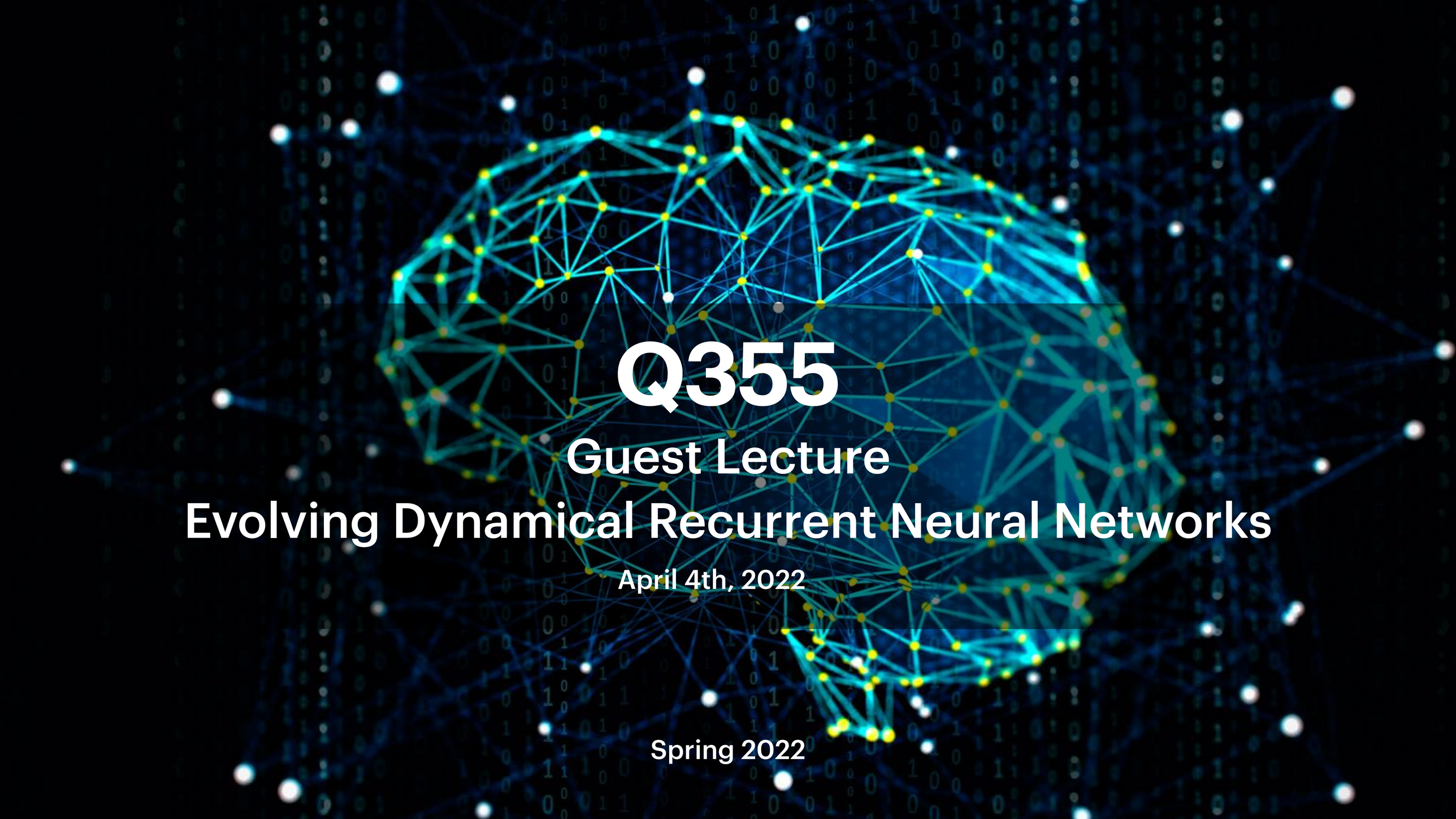


Q355

Guest Lecture

Evolving Dynamical Recurrent Neural Networks

April 4th, 2022

Spring 2022

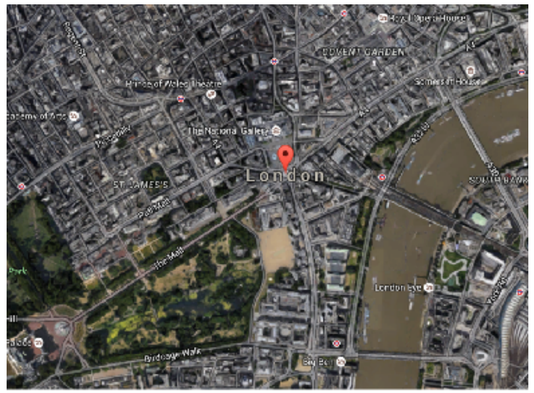
Part 1: DRNNs

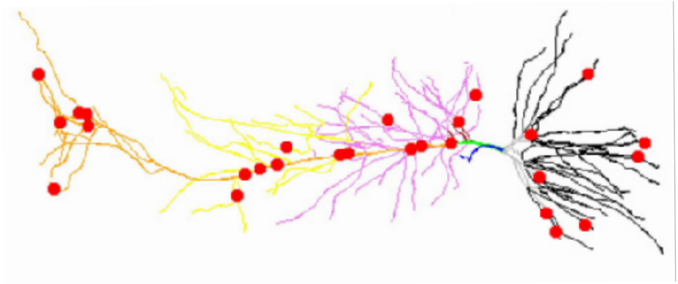
A. Motivation for dynamical models and their numerical integration (Euler).

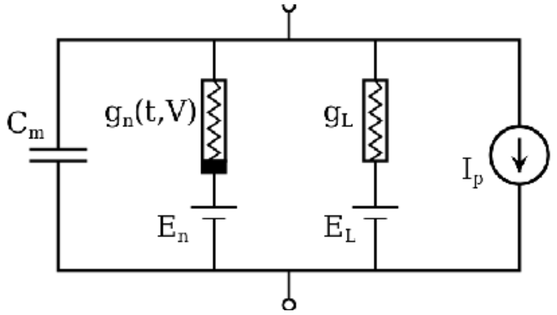
B. Discussion of two dynamical models: Lotka-Volterra (and generalization), CTRNN (firing rate model and extensions), Izhikevich (spiking neuron and network extension) — list of ideas from Sussillo?

