Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Petr Š i n k o v e c

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Conditional Probability Models in Transportation

Guidelines:

Multi-agent Activity-based Transportation Models work in three steps: 1) generation of synthetic population of citizens, 2) scheduling activities (e.g., work, school, leisure, sleep) and 3) executing the activity schedules in a simulated transport network. This work focusses on the data-driven activity scheduler. The scheduler is composed of several subcomponents based on machine learning methods. In many cases these involve modelling of conditional probability (e.g., activity start time/duration probability distribution conditioned by agent´s sociodemography).

The objectives of this work are:

1. Study conditional probability modelling (focus on an approach described by Bishop, see references below).
2. Examine possibilities of non-gradient model learning. Consider methods based on genetic programming.
3. Implement the methods.
4. Experiment and evaluate approaches using real-world data.

Bibliography/Sources:

[1] Bishop, Christopher M.: Pattern recognition and machine learning. Springer, 2006.
[2] Drchal, J.: Base Algorithms for Hypercube-based Encoding of Artificial Neural Networks, dissertation thesis, 2013.
[3] Drchal, Jan, Michal Čertický, and Michal Jakob: Data Driven Validation Framework for Multi-agent Activity-based Models, arXiv preprint arXiv:1502.07601, 2015.

Bachelor Project Supervisor: Ing. Jan Drchal, Ph.D.

Valid until: the end of the summer semester of academic year 2016/2017

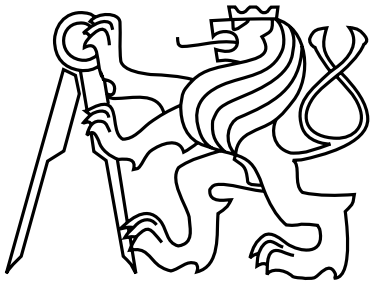
L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, December 14, 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics



Bachelor Work

Conditional Probability Models in Transportation

Petr Šinkovec

Supervisor: Ing. Jan Drchal, Ph.D.

Study Programme: Open Informatics, Bachelor

Field of Study: Computer and Information Science

May 15, 2016

Acknowledgement

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

Abstract

Conditional probability models in transportation are essential part of modeling real world multiagent systems. These models take part in simulation of a wide range of traffic infrastructure problems. This work discusses a comparison of several approaches to searching of these models by investigation of their achieved accuracy and computational requirements. Real world datasets and the particular form of conditional probability models were given.

Optimization task in this work is considered as an instance of a supervised machine learning task with the specific error function and given datasets with training and validation data. The problem was solved with Artificial Neural Networks (ANN) by the method called mixture density networks. Another class of approaches was based on the principles of Genetic Programming (GP) – especially the symbolic regression. The methods for conditional probability distribution modeling are well known for ANN approach unlike for the GP approach. Therefore the work is primarily focused on the GP and its advanced techniques that will try to surpass the ANN in its performance.

Abstrakt

Modely podmíněné pravděpodobnosti v dopravě jsou základní součástí popisu multiagentních systémů s aplikací v reálném světě. Tyto modely se uplatňují při simulacích širokého spektra záležitostí v dopravní infrastruktuře. Práce porovnává několik přístupů pro hledání těchto modelů, prostřednictvím vyšetřování jejich přesnosti a výpočetní náročnosti. Reálná data a konkrétní typ modelu podmíněné pravděpodobnosti byly zadány.

Optimalizační úloha, která je řešena v této práci se dá považovat za úlohu strojového učení s konkrétní chybovou funkcí a danou množinou trénovacích a validačních dat. Problém byl řešen umělými neuronovými sítěmi (ANN) ve formě sítí pro popis hustoty směsi pravděpodobnostních rozdělení (*mixture density networks*). Další třída metod pro řešení byla založena na principech genetického programování (GP) – především symbolické regrese. Metody pro modelování podmíněného rozdělení pravděpodobnosti jsou pro přístup pomocí ANN dobře známé na rozdíl od přístupu pomocí GP. Tudíž se práce zaměřuje především na GP a jeho pokročilé techniky, prostřednictvím kterých bude snaha předčit ve výkonu přístup pomocí ANN.

Contents

Chapter 1

Problem introduction

Examined problem was part of project Agentpolis [9] which is being solved on the Department of Computer Science of Czech Technical University in Prague. This project deals with modeling the city infrastructure, especially public transportation networks. This is useful in order to prototype and execute data-driven simulations of wide range of transportation matters, e.g. investigation of traffic density at any given time and location, commuting times between two areas at given time of the day, usage of given public transportation bus service, pollutant emissions by the transport system in some area and similar things. Useful simulations and their analysis can save a lot money and time, when designing new, or optimizing existing traffic infrastructure.

Data driven simulations in this project require synthesis of simulated population – agents. Agents could be perceived as virtual people described by a set of parameters. To ensure that these agents are described in accordance with the reality (or at least being its close approximation), there has to be a collection of real world data samples, which are used as a template for generating these agents. Mentioned real world data examples (called training examples) used in this work were collected from census data. Training examples were associated to training set T. The number of agents needed to be generated (approximately 10^6) was much greater than the number of training examples (approximately 10^2), therefore it had to be found a way, how to properly generate random samples from a training set in a way, that newly generated samples will come from the same probabilistic distribution, as training set samples.

The core of this work is creation of *generative model* – a model that specifies a probability distribution over an input data. Generative model has specified its underlying mathematical form (e.g. Gaussian distribution), which is parametrically adjusted (optimized) to have possibly the lowest error on training and validation data. There were several techniques developed for this optimization, e.g. maximum likelihood estimation, expectation maximization algorithm and finally mixture density networks and GP, that are described here in detail.

1.1 Formulation & Interpretation

We are interested in *conditional* probabilistic distribution D of several types of models, used in transportation networks modeling. The easiest way to explain what the distribution D might describe, is by following example:

We model exact time instances of when an employee returns from work to home. An employee has some characteristics described by sociodemographical data; this can be simply a vector or a single value, which means e.g. the number of employee's children. From census or another statistic exploration we obtained a training subset T of a whole population. This means, we have a representative list of pairs [number of children; time when employee arrives from work to home]. From training set T we construct a *conditional* probabilistic distribution D, which we use for generation of new examples. Supposing we want to have more data about when employees with 3 children arrive from work to home, we simply randomly choose multiple times from D and obtain generated (artificial) data of employees with 3 children. If the most number of employees from set T with 3 children had home arrival times at 5 p.m., then it is likely to obtain results from distribution D, that have home arrival time close to 5 p.m. as well.

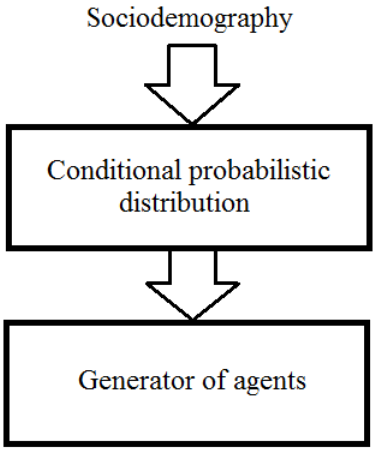


Figure 1.1: Sociodemography is a set of vectors where each vector represents a single human with his characteristic properties and characteristic target values. Target values depend on the type of the model (overview 1.1). The purpose is to be able to generate statistically credible samples of a particular type of human, because collecting real world examples is very expensive.

Distribution D is called *conditional*, because it changes its form depending on the input parameter (number of children – as in the previous example). Conditional distribution D is in its complexity a level above normal probabilistic distribution, because D is a function, that returns a probabilistic distribution. In other words, D can represent infinitely many distributions depending on input parameter. Input parameter is in practice a vector, which describes a single human ([-1, 1] means real number interval from -1 to 1).

Name	Range
age	[-1, 1]
owner of car	{-1, 1}
sex	{-1, 1}
education level	{-1, 0, 1}
driver's license	{-1, 1}
public transportation card owner	{-1, 1}
is in household	[-1, 1]
is a student	{-1, 1}

Table 1.1: Format and meaning of data vector representing a particular human. Note, that values are normalized in order to secure proper run of machine learning algorithms. Data normalization part was already realized and will not be described here.

Easier histogram sampling approach was not examined in this work, but it supposedly would not achieve as good results as approaches in this work. Very important thing to realize is that *conditional* distribution D takes into account more statistical details and brings more credibility then we would be obtaining by intuitive random sample generating from histogram of training data. That is the reason, why this work introduces such a complicated method of creating distribution D.

Here will be briefly described investigated probabilistic models. Each one of these models was aimed to 3 groups of people: *students, employees, people at home* (people, that are not employed, neither regularly visiting any educational institution). Every model had the same format of input data (table 1.1). People's daily activities are divided into *fixed* (sleep, work, school) and *flexible* activities (leisure, daily shopping, longer shopping)

- *time model*: Model input is vector 1.1. This model searches for probabilistic distribution of 4 time instants, when given human does these activities:

  I. leaves home (after sleep)

  II. arrives at school/work [not for people at home]

  III. leaves school/work [not for people at home]

  IV. arrives home (and does not go anywhere else)

- *flexible duration model*: Model input is vector 1.1. This model searches for probabilistic distribution of time duration of following activities per day:

  I. leisure activity

  II. daily shopping

  III. longer shopping (e.g. weekends shopping)

- *flexible count separated model*: Model input is vector 1.1, plus time duration of flexible activities and start times of flexible activities. This model searches for probabilistic distribution of the number of flexible activities. For example employee may have in average lower number of flexible activities than people at home.

