Growing Cellular Automata as Differentiable Self-Organizing systems

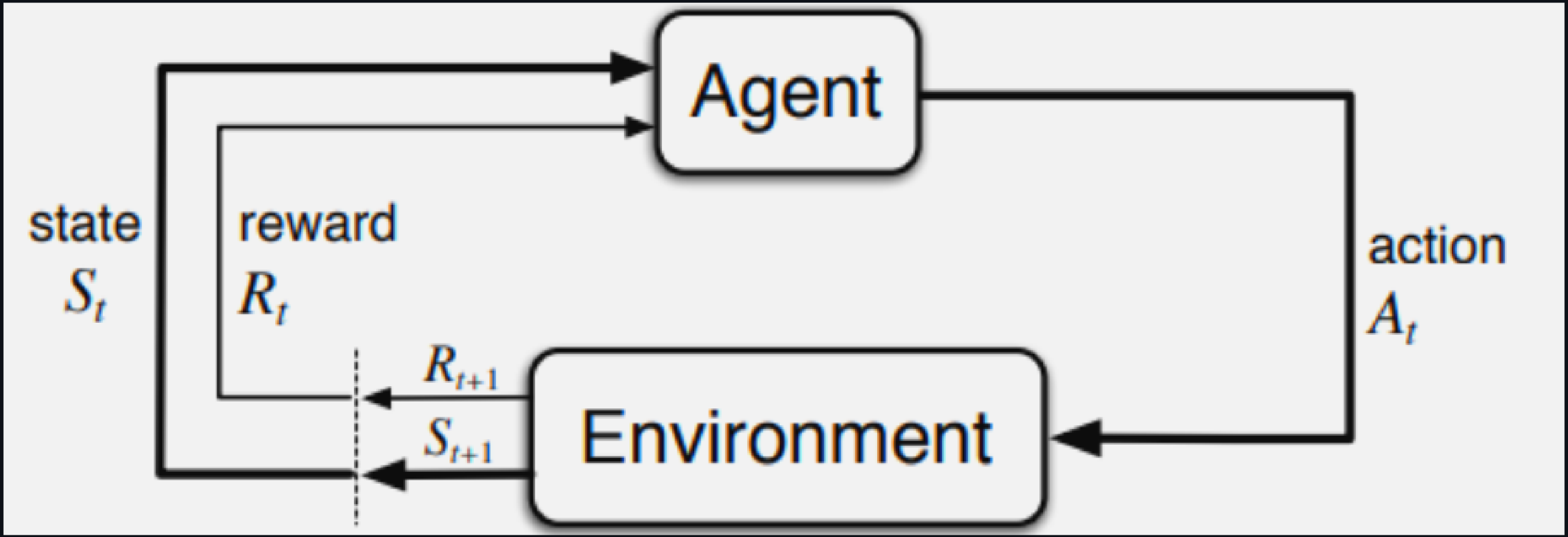
Author: Vincenzo Gargano

Can we consturct robust general purpose self organizing system?

Self-organization is a property omnipresent in all scales of biological life, from complex interactions between molecules forming proteins that can achieve different results only by how they are folded (protein folding is a very complex biochemical setting). But on the bigger scale a group of cells forms a tissue, and we humans are made of different tissues. Humans also can co interact with each other in "tribes" (small societies) and so on.

Reinforcement Learning to treat rules

So we want to focus on practical way to design self-organizing (S-Os) systems. In particular we will use a lot of **Differentiable Programming** (optimization) to learn agent-level policies in a very Reinforcement Learning (RL) fashion since that is the most versatile way of learning a model of the world. Usually in RL we like to have this type of loop, since we're modeling a system.



Introducing Cellular Automata

In our case we will use the framework of Cellular Automata (CA), that is very reminiscent of RL, in fact we have an enviroment and our agents (usually all the agent are copy of a single instance) will follow a set of rules given the local state of our system near the agent, if we "zoom out" from those local structure, we can discover a lot of self emergents behaviour on the entire enviroment. That's what we're instrested in.

CA are biologically inspired to what we just said about global emergent behaviours: from a *simple* set of rules, we get a quite complex behaviour and the collective dynamics of the system are our interests.

Binding CA to RL

A little curiosity that i got during the writing of this presentation was: What are the reward in a CA self-organized system? Well at every step we must follow a rule based on the *local* state, if the rule is not unique we can choose randomly or use a reward function with a discount like we do in RL, so we can purse the "exploration-exploitation" property from RL by using CA, indeed there is this a quite intresting relationship between this description and the Q-learning

Q-learning

Definition

Q-learning: Can we find a function let's call it h such that

$$h(P, M)$$

Which returns a scalar telling us if being in a position P (state of CA) is worth exploring with a move M (Following a rule).