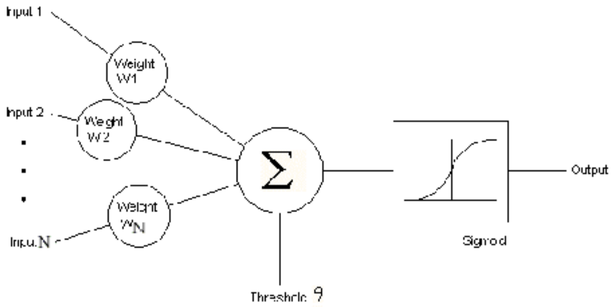
C. Walk through the design, implementation, and use of (some of) those models in Python (code available before hand).

Levels of modeling



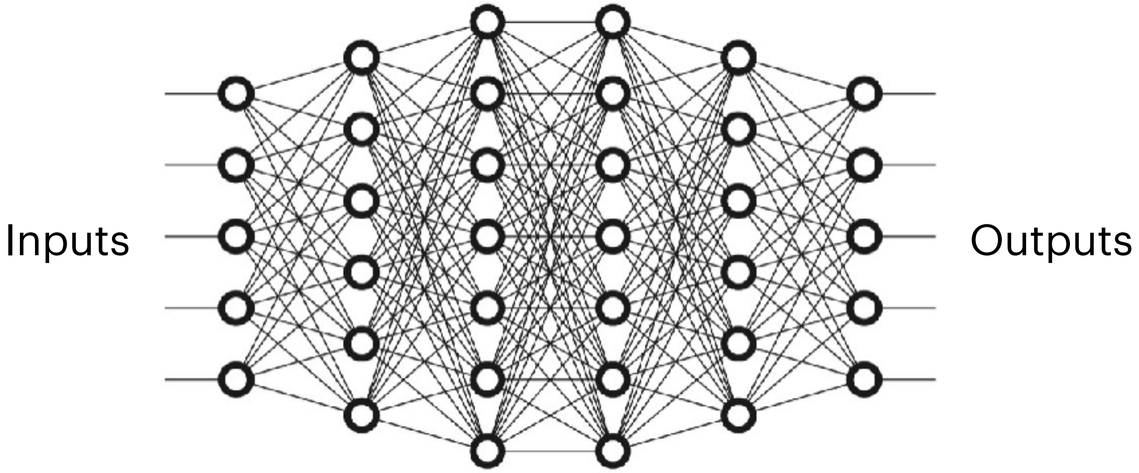




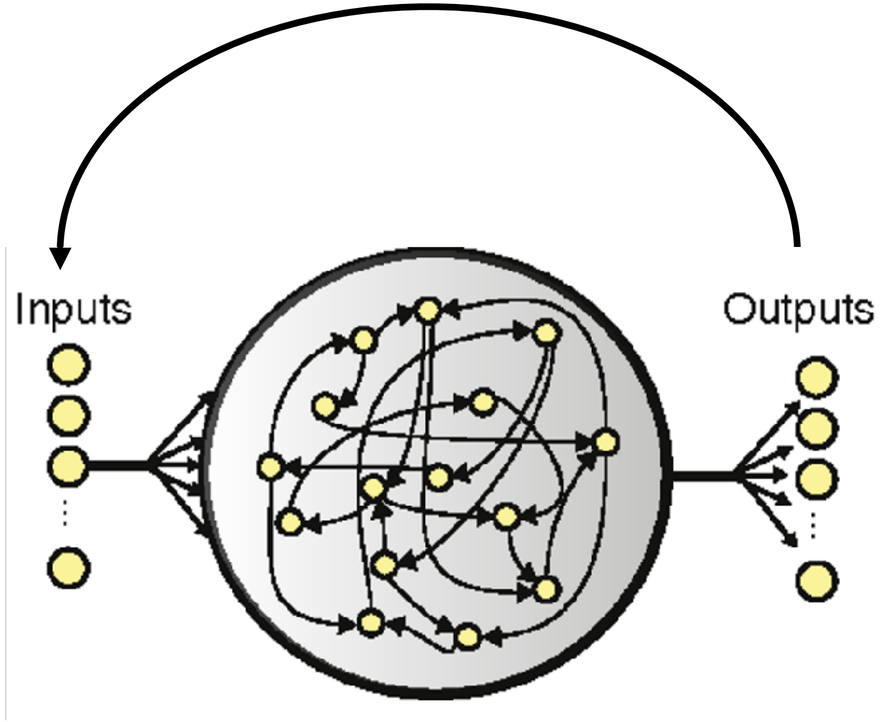


"All models are wrong, but some are useful."

Neural Networks



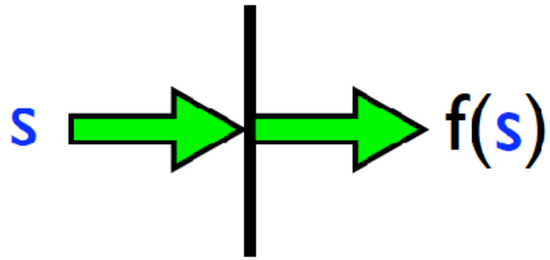
Feedforward
Discrete-time
Typically used for I/O tasks



Recurrent
Continuous-time
Closed loop tasks

Why Dynamical Agents?

A *reactive* agent is at the mercy of its environment.



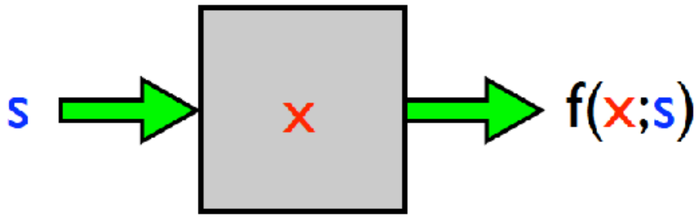
A *dynamical* agent can:

Initiate behavior independently of its immediate sensations.

Respond differently to identical sensory stimuli at different times.

Organize its behavior in anticipation of future events.

Modify its future behavior based on its history of interactions.