Following example shows more closely *student's time model*:

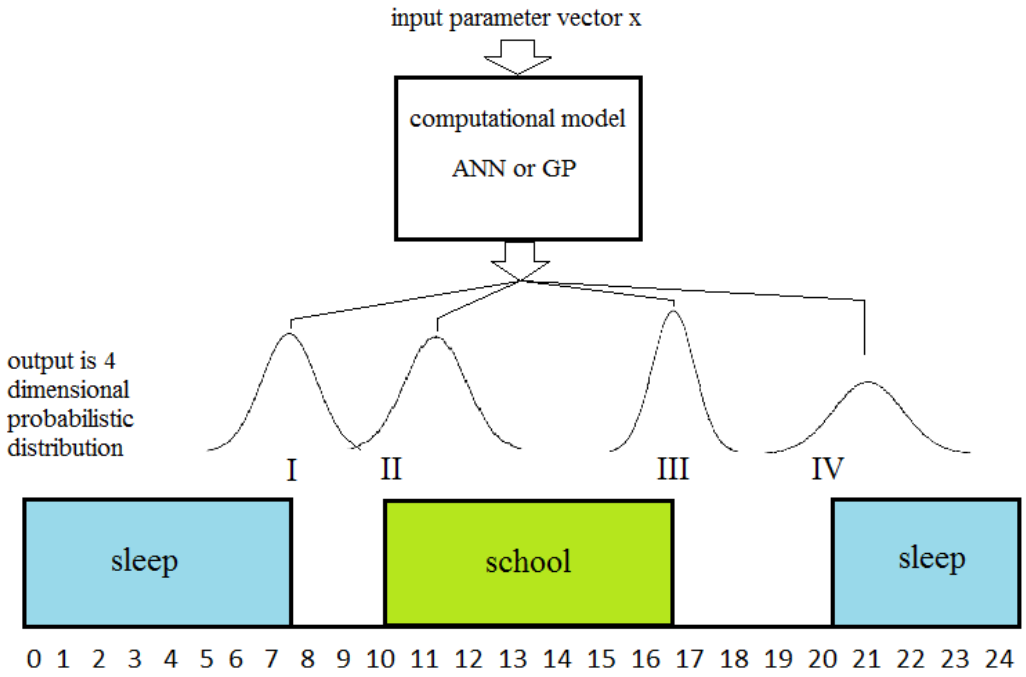


Figure 1.2: Horizontal axis shows day time hours. Time instants I., II., III., IV. represent, when one activity ends, or another starts. For example, time instant III. shows that main part of students, which are described by characteristic given by vector x, ends school at 4.30 p.m.

1.2 Objective specification

Computational model is represented by either ANN or GP, as a tool for calculation parameterized activity probability distribution. This distribution D was selected as a mixture of multivariate normal (Gaussian) distributions in form:

$$D = p(\mathbf{t}|\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\mathbf{x}) N(\mathbf{t}|\mu_k(\mathbf{x}), \sigma_k^2(\mathbf{x})) \tag{1.1}$$

where $N(\mathbf{x}|\mu, \Sigma)$ denotes:

$$\frac{1}{\sqrt{(2\pi)^m |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}} \Sigma^{-1} (\mathbf{x}-\mu)\right) \tag{1.2}$$

Due to quadratic increase of unknown parameters in general covariance matrix Σ with increasing target dimension, Σ was substituted by a scalar product of identity matrix and only one unknown parameter σ_k, i.e. $\Sigma = \sigma_k I$. This covariance simplification has not significant influence of resultant accuracy. It manages to change expression 1.2 to much more convenient form, where it does not have to be dealt with computing a determinant of Σ.

$$N(\mathbf{x}|\mu, \Sigma) = N(\mathbf{x}|\mu, \sigma) = \frac{1}{(2\pi)^{m/2}\sigma^m} \exp(-\frac{||\mathbf{x} - \mu||_2^2}{2\sigma^2}) \tag{1.3}$$

Mixture of Gaussian can be understood as a convex combination of its components, weighted by coefficients $\pi_k(\mathbf{x})$. K is a number of mixture components.

Let us have a training set T:

$$T = \{(\mathbf{x}_i, \mathbf{t}_i)\}, \text{ for } i = 1..M$$
$$\text{where } M = |T|, \mathbf{x}_i \in \mathbb{R}^n, \mathbf{t}_i \in \mathbb{R}^m$$

Letter n denotes input vector dimension (e.g. $n = 8$ see table 1.1). Letter m denotes dimension of output probability distribution (e.g. $m = 4$ see picture 1.2). The objective is to minimize something called error (or objective, or fitness) function. The error function has form:

$$E(\pi, \mu, \sigma, T) = -\sum_{i=1}^{M} ln \sum_{k=1}^{K} \pi_k(\mathbf{x}_i) N(\mathbf{t}_i|\mu_k(\mathbf{x}_i), \sigma_k^2(\mathbf{x}_i)) \tag{1.4}$$

Step by step derivation of 1.4 can be seen in [1] or [2]. It states, that:

$$\underset{\pi,\mu,\sigma}{\operatorname{argmin}} E(\pi, \mu, \sigma, T) = \underset{\pi,\mu,\sigma}{\operatorname{argmax}} \prod_{i=1}^{M} p(\mathbf{t}_i|\mathbf{x}_i) \tag{1.5}$$

and thus this optimization problem can be seen as maximum likelihood estimation problem, where unknown parameters π, μ, σ are not variables, but functions. By solving 1.5 we obtain optimal arguments π, μ, σ. These arguments can be considered not only as constants, but also as more complex mathematical expressions. Total number of these parameters is $L = (m + 2)K$. In general, every desired parameter is a function $f : \mathbb{R}^n \to \mathbb{R}$, where $\mathbb{R}^n$ denotes an input vector space (e.g. space of parameters describing every student). Since all desired parameters (means, variances, apriori probabilities) are functions, that have different values for different input arguments, these parameters construct a distribution, that changes its properties dependent on its input. That is why the probabilistic distribution represented by equation 1.1 is called *conditional*.

The quality measure – the fitness of some probabilistic distribution on training set tells, how well the distribution models data in training set. In this work it is stated, that the lower fitness ensures better performance (sometimes the bigger fitness the better performance – it is only a question of convention).

Chapter 2

Solution methods description

Introduced problem can be solved by ANN in a form of mixture density networks ([1] or [2]), which will be briefly explained. Though the work is primarily focused on investigating GP methods, which are described in greater detail. This choice makes contribution of this work more useful, as effects of ANN in mixture density networks are known much better than GP.

2.1 ANN vs GP as solution tool

Minimization of expression 1.4 is not a traditional parameter optimization task, because parameters π_k, μ_{ki}, σ_k, for $i = 1..m$, $k = 1..K$, (μ_k is m-dimensional mean vector with elements μ_{ki}) are not constants. The target probabilistic distribution D is *conditional* and therefore these parameters have to be dependent on input x. Input x affects what form will distribution D have through influencing its properties on apriori probabilities π, mean values μ and dispersions σ, as they were functions. The total number L of these parameters is dependent on the number of Gaussian components K in a way, that $L = (m + 2)K$ (m-dimensional mean, 1-dimensional dispersion, 1-dimensional apriori probability per component).

2.1.1 ANN solution

In mixture density networks the functions π_k, μ_{ki}, σ_k are interpreted as outputs of ANN. For this task a feedforward ANN was selected with input layer of dimension $n + 1$ (one neuron is bias unit) and output layer with dimension L. Specification of the hidden layers, that achieve the best results on given task is a subject of hyperparameter optimization.

Particular form of functions π_k, μ_{ki}, σ_k is following:

$$\alpha_k = \frac{exp(z_k^\alpha)}{\sum_{i=1}^{K} exp(z_i^\alpha)} \tag{2.1}$$

$$\sigma_k = exp(z_k^\sigma) \tag{2.2}$$

$$\mu_{ki} = z^{\mu}_{ki} \tag{2.3}$$

z^{α}_{k}, z^{σ}_{k}, z^{μ}_{ki} are labels of ANN outputs. Therefore these functions have specific form, thus it is clear, how these functions look like only by the type of selected ANN. The only necessary thing, is to numerically optimize ANN weights to minimize the objective function 1.4 by backpropagation gradient descent algorithm.

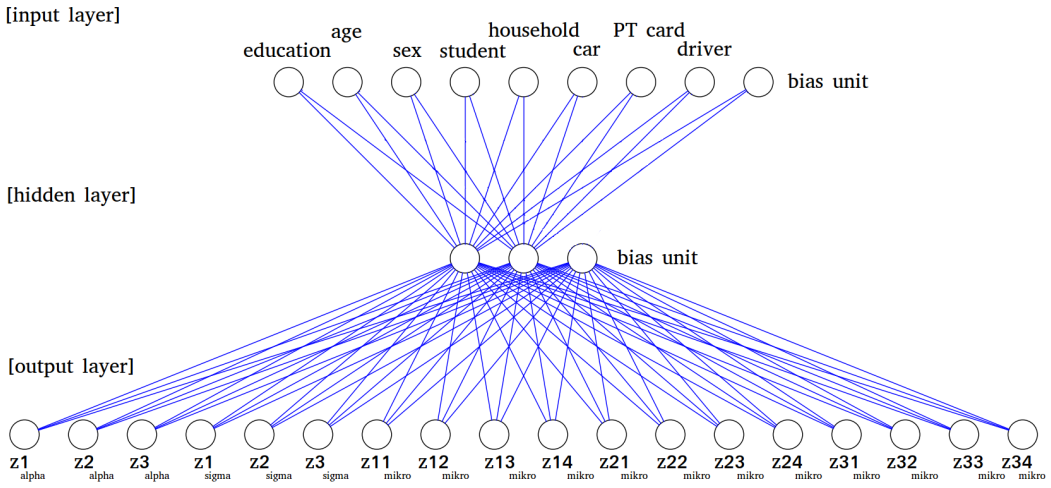


Figure 2.1: Example of ANN used to compute probabilistic distribution of students time models as a mixture of $K = 3$ Gaussian components. Input layer is consisted of 8 input neurons, corresponding to 8 input parameters (table 1.1), plus 1 bias neuron. Hidden layer consists of only 2 neurons, plus 1 bias neuron. Output layer consists of $L = 18$ neurons, corresponding to expressions z^{α}_{k}, z^{σ}_{k}, z^{μ}_{ki}, used for desired functions computation. This network has 72 weights (denoted by connecting lines).