So basically Q-learning is trying to learn an optimal policy $\pi^*(p, m)$. (This is the state-action function from the Bellman Optimality)

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q^*(s', a')]$$

And we can togle the exploration and exploitation to find fucntion that yield the most high expected Q-value. Remember we want to maximize $q^*(s, a)$ (* is signaling the optimal policy).

Regeneration as Self-Organization

Evolution and Regeneration

Most multicellular organism start from a single egg cell, we humans comprised, from that single cell then there is a process of self assemblation in an highly complex structure that can keep regenerating itself, in the case of the human organism the evolution brought us to exploit the rules of nature and computation to implement a set of rules that are highly robust to changes and encoded the software in a gene cellular hardwere that is: DNA. Now, the rules are not simple as the one from the Game of Life from Conway, but in our DNA we have the instruction for build complex structure from a set of amminoacids (21). As we know from high school cells are the building blocks and the ability to reconstruct in the same precise arrangement is the ability of living creature that is most fundamental.

Maintain Equilibrium

Also our body continuatively receives input from external sources that modify it and we're subject on an *Homeostatic feedback loop* also under the biochemical perspective (we need to stay at a certain temperature and have a certain amount of chemical equilibrium to be preseved). If only one of those start being less present (high or low temperature ad example can affect what reaction happen in our body that can be dangerous). Another important step of regeneration is when to stop it, because an uncontrolled growth of cells reminds us of how tumoral cells divide (broken DNA in cells can cause this uncontrolled growth that is bad for us).

The biggest problem: what to build, when to stop

So we have still this big problem of not knowing how a self-emergent behaviour knows when to start rebuilding what and when to stop. If we think to a mathematical formulation we can reconduct this to a complex system (and CA are complex system indeed) and think to the fact that we want to reach a stable state following a trajectory knowing beforehand where we will end up in the state space.

An important step would be to connect computer science to the biomedicine part that studies this phenomena, we can think of some simple rules that progressively builds complex rules and compare to little pieces of code described in the DNA: small subrutines that compose a big one in this type of process: first we start by producing and replicating a certain type of cell with characteristics let's say with photoreceptors, then with all those complex cells we want to create structure and in the end with our *Eye* that is the results we want place it in a certain part of the body to grow. There is surely in our DNA a description detailed for this to happen but it's hidden and we're not even aware of how this happens.

By designing system with the same properties of plasticity and robustness to noise of our organisms we could produce self repairing robots that works in swarms, this could be a great deal in medicin with the hope it doesn't go wrong like in Matrix.

Bulding a Model

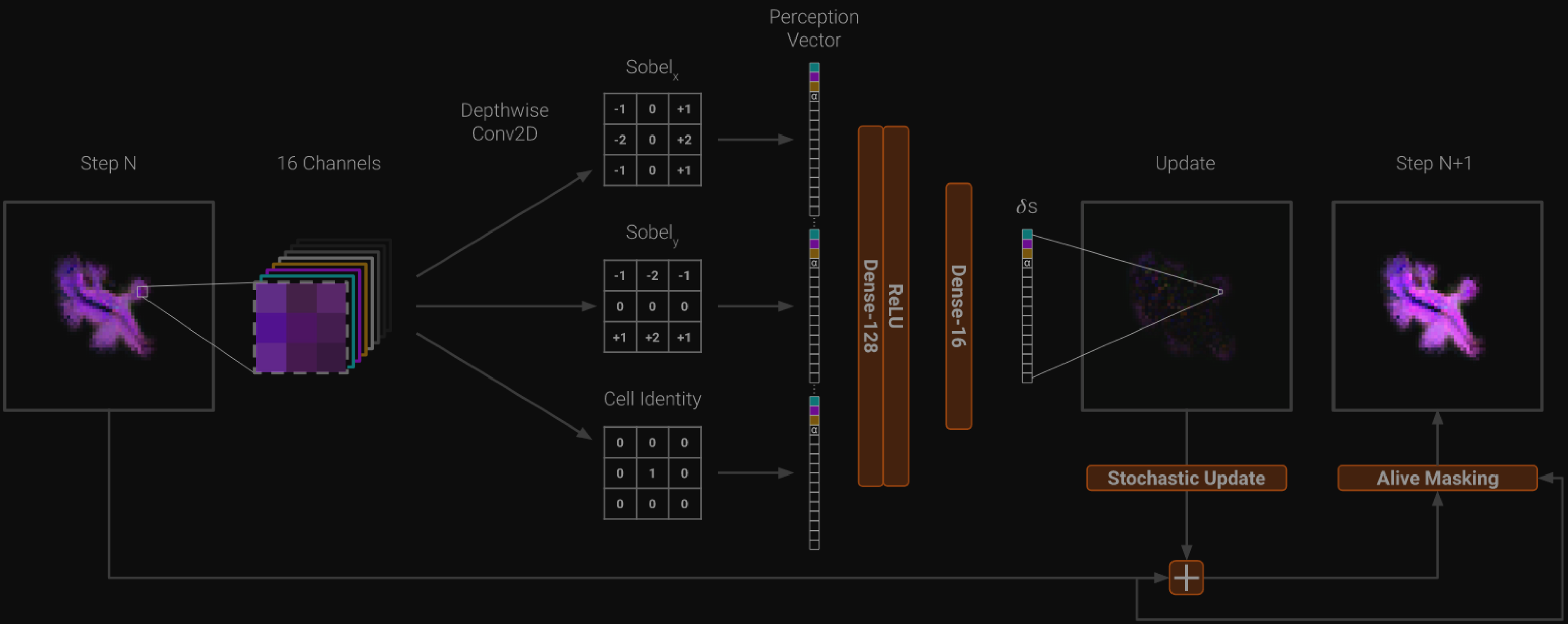
Let's focus on trying to find a set of rules on operating on a CA enviroment that can result in a complex regenerative behaviour, so let's start "simple".