Dynamical System Models

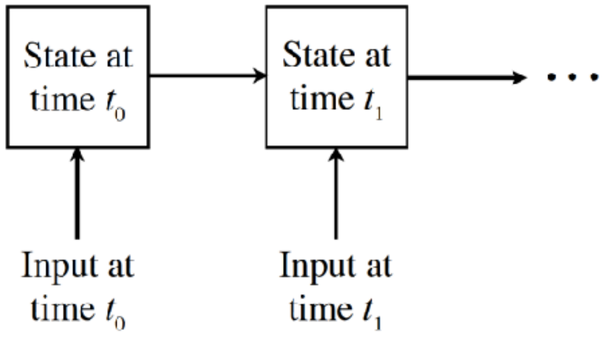
A dynamical system is one whose future state depends on its current state and inputs in some principled way.

To create a dynamical system we simply need to decide:

(1) What is the "something" that will evolve over time (state space).

(2) What is the rule that specifies how that something evolves with time (evolution operator).

In this way, a dynamical system is simply a model describing the temporal evolution of a system.



Differential Equations

Differential equations describe systems undergoing change.

They are particularly useful when modeling natural phenomena.

Solution to ODEs are functions or a set of functions. (Unlike the solutions to algebraic equations which are numbers).

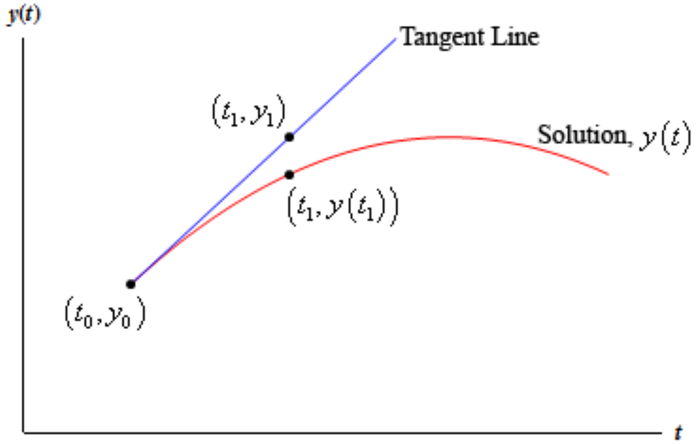
Often, systems described by differential equations are complex enough, that a purely analytical solution to the equations is not tractable. It is in these complex systems where computer simulations and numerical methods are useful.

Simplest method to numerically integrate a dynamical system

Techniques for solving differential equations based on numerical approximations were developed before programmable computers existed.

We will describe the simplest method for programming simulations of differential equations.

From any one point in the curve, it's easy to find an approximation to a nearby point on the curve by moving a short distance along a line tangent to the curve.



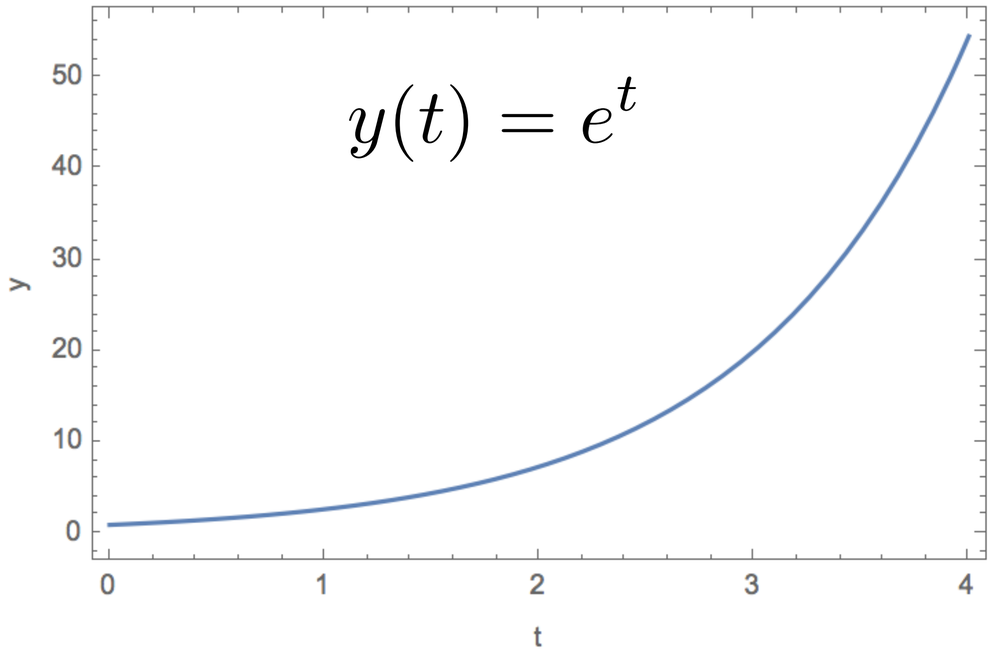
$$\frac{dy}{dt} \approx \frac{y_{t+h} - y_t}{h}$$

$$y_{t+h} = y_t + h\left(\frac{dy}{dt}\right)$$

Simple Example of Numerically Integrating a Dynamical System

$$\frac{dy}{dt} = y$$

$$y(0) = 1$$



Python Demo: Stepsize and Integration Error

Example #2: Lotka-Volterra System Predator-Prey Dynamic Model

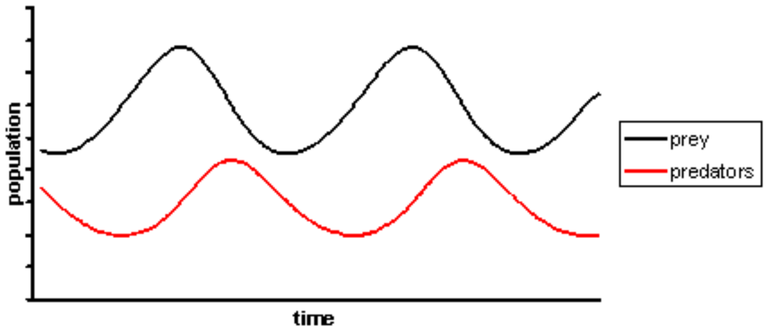
Differential equations are frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey.

$$\frac{dx}{dt} = \alpha x - \beta xy$$

The populations change through time according to the pair of equations.

$$\frac{dy}{dt} = \delta xy - \gamma y$$

x is the number of prey (e.g. rabbits), y is the number of predator (e.g. foxes). The rates of change represent their growth rates. t represents time.