2.1.2 GP short description

Genetic programming ([3], [4], [6]) can be seen as an optimization tool, which creates some sample of possible solutions of given problem, which is called population and then tests this population on how well it deals with target expression optimization task (this is called, that population is evaluated). One sample in generation is called an individual. After evaluation of population, each individual has its performance. Population is then sorted according to the individuals' performance. After that, some of the best individuals are copied for the next generation (next iteration of algorithm), and part of the best individuals is selected to mutation or crossover. Mutation creates new individual from old individual, by applying some change to old individual structure. Crossover takes two individuals and returns a new individual, that has part of structure from the first individual and the rest of the second individual. These new individuals are inserted into new generation and the

process is repeated, until some target level of performance is reached, or the maximum allowed number of generations is exceeded.

Individual I is forest of tree expressions $tree_i^I$. By applying mutation on individual is meant two kinds of mutation, which are applied on its every $tree_i^I$:

- Point mutation: random node n is selected from $tree_i^I$ and:
  1. If n is a function symbol, then n is substituted by another random function symbol from the primitive set (set of allowed functions).
  2. If n is a variable symbol, then n is substituted by another random variable symbol from the variable set.
  3. If n is a constant symbol, then the value of n is multiplied by a random value from the standard Cauchy distribution [4].
- New branch mutation: random node n is selected from tree the $tree_i^I$, new random expression (new random tree) n' with the depth d is generated and then the node n in $tree_i^I$ is replaced by n'. The node depth is the lowest number of edges, that lead to the top node. There is naturally a limitation of maximal tree depth, which cannot be exceeded. This value strongly influences the GP potential of solving complex tasks. Originally was the new random tree depth d in this project randomly selected from the set $\{1, 2, 3\}$. Later it was changed to deterministic $d = 1$ in order to reduce possible undesired effect of inappropriate new branch on the whole fitness. This option had the improving effect, which was experimentally verified for the problem of conditional distribution in studied models).

Crossover creates new individual by randomly selecting two individuals A and B from population. Then it is for every pair of the trees: $tree_i^A$, $tree_i^B$ for $i = 1..L$ realized (L is the number of trees in the individual):

1. create copy $new\_tree_i^A$ of $tree_i^A$
2. select random node n^A from $new\_tree_i^A$ and create copy n^B of randomly selected subtree from $tree_i^B$
3. replace subtree beginning from n^A by n^B and return $new\_tree_i^A$

GP is inspired by the natural selection mechanism, where only the most able group of individuals is allowed to reproduce. This is implemented by sorting individual according to their fitness. *Reproduction_ratio* is a part of population, which will be allowed to take part in creating new generation. *Elitelist_proportion_size* is another ratio, which controls the size of population, which will be copied to next generation without any change.

2.1.3 GP solution

GP realizes optimization by symbolic regression. Function in this problem can be seen as mathematical expression, which can be represented in its tree structure. For example

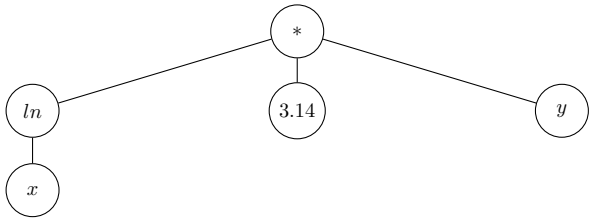
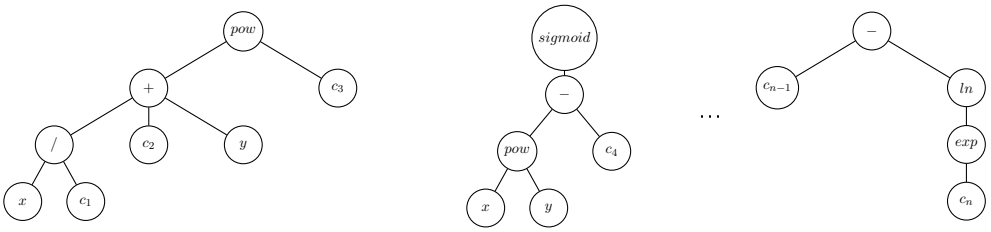


Figure 2.2: Example of a tree representing the expression $(* \ (ln \ x) \ 3.14 \ y)$.

function $f(x, y) = ln(x)3.14y$, or equivalently $f(x, y) = (* \ (ln \ x) \ 3.14 \ y)$ (in somewhat comfortable preorder notation), can be represented as tree in figure 2.2.

Example of an expression forest $[tree^{\mu}_{ki}, \ tree^{\sigma}_{k}, \ tree^{\alpha}_{k}]$ with the primitive set {+, -, /, power, ln, exp}, the variable set $\{x, y\}$ and the real valued constants $c_1 ... c_n$ is in the following picture:



In symbolic regression we are searching for an optimal form of expression (optimal tree structure), which solves a given problem. Effectivity of the current solution on a given problem measured as a value of error function. Let the function $f(x, y) = (* \ (ln \ x) \ 3.14 \ y)$ give performance A. If we for example change upper node from multiplication (*) to addition (+), we obtain performance B, which might be better than A. If B $>$ A, we accept new form of optimized function and continue in improving, or terminate optimization in case we are satisfied with the performance. Otherwise we continue by applying new structure and investigating, whether it helped. Applying new structure in expression involves several techniques, which are described in 2.1.2.

Objective remains the same: find L functions π_k, μ_{ki}, σ_k, which minimizes 1.4. Each of these L functions have their own tree expression, which is tuned to appropriate form. This is equivalent to ANNs values z^{α}_{k}, z^{σ}_{k}, z^{μ}_{ki}.

For better distinction in ANN and symbolic regression notation let be these values denoted as: $tree^{\alpha}_{k}$, $tree^{\sigma}_{k}$, $tree^{\mu}_{ki}$.
Following mapping is employed for obtaining desired functions: π_k, μ_{ki}, σ_k:

$$\alpha_k = \frac{sigmoid(tree^{\alpha}_{k})}{\sum_{i=1}^{K} sigmoid(tree^{\alpha}_{i})} \tag{2.4}$$

$$\sigma_k = sigmoid(tree^{\sigma}_{k}) \tag{2.5}$$

$$\mu_{ki} = sigmoid(tree^{\mu}_{ki}) \tag{2.6}$$

where $sigmoid(x) = \frac{1}{1+exp(-x)}$.

Note that every value σ_k and μ_{ki} is projected by sigmoid function to the interval $[0, 1]$. This is not a problem, because target vector attributes in training set were normalized to the interval $[0, 1]$ as well. For example if mean value of some time instance corresponds to 5 p.m., then it is represented by $\mu_{ki} = \frac{17}{24} = 0.708$.

As an example of possible look of $tree^i_{jk}$ expressions, see the example 3.5 of an individual.

2.2 Advanced GP methods

These methods improve performance either directly by influencing the selection mechanisms, or indirectly by increasing effectivity of mutational operations, or by combination of both.

2.2.1 Genetic Programming with Explicit Fitness Sharing (GPEFS)

GP, ANN and other optimization methods for minimizing very complex nonconvex functions have problem with stucking in local extrema. This can be partially solved by running the optimization algorithm repeatedly, with different initial values of parameters, or application of stochastic gradient descent (or combination of both). GPEFS facilitates search for better local minimum, by analyzing broader search space. It is done by splitting population to several species, which compete only amongst its members. In classic GP individuals compete with the whole population – not only its parts, therefore they are more likely to end up in worse local extrema. Thus the GPEFS can be roughly seen as n simultaneously running GP algorithms, which have assured, that every instance of these n GP runs, searches in different direction.

Target number of species	positive integer
Elitelist proportion size	[0, 1]
Initial speciation threshold	[0, infinity)
Species reproduction ratio	[0, 1]
C	[0, infinity)
K	[0, infinity)
Not matching node exit	{True, False}
Descent null trees	{True, False}

Table 2.1: Parameters introduced by GPEFS

GPEFS workflow is described by niching algorithm in very understandable form in [4]. Introduced parameters (table 2.1) are used primarily to sustain number of species throughout evolution process. Parameters C, K, $Not\_matching\_node\_exit$, $Descent\_null\_trees$ are used in determining to which species an individual will be assign. This is done by measuring a similarity of two given individuals. Similarity of two individuals is measured by sum of distances of their corresponding trees. Thus a proper metrics have to be defined for measuring structural difference of two tree expressions. Properties of this metrics is controlled by C, K, $Not\_matching\_node\_exit$, $Descent\_null\_trees$. Further details about these metrics and reasons of choosing given configuration 3.2.2 is in [4].

2.2.2 Tree optimization via error backpropagation (BPG)

It is the way, how to numerically optimize values of constants in expression forest. This ensures faster convergence to local optimum and whole system has therefore bigger potential for achieving better performance. Base algorithm for backpropagation gradient descent is a simple gradient descent, which is the way of finding local minimum of differentiable real function, by iteratively altering value of its arg min point by a value of function derivative at that point, multiplied by learning rate (in vectorized form):

$$\mathbf{x}_{t+1} := \mathbf{x}_t - \alpha \Delta \mathbf{x}_t \tag{2.7}$$

Starting point $\mathbf{x}_0$ and learning rate α are given. Therefore it is only necessary to compute gradient Δ.

