AI Summary

Massimo Perego

Contents

1 Neural Networks

1.1 Multi Layer Perceptrons MLP

1.1.1 Definition

An r-layered perceptron is a feed-forward neural network with a strictly layered structure and an acyclic graph.

Each layer receives input from the preceding one and passes its output to the subsequent layer, jumps between non-consecutive layers are not allowed. Usually, each neuron is fully connected to the neurons in the preceding layer.

They are "feed-forward", information flows in one direction only, there can't be cycles. They can have significant processing capabilities, since they can be increased by increasing the number of neurons and/or layers.

There are different types of layers (and thus neurons):

- Input: receive inputs from the external world
- Hidden: one or more layers that perform transformations on the inputs
- Output: produces the final outputs of the network

Each neuron receives multiple inputs, either from the previous layer or from the external world, each one with an associated weight. The network input function combines these input, usually with a weighted sum, and thus determines the global solicitation received by the neuron.

The activation function determines the neuron's activation status based on its inputs. It is a so-called sigmoid function, a monotonically non-decreasing function with a range [0,1] (or [-1,1] for bipolar sigmoid functions). It (usually) introduces non-linearity, giving the neuron its processing capabilities and allowing the network to learn complex patterns (if they were linear we could merge all functions into one).

The output function produces the final output of the neuron based on its activation status and the output is then passed to the next layer. It is often the identity function. It might be useful to scale the output to a desired range (linear function).

1.1.2 Function Approximation

Any Riemann-integrable function can be approximated with arbitrary accuracy by a four-layer multi-layer perceptron. We approximate the function into a step function and construct a neural network that computes said function.

The input is taken by a single input neuron, and the first hidden layer is composed of a neuron for each of the step borders of our approximated function. Each neuron determines on which side of the step border an input lies.
In the second hidden layer there's a neuron for each step, that receives input from the two neurons that refer to the values marking the border of the step. The weights and threshold are chosen is such a way that neurons in the second layer are active only if the function value is inside the step, i.e., only one of the neurons in the preceding layers is active. Only a single neuron in the second layer can be active at a time.
The output layer has the identity function as activation and receives only the value of the step as input.

The accuracy can be increased arbitrarily by increasing the number of steps.

We can remove a layer and simplify the approach by considering the variation of the function value between each step. There is one hidden layer and outputs from this layer are weighted with the relative difference between steps.

1.1.3 Regression

Training a NN is closely related to regression, the statistical technique for finding a function that best approximates the relationship in a data set; both regression and MLP training involve minimizing an error function, usually the mean square error. We need to adapt weights and parameters of the activation function to minimize said error.

Types of regression:

- Linear: When a linear relationship between quantities is expected;
- Polynomial Regression: Extends linear regression to polynomial functions of arbitrary order;
- Multi-linear Regression: Used to fit functions with multiple arguments;
- Logistic Regression: Particularly relevant to ANNs because many such networks use a logistic function as their activation function. If we can transform a function to a linear/polynomial case we can determine weights and thresholds for the system, for a logistic function this can be done by

way of the Logit transformation, allowing a single neuron to compute the logistic regression function.

1.1.4 Training

With the term "training", for an MLP, we're talking about minimizing the error function on the data set given for the training.

Gradient Descent: We can derive from the error function a direction in which to change the weights and thresholds to minimize the error.
The gradient is a vector in the direction of the steepest increase of the function. We make small steps (size determined by a learning rate) in the direction indicated by the gradient on the error function until convergence, i.e., a local optima of the error function is found.
This requires having a differentiable activation and output function.
The algorithm will essentially be:

- Compute the gradient
- Small step in the opposite direction of the gradient
- Repeat until convergence

Some variants have been developed to address the challenge of learning rate selection and overcoming local optima:

- Random restart: train the network multiple times, with different starting points
- Momentum: Adds a fraction of the previous weight change to the current step
- Manhattan Training: Uses only the sign of the gradient to determine the direction of the step, simplifying computation
- Adaptive Learning Rates: Adapt the learning rate for each parameter based on the history of gradients

Error Backpropagation: Only the output neurons are connected to the error, but we need to train the whole network.
The error values of any (hidden) layer of a multi layer perceptron can be computed from the error values of is successor layer. The error is computed at the end of the network and then backpropagated through the whole network.
General structure of the algorithm:

1. Setting and forward propagation of the input
2. Calculate the error and adapt the weights for the last layer

3. Error backpropagation, the "new" error factor is computed starting from the error of the subsequent layer, and this is done layer by layer

This allows to calculate how much each neuron "contributes" towards the final error.

The weight adaptation depends on a learning rate, which has to be initialized properly in order to not "jump" over the minimum without ever converging (i.e., it's too high).

The error can't completely vanish due to the properties of the logistic function, there will always be some residual errors due to the computation of the various parameters.

1.1.5 Deep Learning

The term "Deep Learning" refers to a NN with several hidden layers. With "depth" we mean the number of layers that separate input and output.
Approximating a function with only three layers might require a large (even exponential) number of neurons, while more hidden layers allow for the same approximation with fewer neurons. Increasing the number of hidden layers decreases complexity. Having less neurons also allows using smaller data sets.

The main problems with Deep Learning are:

- Overfitting: we're increasing the quantity of adaptable parameters, and thus capabilities, it might lead to overfitting. Some solutions are: weight decay (avoiding large values for the weights), sparsity constraints, dropout training;
- Vanishing gradient: the gradient tends to vanish through the layers, slowing down significantly learning in the first layers. Some activation functions could counteract this by having larger values for the gradient. Another approach is training the network layer by layers, usually as a series of stacked autoencoders: 3-layer perceptrons that maps its inputs to approximation of themselves (layers are: encoding-hidden-decoding); the hidden layer is expected to learn features of the inputs, so we need to limit the number of hidden neurons. After training an autoencoder normally, the decoder layer is removed and another, not yet trained, autoencoder is added

1.1.6 Convolutional Neural Networks

Still inside the field of deep learning, but the "receptive field" of each neuron is reduced, i.e., each neuron is connected only to a partial region of the input data

(preceding layer, we're removing the fully connected constraint). This allows for very deep networks, with a reduced number of connections. The "convolutional" comes from all neurons in the same layer sharing the same weight, like convolution (kinda like sliding a window over an image).
Neurons in the layer after the convolution apply maximum pooling, keeping only the maximum activation of the neurons for each sampled region, maintaining the obtained results but losing knowledge of their location in the original input. This allows to extract features and remove noise.
Further layers allow for more high-level features, building a hierarchy over multiple stages.