Our model has a 2D Grid, starts with a single cell and, in the following timesteps, iterative updates with the same set of rules being applied to every cell at each step. The new state of a cell depends on the ones in the immediate neighborhood. The toy model we're trying to develop is similar of an organism, so we must specifiy the possible states the cell can be in with a set of discrete values (as usually is done in CA), there are variant with continuous values that makes things more interesting. (To notice that if we use a continuous values for it we get a *differentiable function of the cell's neighborhood* states.)

The rule that will lead the dynamics of our system towards an equilibrium point, by running this model for a certain amount of timesteps from a random configuration will emerge the patterning behaviour we want.

Differential Update Rules

Why we're intrested in a *differentiable* rules? If we speak about differential update rules we're speaking the language of Neural Networks with powerful concept of loss function differentiability expressing "how good we're doing". So we can use gradient-based optimization along the concept of "stacking" differentiable function one on top of the other that gave rise to the Deep Learning and DNN phenomena, and since with a MLP we can universally approximate any computable function we have a strong theorical guarantees.



A single update step of the model.

Defining Cell States

A cell state is a vector of 16 real values numbers, the *first three channels* represent the RGB color spectrum, the target pattern has a color in range $[0.0, 1.0]$ and $\alpha = 1.0$ for foreground pixels and $\alpha = 0$ for background ones.

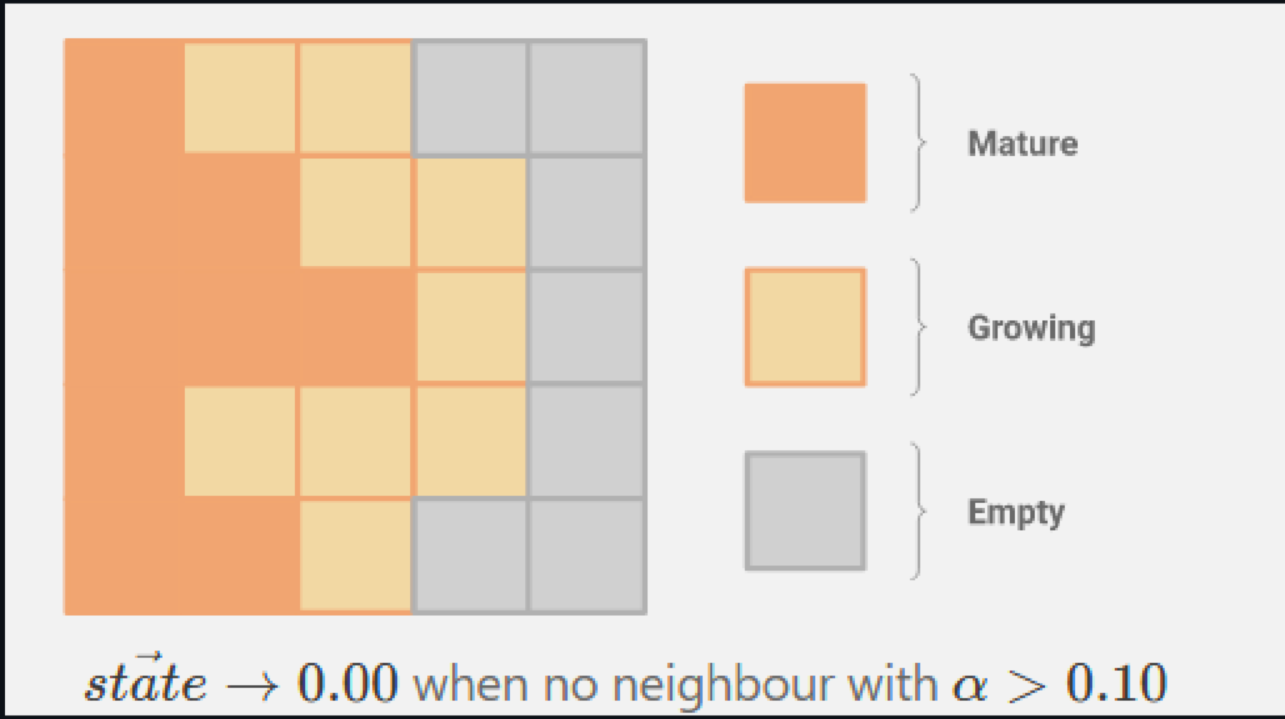
The α *channel* has special meaning, demarcating living cells (those belonging to the growing pattern). If a cell has $\alpha > 0.1$ the neighbors pixels (adjacent) are considered alive, other cells are dead and have their state set to 0.0 at each timestep.