Every differentiable real function has explicit rules for calculation its derivation. Sometimes it could be very difficult to come up with analytical form of gradient. This is typical problem in the process of learning ANN, and for the same reason it is a problem at symbolic regression as well. The problem is that ANN or symbolic forest is in fact composition of many functions. It is known, that derivation of function composition can be realized by the chain rule. This way of computing derivation of function composition is, however, very inefficient (from computing or programmer point of view). Thus it was developed new method to computing derivation – the backpropagation algorithm. It computes derivation by recursive rules:

$$\delta_i^{|L|} = 1 \tag{2.8}$$

$$\delta_i^L = \sum_j \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial z_i^L} \tag{2.9}$$

These rules can be applied on every analytical function. It is only necessary to represent them in the following form:

$$\mathbf{z}^{|L|}(\mathbf{z}^{|L|-1}(...(\mathbf{z}^2(\mathbf{z}^1))...)) \tag{2.10}$$

(very useful is to imagine this representation as a tree expression – see pictures in chapter 2.1.3). Formally the term $\mathbf{z}^i$ is a vector functions $z_j^i : \mathbb{R}^n \to \mathbb{R}$. Number $|L|$ is the depth of the function tree expression. Term δ_j^i means backward error message (derivation) from function $\mathbf{z}^i$ by input $\mathbf{z}^{i-1}$. There is a dependency of the derivative in lower layer l on the derivative of the upper layer $l+1$; this can be expressed as:

$$\delta^1(\delta^2(...(\delta^{|L|-1}(\delta^{|L|}))...)) \tag{2.11}$$

We are interested in δ^1. The backpropagation workflow is following: we start with evaluation of the expression (2.10). Then we use rules (2.8) and (2.9) in order to evaluate expression (2.11) (it is also said, that the error is backpropagated), and we take δ^1 as result.

Desired gradient Δ in 2.7 is the result δ^1. Important thing to realize is, that it is necessary only to offer rules, for computing derivatives of functions in primitive set and be able to evaluate them. Derivative computation is much faster than using chain rule due to clever backpropagation evaluation.

Full comprehension of this algorithm may be achieved from practical experience with exemplary toy expressions. Very useful explanation of equations (2.8) and (2.9) is in [11] – that source was sufficient for custom simple implementation used in this work.

2.2.3 Automatically Defined Functions (ADF)

This extension of GP algorithms enables more effective problem solving, by repetitive utilization of whole blocks of structure (e.g. tree expressions). In other words, it automatically introduces reusable subroutines – functions. This may have positive effect on some problems, which are decomposable into subproblems, or on problems, where some task is repeated multiple times. Very extensive explanation of this problematics is in [6].

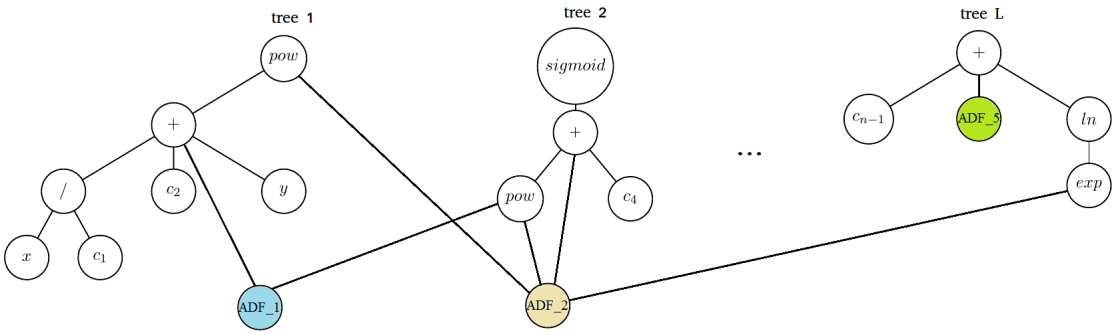


Figure 2.3: ADF can be considered as terminals, that are reusable on multiple places.

There are many ways, how to implement basic idea of ADF. One particular approach was implemented, which introduces these parameters:

Population shared ADF	{True, False}
Recursive ADF	{True, False}
Number of ADF	positive integer
ADF mutation frequency per epoch	[0, 1]
Generate terminal as ADF	[0, 1]
Maximal ADF tree depth	positive integer

In case of symbolic regression ADF are symbolic trees, that in case of *Population shared ADF = True* can by used throughout the whole population, i.e. that the same expressions are used multiple times and in multiple individuals as well. $Population\_shared\_ADF = False$ ensures, that ADF set will not exist for whole population, but every individual will have its own ADF set. Total number of ADF is limited by $Number\_of\_ADF$ and this number remains the same. $Maximal\_ADF\_tree\_depth$ controls complexity of these functions. If $Recursive\_ADF = True$, than one function can be part of another. There is a necessity of avoiding self reference (it would cause infinite loops during evaluation) and thus it has to be ensured, that one function from ADF set has not reference on itself. This illustrates following picture:

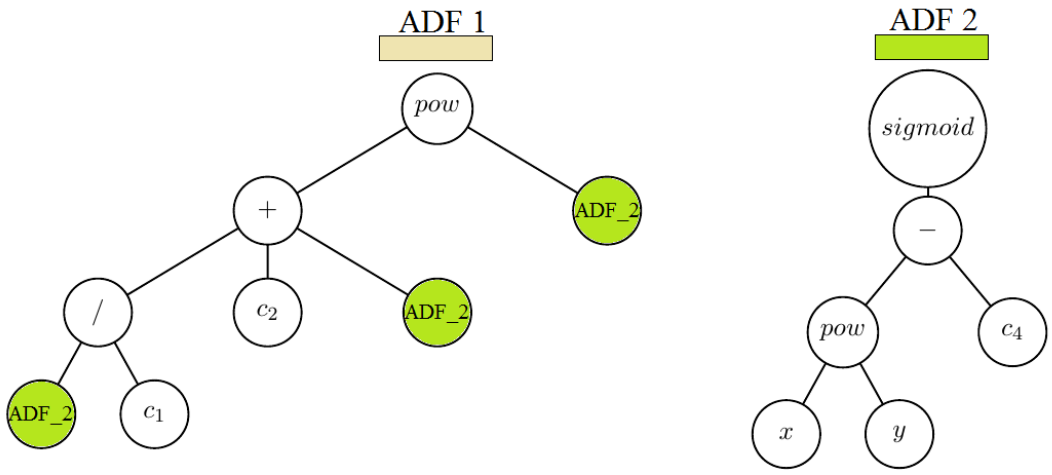


Figure 2.4: Recursive ADF composition ensures bigger potential for solving more complex tasks. Every ADF A has its number $i = adf\_order(A) \in \{1..Number\_of\_ADF\}$. When ADF B participates on the structure of ADF A, then it has to be ensured, that $adf\_order(A) < adf\_order(B)$. This leads to avoiding of an inadmissible self-reference, which would cause an infinite loop during evaluation.

Another parameters relate to mechanisms of ADF usage. ADF can be considered as terminal. Parameter $Generate\_terminal\_as\_ADF$ controls the probability of terminal selection to be one of the ADF. Complementary probability is probability of applying constant or variable as terminal. ADF have also a chance of evolving, this is done by their mutation in epoch, which is realized by probability *ADF mutation frequency per epoch*.

ADF are not universal tool for every problem and there are no explicit rules, how to find out, whether they will be useful. ADF were succesful, for example, in problems from molecular biology, biochemistry and some robotics tasks.

Chapter 3

Experiments and Results

First part of the experiments was about searching the best configuration of ANN for every studied model. This included hidden layers and number of Gaussian components K for optimization of expression 1.4. GP had problems with increasing K in the sense, that higher values of K caused linear, still very huge increase of computing demands. While the ANN was able to deal with $K = 20$, the GP in general was unable to compute applicable results even for $K = 5$. Good compromise in the choice of the value K was to set $K = 3$. This value made computation much faster and moreover it achieved very similar results as $K = 4$ or $K = 5$ on training and validation error.

This limitation of the number of Gaussian components brings noticeable dimension reduction as the number of unknown components is naturally lower. It can be said, that the choice of $K = 3$ causes remarkable reduction of the whole potential for model performance. This is because the best ANN performance tends to be higher with increasing K. This turned out to be true when K was 10, 15, 20.

One important target of this work is to compare the effectivity of GP against the effectivity of ANN. Therefore the conditions for their comparison have to be as similar as possible. With regard to the number of investigated models, it was chosen one referential model, where the comparison will be analyzed. Student's time model which describes probabilistic distribution of 4 time instants in a student's day (see 1.2) was chosen as a referential model. Since the number of these time instants is $m = 4$, it is the most complicated investigated model. Flexible duration and flexible count separated models had target dimension $m < 4$. Moreover it states, that when one method will be better in dealing with some complicated model, then is very likely to do better in simpler models.

Resultant model (GP model with the best performance at minimizing error function) is composition of previously described advanced methods GP/GPEFS, BPG optimization and ADF. In order to obtain better idea of improving effect of single introduced methods, they were gradually composed together, so the increase of performance could be observed (figure 3.3).

3.1 Hyperparameter optimization

Finding optimal parameter configuration is an instance of hyperparameter optimization problem and was solved with simple grid search algorithm on every value of Cartesian product of allowable and reasonable elected values for every controlling parameter.

There is a natural necessity of being able to compare two parameter configuration based on their performance. Performance of GP (resultant fitness) is random variable, therefore it is necessary to employ statistical methods for comparing hypothesis about one is greater than another. For this task was chosen an analysis of boxplot graphs 3.1. Following example shows a typical situation, which occurred on every better/worse comparison between two configurations, which differed in the value of exactly one parameter.