1.2 Radial Basis Function Networks RBFN

1.2.1 Definition

RBFNs are a feed-forward (the data flows only in one direction, forward, no cycles) neural networks with always three layers. Each previous layer is fully connected to the subsequent.
In the hidden layer radial basis functions are employed as activation functions. A RBF is a function that peaks at zero and decreases in all other directions.

From hidden to output neurons the activation function is the "classical" weighted sum of all inputs. The activation function of the output neurons is linear and propagates to the output.

The input function of each hidden neuron is a distance function (there can be different types) between input and weight vectors. Each hidden neuron receives as input a distance. The activation function is a radial function, which decreases monotonically. The catchment region, defined by the reference radius σ, defines the shape and width of the function, i.e., the function is 1 at 0, decreases in some way until σ. Each neuron can have its own function.
Since the function is activated based on distance, this draws a circle (or equivalent for a certain definition of distance) in the feature space, where the data considered is located.

1.2.2 Function Approximation

RBFNs are universal approximators, like MLPs, but give more choices on how to approximate, improving the approximation, alternatively, obtaining the same approximation with fewer neurons.

Each function can be approximated by a delta approach in which the resulting function is the weighted sum of the function that define the RBFN. Using rect-

angular functions give the same approximation as an MLP, while triangular or Gaussian function allow smoother transitions between steps.

1.2.3 Training

Considering a case of supervised learning with a "simple RBFN", where each training example is covered by its own RBF, i.e., a neuron for each training example.
In this case the weights to the hidden neurons are initialized with the value of the respective training example.
The radii of the activation functions can be chosen heuristically in such a way that each function doesn't interfere with other patterns. We center each radial function around a specific pattern.
We then can find analytically find the value of the weights from hidden to output neurons, it's the vector of desired outputs multiplied by the inverse of the matrix containing the hidden layer outputs.
This method guarantees perfect approximation, it's not necessary to train a simple RBFN.

General Radial Basis Function Networks possess fewer neurons than training examples, we need to select a subset of the training patterns as centers, which will become the input weights for the hidden neurons, then we find the weights for the output layer as before, with an over-determined matrix.

We need to find "good" centers, since they influence the training, they become what the radial function is based on. A way to find the centers is C-means Clustering: considering a number c of clusters to be found

1. Initialize the cluster centers randomly
2. Assign each training data point to the closest cluster center
3. Recalculate the centers from the new data points assigned
4. Repeat the last two steps until convergence

Then we can add backpropagation to train the network since it will not be perfectly accurate, following standard backpropagation rules.

A 3-phases RBF training consists of:

- Find output connection weights with inverse
- Find RBF centers (clustering)
- Error backpropagation

1.3 Learning Vector Quantization LVQ

1.3.1 Definition

Learning Vector Quantization is a way to find a suitable quantization (many-to-few mapping, often to a finite set) of the input space, a way to divide in cluster the input space.The clusters are represented by a center or reference vector.

The data points are processed one by one and only one reference vector per data point is updated. Competitive learning: only the "winner neuron", i.e., the one with the highest activation, is adapted

1.3.2 LVQ Networks

A LVQN is a feed-forward 2-layered neural network. It can be seen as RBF without the output layer, the activation for each of the neurons in the hidden layer is used as output.

The network input function for each output neuron is a distance function of input vector and weight vector, for some definition of distance.
Each neuron in the hidden layer has its own radial function centered in a different place whose input is a distance from input to weight vector.

The output function of each output neuron also takes into consideration the activation of all input neurons: winner takes all, only the biggest activation will lead to a value of 1, all other neurons will be 0. We need a connection between all neurons in the output layer to "check the winner".

Training: We want to learn the position of the reference vector (center of the radial function). For each training pattern we find the closest reference vector (winner neuron) and adapt only that one.

To adjust the weights we can use (assuming supervised learning):

- Attraction rule: reference vector and point have the same class, move the reference closer to the point
- Repulsion rule: reference vector and point have different classes, move the reference away

There is a learning rate η that determines how much the new information matters, this rate should be time-dependent, if it doesn't decrease as time goes on it could lead to oscillations without convergence.

We also could update the two closest reference vectors, provided that they are of two different classes, by using attraction and repulsion rules. The point is in the middle of two clusters, update both the correct and wrong one (in the proper direction).

Standard LVQ may drive reference vector further and further apart. The window rule consists in updating the data only if the point is close to the classification boundary, it's inside a "window". This way the adaptation stops as soon as the classification borders are far enough away.

1.4 Self-Organizing Maps SOM

1.4.1 Definition

SOM (or Kohonen feature maps) are a feed forward neural network with 2 layers, with local connections only among neighboring hidden/output neurons.

The input to each output neuron is a distance function between the input vector and a weight vector. The activation function of each output neuron is a radial function.

The output function of each output neuron is the identity, and the output is often discretized according to the winner-takes-all principle.

There is a neighborhood relationship defined on the output neurons, the centers of the radial function are connected together, leading to a (usually two-dimensional) grid. Each neuron is connected to its nearest neighbor and has a region assigned to it. Reference vectors close to each other in the input space belong to neurons not too far away from each other.

1.4.2 Topology Preserving Mapping

The SOM perform a meaningful dimensionality reduction, preserving topology, i.e., maintaining spatial relationships present in the input data. We can map high-dimensional structures (input space) onto low-dimensional spaces.

Training has to be adjusted to incorporate maintaining the neighboring structure, which works only on unsupervised learning.

We need to modify the attraction rule in a way that considers the concept of neighborhoods, we do this by incorporating a radial neighborhood function,

which determines if a neuron has to be updated (is inside the neighborhood radius). More than one neuron at a time is updated, in a way that takes into account the distance between winner neuron and the one to update (determined by the neighborhood function), i.e., all neurons.

