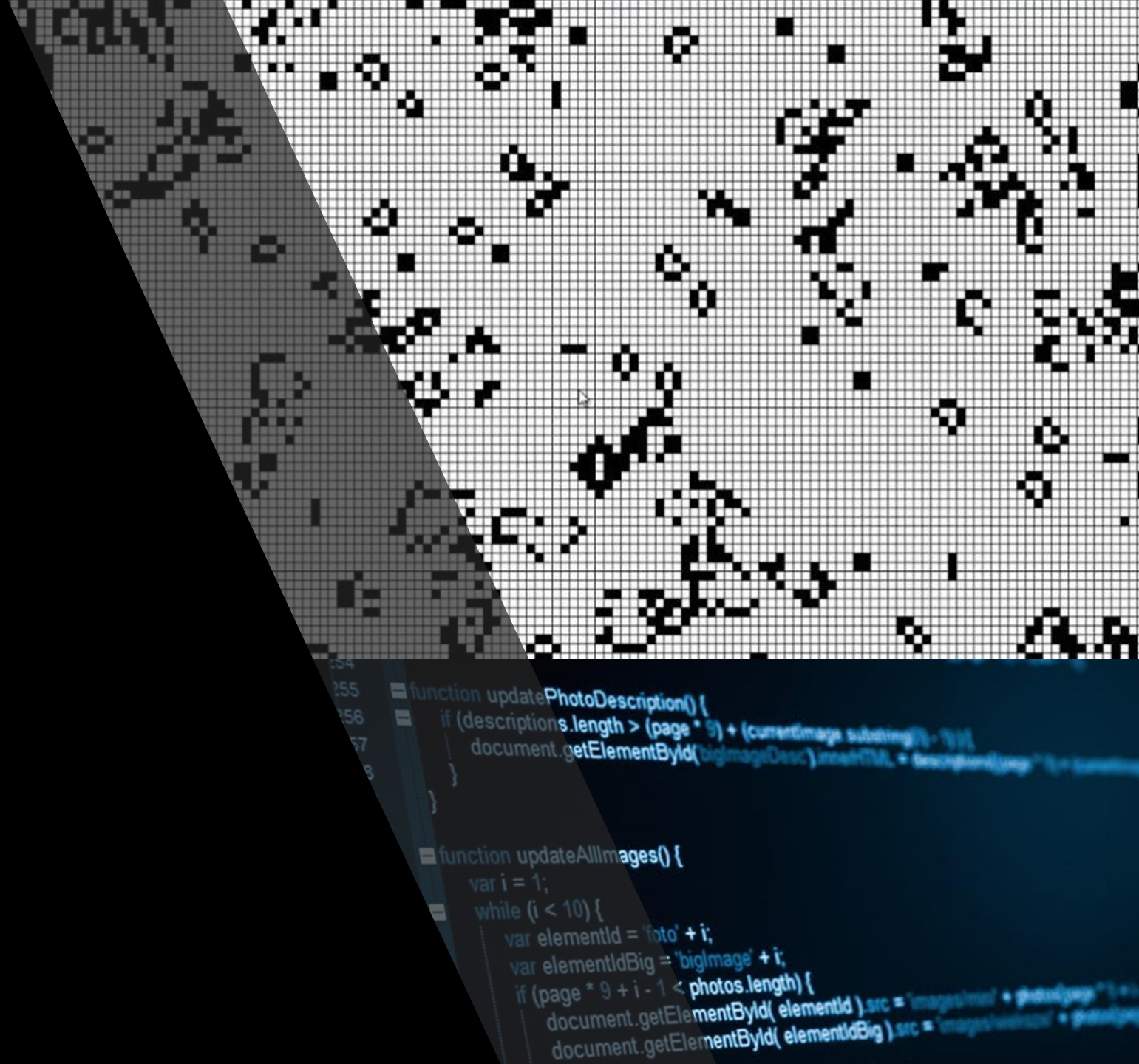
- There may be a future where we do not need Software Engineers anymore (just network designers)

Composability allows for autonomous network design and generation (i.e. it has the potential to automize the architecture generation for any given program.)s

- Solving the training sample generation problem could lead to software which automatically conforms to ever changing software requirements!

(no need for agile programming anymore!)

Thank you!



Luca Scimeca &
Andrzej Banburski