Type of Cells

We define thus three type of cells

- With $\alpha > 0.1$: *Mature*
- With $\alpha \leq 0.1$: *Growing*
- with $\alpha = 0$: *Empty*

The growing cells can become mature if the α raises above the threshold.


Mature
Growing
Empty
$\vec{state} \to 0.00$ when no neighbour with $\alpha > 0.10$

The remaining channels are our parameters of the neural net, so let's call them θ-channels or **Hidden channels**.
If we want to give a more explainable interpretation they can be tought as the chemical concentration of ions or electric potentials or some signaling mechanism that orchestrate the cells behaviour of the set of rules that control the growth. All the cells share the same genome (differentible update rules) and are distinguished by encoded information into the hidden channels: So a cells receive the [hidden channel + life channel + color channel] that is our *final state vector*, stores it and emit after applying a rule on those.

Cellullar Automaton Rules

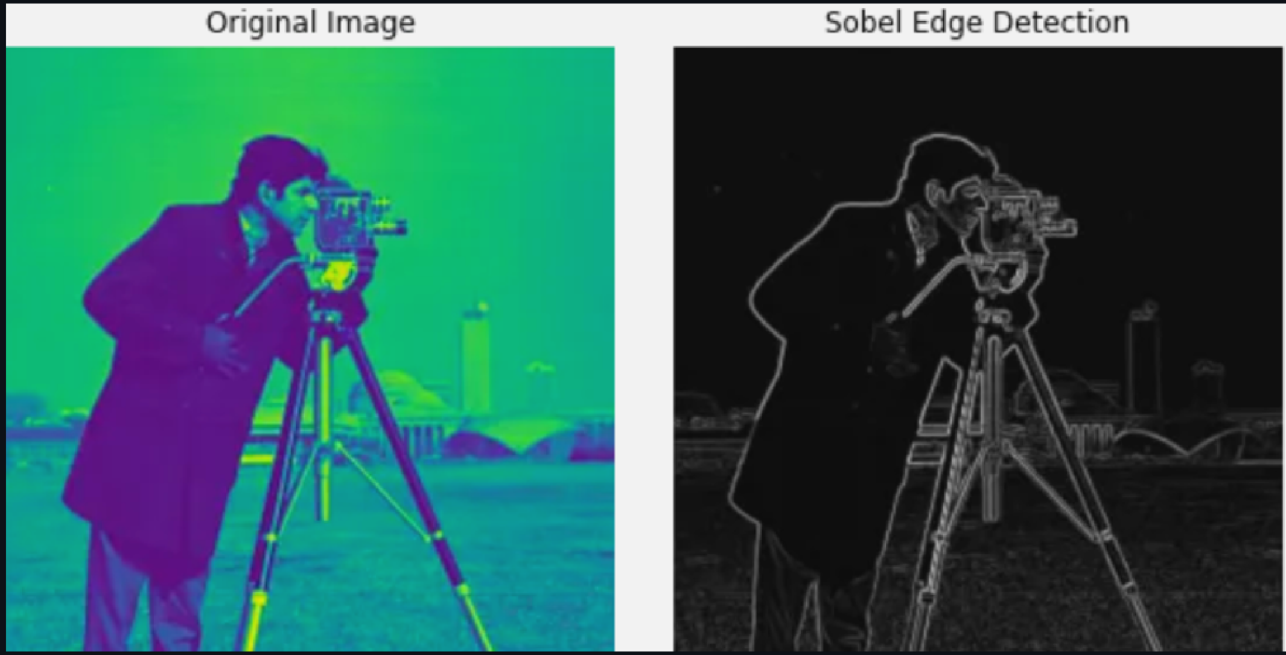
Now let's define the update rule, since we are working with a 2D grid $H \times W$ of 16-Dimensional vectors, essentially we have a 3D array of shape $[heigh, width, 16]$. We are going to apply the same update on each cell, and each one depends on the 3×3 neighborhood of the cell. So we're applying an operation on a cell given the ones near it, this is like doing a convolution operation.