Learning rate and neighborhood function radius should decrease with time, to avoid oscillations and ensure convergence.

For the training, the weight vector of the neuron in the self-organizing map is initialized randomly, the reference vectors are placed randomly, using random training examples.
The training itself consists of:

- choosing a training data point
- finding the winner neuron, w.r.t. the distance function
- adapt all neurons in the radius given by the time dependent neighborhood function

1.5 Hopfield Networks HN

1.5.1 Definition

HN are a type of recurrent neural networks (it contains cycles, but no self-loops), in which every neuron is both input and output and all neurons are connected to every other.

The connection weights are symmetric.
The network input function of each neuron is the weighted sum of the outputs of all other neurons.
The activation function of each neuron is a threshold function.
The output function of each neuron is the identity.

Since it's recurrent, the behavior depends on the update order of the neurons. It can be

- synchronous: all at the same time, can lead to oscillations
- sequential: one after the other

Convergence theorem: If the neurons are updated sequentially, a stable state is always reached within a finite number of steps.
To prove this, we can map every state of a HN to a real number, reduced with every state transition. This is called energy function.
So the energy of the system can only decrease, we can't go back to higher energy

states, eventually reaching a local minimum of the function, i.e., a stable state, a traversal of all the neurons without a state change.

1.5.2 Associative Memory

HN can easily implement associative memory, i.e., address memory by its contents. When presented a pattern, it returns whether this pattern coincides or not with an existing one, and it doesn't need an exact coincidence.

The associative memory works by exploiting the stable states of a HN: the weights and thresholds make the pattern coincide with the stable states.

To store patterns in a HN (i.e., make them stable states), we need to find the weight and threshold that make each pattern a stable state of the system. The final weight matrix is the sum of all matrices computed for all patterns.

1.5.3 Solving Optimization Problems

We can exploit the minimization of the energy function to solve optimization problems by converting the function to optimize in an energy function to minimize and constructing the HN with weights and thresholds given by the energy function. Then we initialize the HN randomly and update until converge, each stable state is a local optima of the original function.

The way a HN is updated limits the "moves" that can be performed since compound moves are not allowed if they increase energy before reducing it.
We only know that the solution found will be a local optima, in a process similar to gradient descent/hill climbing.

Simulated Annealing: To solve getting stuck in a local optima, we start from the idea of Simulated Annealing: starting from a random solution, choose other random solutions (usually in a neighborhood) and evaluate them: accept them if they're better, while if they're worse accept them with a probability based on a parameter temperature that decreases over time.

For a HN this can be translated into accepting a transition to a higher energy state with a certain probability, in hopes of finding a better solution.

1.6 Boltzmann Machines BM

1.6.1 Definition

Instead of modeling a single pattern like a HN, BM want to model a probability distribution of the data over the whole input space. They differ from HN mainly in the way neuron states are updated.
They're stochastic recurrent neural networks that model complex probability distributions.

A probability distribution is defined on the states, based on the Boltzmann distribution, with the help of the energy function; more probable configurations are represented by lower energy states.
The energy function takes as input the state of the system, represented by the vector of neuron activations.
The probability of a neuron being active is a logistic function of the (scaled) energy difference between its active and inactive state.

Update procedure: Each update step is:

- A neuron is chosen (randomly)
- The energy difference and then the probability of the neuron having activation 1 are computed
- The neuron is set to activation 1 with this probability (and to 0 with the complement of said probability)

Repeated many times, for randomly chosen neurons. This is a Markov-Chain Monte Carlo (MCMC) procedure.

After enough steps, the probability of the network being in a specific activation state depends only on the energy of that state, independent of the activation state the process started with.

1.6.2 Training

The probability distribution represented by a Boltzmann machine via its energy function can be adapted to a given sample of data points, by adapting weight and threshold values.
The data points must be compatible with a Boltzmann distribution, but we can mitigate this by dividing the neurons into visible and hidden; the activations of the latter are not fixed by the data points (deviation from the HN). The probability distribution is thus fixed only on the visible neurons.

The approach is to measure the difference (needs some measure of distance/loss) between two probability distributions and use gradient descent to minimize said difference. Starting from random values, minimize through gradient descent the difference between distribution created and data given.

There are two phases to each training step:

- positive: visible neurons are fixed to a random data point, hidden neurons are updated until a stable state is reached (we're using the data set)
- negative: all units are updated until a stable state is reached (the system is free)

The update to weights and thresholds is given by the difference in probability of activation between the two phases. If a neuron is not activating enough times during the negative, the threshold should be lowered. If two neurons are active together more often during the positive, the weight among them should be increased.

1.6.3 Restricted Boltzmann Machines RBM

The training of standard BM is impractical unless the network is very small. To limit complexity, RBM use a bipartite graph instead of a complete one, connections go only from neurons in the visible layer to the hidden one, there are no connections among the same group (all input neurons are still also output neurons, hidden is just a name here).

Due to lack of connections, the training can now proceed by repeating three steps:

1. visible units are fixed to a random data sample $\vec{x}$, hidden ones are updated once and in parallel, obtaining $\vec{y}$and retrieving positive gradient xy^T
2. hidden neurons are fixed to $\vec{y}$, visible ones are updated once and in parallel, "reconstructing" the training example $\vec{x}^*$. Fix the visible neurons to the reconstruction, update the hidden ones and obtain $\vec{y}^*$ and thus the negative gradient x^*y^{*T}
3. update the connection weights with the difference of the positive and the negative gradient (with a learning rate)

RBM can be used to build deep networks, similar to stacked autoencoders for the MLP.

1.7 Recurrent Networks RNN

RNN are NN without the constraints of all networks seen until now.
The output is generated when stability is reached.

The configuration can be

- by construction if the structure of the computation is known
- by extending the error backpropagation algorithm in time to deal with recursion

RNN can be used to represent differential equations. The output of the network can be reconstructed by considering the output in the previous instant of time, i.e., using the derivative. This allows to transform the equation into an RNN, creating for each variable a node inside the graph and associating to the connections the value of the differential.

Vectorial Neural Networks: RNN can be composed of multiple recurrent sub-networks, allowing to compute vectorial differential equations.