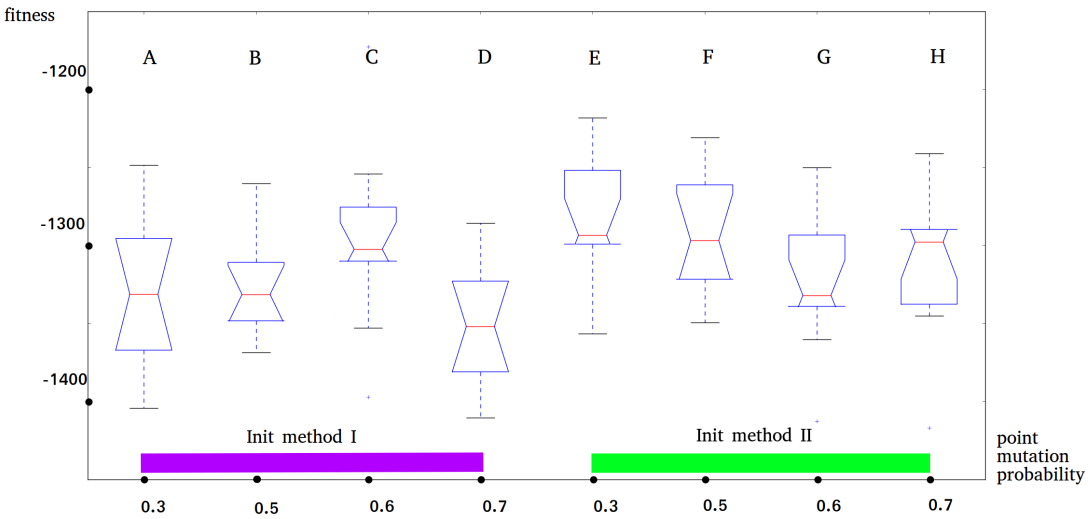


Figure 3.1: Example of boxplot diagram for effect analysis of different values of point mutation probability and initialization method (other parameters are the same for each column). Lower value is better. Based on a shape of these boxplot columns notches, it could e.g. not be said, that configuration D has a significant difference of median against configuration B, because their notches overlap (width of the notches is proportional to the interquartile range of the sample, for details see [7]). On the other hand preference of configuration D against configuration F is justified by lower achieved fitness and by non-overlapping boxplot notches and thus this preference is statistically significant.

3.1.1 ANN optimization

For every investigated model optimal ANN was identified by grid search algorithm on the set $\mathbb{K} = [2, 3, 5, 10, 15, 20]$ of possible number of components K. Next set was $\mathbb{L}$, which elements were lists of numbers, that represent number of neurons in layer; e.g. (4, 4) means, that number of hidden layer is 2, and every hidden layer has 4 neurons. Set $\mathbb{L}$

was specified intuitively as: $\mathbb{L} = [(2), (3), (4), (5,), (7,), (7,7), (10,10), (6,), (5,5), (14,14), (5,5,5), (10,), (8,9), (12,12), (8,10,8), (14,), (6,6), (8,8), (12,12,12), (12,), (9,10), (11,11), (10,12,10), (6,3,6), (12,10,12), (10,6,9), (6,3,3,6), (4,2,2,4)]$

Every randomly initialized optimization run was repeated 10 times, every repetition was cross-validated on 10 folds of validation set. $|\mathbb{L}| = 28, |\mathbb{K}| = 6$. Maximum number of training epochs was 1000 (in case that the validation error decreased before 1000 iteration, the optimization was terminated). Therefore the search for one particular optimal model took exactly $28 \cdot 6 \cdot 10 \cdot 10 = 16800$ ANN optimization procedures, which was computationally time-consuming.

3.1.2 GP vs GPEFS

Comparing these two approaches by measuring the performance upon the same parametric configuration has not much sense. It has showed, that in the case, when GP and GPEFS have the same population size, and target number of species in GPEFS was greater than 1, then GP was significantly better. That is naturally because GPEFS splits whole given population into species, which minimizes error function in different directions. Splitting population to species causes reduction of competitive potential per species, compared with competitive potential in GP population. Therefore it had to be increased the population size, while running GPEFS. Idea of considering GPEFS as GP with $Target\_number\_of\_species = 1$ is not correct, because real number of species fluctuates during evolution process. *Target number of species* is only number of species, which system tries to achieve.

There has been also seen, that GPEFS slightly changes details of mutation process, more precisely – some optimal configurations for GP does not have to be necessarily optimal for GPEFS. This has been observed between 3.2.1 and 3.2.2. Similar observation was made on picture 3.2

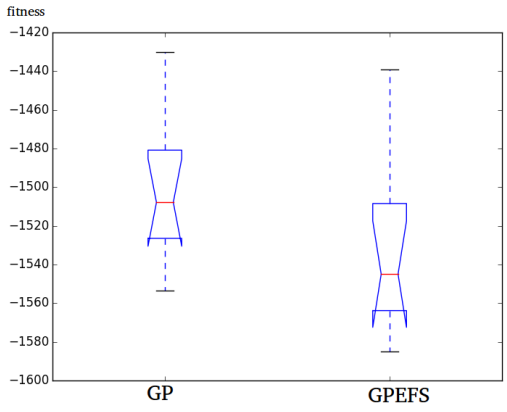


Figure 3.2: Boxplot of best reached fitnesses (lower value is better) made on 10 trials of GP and GPEFS. Both models had the same population size and other parameters same. $Target\_number\_of\_species = 1$, therefore was GPEFS closer to simple GP. As showed, GPEFS did slightly better, but not significantly.

3.2 Optimal models

Every parameter search trial was repeated 10 times. Number of evolution epochs per trial was set to 500; above this number of epochs the system was not likely to change its best fitness. Population size for model search was set to 200 individuals for GPEFS with 3 target number of species and 100 individuals for GP. There can be some objection about choosing these values of population size. One empiric recommendation stated, that number of individuals per species should be circa 20. Thus at 200 population size this number should be 10, but this configuration had very poor results. On the other hand setting number of species to 1 would not bring much difference from standard GP run. As good compromise with very good final results, was to set number of species 3 (which was considerably different from empirical rule). Due to extensive computational requirements, was even this choice of population size and epochs quite high. Cross validation was omitted for this reason as well. Still the single model evaluation took even multiple days.

Hyperparameter sets for (GP/GPEFS) optimization [1200 runs in total (1·5·4·3 = 60, 60·10 (per 10 repeats), 600·2 (per GP/GPEFS))]:

```
initial depth = [2]
maximal depth = [2, 3, 4, 5, 6]
point mutation probability = [0.3, 0.5, 0.6, 0.7]
generate terminal as variable = [0.3, 0.5, 0.7]
```

Hyperparameter sets for (BPG GP/GPEFS) optimization [1080 runs in total]:

```
initial depth = [2]
maximal depth = [5]
point mutation probability = [0.3, 0.5, 0.7]
generate terminal as variable = [0.3, 0.5, 0.7]
probability of BPG optimization per epoch = [0.3, 0.5, 0.7]
batch size = [10, 20]
```

Hyperparameter sets for (ADF BPG GP/GPEFS) optimization [960 runs in total]:

```
initial depth = [2]
maximal depth = [5]
point mutation probability = [0.5]
generate terminal as variable = [0.2, 0.35]
generate terminal as constant = [0.2, 0.35]
probability of BPG optimization per epoch = [0.5]
batch size = [10]
population shared adf = [True, False]
adf mutation frequency per epoch = [0.2, 0.5, 0.8]
number of adf = [3, 5]
```

One evaluation of ANN parameter configuration took 50 seconds on average per trial (it was made 16800 evaluations). One evaluation of GP parameter run took 90 minutes in average per trial (it was made 3240 evaluations). Total CPU time on record in Metacentrum was at the end of experimenting 354 CPU days. On home computers was calculated approximately 40 CPU days.

Primitive set was chosen based on positive reference of [4] as:

Function	Expression
Addition	$\sum_{i=1}^{n} x_i$
Multiplication	$\prod_{i=1}^{n} x_i$
Arctangent of sum	$\arctan(\sum_{i=1}^{n} x_i)$
Gaussian kernel of sum	$e^{-(\sum_{i=1}^{n} x_i)^2}$
Sine of sum	$\sin(\sum_{i=1}^{n} x_i)$

Table 3.1: Primitive set used in every GP model.

Arity	Probability
2	0.9
3	0.08
4	0.02

Table 3.2: Functions arities and their probabilities.

Initialization of forest was realized with grow method [3] with maximum depth equal to 2. Another initialization alternative for symbolic regression was to set all trees to zero expression – e.g. (+ 0 0); this method very quickly found relatively good minimum, however then it almost stopped to improve at all. The point mutation of node means, how will be some expression in node changed. It depends on type of expression the node contains:

- For function node it randomly selects function from primitive set 3.1 and substitutes existing type of function. Used primitive set was useful as well because there was not necessary to check arity of function.

- Value of constant was altered by multiplying a random value from standard Cauchy distribution.

- Variable was swapped with another variable.

Branch mutation was limited to random selection of node from tree expression and replace it by randomly generated function with terminals as its arguments. It means, that newly provided structure had depth at most 1.

3.2.1 GP model

As a best configuration was evaluated these settings:

Forest initial method	grow method with maximal depth = 2
Point mutation probability	0.7
Generate terminal as constant	0.5
Generate terminal as variable	0.5
Maximal tree depth	5
Population size	100

Elitelist proportion size	0.2
Individuals reproduction ratio	0.4

Table 3.3: Table shows population diversification control parameters. Elitelist proportion size = 0.2 means, that 20% of best individuals will be inserted to new generation without change. The remaining 80% of population will be generated from best 40% of individuals in previous generation (Individuals reproduction ratio = 0.4).

3.2.2 GPEFS model

As a best configuration was evaluated these settings:

Forest initial method	grow method with maximal depth = 2
Point mutation probability	0.6
Generate terminal as constant	0.5
Generate terminal as variable	0.5
Maximal tree depth	5
Population size	200

Recommended settings for tree distance algorithm 2.2.1 was set to:

Target number of species	3
Elitelist proportion size	0.2
Initial speciation threshold	1.0
Species reproduction ratio	0.4
C	0
K	1
Not matching node exit	True
Descent null trees	True

3.2.3 BPG GPEFS model

Configuration of gradient descent during optimization procedure 2.2.2 was following:

Maximal iteration per epoch	20
Batch size	10
Stochastic	True
Initial learning rate	0.05
Probability of BPG optimization per epoch	0.5

Parameters of niching algorithm were set equally as in previous models 3.2.2. Other parameters providing best performance were:

Forest initial method	grow method with maximal depth = 2
Point mutation probability	0.5
Generate terminal as constant	0.7
Generate terminal as variable	0.3
Maximal tree depth	5

Note that probability of generation terminal as constant was found to be 0.7. It quite corresponds to intuitive notion, that more constants in expression, which are optimized by constant tuning method, the better fitness will be reached. On the other hand, if the expression was consisted only of variables, the BPG optimization algorithm would have zero effect.