The rest are the parameters of the system (positive, real) that describe the interaction of the two populations.

Python Demo available if interested.

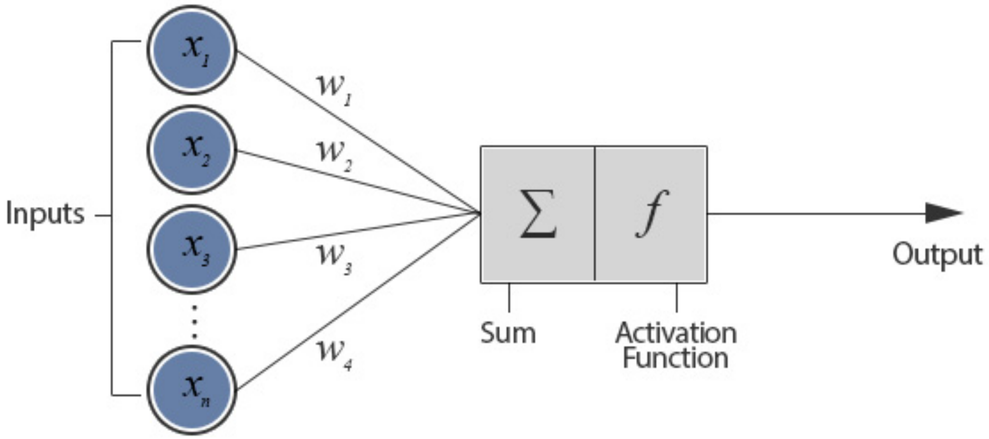
Simplest Artificial Neuron (Perceptron)

Neurons output a binary state (0 or 1).

Time evolves discretely.

Neurons receive excitatory or inhibitory input from other neurons through a weighted connection.

Neurons evolve in time according to a rule that involves the state of the incoming connections (weighted) and a threshold.

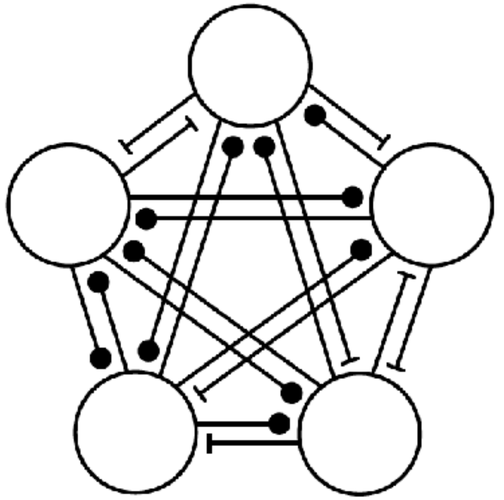


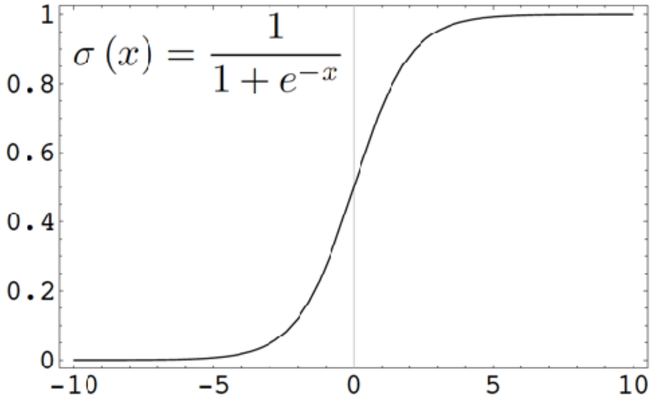
Continuous-Time Recurrent Neural Network

The state of a neuron is described by a real number (continuous state space).

The activation of a neuron *i* changes as a function of the sum of all weighted inputs from presynaptic neurons *j* and a decay rate (continuous evolution operator):

$$\dot{y}_i = \frac{1}{\tau_i}\left(-y_i + \sum_{j=1}^{N} w_{ji}\sigma\left(y_j + \theta_j\right) + I_i\right)$$





Leaky component

Summed input from all presynaptic neurons

Transfer function (typically a sigmoid)

Neuron activation (y) (the state of the neuron)

$$\dot{y}_i = \frac{1}{\tau_i}\left(-y_i + \sum_{j=1}^{N} w_{ji}\sigma\left(y_j + \theta_j\right) + I_i\right)$$

External input to neuron i (typically a function that changes over time)

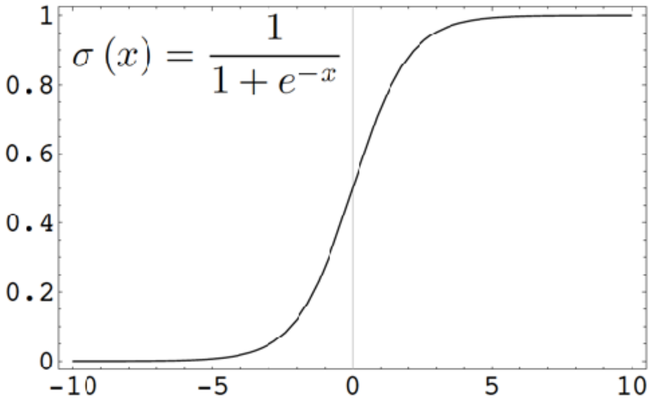
Rate of change of neuron's i activation (dy/dt)

Time-constant (t>0, how fast or slow that neuron is at integrating information and changing activation)

Weight (strength of the connection between the presynaptic neuron j and the current neuron i).

Bias for presynaptic neuron j (affects the output level in relation to the activation level)

Model parameters



Why CTRNNs?

Simplest nonlinear continuous time dynamical neural model.

Biological interpretations:

(a) Mean firing rate model.

(b) Model of nonspiking neuron with input nonlinearities.

Computationally and analytically tractable.