Error backpropagation in time: Backpropagation is not directly applicable since loops propagate errors in a cyclic manner. We need to "unfold in time" the network, by considering all neurons for every time slice one after the other. Backpropagation is then computed on the "unfolded" network and adjustments of the same weight are combined to generate the final value.

2 Fuzzy Systems

2.1 Fuzzy Sets

2.1.1 Definition

Classical logic allows for only 2 truth values: *true* and *false*, but many propositions in the real world are not always that well-defined and we use some imprecise linguistic terms. Also, as the complexity of a system increases, making precise and significant statements about its behavior becomes more difficult, rendering abstraction of details (i.e., approximation) needed.

To model linguistic imprecision we use fuzzy systems, which allow approximate reasoning in automated systems, by considering "degree of truths", every proposition is true to a certain degree.

A fuzzy set μ is a set of elements with a continuum of membership grades, i.e., a mapping

$$\mu : X \mapsto [0, 1]$$

assigning to each element $x \in X$ a degree of membership $\mu(x)$ between 0 (no membership) and 1 (full membership). Values of 0 and 1 correspond to the equivalent values in crisp sets.

Membership function: Expresses the degree to which an element belongs to a fuzzy set, without a discrete threshold, it aims to be an interface between linguistic model and numerical representation. The membership degrees are fixed only by convention and the unit interval between membership grades is arbitrary.
Furthermore, the membership function attached to a specific linguistic description is dependent on the context ("young" in "young retired person" is different than in "young student").

Linguistic Variables and Values: Linguistic variables assume linguistic values and represent attributes in fuzzy systems. Linguistic values partition the possible values of a variable subjectively, taking into account the value, set of values and context (for "size": "small, medium and large" mean 3 different "ranges" of possible membership). The grammar and fuzzy set for the linguistic expressions must also be defined.

Semantics: Fuzzy sets, and thus membership grades, can have different interpretations and can be used to model:

- Similarity: the membership represents a degree of proximity from a prototype element; used in pattern classification, regression, cluster analysis
- Preference: the membership represents intensity of preference and/or feasibility of a choice, the fuzzy set represents criteria/flexible constraints; used in fuzzy optimization, decision analysis
- Possibility: the membership represents a degree of possibility that a certain parameter has the considered value, distinguishing plausible clauses from less likely ones; used in expert systems, artificial intelligence

2.1.2 Representation of Fuzzy Sets

Vertical Representation: Fuzzy sets can be described by their membership function by assigning a degree of membership to each element of the set mapped by the fuzzy set. Considering $\mu : X \mapsto [0, 1]$, assign $\mu(x)$ to all $x \in X$; this is more natural but harder to use in automated systems.

Horizontal representation: For how many membership degrees we want, taken as a subset of $[0, 1]$, list the subset of elements that have at least that membership value. Considering elements with value $\geq \alpha$, this is called α-cut of the fuzzy set. Basically, for some α, identify the set of input values with a membership degree higher than α (with $>$ are called strict α-cuts).

Any fuzzy set can be described using its α-cuts, and the representation of a fuzzy set can be obtained as an upper envelope of its α-cuts, i.e., I can obtain the fuzzy set starting from all its α-cuts by considering, for each possible input value, the maximum membership degree (sup) that contains said value.

This representation is easier to process in computers, using finite discrete values for membership and input values, obtaining an approximation of the original fuzzy set. They are usually stored as a chain of linear lists, useful for arithmetic operators.

Support: Set of all elements of the input space that have non-zero membership.

Core: Set of all elements of the input space that have membership of one.

Height: The maximum membership values of any element in the input set. A fuzzy set is called "normal" if the height is 1.

Fuzzy Number: A fuzzy set is a fuzzy number iff it's normal, bounded, closed and convex.

2.1.3 Fuzzy Logic

We need to define "classic" operators on fuzzy sets, extending the truth table of each operator to deal with degrees of membership instead of crisp values. We need the ability to combine fuzzy truth values. A minimum requirement of this extension is that, when restricted to values 0 and 1, it should coincide with the classical operator.

T-Norms: A possible operator for the conjunction should keep commutativity, associativity and monotonicity of the classical operator. We also want that the conjunction between any proposition and truth value 1 results in the value of the first proposition.
Any function that takes the unit interval squared (two fuzzy variables) and gives back a value in the interval, which respects these axioms is called a T-Norm. Corresponds to conjunction, and thus intersection since it's just the conjunction of each element being in the sets.

T-Conorm: Similarly to before, we want commutativity, associativity, monotonicity and that the disjunction between 0 and a proposition is equal to the value of the proposition. Any such function is a T-Conorm. Corresponds to disjunction and thus union, since it's just the disjunction of each element being in the sets.

Negation: The requisites for a negation operators are that the crisp values get "inverted", together with monotonicity. A simple negation operator is the complement: $\neg\mu(u) = 1 - \mu(u)$. It's called "strict" if the operator is strictly decreasing and "strong" if it's involutive $\neg\neg\mu(x) = \mu(x)$.

Implication: A simple way of defining it could be by using the already defined operators: $I(a, b) = \neg a \wedge b$. There can be many more, but they all come from the generalization in some way of the classical implication and they collapse to that when restricted to values 0 and 1.

For all operators, there can be many ways of defining them, provided they respect the axioms/property stated. They can be defined based on the usage needed, making them dependent on the context.

Extension principle: We want to generalize mappings or functions defined on crisp values to fuzzy sets, using the interpretation of the partial truth of a membership degree. Starting from a crisp function $f : X \to Y$ and considering the fuzzy set μ on X, with membership function $\mu(x)$, the extension principle tells us how to determine the membership function of the fuzzy set on the co-domain after applying the function f.

Basically, for each output value ($y \in Y$), we look at all possible input values ($x \in X$) that could produce y through f and take the highest membership degree among those.

Arithmetic operations on fuzzy sets: Arithmetic operations can be defined on fuzzy sets using the extension principle. It's desirable to reduce fuzzy arithmetic to ordinary set arithmetic and applying the elementary operations of interval arithmetic. We can do this by using a set representation, but the representation can be determined directly from the α-cuts of an extension only if the function is continuous and and the fuzzy sets are upper semi-continuous (can be described by an upper semi-continuous function). Then the α-cuts of the mapping are the mapping of the α-cuts.