3.2.4 BPG GP model

Gradient descent configuration was the same as in GPEFS version.

Other parameters providing best performance were:

Forest initial method	grow method with maximal depth = 2
Point mutation probability	0.5
Generate terminal as constant	0.7
Generate terminal as variable	0.3
Maximal tree depth	5

3.2.5 ADF BPG GPEFS model

This model introduced additional parameters, whose values had to be optimized during hyperparameter optimization process. Their values are here:

Population shared ADF	True
Recursive ADF	True
Number of ADF	5
Maximal ADF tree depth	4
ADF mutation frequency per epoch	0.5
Generate terminal as ADF	0.3
Generate terminal as constant	0.35
Generate terminal as variable	0.35
Maximal tree depth	5

Table 3.4: Population shared ADF = True means, that the same ADF were used over the whole population (they were not defined for each individual separately). Model was mentioned to be as simple as possible, thus the number of ADF was static, equal to 5. One ADF could be part of another, but only in case it would not cause an infinite loop during evaluation. ADF mutation frequency per epoch controlled the probability of ADF mutation per one epoch. Using ADF in expression could be considered as it was simple terminal, thus its using has to be determined by some probability; Generate terminal as ADF = 0.3.

Examined option of using individual specific ADF (Population shared ADF = False), caused frequent fluctuations of best fitness during evolution. System could not stabilize itself and therefore it could not evolve any persistent structure, which would survive for multiple epochs and allow to reach suitable local minimum. On the other hand setting several ADF, that can be accessed by every individual in population, brought surprisingly high stability of best achieved fitness – even in comparison with the GP variant, that did not use ADF in any form (see lower variance at ADF best results in figure 3.3).

3.2.6 ADF BPG GP model

This GP variant of previous ADF BPG model had the same best configuration, for mutation control, backpropagation algorithm and GP algorithm.

3.3 Optimal models comparison

To have a better idea about the performance of evolutionary methods based on the number of involved constants (i.e. the dimensionality of optimization problem), the referential ANN was trained with approximately the same number of constants, that were involved in evolutionary models. For the number of components $K = 3$ and the input dimension $n = 8$ the investigated ANN had following layers:

- input layer: 8 neurons + 1 bias
- hidden layer: 2 neurons + 1 bias
- output layer: 18 neurons

These settings (picture 2.1) contains exactly 72 constants to tune – the number which was close to the number of constants involved in evolutionary expression forests. The network was feedforward with arctangent activation function and was trained by standard backpropagation algorithm.

3.3.1 Achieved fitness

	ANN	GP	BPG GP	ADF BPG GP
training error	-1973.98	-1514.65	-1626.79	-1424.12
validation error	-477.61	-360.82	-396.12	-332.39
number of constants	72	46	32	157

	GPEFS	BPG GPEFS	ADF BPG GPEFS
training error	-1546.63	-1740.84	-1725.44
validation error	-363.09	-423.55	-420.00
number of constants	66	42	87

Table 3.5: Best achieved fitness for best found models on the same training and validation dataset. Validation dataset had 168 samples. Training dataset had 301 samples.

Inspection of value of error function is one way of deciding how well did the system. There are several indirect methods how to examine the quality of discovered system. They are based on checking, whether some random sample belongs to some reference probability distribution. Reference probability distribution is given as empirical distribution by training data. Random sample can be generated by found models.

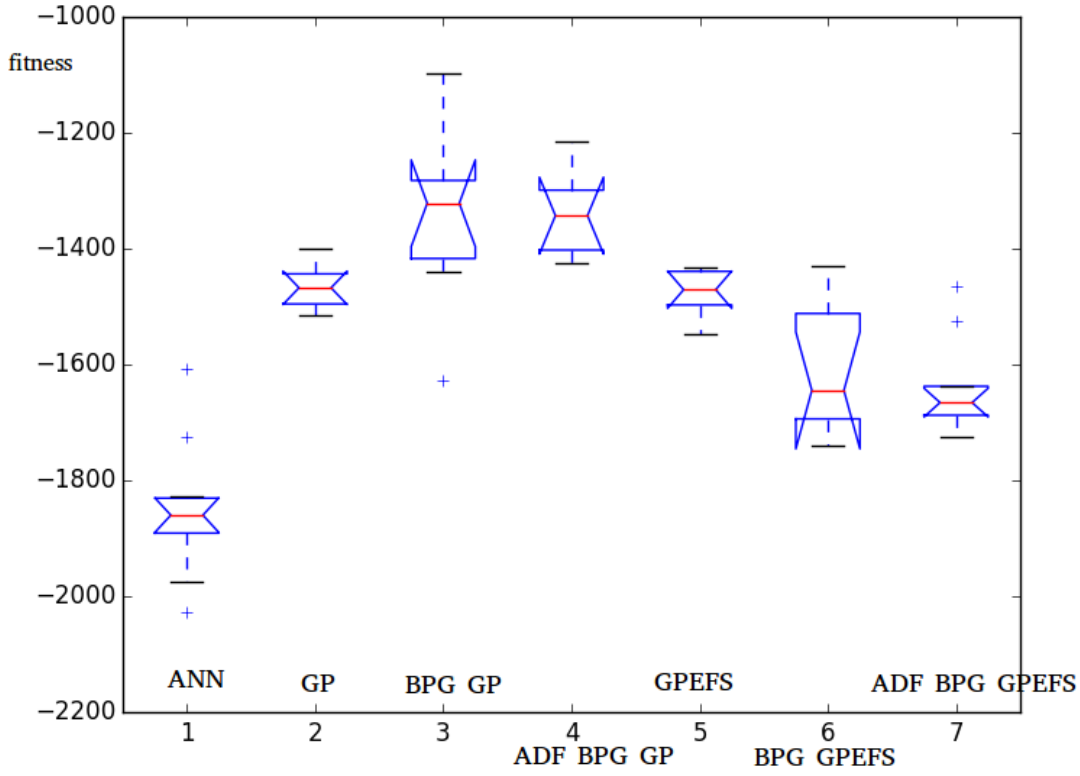


Figure 3.3: Boxplot diagram shows the best achieved fitness on training set, after 10 trials for each examined model (lower is better). GPEFS proves to be significantly better in combination with backpropagation optimization. ADF BPG best results have lower variance in comparison to BPG, but not achieving better performance. ANN was undoubtedly the best (even in validation error).

3.3.2 Kolmogorov-Smirnov (KS) test

The Kolmogorov-Smirnov deviation statistic is used to test, whether some sample comes from given probability distribution. It quantifies a distance between the empirical distribution function of the sample (validation or training data) and the cumulative distribution function of the reference distribution (that is desired distribution 1.1). With regard to very small *p-values* during the test, it was investigated rather the KS-statistic of the test. In fact, the KS-statistic can be detected from p-value and vice versa.

Useful details about this test with other methods for activity-based models validation are in [10].

training data	ANN	GP	BPG GP	ADF BPG GP
dimension I	**0.255**	0.267	0.333	0.458
dimension II	**0.178**	0.297	0.214	0.315
dimension III	**0.107**	**0.172**	0.196	0.375
dimension IV	**0.101**	0.309	0.303	**0.261**
validation data				
dimension I	**0.189**	0.302	0.388	0.558
dimension II	**0.205**	0.461	0.342	0.325
dimension III	**0.069**	**0.146**	0.186	0.375
dimension IV	**0.053**	0.222	0.169	**0.149**

training data	GPEFS	BPG GPEFS	ADF BPG GPEFS
dimension I	**0.130**	0.208	0.250
dimension II	**0.178**	0.488	0.285
dimension III	0.380	0.291	0.214
dimension IV	0.398	0.380	0.375
validation data	GPEFS	BPG GPEFS	ADF BPG GPEFS
dimension I	**0.192**	0.335	0.305
dimension II	**0.295**	0.594	0.395
dimension III	0.292	0.169	0.275
dimension IV	0.255	0.325	0.292

Table 3.6: Statistics of KS-test for every model on training and validation data (lower is better). Bold values are the best results (ANN is treated separately). It can be seen, that if one component has the lowest training error, then it has the lowest validation error as well.

3.3.3 χ^2 test

This is another alternative for measuring quality of how well we can simulate some unknown probabilistic distribution. Mathematical principles of this test are similar to the KS test, in a sense, that this test uses only a different form of expression for calculation value of statistics. Necessary part of being able to run this test is to discretize data into histogram bins. Empirical rule recommends to ensure that the number of elements in the bin with the lowest number of samples, must not be less than 5. This was actually the way, how was discretization of data samples for χ^2 test in this project realized.

Important thing to notice is, that KS and χ^2 tests in this implementation do not necessarily preserve partial ordering of their results. This means, that most of the time, KS test states: $A \succ B$ and also χ^2 test states that: $A \succ B$, but there were seen examples, when KS test stated: $A \succ B$, but χ^2 test stated: $A \prec B$. Therefore it should be considered personally, whether to choose KS or χ^2 test, for emphasizing its results.

training data	ANN	GP	BPG GP	ADF BPG GP
dimension I	**52.78**	24.05	21.43	273.48
dimension II	**7.78**	6.56	9.72	43.28
dimension III	**1.34**	20.66	**20.39**	81.10
dimension IV	**17.47**	92.57	**74.05**	88.72
validation data				
dimension I	**22.31**	177.73	52.93	826.30
dimension II	**76.97**	552.41	295.23	**271.87**
dimension III	**5.70**	**40.33**	76.46	482.66
dimension IV	**17.59**	168.67	**80.62**	87.68

training data	GPEFS	BPG GPEFS	ADF BPG GPEFS
dimension I	12.85	**10.78**	12.62
dimension II	19.66	**6.25**	19.66
dimension III	115.70	49.41	36.66
dimension IV	136.55	165.15	187.01
validation data	GPEFS	BPG GPEFS	ADF BPG GPEFS
dimension I	**19.44**	20.97	68.60
dimension II	269.77	954.07	502.45
dimension III	102.76	55.11	94.21
dimension IV	277.71	395.10	304.70

Table 3.7: Statistics of χ^2-test for every model on training and validation data (lower is better). Bold values are the best results (ANN is treated separately). Note that best results according to the χ^2-test do not necessarily correspond to the KS best results. χ^2-test was used additionally in order to be sure about validity of the output conditional distribution. As it can be seen, χ^2-test is not very useful here. It is probably because of a lack of data and the reason, that this test is primarily used for investigation of two discrete distributions – not a conditional continuous distribution (even if is discretized) and discrete sample. In this situation it should be taken into account more the results of the KS test.