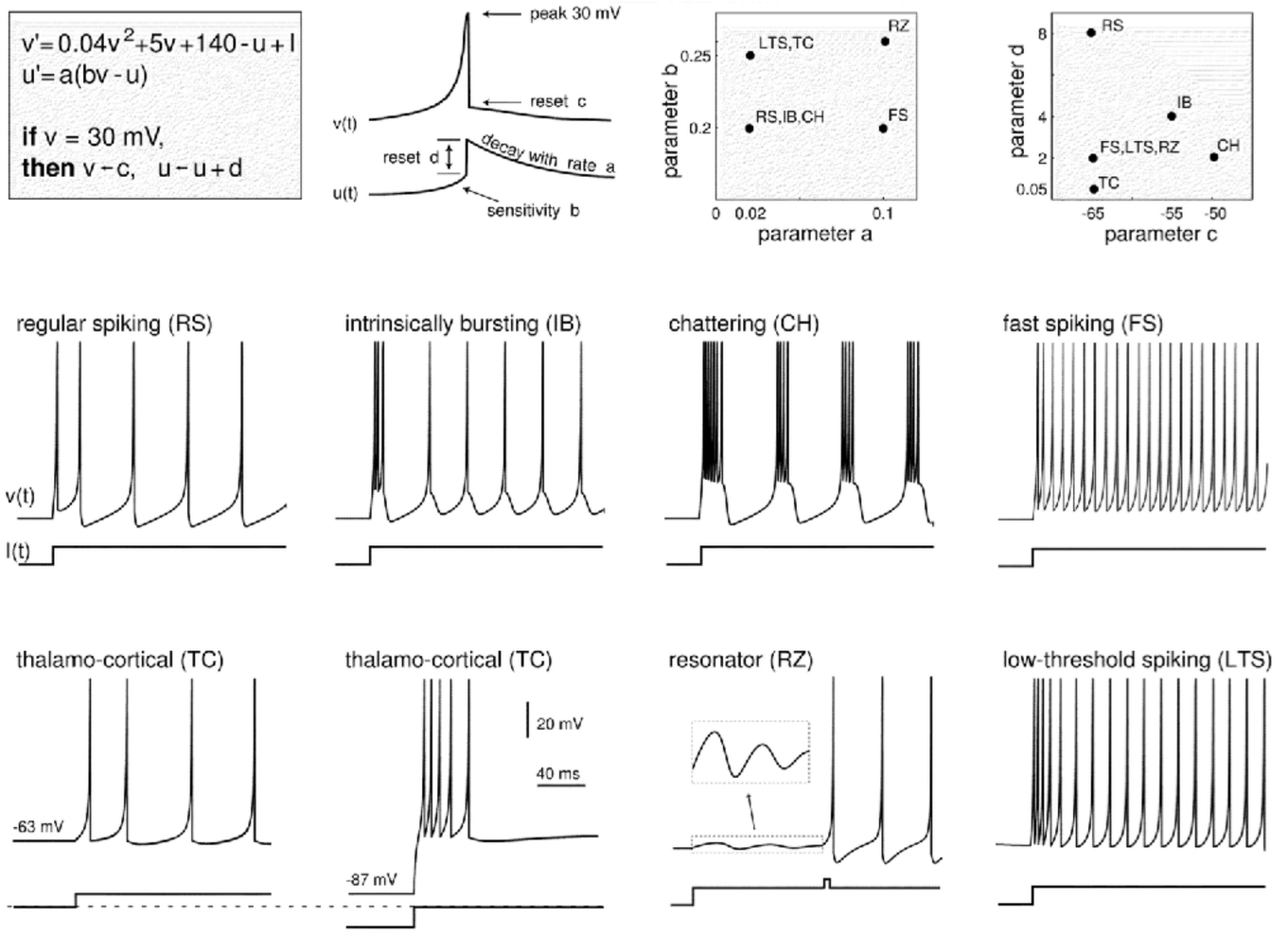
Universal approximators of smooth dynamics.

Model neurons or convenient basis of dynamics.

See Python Demo.

Izhikevich Neural Model

Neuron model with an additional state variable per neuron that produces spiking and bursting behavior.



Python Demo available if interested.