2.1.4 Fuzzy Relations

We want to generalize the idea of crisp relations to allow for degree of associations between elements. We want to represent the "strength" of the relation among elements.

A fuzzy relation is a fuzzy set defined on tuples that may have a varying degree s of membership within the relation. The membership grade indicates the strength of the relation. A crisp relation tells us only if the element share the concept while a fuzzy relation tells us how much of that concept they share.

The Cartesian product of fuzzy sets is usually a fuzzy relation, with (usually) membership degree equal to the minimum (T-Norm). A subsequence is a partial subset of tuples in a Cartesian product.

Projection: Reduce a multi-dimensional fuzzy relation to another one with less variables, removing some dimensions (literally a projection on one of the axis). We project a relation on a subset by taking the maximum in the relation for all subsequences considered. The max can be generalized to another T-Conorm.

Cylindric Extension: Opposite of projection, extend a relation to all original dimensions. For each subsequence we want the largest, least specific, fuzzy relation compatible with the projection. No information not included in projection is used to determine the extended relation.

Cylindric Closure: We can reconstruct the original relation starting from several of its projections. We take the intersection of all cylindric extensions.

2.1.5 Binary Fuzzy Relations

Binary relations are generalized mathematical functions, fuzzy binary relations introduce membership grades for the strength of the relation among two sets.

Considering a fuzzy binary relation $R(X, Y)$:

- **Domain:** maximum grade of each $x \in X$, considering any possible $y \in Y$
- **Range:** maximum grade of each $y \in Y$, considering any possible $x \in X$
- **Height:** maximum grade for each pair $(x, y) \in R$
- **Inverse:** a relation can be represented by a matrix (think adjacency matrix for a graph), so the inverse of the relation is the transposed matrix

Binary Relations on a single set: It's possible to define fuzzy binary relations among elements of a single set. A possible representation is a directed graph. A fuzzy binary relation is:

- Reflexive iff $\forall x \in X : R(X^2) = 1$
- Symmetric iff $\forall x, y \in X : R(x, y) = R(y, x)$
- Transitive iff $\forall (x, z) \in X^2 : R(x, z) \leq$ than the maximum possible value of $\min\{R(x, y), R(y, z)\}$, $\forall y \in Y$

A fuzzy binary relation which satisfies the properties above is called a fuzzy equivalence relation.

2.2 Fuzzy Control

Fuzzy Control is a way of defining a nonlinear table-based controller, the nonlinear transition function can be defined without specifying every entry of the table individually, it wants to specify a behavior.
All control systems share a time-dependent output variable and said output is

controlled by a control variable. Disturbance variable might influence the output.

Since it's often very difficult to specify an accurate mathematical model for a system the fuzzy approach uses a knowledge-based analysis, via some linguistic rules.
To define a fuzzy controller we need to formulate a set of linguistic rules, each input variable has to be partitioned into fuzzy sets representing the different states/levels of the variable.
We need to identify, for each variable, the relevant linguistic terms and represent each one via a fuzzy set (what's the fuzzy set corresponding to a "high" load? What about "low" load?). Then we need to partition the domain of the input variable among all fuzzy sets.

2.2.1 Architecture of a Fuzzy Controller

The architecture of a fuzzy controller is composed of:

- Fuzzification interface: receives the current input value (eventually maps to a suitable domain), determines the degree to which each input belongs to each fuzzy set
- Knowledge base: consists of data base and rule base; the data base contains information about boundaries, possible domain transformations and fuzzy sets with corresponding linguistic terms, while the rule base contains linguistic control rules
- Decision logic: represents the processing unit, it computes output from the measured input, according to the knowledge base. It applies the fuzzy relation (the rules)
- Defuzzification interface: determines the crisp output value (and eventually maps it back to the appropriate domain).

A normal process of a fuzzy controller might be:

1. Input acquisition (not fuzzy output)
2. Fuzzification (fuzzy output)
3. Rule evaluation (fuzzy output)
4. Defuzzification (not fuzzy output)

Defuzzification: We need to convert fuzzy signals from the controller to a single, crisp, non-fuzzy value. After all the fuzzy rules are evaluated the outputs are combined into a single fuzzy set and this set is then defuzzified to produce a crisp output. Common defuzzification methods are:

- Max Criterion Method MCM: select an arbitrary value for which the maximum membership is reached; it can cause discontinuous behavior
- Mean of Maxima MOM: takes the average of the values with the maximum membership degree in the output fuzzy set; discontinuous behavior
- Center of Gravity COG: considers the center of the area under the membership function curve for the whole fuzzy domain; more complex but results in a more continuous behavior

2.2.2 Models

Mamdani Controller: The first model of fuzzy controller. It's based on a series of rules like "if X is M_n then Y is N_m", where M_n and N_m are intervals that represent linguistic terms, X is an input variable and Y is an output variable. All rules are in the form "*if-then*" form and should be interpreted as partial definition of a function. For multiple inputs a T-Norm is used.

For each rule, we calculate its firing strength, then the value of all rules is aggregated (union of the Cartesian product of all sets) to obtain a fuzzy set that represents a description of the final output, which has to be defuzzified, usually through the COG method.

This is pretty easy and intuitive but the number of rules can grow quickly, increasing complexity, and defuzzification is computationally expensive.

Takagi-Sugeno-Kang Controller: The idea is similar to the Mamdani controller, but rules are in the form

$$R : \text{if } x_1 \text{ is } \mu_1 \text{ and } x_n \text{ is } \mu_n, \text{ then } y = f(x_1, \ldots, x_n)$$

where $f(x_1, \ldots, x_n)$ is a crisp function, generally linear. The idea is that, if we can define a "good" function for controlling each region described by the corresponding linguistic term then we can obtain a crisp output, removing the need for defuzzification. The final output is obtained by combining the result of each function with the firing strength of the associated rule.

Similarity-based Reasoning: The concept of "similarity relations" is used to make reasoning; they are a type of fuzzy relations that describe how similar pairs of elements are (equivalence relations but fuzzy).