3.4 Visual fitness comparison

Following histograms show the output of found conditional probabilistic distribution D for student time model together with empirical distribution of corresponding samples from validation dataset. Histogram bins for generated samples are in transparent colors (or lighter shades of colors). Histogram bins for validation samples are in opaque colors.

The showed model is exactly the model showed in picture 1.2. Horizontal axis shows day time hours. Time instants I., II., III., IV. represent, when one activity ends, or another starts (see explanation in section 1.1).

The more accurately the distribution D models the empirical distribution of validation samples, the more will their histograms overlap.

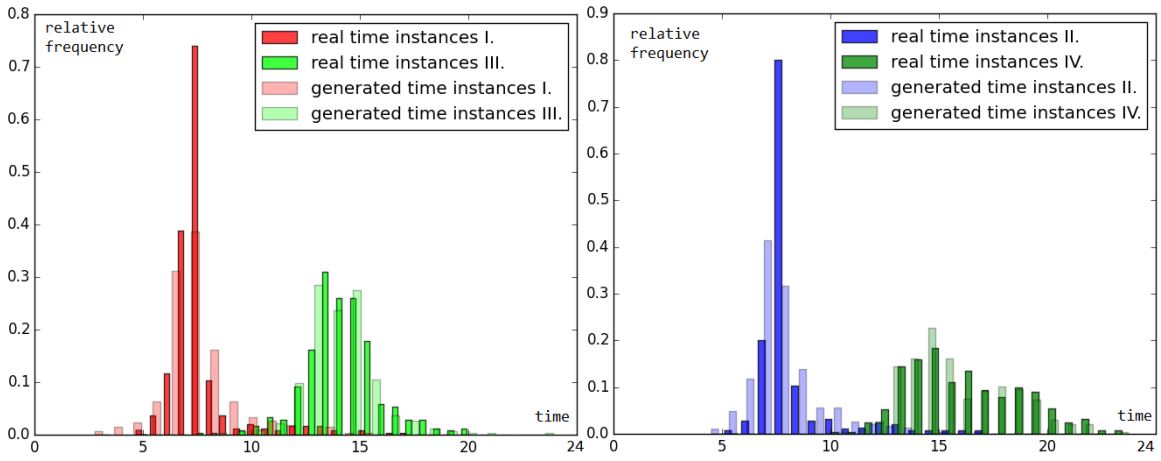


Figure 3.4: There are $m = 4$ histograms of (m-dimensional) conditional probability distribution D modeled by network with 1 hidden layer with $2 + 1$ neurons (see network in figure 2.1). The generated histograms are in transparent colors. Histograms of validation samples are in opaque colors. The meaning of time instants I., II., III., IV. is following: I. time when a student leaves home, II. time when a student arrives at school, III. time when a student leaves school, IV. time when a student arrives home. For example most of the real world students leave their homes at 7:30 a.m. – see the peak in the red histogram of real time instances I.

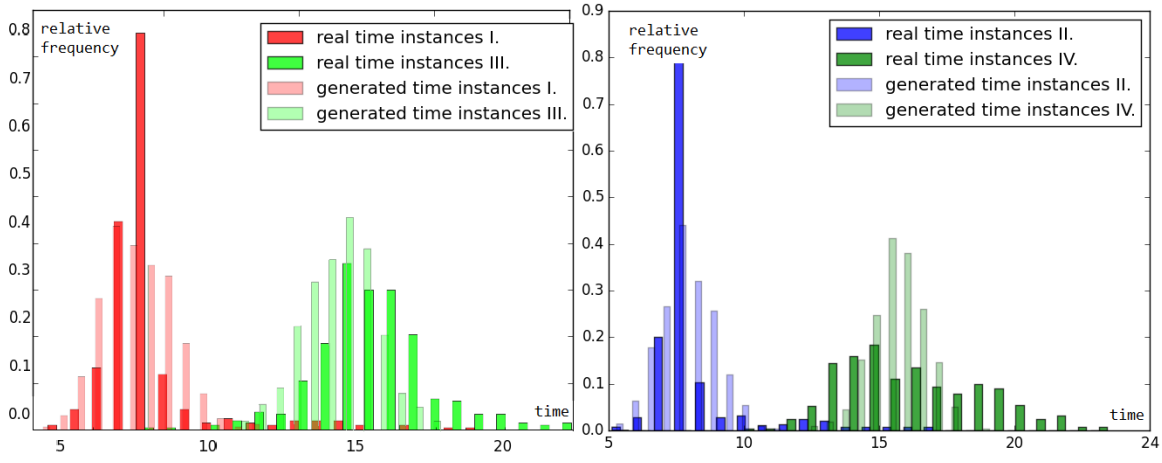


Figure 3.5: Analogous figure of conditional distribution D modeled by expression searched by GPEFS algorithm (with configuration in table 3.2.2). Note that the histograms do not overlap as good as the previous ANN histograms.

3.5 Illustration example of individual

This is exactly the best individual for BPG GPEFS model, which was included in the final comparison. Note, that resultant expressions were not simplified during evolution.

Number of components $K = 3$, maximal tree detph = 5.

$tree_1^{\alpha} = arctan(-0.449004122843+exp(-(0.294651386734+arctan(student+education))^2))$

$tree_2^{\alpha} = exp(-(0.44969788205 + pt\_card + car + sex)^2)$

$tree_3^{\alpha} = sin(exp(-(student + 0.533099626115)^2) + (exp(-(student + sex)^2) * pt\_card))$

$tree_1^{\sigma} = (arctan(sin(-0.405960165464+exp(-(age+arctan(household+pt\_card))^2))+ license) + sin(age + -0.551156432354) + (education + -0.912860317665))$

$tree_2^{\sigma} = (-2.02386803718 + arctan(age + license))$

$tree_3^{\sigma} = ((student + education) + sin((((car * student) + age) * student) + household + (age + sex)) + pt\_card)$

$tree_{11}^{\mu} = sin(sin(education+arctan(sin(license+household)+education))+sin(exp(-((license* education) + education)^2) + sin(car + license + sex) + arctan((age + 0.22776210121) + household)))$

$tree_{12}^{\mu} = arctan((-0.147147298186 * arctan(exp(-(household + (-0.0122084221538 * student))^2) + sex)) + -0.946644834493)$

$tree_{13}^{\mu} = (sin(exp(-(car + exp(-(pt\_card + -0.171649043482 + license + car)^2))^2) + sin((car*car)+age))*sin(license+education+0.621579954787+-0.722262517369)*age)$

$tree_{14}^{\mu} = exp(-((arctan(age+household)+(arctan(education+arctan(-0.18966721874+ 0.870534234414)) * education)) + education)^2)$

$tree_{24}^{\mu} = arctan(arctan(arctan(-0.367706154523 + exp(-(license + sex)^2)) + (age * pt\_card)) + arctan(-0.226583914203 + (age * education * sex)) + -0.680878322007)$

$tree_{22}^{\mu} = sin((-0.13550907753 + license) + exp(-(-0.055725986992 + arctan(license + -0.938595921979))^2))$

$tree_{23}^{\mu} = (arctan((-2.10885823058*exp(-(-0.285331905219+education)^2)*student)+ pt\_card) * (exp(-(-0.528141878851 + 0.528133866054)^2) * age))$

$tree_{24}^{\mu} = (exp(-(education+arctan(student+(-0.16822491128+student))+exp(-(pt\_card+ -0.345196518147)^2))^2) * exp(-(-0.992731009344 + student)^2))$

$tree_{31}^{\mu} = sin((arctan(arctan(-0.0114736181175+-0.577176201548)+-0.375443649847)+ education + (arctan(sin(age + -0.662071180284) + -0.765168490762) * (0.964524381167 + sex))) + arctan(license + pt\_card))$

$tree_{32}^{\mu} = exp(-(exp(-(1.06277040388+(pt\_card+household))^2)+(age*household))^2)$

$tree_{33}^{\mu} = ((exp(-(household + arctan(household + household))^2) + (0.187414758931 + sex)) + arctan(((student + education) + -0.184415877206) + exp(-(car + arctan(student + license))^2)))$

$tree_{34}^{\mu} = ((age * -0.456067798165) + (sin(education + 1.6691551176) + exp(-(student + (household * license) + 0.832856713804 + student)^2)))$

3.6 Conclusions from the experiments

Interesting observation is, that GP models do not overfit data, or in other words – the GP system, that has better training error, has better validation error too. This fact definitely does not apply to ANN, where one can have excellent training error, but poor validation error, caused by overfitting. But none of the GP models did achieve as good training and validation error as chosen intentionally simple ANN.

Best GP models were models with backpropagation optimization with combination of GPEFS evaluation logic. I cannot say whether to use ADF or not, because their median best result was approximately the same. I would personally use simpler BPG GPEFS model ADF BPG GPEFS, because of the greater simplicity and faster evaluation time. Another interesting observation is, that using backpropagation optimization without GPEFS gave very poor results – worse than if it would not be used at all.

Due to apparent results, there are not many doubts in opinion, that ANNs are distinctly more effective at this type of regression task. At the same time ANNs are easier implementable and can be accelerated on graphical processing unit, which causes up to 1000 times more training speed as in case of GP. There also exists many high quality, very fast and scalable systems for neural networks, such as Google's TensorFlow or Theano library. And there is not such a quality counterpart to these libraries for genetic programming.

