# AI Summary

Massimo Perego

## Contents

**1 Neural Networks 2**

1.1 Multi Layer Perceptrons MLP . . . 2

1.1.1 Definition . . . 2

1.1.2 Function Approximation . . . 3

1.1.3 Regression . . . 3

1.1.4 Training . . . 4

1.1.5 Deep Learning . . . 5

1.1.6 Convolutional Neural Networks . . . 5

1.2 Radial Basis Function Networks RBFN . . . 6

1.2.1 Definition . . . 6

1.2.2 Function Approximation . . . 6

1.2.3 Training . . . 7

1.3 Learning Vector Quantization LVQ . . . 8

1.3.1 Definition . . . 8

1.3.2 LVQ Networks . . . 8

1.4 Self-Organizing Maps SOM . . . 9

1.4.1 Definition . . . 9

1.4.2 Topology Preserving Mapping . . . 9

1.5 Hopfield Networks HN . . . 10

1.5.1 Definition . . . 10

1.5.2 Associative Memory . . . 11

1.5.3 Solving Optimization Problems . . . 11

1.6 Boltzmann Machines BM . . . 12

1.6.1 Definition . . . 12

1.6.2 Training . . . 12

1.6.3 Restricted Boltzmann Machines RBM . . . 13

1.7 Recurrent Networks RNN . . . 14

# 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 $\sigma$, defines the shape and width of the function, i.e., the function is 1 at 0, decreases in some way until $\sigma$. 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 rectangular 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 $\eta$ 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.