A function $E : X^2 \to [0, 1]$ is defined as similarity relationship respect to a T-norm if and only if it satisfies:

- Reflexivity: $E(x, x) = 1$

- Symmetry: $E(x, y) = E(y, x)$
- Transitivity: $\top(E(x, y), E(y, z)) = E(x, z)$

We interpret fuzzy sets as a "grades of similarity", starting known situations we can extract some "similarity classes" and thus rules for determining how the system behaves.

2.3 Fuzzy Data Analysis

The term "fuzzy data analysis" can refer to two different approaches:

- fuzzy analysis of crisp data (fuzzy clustering)
- analysis of data in the shape of a fuzzy set (random set)

2.3.1 Fuzzy Clustering

We want to divide a set of data (unsupervised learning) in a way that makes objects in the same cluster the most possible similar and objects in different cluster the most possible different. With "difference" being interpreted as some measure of distance between objects, less is more similar; we'll assume a Euclidean distance.

We've already seen the idea behind `crisp c-means clustering`. With crisp clusters, a cluster partition is optimal when the sum of distances between centers and elements is minimum. This way, when a point is in the middle of two clusters the assignment is arbitrary.
Fuzzy clustering solves this problem by introducing a continuous membership $\in [0, 1]$ to each cluster, each point can belong to more than one cluster. There are two types of fuzzy clustering: probabilistic and possibilistic. In the first case we have the assumptions that no cluster can be empty and that the sum of all membership for each point is 1, for the latter we keep only the first assumption.

Issues: How do we know how many clusters the algorithm is supposed to consider? We could try visualizing, but with a high number of dimensions it becomes unfeasible. We need some evaluation criteria:

- Clarity of separation: how distinct are the clusters
- Minimum volume between clusters: we want to minimize cluster overlap
- Maximum number of points focused near the centroid of the cluster: we want the uncertainty of the points in each center at a minimum

Some common metrics are:

- Partition Coefficient PC: it measures the clarity of the partition

- Average Partition Density APD: it measures the average density of a cluster
- Partition Entropy PE: it measures the entropy of the partition, giving an idea of how uncertain it is

Variants: Using Euclidean distances limits the types of clusters since they can only be spherical, variants have been proposed to relax this constraint. Gustafson-Kessel's algorithm uses Mahalanobis distance, which is computationally more expensive but allows for more flexibility in the clusters' dimensions (elliptical shape).

Another approach consists in allowing non-convex clusters, such as the shell clustering algorithms; useful for image recognition and analysis.

In the case of non-vectorial data (graphs, trees, ...) it might be useful a kernel-based clustering. It uses a map from the input space to a Hilbert space, in which data is transformed. The similarity between points is computed through a kernel function, i.e., the scalar product of the points mapped to the Hilbert space. It doesn't generate an explicit representation of the clusters but this allows for a more flexible representation.

When there are outliers it might be useful to add a noise cluster to manage them. It's not explicitly associated to any prototype, but each point has a probability of being part of the noise cluster, adapted during the optimization phase.

2.3.2 Random Sets

We want to extend the fuzzy techniques to work on fuzzy data. A random variable is a mapping from the probability space (cases) to a set $X : \Omega \to U$ (usually $\mathbb{R}$), a random set is the generalization of this concept, the mapping doesn't go to a single value but to a subset of values $\Gamma : \Omega \to 2^U$.

To analyze random sets, we use upper and lower probabilities, which represent the probability that the random set

- has an intersection with the upper probability
- is completely contained within the lower probability

We can generalize this concept, allowing the function Γ to map random sets to a fuzzy function, to better represent uncertainty.

2.4 Neuro-Fuzzy Systems

Fuzzy systems manage high level reasoning, but can't adapt to new environments like NN can. It makes sense to combine the learning capacity of NN with the high-level representation of fuzzy systems to have a better interpretation of the internal states of a NN during computation.

There are two modes of collaboration:

- cooperative model: the two systems work independently; the NN generates some parameters (offline) and then optimizes (online) for the fuzzy controller
- hybrid model: fuzzy sets and rules mapped inside a NN, the two data structures are integrated with no overhead; fuzzy sets can be modeled both as weight between neurons that as activation function

3 Evolutionary Algorithms

An optimization problem can be described by a triple $(\Omega, f, \prec)$, where Ω is the search space, f is an objective function in the form $f : \Omega \to \mathbb{R}$ and $\prec$ a comparison relationship. We want to find an element $x \in \Omega$ which optimizes the function in the search space.

3.1 Metaheuristics

Metaheuristics are computational techniques which aim to solve approximately CO problems in several iterations, usually applied to $\mathcal{NP}$ problems. They often operate by improving upon a set of candidate solutions.

Local Search Methods: Given $(\Omega, f, \prec)$, starting from a solution, we can search around it in a neighborhood searching for a local maximum. The search can proceed via gradient, if f is differentiable (gradient ascent/descent), otherwise evaluating, either exhaustively or heuristically, the points around the solution (steepest descent/hill climbing).

Simulated Annealing: We want a higher chance of moving from worse to better than the opposite. Starting from a solution, random variants are created and evaluated: they always replace the current reference solution if better, if worse they're accepted with a probability dependent on how much worse they are and a temperature parameter, which decreases with time.

Threshold Accepting: Similar to simulated annealing, but with a limit on quality degradation of the solution.

Gread Deluge Algorithm: Similar to before, but solutions are always accepted within an absolute bound.

Record-to-Record travel: Similar to Gread Deluge, but the threshold is relative to the best solution found instead of absolute.

Tabu Search: Considers history, forbids exploring previously considered solution candidates, usually for a set amount of time (tenure).

There's more, but I'm kinda bored and know them already. I don't think they're too useful so I'm just non gonna write them.

3.2 Actual Evolutionary Algorithms

The idea behind evolutionary computing is to mimic biological evolution, positive traits obtained by random mutations should be favored by natural selection. Starting from a population, we want to see how it evolves.