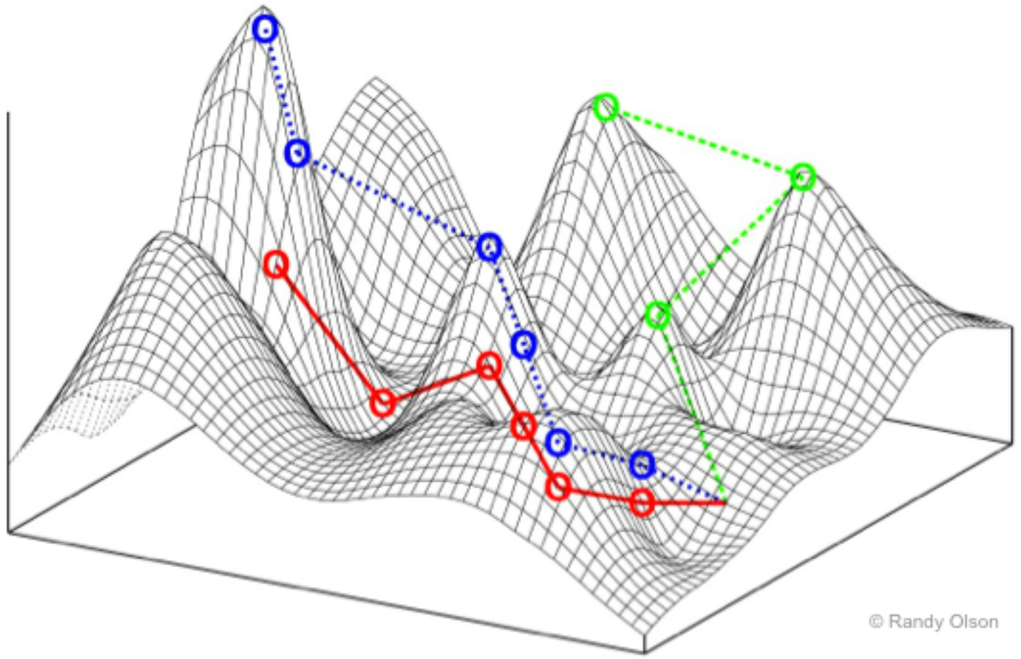
Part 2: EAs

A. Motivation for the use of evolutionary algorithms to find ensemble of model solutions.

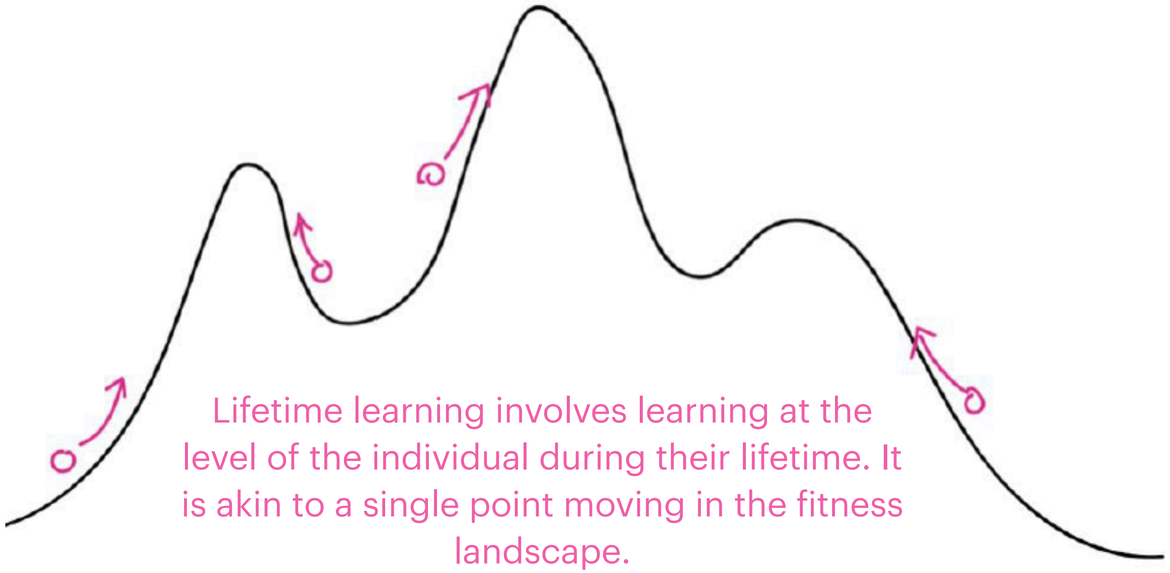
B. Basics of an evolutionary algorithm strategy

C. Walk through the design, implementation in Python (code available before hand).

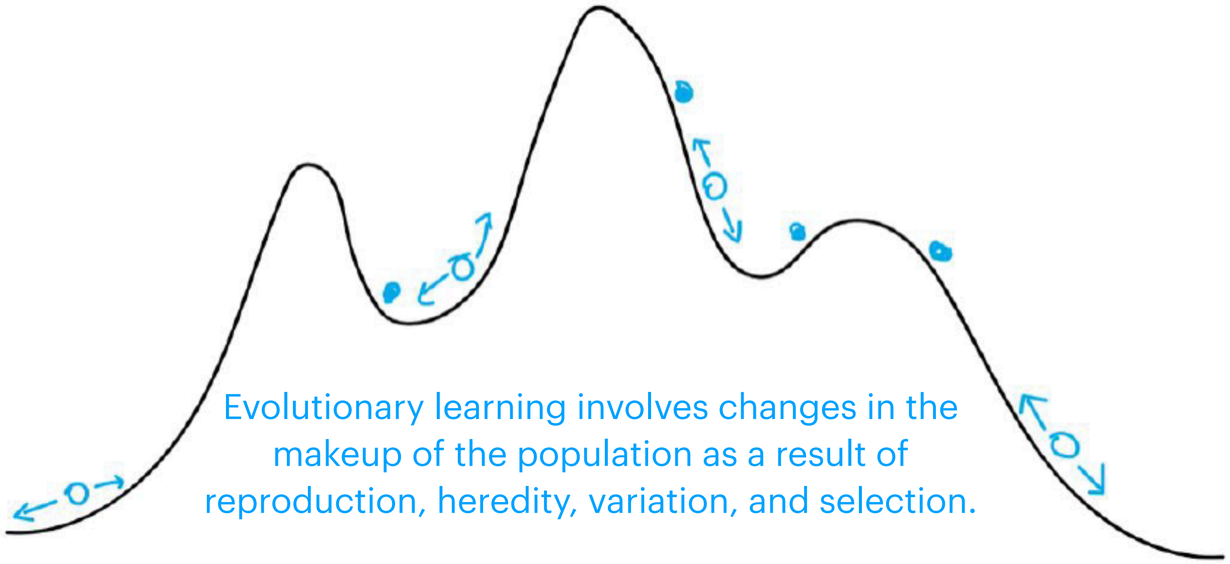
Lifetime learning and Evolutionary Learning



Learning as any kind of movement over the fitness landscape that results in improvements in the adaptive behavior.

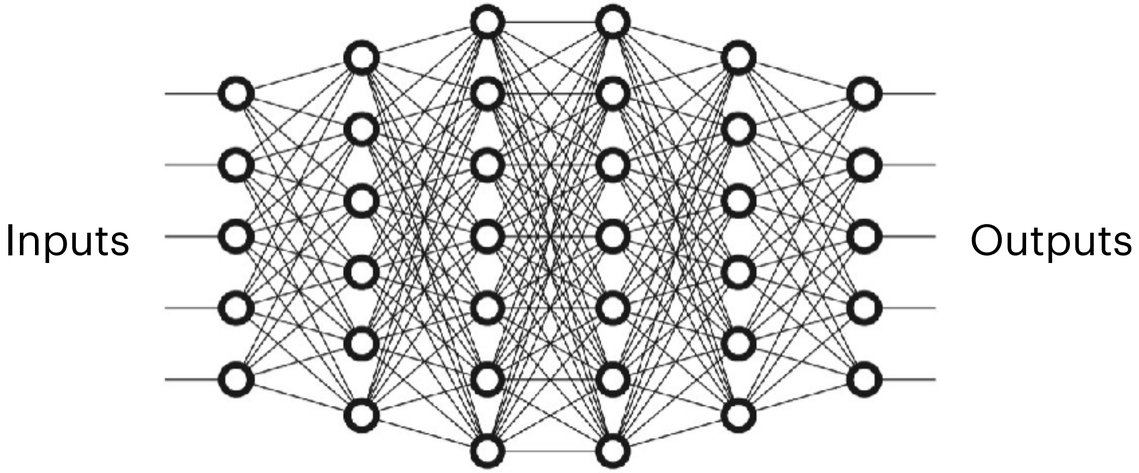


Lifetime learning involves learning at the level of the individual during their lifetime. It is akin to a single point moving in the fitness landscape.