A convolution is a linear operation but can be combined with pre-cell update to produce a complex update rules with nonlinear dependances so it can learn the desired behaviour.

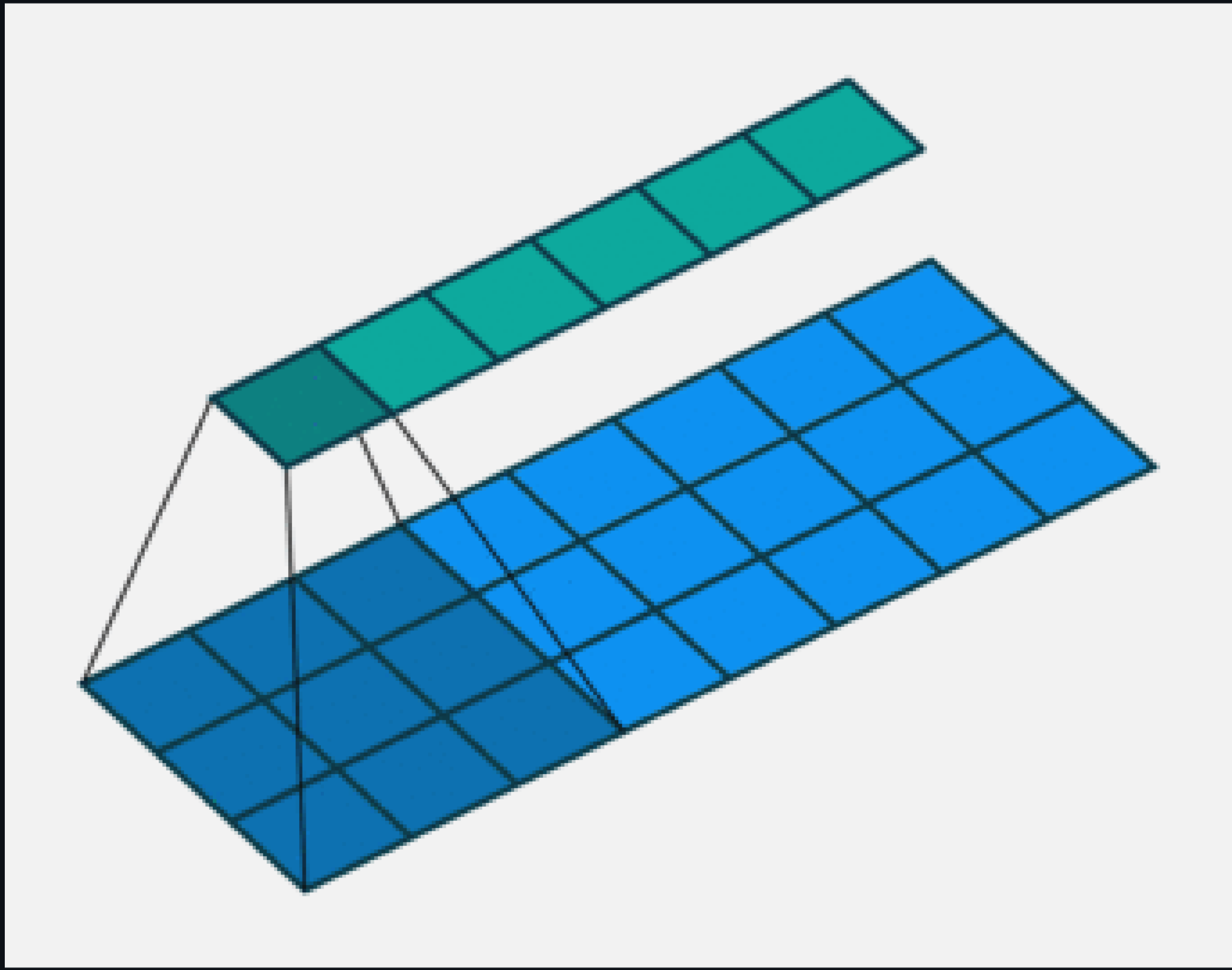
We can split the update rule into 4 sequential phases:

First phase: Perception

First we want to define what is perceiving a cell in the sourrounding thus we can use a convolution operation that we can visualize below, our receptive field will act on a 3×3 matrix extracting a single value for each of our channel of interest. The **Kernel** of the convolution in our case is fixed to be a particular one: *Sobel filter*.