For an evolutionary algorithm we need:

- An encoding of the solutions, dependent on the problem
- A method to create an initial population (usually random or by a simple heuristic)
- An evaluation function to evaluate the fitness of each individual
- A selection criteria, in relation with the evaluation function, needed to select individuals that should go into successive generations
- A set of genetic operators to modify chromosomes (pieces of the encoding); the most common two are mutation and crossover
- Parameters for population size, mutation, …
- Termination criteria

EAs separate the search space Ω into genotype $\mathcal{G}$ for the encoding on which genetic operators act, and phenotype $\mathcal{F}$, on which the fitness function f is defined. Fitness is what determines the chance of survival of an individual, and the strength of fitness in such probability is called "selective pressure", the higher the pressure the more good individuals are favored, it determines the balance between intensification on already found solutions and exploration of the search space.

3.2.1 Encoding

The encoding of the solutions should allow visiting the whole search space, be similar for similar candidates and the fitness function should return similar values for similar candidates. These are highly dependent on the problem and might not always be easily satisfiable.
The encoding/decoding process should not be too computationally expensive.

3.2.2 Selection

At each generation, we want to create a new population starting from the previous one. There are various methods to select the individuals:

- Roulette-wheel selection: random selection but each individual has probability proportional to its fitness; it's simple but has no guarantees, and might soon lead to stagnation if candidates end up having similar fitness

- Rank-based selection: sort the individual by descending fitness, give probability to each of them based on their rank in the list and extract at random; it's more computationally complex due to the sorting but allows the decoupling of fitness value and selection probability
- Tournament selection: a number of subsets of the population is chosen and there's a "tournament" for each subset, the best individual from each subset is kept
- Elitist: keep only the best
- Crowding: Individuals are replaced by similar ones

3.2.3 Genetic Operators

To generate new elements starting from a population, we apply genetic operators on some or all the individuals. The operators can start from one, two or multiple parents.

Mutation: One-parent operator, it introduces changes on the genome of the candidate considered, useful for introducing biodiversity in the genetic pool. The specific mutation applied usually depends on the encoding of the solutions, but common ones are swaps of values, shifts and permutations.

Crossover: Two-parents operator, given two solutions it creates a new one:

- One-point crossover: determine a random position inside the encoded solution and swap the "tails"
- N-point crossover: determine N random position and swap all odd(/even) parts
- Uniform crossover: randomly extract a mask that determines whether or not to swap each gene in the encoding
- Shuffle crossover: permute the genes randomly, one-point crossover, then un-mix the genes

The crossover can also be a multiple-parents operator: diagonal crossover, given k parents, choose $k-1$ points and shift the pieces diagonally.

3.2.4 Introns

EAs tend to produce increasingly more complicated candidates, in part due to the presence of introns, pieces of DNA that carry no information and thus, for our case, that have no effect on the fitness. Mutation/recombination on introns has no effect.

Introns can easily grow in number and complexity since they cause no direct penalty on the fitness function, but it's better to prevent introns as much as possible given that they increase the time needed to find significant data and make solution harder to interpret.

Some solutions could be: modifying the fitness function to penalize introns, change the recombination phase in order to prevent them or prioritizing simple chromosomes, thus avoiding too many introns (sometimes at the expense of genetic diversity).

3.3 Swarm and Population based optimization

Swarm intelligence (SI) is a computational approach inspired by the collective behavior of social animals and insects, entailing simple individuals following simple rules with only local interactions that lead to a collective behavior that cannot be concluded based on the rules of a single individual. This concept can be used to develop multi-agent intelligent systems, able to perform a cooperative behavior. Single individuals can exchange information without a central control unit. A population of individuals moves in the search space.

3.3.1 Particle Swarm Optimization PSO

Inspired to the biological pattern of food research of birds and fishes. A swarm of candidates solutions aggregates information to guide the search. Each candidate updates its position based on personal and global memory. Informations are shared and the collective memory is used to find solutions.
Each individual is a possible candidate for being in the solution.

3.3.2 Ant Colony Optimization ACO

Inspired to the biological pattern of the ants searching for food. More promising paths are marked with pheromones, the individuals exchange information by editing the environment. Each candidate leaves a trail that modifies the search space for all others. Each individual is a possible candidate for being in the solution.

3.3.3 Population-based Incremental Learning PBIL

A population of individuals is generated randomly according to a probability distribution. Instead of explicitly conserving the memory of individuals, PBIL focuses on maintaining the statistics of the population.

As a recombination operator is used the uniform crossover. The selection process only chooses individuals that improve the statistics of the population. The mutation is a simple bit flip.

The distinctive feature is the learning rate, which tunes the possibility of movement of the individuals in the space. This changes in the time and is reduced with the number of iteration. This allows great mobility initially, eventually reducing when an optimum is found.

3.4 Theoretical foundations

3.4.1 Schema Theorem

To analyze why evolutionary algorithms work, we can look at the schema theorem. Since these algorithms work on bit strings, we can consider only binary chromosomes, more precisely, we consider schemata partly specified binary chromosomes. We then investigate how the number of chromosomes matching a schema evolve over several generations. The objective is to give a rough stochastic statement that describes how these algorithms explore the search space.

Definition 3.1. A *Schema* h is a character string of length L over the alphabet $\{0, 1, *\}$, $h \in \{0, 1, *\}^L$. The $*$ is a wildcard character.

Definition 3.2. *Matching*, a binary chromosome $c \in \{0, 1\}^L$ *matches* a schema $h \in \{0, 1, *\}^L$, written as $c \triangleleft h$ iff it coincides with h at all positions where h is 0 or 1, not taking into account $*$.

We want to measure the effect of selection on the fitness. The individuals matching a schema in the next generation will in some way proportional to the measure of fitness of the schema. As a measure, mean relative fitness is usually chosen.

To calculate the influence of mutations we need to measure the probability of preserving the schema even after the mutation.

Definition 3.3. The *Order of a schema* h is the number of zeros and ones in h, $\mathrm{ord}(h) = \#0 + \#1 = L - \#*$.

Considering one-point crossover, the probability that mutation preserves the schema h is $(1 - p_m)^{\mathrm{ord}(h)}$, where p_m is the "mutation probability".
So, the probability of preserving the schema is measured w.r.t. the defining length.