The fact is, that every task, which can be solved with ANN, is solvable by GP, only after modification of the range of some controlling parameters, or simply after addition another function to primitive set. However there are many problems, where the GP gives poor results, or it takes large amount of time to even achieve the result, which gives simple ANN after several seconds of training. Unfortunately, this problem was one of them and therefore this work cannot be considered as something, that revealed a usable alternative to ANN for probabilistic distribution modeling.

Chapter 4

Development & Problems

As usual on the Faculty of Electrical Engineering on CTU, bachelor works are developed from 5th semester started with Software or Research Project course. In its first phase I was given a functional implementation of ANN (programmed by Jan Drchal) for searching of activity models in transportation. This phase involved mainly the basic skills for solving machine learning tasks: learning Python programming language, studying ANNs and machine learning techniques and studying solved problem principles and logic of given ANN implementation.

After the introductory part, it was clear, that I will not be able to process experiments on a single computer, therefore I registered to Czech National Grid Organization (Metacentrum) for an access to grid computing environment. I chose appropriate hyperparameter values (numbers of neurons in layers) for ANN optimization and ran those tasks in Metacentrum (for models 1.1). There was naturally several implementation issues, which had to be debugged (process overusing, memory leaks).

In the second phase I was assigned to study and implement solution through symbolic regression using evolutionary algorithms. This involved studying of the GP principles and implementation of a basic solution, which was verified and debugged on a simple regression task.

During implementation of the Gaussian mixture model for probabilistic distribution modeling, I encountered an unpleasant issue with numerical instability during tree evaluation. First, there was possibility of reaching zero as an argument of the logarithm; this was solved by adding small constant 10^{-20} and thus the value of logarithm could not go to $-infinity$. Another numeric issue was the math range error caused by too big exponent in the output sigmoid function. This was solved by reducing input argument of the sigmoid function proportionally into the interval $[-15, 15]$.

I tried an adaptive power of mutation, which was based on decreasing of a probability of the new branch introduction, when the best fitness did not improve for many epochs. With lower new branch mutation probability the system will use more point mutation, which do not influence performance as much and therefore serves for better focusing to local optimum. Adaptive power of mutation turned out to have zero effect on performance.

During the hyperparameter optimization of the first GP method I encountered the problem with a very low effect on computational speed up by using a multithread parallelism in

Python, thus I used parallelism on the multiprocess level. In order to reduce computational requirements I tried to omit apriori probabilities in 1.1; more precisely to set $\pi_i(\mathbf{x}) = \frac{1}{K}$ for $i = 1..K$. But this approach showed noticeably worse results than the full-featured form of distribution 1.1 without this limitation.

Next phase was focused on the introduction of advanced techniques to increase the probability of reaching the global optimum. This involved studying of GPEFS conception in [4] and implementation of the tree distance metrics and niching algorithm (based on provided java implementation). After GPEFS implementation very similar hyperparameter optimization methods was run as in the case of GP. In this phase I also examinated the p-values of the Kolmogorovov-Smirnov test and the discretized χ^2 test for all previous models (ANN, GP, GPEFS).

Due to apparent worst performances of the GP approach in comparison to ANN approach, I was recommended to consider using optimization method based on the gradient descent. This included the basic backpropagation algorithm studying and its modification to its more general form for tree expression and arbitrary function (I made this implementation on my own from the course lecture [11]). I debugged my implementation on simple regression tasks and performed the hyperparameter optimization as in previous cases.

In the GPEFS niching algorithm there is the coefficient δ for adaptive tree distance metrics modification, that tries to set real number of species to the *target number of species*. In [4] was originally proposed to alter δ by multiplying by 2 or by $\frac{1}{2}$ based on current feedback. This is not very useful in order to achieve more accurate value of δ, because this oscillates only on two's multiples of initial δ value and thus the system is not likely to do well in focusing on more suitable value of δ. Therefore I changed multiplication constant 2 to 1.2 and $\frac{1}{2}$ to 0.7.

After analyzing a contribution of backpropagation optimization, I spent some time trying to simulate synthesis of ANN in the form of tree expression, because ANNs still showed significantly better performance against GP methods. Modeling neuron as a tree expression is straightforward at first sight, as an output of one single neuron in the network can be modeled as an output of function $\mathbb{R}^n \to \mathbb{R}$:

$$output = activation\_function(\sum_{i=1}^{n}(weight_i * input\_neuron_i))$$

Which can be perceived as tree expression (lisp notation).

```
output = (activation_function
            (+ (* weight_1 input_neuron_1)
               ...
               (* weight_n input_neuron_n)))
```

But neurons in network are in principle functions $\mathbb{R}^n \to \mathbb{R}^m$, where m is generally greater than 1. That causes big implementation issue because the neuron has to have references on its succesors and thus the whole neural network must be represented as oriented graph. Over this graph can be naturally applied evolutionary operators which alter its structure [5]. I did not implement this algorithm, because this thing was already deeply studied and I was redirected by Jan Drchal to Automatically Defined Functions.

One of the contribution of ADF is, that they partially allow to employ functions $\mathbb{R}^n \to \mathbb{R}^m$ (as in neural network) – it is because functions are *shared* – they have in principle more than one outputs. The idea of ADF leaves a lot space in their specific implementation; details in 3.2.5. Workflow activities were similar to previous: studying the idea behind ADF, implementation plus debug, and finally hyperparameter optimization.

After finishing work on ADF, I moved on final comparison part. I tried to compare different methods in the way, which would be able to demonstrate, that implemented advanced techniques really do have the improving effect on the performance. Therefore I tried to emulate the same settings; e.g. the population size during evaluation, the number of repeat trials, number of epochs was the same.

I encountered a problem with much slower convergence of GPEFS compared to GP. More precisely GP was achieving significantly better results than GPEFS with multiple target species, the same total number of individuals and training epochs. Therefore I compared performance of GP and GPEFS with only 1 target number of species and the same population size (picture 3.2). The difference in performance was quite surprising, which led to retraining all previously investigated model configurations (GP/GPEFS, BPG GP/GPEFS and ADF BPG GP/GPEFS) with ensuring, that GPEFS will have larger population in order to better utilize its benefits.

Chapter 5

Conclusion

This work discussed a modeling of the conditional probability distributions of the transportation models. These probaillistic distributions are designed to model real world behavior and in practice they serve as a tool for real world simulation – especially simulation of the traffic infrastructure at given space and time. The more effective method for modeling of these distributions, the bigger amount of time and money it can save from demanding real world data samples collection.

The goal of this work was to specify optimal conditional probability distributions that belong to agent based transportation models. This task led to usage of two approaches: mixture density networks and genetic programming. I examined properties of these two paradigms, computed optimal probability models for every given instance of transportation model and compared properties of those two approaches. As the genetic programming offered more space for experimenting with combination of its advanced techniques, I spent most of the time by their examination and implementation. For this purpose I implemented a GP optimization toolbox, where I run all my experiments. The total CPU time that took my experiments was 400 CPU days.

It turned out that mixture density networks are more effective approach for solved problems. It was also more comfortable to work with in terms of lower control parameters and faster training process.

Mentioned GP toolbox is also useful for symbolic regression task. It supports every described feature in this report and thus one can use it for own experiment, besides that the code analysis can offer a deeper comprehension of used methods.

Whether to continue in investigation of this problem, I am convinced, that in the problem scale which I was experimenting in, there can not be achieved much better results than I showed.

Bibliography

[1] Bishop C. M. *Neural Networks for Pattern Recognition* 2005: Oxford University Press

[2] Bishop C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)* 2006: Secaucus, NJ, USA: Springer-Verlag New York, Inc

[3] Riccardo Poli, William B Langdon, Nicholas Freitag Mcphee *Field Guide to Genetic Programming* 2008: Number March. Published via http://lulu.com

[4] Jan Drchal *Base algorithms for hypercube-based encoding of artificial neural networks* 2012: A Faculty of Electrical Engineering, Czech Technical University in Prague

[5] Kenneth Owen Stanley *The NeuroEvolution of Augmenting Topologies (NEAT)* 2012: Users Page, http://www.cs.ucf.edu/ kstanley/neat.html

[6] John R. Koza *Genetic Programming II, Automatic Discovery of Reusable Programs* 1994: The MIT Press, Cambridge, Messachusetts

[7] R. McGill, J. W. Tukey, W. A. Larsen *Variations of Box Plots* 1978, The American Statistician, Vol. 32, No. 1., pp. 12-16

[8] Matplotlib, Pylab examples, [Cit. 20. 3. 2016], Available on: *http://matplotlib.org/examples/pylab_examples/boxplot_demo.html*

[9] M. Čertický, M. Cuchý, J. Drchal, M. Jakob *AgentPolis Model Report* 2015: Agent Technology Center, Czech Technical University in Prague

[10] J. Drchal, M. Čertický, M. Jakob *VALFRAM: Validation Framework for Activity-based Models* 2015: Agent Technology Center, Czech Technical University in Prague

[11] Nando de Freitas *Deep Learning course 2015, Lecture 8: Modular back-propagation, logistic regression and Torch* 2015: Department of Computer Science, University of Oxford

Appendix A

CD contents

The most useful item, which is provided on CD, is prepared GP/GPEFS optimization toolbox. It manages to solve regression tasks of training data points, given by list of vectors $\mathbf{x}$ and list of target values $\mathbf{t}$. Regression can be made with respect to error function 1.4, or to much simpler mean square error function, which is due to its simplicity and universality very frequently used error function in machine learning regression.

Toolbox source code is written in the popular Python language and does not require any commercial libraries. Program utilization is very easy; it only requires to define training data and control parameters. This is demonstrated in *demo.py* module.

CD also contains LaTeX source file of this text report.

There are also provided serialized python objects of optimized final models for GP, GPEFS, BPG GP, BPG GPEFS, ADF BPG GP, ADF BPG GPEFS. Purpose of these files is, that one may run a python debugger upon these serialized objects and study their structure, in order to better comprehend its inner logic. Then it would be easier to modify it with details of personal use.