Usually when dealing with Convulutional Neural Net we are assumed that the kernels are naturally learned by the optimization scheme but in this case we have a fixed and differentiable filter, this choose is supported by real life cells that rely on chemical gradients to guide organism development, sobel filters are intresting because when they are applied to an image they are responsive to *edges* of the image.



The filter will estimate the partial derivatives of cell state channels in $\vec{x}$ and $\vec{y}$ directions, giving us a $2D$ gradient vector in each direction, for each state channel. Afterwards we concatenate those gradients with the cell's states getting a $16 * 2 + 16 = 48$ dimensional *perspected vector* for each of the cells in our grid.

```
def perceive(state_grid):
sobel_x = [[-1, 0, +1],
[-2, 0, +2],
[-1, 0, +1]]
sobel_y = transpose(sobel_x)
# Convolve sobel filters with states
# in x, y and channel dimension.
grad_x = conv2d(sobel_x, state_grid)
grad_y = conv2d(sobel_y, state_grid)
# Concatenate the cell’s state channels,
# the gradients of channels in x and
# the gradient of channels in y.
perception_grid = concat(
state_grid, grad_x, grad_y, axis=2)

return perception_grid
```

Second Phase: Update rule

Each cell now has applied a linear operation and yield a *perspected vector* that is fully differentiable, in fact a single vector can be further composed by $1D$ convolutions followed by non linearities like ReLu functions, these will be our update rules.

We want to learn by the data the update rule but each cell has to apply the same rule in our case, the network parameterizing has 8.000 parameters, this is a type of network called *Residual Neural Network*, the update rule output will return an incremental update to the *cell's state*, that will be applied to the cell before the next time step.

The rule update initial behaviours: *Do-nothing* that is implemented by setting all the initial weights to the final convolutional layer to zero, in addition we're not applying the ReLu non-linearity to the last layer because we want to be able to only do addition and subtraction on that step.

```
def update(perception_vector):
# The following pseudocode operates on
# a single cell’s perception vector.
# Our reference implementation uses 1D
# convolutions for performance reasons.
x = dense(perception_vector, output_len=128)
x = relu(x)
ds = dense(x, output_len=16, weights_init=0.0)
return ds
```

Third Phase: Stochastic cell update

In the CA framework all the cell are *syncronously updated*, but a real life system is not governed by a global clock for each cells what we can expect is that every cells has it's own clock so we can relax this quite involved syncronized global update by assuming *independance* of cells update, so each cell has a random time interval between one update and the other, the solution is to mask the entire grid of *update vectors* with a time-delay for the stochastic update, so each cells has a probability to be updated :in the paper is settled to 0.5 during training.

This can also be seen as a type of dropout technique during training getting our update to not enforce the same cells every step but to encourage sparsity of update and thus augments generalization of learned update.

```
def stochastic_update(state_grid, ds_grid):
# Zero out a random fraction of the updates.
rand_mask = cast(random(64, 64) < 0.5, float32)
ds_grid = ds_grid * rand_mask
return state_grid + ds_grid
```

Fourth Phase: Living cell masking

We're modelling the growth of an organism starting from a single cell being at least *Growing* and cells that are empty don't need to participate in the growing mechanism and will not matter in any computation or carry *hidden states*. We enforce this by setting all the *channels* of empty cells to zero. Remember a cell is empty if there is no *Mature* ($\alpha > 0.1$) cell in the 3×3 neighborhood.

Learning to Grow

So the first naive experiments is to use a target ground truth ($\hat{y}$) the target image and then train the CA model after some stochastic updates of the state.

- A center cell is set to be alive having all channel exepct RGB set to 1.
- We iteratively apply update to the cell by sampling a random number of steps for each training step. Since we want to enforce stability in the pattern transformation "trought-time"
- At the last step we just compute the L_2-loss pixel-wise between RGB-Alpha channels in the grid and the ground pattern.