Definition 3.4. The *Defining length* of a schema h ($\mathrm{dl}(h)$) is the difference between the position of the last 0(1) and the first 0(1) in h.

Definition 3.5. The probability that a chromosome is cut in such a way that some of the fixed characters of a schema lie on one side of a cut and some on the others is $\mathrm{dl}(h)/L - 1$.

Definition 3.6. $N(h,t)$ is the expected value of the number of chromosomes that match the schema h during the t-th generation.

Definition 3.7. $N(h, t + \Delta t_s)$ is the expected value of the number of chromosomes that match the schema h during the t-th generation after selection.

Definition 3.8. $N(h, t + \Delta t_s + \Delta t_x)$ is the expected value of the number of chromosomes that match the schema h during the t-th generation after selection and crossover.

Definition 3.9. $N(h, t + \Delta t_s + \Delta t_x + \Delta t_m) = N(h, t + 1)$ is the expected value of the number of chromosomes that match the schema h during the t-th generation after selection, crossover and mutation. The various Δ will depend on the probabilities of the single operators defined.

Schema theorem: It says that the schemas with a fitness score above the average, a short defining length and a low order, reproduces in an approximately exponential way.

3.4.2 No free lunch theorem

The theorem says that there is no evolutionary algorithm that can be used for all problems, no other algorithm is superior to the average.
The "right" choice and encoding of the "correct" genetic operators depend on knowledge of the specific problem.

3.5 Genetic Programming GP

The idea behind GP is to automatically create programs able to solve problems, many of them can be seen as a search for a program; we need to connect input and output. GP is a family of evolutionary algorithms that allows the automatic creation of programs that solve problems.

Similarly to genetic algorithms, we need an encoding that allows representation and manipulation of the single programs. Usually, the parse tree represen-

tation is chosen, where internal nodes are operations and the leaves are variables/constants. We define the problem-specific sets: F of function symbols and operators and T of terminal symbols (constants and variables). GP can solve problems efficiently and effectively, if such sets are sufficient and complete, to ensure that an appropriate program can be found. Finding the smallest sets is usually $\mathcal{NP}$-hard.
For a boolean function the sets might be $F = \{\vee, \wedge, \neg\}$ and $T = \{x_0, \ldots, x_n, 0, 1\}$.

The algorithm is:

1. Generate an initialization population of random expressions
2. Evaluate the fitness of said expressions
3. Selection with one of the EA strategies
4. Genetic operators, usually only crossover

3.5.1 Initialization

We need to generate a random population. There's no fixed length for the chromosomes (the programs), so we need some parameters like maximum height and maximum number of nodes. There are various methods to initialize the parse trees:

- Full: leaves occur only at maximum depth, resulting in (full) symmetric trees
- Grow: at any level there's uniform probability of having either an internal node or leaf, resulting in asymmetric trees
- Ramp-half-and-half: combines both methods across different depths to increase diversity

3.5.2 Genetic Operators

The evolution of the population takes place via genetic operators. The 3 most important are mutation, crossover and clonal reproduction. A typical crossover operator is exchanging two sub-expressions, i.e., choosing an internal node and switching the sub-trees. A mutation could be exchanging a sub-expression (sub-tree) with one randomly generated.

3.6 Evolutionary Strategies ES

In an ES we try to optimize the entire evaluation process instead of the single individual. We need parameters that depend on the choice of genetic operators. We consider an optimization problem as a function $f : \mathbb{R}^n \to \mathbb{R}$ to minimize. Chromosomes (solutions encoding) are represented by a real-valued array. We

then use mutation to move the vector inside the search space, modifying it by adding a vector randomly taken from a normal distribution. Only the best (mutated) elements proceed in the subsequent generation (elitism). Possible selection approaches are:

- Plus strategy: selection works on both mutated and non-mutated elements (fitness can only improve, can get stuck in local optima)
- Comma strategy: generate a lot of mutated individuals and choose solely among them (increases diversity)

We can dynamically adapt the variance of the added vector in order to optimize convergence; small variance leads to small changes (exploitation), high variance leads to big changes (exploration).

The variance is "good" when 1/5 of the mutated elements are successful, i.e., have better fitness.
We can also save, along with each chromosome, its variance, in order to determine which ones have "bad" variance and thus will lead to worse descendants; we can then remove them from subsequent generations.

3.7 Multi-Criteria Optimization

There are cases in which optimization problems have multiple, possibly conflicting, objectives and constraints, each one represented by a fitness function $f_i : \Omega \to \mathbb{R}$. The easiest approach is to aggregate all functions into one, each one with its own weight reflecting its relevance. But the choice of weight might not be easy, how do we define relevance?

Arrow's impossibility theorem: It's impossible to have an aggregated function that maximizes all single functions.

3.7.1 Pareto-Optimal solutions

A solution to Arrow's impossibility theorem is to choose Pareto-optimal solutions, i.e., a solution which can't increase the value of any function without another one getting worse.
We don't need to aggregate all functions, removing the need for weights, but there are usually multiple Pareto-optimal solutions (Pareto frontier), and thus there's no single solution to the problem.

It's possible to use EAs to find solutions inside (or close to) the Pareto frontier. The simplest approach consists in using a weighted sum of the objective functions as fitness function.

Vector Evaluated Genetic Algorithm VEGA: Given k different criteria and the relative fitness functions, select k solutions such as each one optimizes a different function. This is a simple approach and computationally easy, but penalized solutions that satisfy each criterion well but not perfectly, the search is focused on marginal solutions.

Definition 3.10. An element *dominates* another if it has a better or equal (only better for *strict dominance*) value for each objective function.

It's possible to take an approach based on dominance. We rank individuals based on dominance relationships; we find all the non-dominated solutions and assign them the highest rank, remove them from the population and repeat with the second-highest rank; continue until the population is empty. After ranking all solutions, the selection is based on the ranks.

All individuals on the Pareto frontier are considered equally good and this can lead to genetic drift and convergence to a random extreme point. To avoid this we can use techniques to decide among elements of the same rank, e.g., power law sharing: if more individuals have similar fitness scores, new individuals similar to them will have lower fitness.

These are the basis for Non-Dominated Sorted Genetic Algorithms, which also include generating offspring by way of genetic operators.