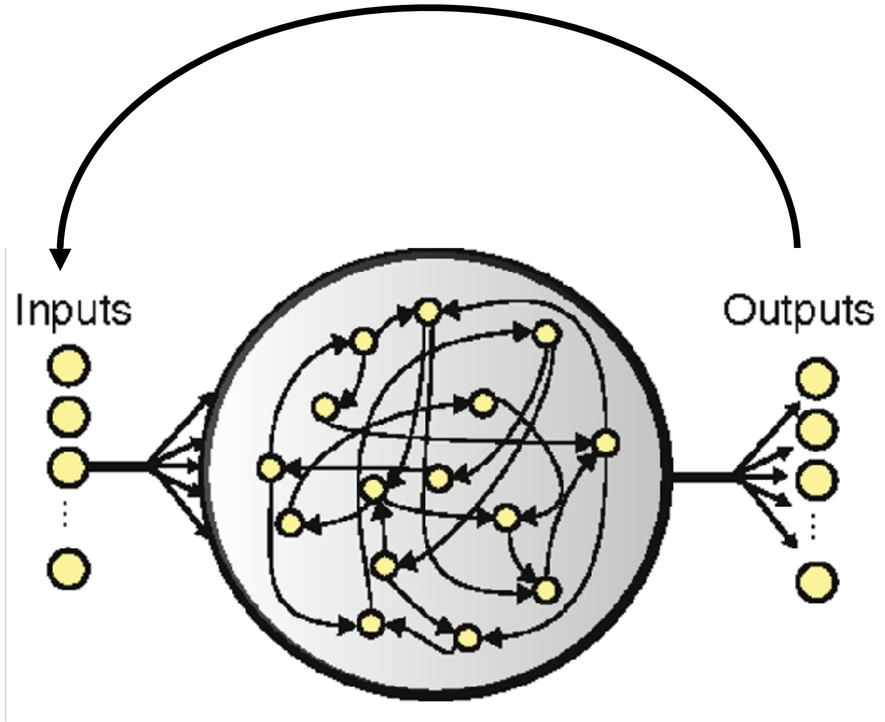


Evolutionary learning involves changes in the makeup of the population as a result of reproduction, heredity, variation, and selection.

Training Neural Networks

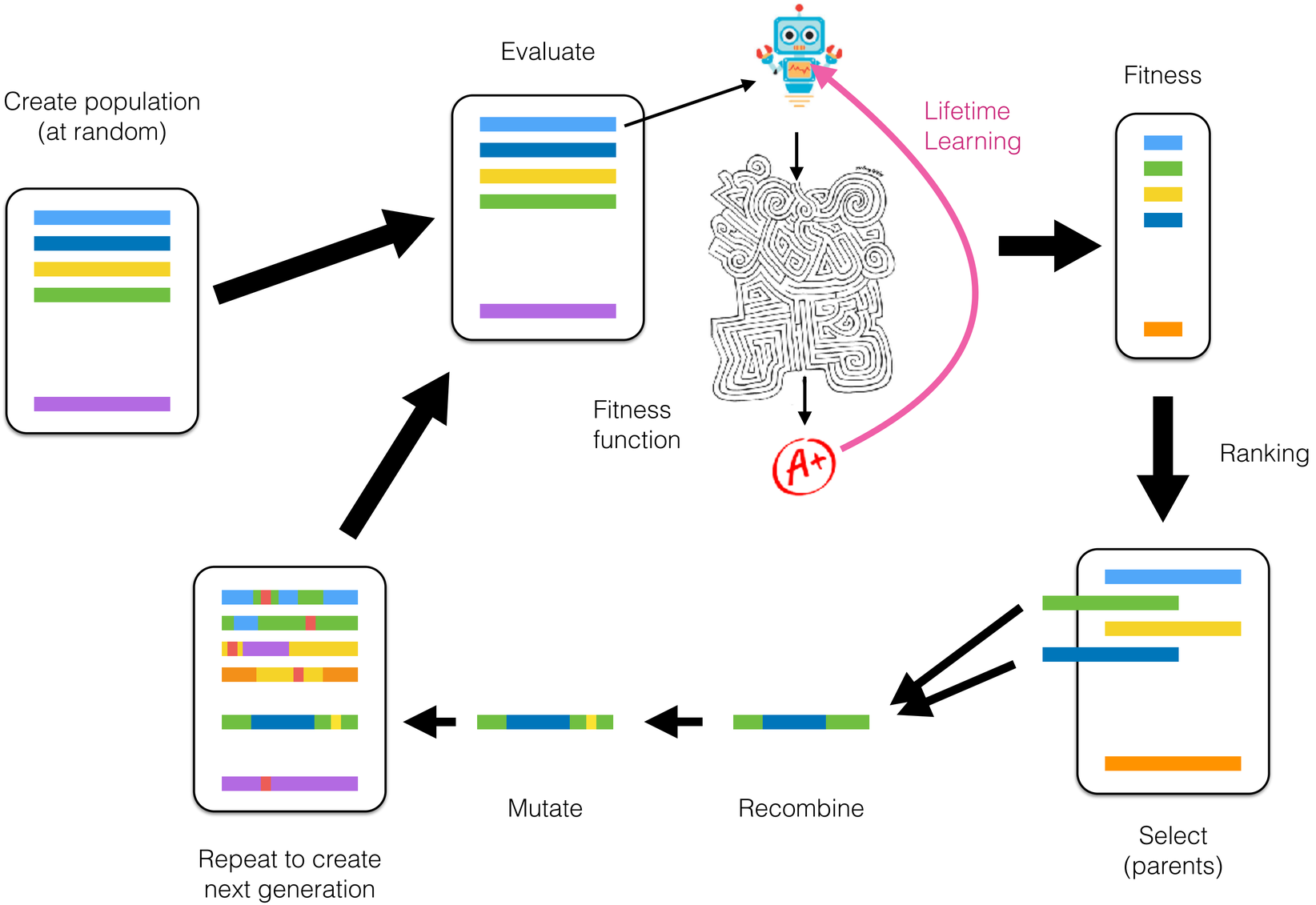


Gradient-based methods are the dominant method for training feedforward ANNs.
However, neuroevolution approach also used.
See https://eng.uber.com/deep-neuroevolution/ for more info.