The loss is also differentiably optimized with respect to the update rule parameters by backpropagation trought time (BPTT) a method used for RNNs training, if we get a too long BPTT update we can get instability in those gradient either explosion or shrinking so we normalize an L2- normalization per variable to the parameter gradients this is similar to weight normalization (Inspired by BatchNorm in standard architectures)

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

After the optimization converges we can run simulations of learned CAs growing pattern starting from the seed cell.
Now since we learn to reconstruct our pattern by a certain number of *fixed* timesteps what happens if we keep our system running and updating in further timesteps? What will the system predict in the future evolution of our pattern? We can see it in the following example:

Long-term instability

As we can see different pattern results in different outcomes in the long term, after the trained and controlled part we are lead to instability that will get us out from the stable pattern and some pattern dies out and some other doesn't know when to stop growing. Some of them seems to remain stable but capture some recurrent pattern in the data we fed. How do we design now a robust and persistent pattern overtime?

What persists, exists!

Now we have to talk about dynamical systems and understand why is emerging an instability in our previous experiment of growth. Each cell is in fact a dynamical system and each of those share dynamics with the one near it, because cell are locally coupled between themselves. When we train the update model with the rules we're adjusting the dynamics of the system and we want to find dynamics that satisfies some properties. We want the system to evolve from seed pattern to a target pattern that's a trajectory in the state space we have to follow. So we need to make the target pattern an **attractor** in the dynamical system.

A simple strategy is letting the CA iterate for a much longer number of time steps and periodically apply the loss against the target to enforce stability around that point. So we train trough BPTT with intervals that becomes longer and longer and by iteratively doing this we are molding dynamics of the cells to the fixpoint we want.

But this comes at a cost: since we're doing a BPTT longer timesteps takes more time for the training and furthermore the memory requirements increases since we've to store every activation in the entire episode in memory before doing the backward pass.

A proposal is to apply a *sample pool* based strategy to get similar effect: we have a pool of initial seeds to start iterating with the usual single alive pixel, then we sample a batch from this pool that will be part of our training step.

Then in order to prevent the system to catastrophically forget information each time we replace once sample in the batch with the original single pixel cell state. After a train step we replace samples drawed from the pool with the new updated version and we keep going (remembering to still getting one sample to remain the single pixel) as show the animation

At the start of the training the random dynamics allow the model to end up in various uncomplete and random states the ones with the strange green noise, but with the pool sampling we refine the dynamics to be able to go back and recover the bad states, and of course the samples in the pools at each steps are closer to the target pattern.

With this method we're using previous final state (random) as a new starting point to guide CA to learn how to persist or *improve* (we will see later) an already formed pattern. This allows us to interleave the loss for al long number of timesteps encouraging the generation of attractor as to be the target shape in the complex system.

Another thing is that a random reseeding is not the best thing we want since taking the oldest state is the thing that is doing better and so everytime we sample the top-k higher loss samples in the batch, making the training more stable at initial stages, this also cleanup the low quality states at the start.

Learning to Regenerate

As we previously said we want to also have the ability to regenerate, as skin gets replaced but in some organism the very heavy damage to some part can be easily regenerated. How our model behaves is we cut off a part from it?

Well the results are not that good, we damage the final state after we get to it from the initial and we can see that some of the damage types are recoverable in an acceptable way but some other cannot reallt recover to the original state, also we can notice that the lizard developes a quite strong regenerative response to damage without explicit training, maybe also image based lizard are naturally good at it who knows.

So we've a problem but solution is simple, we've teached our system to recover into the attractor from a single cell. These system were training to grow and then stabilize some of those pattern has natural "regeneration ability" but if we change the type of damage we can get different reaction : *Uncontrolled growth*: explosive mitoses, *Overstabilzation*: No response to damage or even *Self destruction*.

What we want to do is increase the *basin of attraction* for the target pattern, so by increasing the space of cell configuration that is able to recover to the correct pattern we will get everytime to it. The solution is simple we just damage some pool sampled state before the training step. So now we're teaching also how to recover from damage.

Rotating Perceptive Field

Until now we treated the perceptive field of a cell by wathing the neighborhood and estimating the state channels using *Sobel filters*, in direction $\vec{x}, \vec{y}$.

A biological analogy of this is that each agent has two sensor acting in orthogonal directions sensing the gradients in the concentrations of chemicals along the axis of that sensor. What happens if we rotate the sensors? Well we apply a rotation on the kernels

$$\begin{bmatrix} K_x \\ K_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \mathrm{Sobel}_x \\ \mathrm{Sobel}_y \end{bmatrix}$$

So by *rotating perceptive field* we are producing a rotated version of the target pattern to reach and we don't have even to retrain our model.