Except on relatively simplified scenarios, gradient-based methods are feasible for DRNNs on embodied tasks.
Neuroevolution is the primary method for training embodied and situated DRNNs.

Both forms of learning (lifetime and evolutionary) occur in tandem in all living organisms



Artificial Evolution

For **n** times round generation loop

Evaluate all the population

Select preferentially the fitter ones as parents

For **p** times round reproduction loop

Pick 2 from the parental pool

Recombine to make 1 offspring

Mutate the offspring

Throw away parental generation, replace with offspring.

Requirements

Heredity — there's a form of replication/reproduction and offspring are (roughly) identical to their parents.

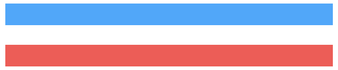
Variability — except not exactly the same, some significant variation introduced through mutations.

Selection — the 'fitter' ones are likely to have more offspring.

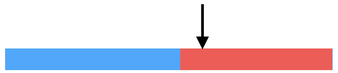
Operators

Recombination

Typically 2 parents recombine to produce offspring.

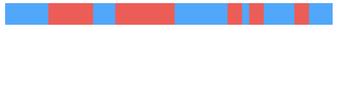


1 point crossover

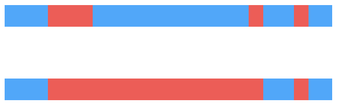


2 point crossover

Uniform crossover



Uniform crossover
with a tunable percentage



80%

20%

Mutation

After an offspring has been produced

Mutate at randomly chosen loci with some probability

If the genotype is not discrete but real valued, then typically a mutation involves changing all the loci with a random variable generated from e.g., Gaussian distribution with mean 0 and a small enough standard deviation.

Selection

Truncation selection. All parents come from top scoring 50% (or 20%, etc).

A different common method: **Fitness-proportionate**. If fitness of (an example small) population are: 2, 5, 3, 7, 4 total 21, then to generate each offspring you select each parent with probability: 2/21, 5/21, 3/21, 7/21, 4/21.

How about if early on all scores are zero (or near zero) except one slightly bigger — then it will 'unfairly' dominate the parenting of the next generation.

If later on all the scores vary slightly about some average (e.g., 1010, 1020, 1017, ..) then there will be very little selection pressure to improve through these small differences.

Rank selection. Line up all in the population according to rank, and give them probabilities of being selected-as-parent in proportion.

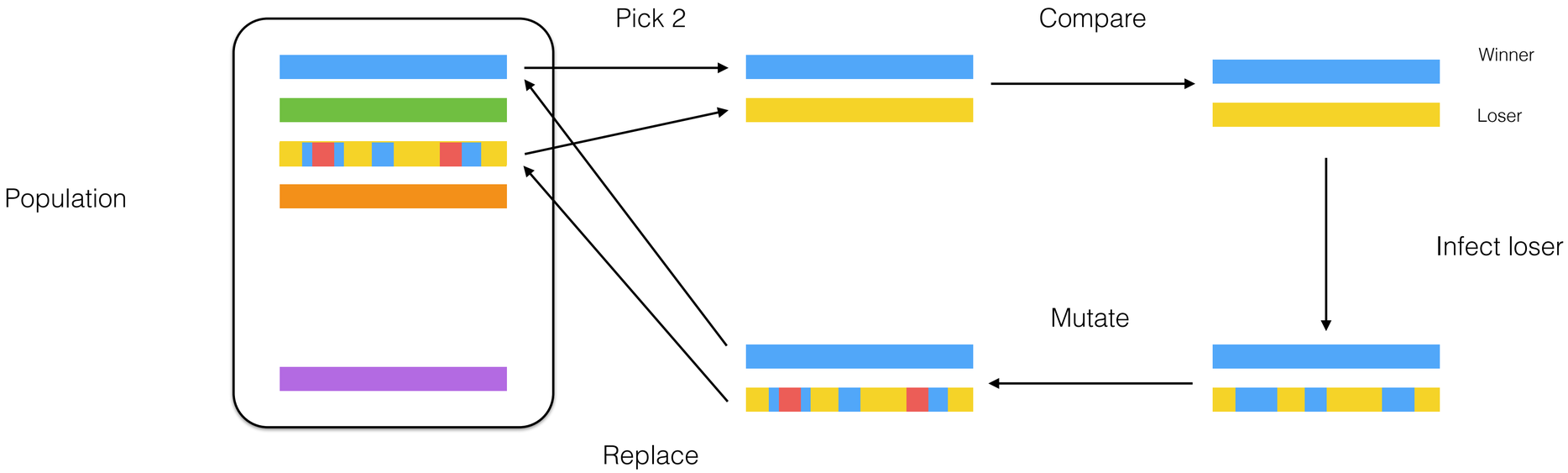
with linear rank selection you ignore the absolute differences in score — focus purely on ranking.

The 'line' in linear ranking need not slope from 2.0 to 0.0, it could e.g. slope from 1.5 to 0.5. You could have nonlinear ranking. But the most common way (I recommend unless you have good reasons otherwise) is such a linear slope from 2.0 to 0.0 as shown.

This means that the **best** can expect to have **twice** as many offspring as the **average**. Even below-average have a sporting change of being parents.

Microbial GA

Microbial sex:
Instead of “Let’s make babies!”
It is: “Want to share some of my genes?”



Inman Harvey (2011). The Microbial Genetic Algorithm.
Advances in Artificial Life. Part II, LNCS 5778, pp. 126-133.