Since we're in a square lattice and now in a "more continuous" setting the perceived gradients along the two direction are expected to be invariant to the angle. But in a pixel based model things are not that simple, when we apply a rotation on a pixel we are computing a mapping that's not bijective and has to interpolate between the current state of reaching the fixpoint and the one after the rotation, now since pixels are coupled this effect is propagated on all the grid and the results of the reached fixpoint can be not the best one.

Related works and final considerations

CA and PDEs

There is plenty of literature covering the CA and PDEs relationships, the application can be implemented in biological systems, physical and even social system. Some example that inspired this paper are from: Alan Turing introducing the famous *Turing Pattern* in 1952 from this paper, suggesting that reaction-diffusion systems can model chemical behaviour during morphogenesis. That's the process that starting from morphogens builds trough a set of rules some structure that has a particular function, starting from an homogeneous equilibrium triggered by random disturbances reaction bring to the complex pattern.

Example of turing pattern obtained trought morphogenesis on a Pufferfish

Another inspired reaction-diffusion model which shows a variety of behaviours is the Gray-Scott model, by controlling some variables we can obtain different results

From this paper

Von Neumann and CA

Since when the seminal paper from Von Neumann introduces CAs as self-replication models many researcher started to study these behaviour, in fact at the same time in the 1950s we have the paper about Lamda Calculus introducing a Turing Complete system that work by functions composition having at it's core the Lambda Operator that is used to replicate variables in the system (a self referential fucntion implementing recurrent behaviour). And the introdcution of the aforementioned *Game of Life* from John Conway. Another great mind working on CAs is Steven Wolfram that invented and showed that Rule 110 (another CA) is Turing Complete in the paper "A new kind of Science". Proposing the paradigm of CA as a tool for understanding the world.

More recently researcher extended the Game of Life to a continuous domain: Rafler's SmoothLife, Lenia - Biology of artificial life. The latter also discovered and classified an entire species of lifeform generated by the model!

Also researchers used evolutionary algorithms to find the rules in CA to get the behaviour they aimed like in CA-NEAT.

Neural Network and Self-Organisation

CNNs and CAs are closely related since both work on a grid, moreover with all the technique they use in the paper and abusing of the Machine Learning folks jargon we could rename this model as : **Recurrent Residual Convolutional Networks with 'per-pixel' Dropout**.

Looking at it broadly, the concept of self-organization find it's way deeper into ML with popularization of Graph Neural Networks (GNNs), typically this set of models run repeated computation across vertices of a (possibly dynamic) graph. Vetices communicate still locally trought edges and aggregate global information required to perform the task over rounds of messages exchanged, just as atoms in a cristalline structure produce emergent properties (different cristalline structure have physically different behavioru), in fact we can model molecules with those models.

Swarm Robotics

Another observation of how powerful is self-organization is when it's applied to swarm modelling. Reynold Boids simulated flocking of birds with a tiny handcrafted rules. Nowadays we can embed tiny robots with programs following set of rules and test their collective behaviour like sciestists do in : Mergeable nervous systems for robots and **Kilobot: A low cost scalable robot system for collective behaviors**. These programs are designed by humans but this work showed that we can use *Differentiable modeling* to learn automatically the rules we want.

Physical Implementation of the model

At the end the paper speaks about a proposal of using a computer for each cell to model the behaviour of the system each computer will need 10Kb of ROM to store the genome (Neural net and control code) and 256kbytes for the RAM for the cell intermediate activations, then cells has to communicate with the 16-value state vector to the ones near it and a RGB-diode for display color. Single cell update would take 10k multiply-add operations and has to be desyncronized with all the others (random updates). There are a lot of issues for this uniformely decentralized system. The conjecture of this system may be good for reliable, self-organized systems.
On the theoretical machine learning part it's showed by decentralized neural network like Federated Learning based approach that this system still learn the global behaviour without using a global modeling scenario, this work overall propose a solid grounding for exploring more decentralized learning modeling.

A little notes about RNN and CA

Yeah another of my observation is that there is a certain type of Recurrent Neural Network architecture that is biologically inspired to "reconstruct" a pattern that is the thing we're trying to enforce now these are called **Hopfield Network**.

Basically are continuous or discrete network (depend on the energy function) that are represented by a fully connected graph whose weights are the synaptic connection between neurons, a neuron fire if a certain treshold is reached upon the linear sum of receinving inputs and a nonlinear function is then applied, that's the update rule we're following in this case and a pattern in stored in those network by doing an outer product between the pattern itself and the weights matrix is settled to that *Autocorrelation matrix*! That's an attractor we built into this type of network.

By following updates we just get to the stable state of our dynamical system! So Hopfield Network are dynamical system
and can be viewed as a type of CA, we know that CA are turing complete that's not a proof but an hint.

References

https://distill.pub/2020/growing-ca/