COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF APPLIED INFORMATICS

Introduction to Artificial Intelligence

Project: CSP or multi-layer network

November 29, 2021

GENERAL INFORMATIONS:

For your project you choose **one** of the following assignments.

A mandatory part of your solution is a report $README.[pdf|doc|txt]$ in which you describe your solution - what methods you tried, what heuristics you chose, what tricks you used to increase efficiency, what had the greatest impact on the problem, what were your results etc. Please keep the formal side of your report on appropriate level.
Moodle allows multiple submissions, so if you submit final version before deadline, please email us so we know that we can grade it.
The deadline is **8. January 2023.** In case your oral exam is earlier, deadline moves to 3 days before exam so we have enough time to grade it.
You can submit it even after the deadline with a penalty of -2 points for each extra day.

PROJECT 1: CONSTRAINT-SATISFACTION PROBLEM

Using methods for solving CSP problems, you will be solving crossword puzzles.

Input data:

In the attached file *words.txt* there is a dictionary - about 30 000 english words which you can use to fill in the crosswords.
In *krizovky.txt* there are 10 empty crosswords, each separated by an empty line. Each crossword consists of "walls"('#') and empty spaces (' '), for example.:

```
#####
#   #
# # #
#   #
# # #
#   #
#####
```

Program:

You are given a prepared skeleton of the code (*crossword.py*), which you can use. This time you can modify anything in it - feel free to implement new definitions, classes, structures, etc., in order to solve the crosswords as efficiently as possible. The crossword is implemented in the class *CrossWord*, and the "map" itself is saved as *CrossWord.grid = list[list[string]]*, where each cell of the list contains a single letter of the crossword. The indexing takes the form of $[row][column]$ (not $[x][y]$!).
Besides the algorithm itself also **submit *krizovky_out.txt***, which will contain crosswords already solved by your algorithm and of course $README$.

The algorithm:

To solve the crosswords use backtracking (as the base you can use the code from the exercise), where you implement various heuristics and/or other "enhancements", allowing the algorithm to solve the crosswords properly and fast.
Think mainly about the CSP lecture, there is a lot of useful information on how to approach solving CSP problems. But here are some suggestions/ideas, which might help you:

- **heuristics** - we saw already on the exercise, that heuristics can be a tremendous help when solving CPS problems efficiently. We talked about heuristics to choose the variable but also the value.
  - choice of the variable: MRV, degree heuristics, ...
  - choice of the value: least-constraining-value, ...
  - heuristics can be combined.
  - are there some heuristics specifically suitable for this task?
  - are there some heuristics specifically efficient when working with a certain data structure?
  - TIP: *on each crossword you can use different heuristics (or a different set of heuristics)!*
- **node and arc consistency** - a way to effectively narrow down the domain sizes.
  - can be applied on the initial domains, or you can maintain the consistency when filling in the crossword.

  - pseudocode of arc consistency implementation can be found for example HERE - page 13
- **data structure** - iterating through a list of all possible words is slow. Can we keep the words in some other data structure - for the sake of our assignment?
- **other** - you can use other technique to solve the crosswords faster (while it is implemented by you).

From the options above you can choose to implement all, some of it or even nothing. The grading is based almost solely on the number (and the difficulty) of crosswords your algorithm can solve in a limited time.

Task (15 + bonus 2p):

Your task is to fill in the empty spaces of the crosswords in a following manner:

- each empty space must be filled
- directions of the crosswords are only *down* and to the *right*
- each sequence of at least two letters between the two walls (#) must be a word from the attached dictionary - this holds for both of the directions.

For example a crossword defined above can be filled as follows:

```
#####
#era#
#x#r#
#ice#
#s#n#
#tea#
#####
```

In the attached file there are 10 distinct crosswords, where the last one is a bonus. You can solve them in any order.

Grading:

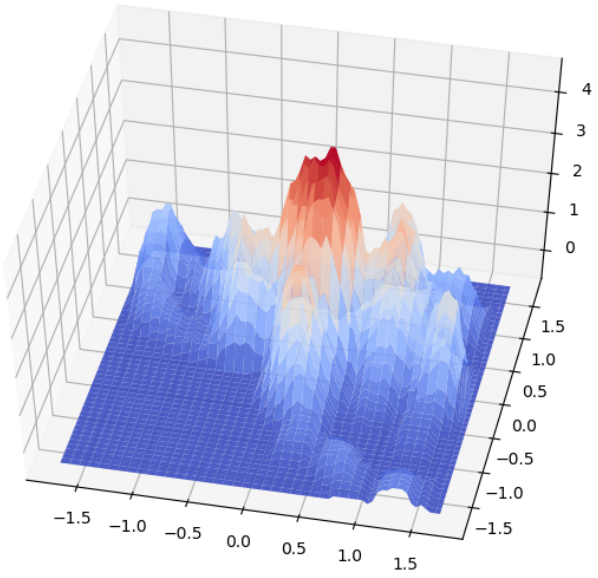
The main condition is to have a correct implementation of backtracking, adjusted to the task. Then, each of the crosswords has its own "worth" - the number of points you get if you solve it correctly. The points are distributed as follows: [0.5, 1, 1, 1, 1.5, 1.5, 2, 2, 1.5, 2]. The crosswords are ordered "roughly" according to their difficulty, although is is hard to judge, because with certain heuristics their difficulty might be different. Therefore, if you want to change the order in which we will evaluate (and time) the ability of your algorithm to solve the crosswords, change the order of crossword indices - in the for loop, in which the individual crossword puzzles are solved. **Do not modify** the file "krizovky.txt" or "words.txt", because we will test your code with the original files.
For neat code (we will be able to run the code without modification and read it easily) you can gain 1 point. A nice and detailed README is worth 2 points.
Time limit for your code to run is 30 minutes. We will evaluate your code on a notebook with Intel(R) Core(TM) i7-10510U processor. In case you are not sure how many points you get (and how long it takes to solve on our notebook) you can contact us and we will tell you how fast your code is.
In case you have any questions regarding this project, please contact me (email: stefan.pocos@fmph.uniba.sk or via MS Teams). I will probably notice an email sooner than a message on MS Teams.

PROJECT 2: MULTI-LAYER NETWORK, ERROR BACKPROPAGATION

Program a multi-layer perceptron and choose the best architecture to approximate a given 2D function.

Data:

You will approximate the function $f : \mathbb{R}^2 \to \mathbb{R}$, which has two inputs and one output:



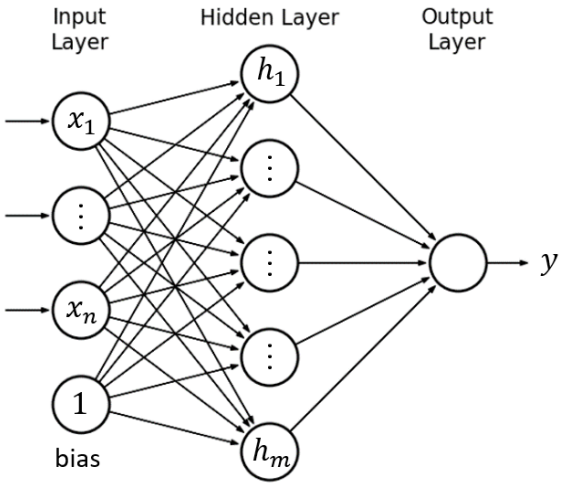
During the exercises we only used a simpler network and just one set of data. In practise, data is usually split into training and testing (optionally also validation) sets. Testing set will be used by us for grading (you don't have access to it). The rest is available for you and you can use it for training and choosing the optimal model. You should split this data into training set and validation set, where the training set is used to train each of the models and the validation is only used for evaluation of generalization error (meaning, how well can the model predict data that it has never seen before) and choosing the best set of hyper-parameters. Be careful to split the data randomly to ensure that both training and validation set are from the same distribution. This can be done simply by shuffling the data before splitting. Train to val. ratio can be for example 80:20 or 75:25, just make sure that the training set is much bigger than validation set.
Optionally you can use k-fold cross validation.

1771 of training points are available, which you can use for training and other 443 testing points (these are only known to us) will be used for evaluation of the model. You can load the data using $np.loadtxt(...)$, where the first two columns are the input points and the last one is the output.

Model:

Your model must meet three conditions:

- It is a multi-layer perceptron, so it has input layer, output layer and **at least one** hidden layer.
- Include bias on input layer. On hidden layer it is optional.
- Model will be trained using error backpropagation, formulas are included below.



Choose the other parameters in a way, that the network achieves as high accuracy as possible. Do not forget, that the network will also be tested on the points, which you do not have access to, so you should mainly optimize the generalization error (error on validation data).
Output computation in multi-layer network with n inputs, m hidden neurons h_i and one output y:

$$net_i^{hid} = \sum_{j=1}^{n} w_{ij}^{hid} \cdot x_j$$

$$h_i = f_{hid}(net_i^{hid})$$

$$net^{out} = \sum_{i=1}^{m} w_i^{out} \cdot h_i$$

$$y = f_{out}(net^{out})$$

Formulas for training (backpropagation) this network:

$$\delta^{out} = (d - y) \cdot f'_{out}(net^{out})$$

$$\delta_i^{hid} = (w_i^{out} \cdot \delta^{out}) \cdot f'_{hid}(net_i^{hid})$$

$$w_i^{out} := w_i^{out} + \alpha \cdot \delta^{out} \cdot h_i$$

$$w_{ij}^{hid} := w_{ij}^{hid} + \alpha \cdot \delta_i^{hid} \cdot x_j$$

Do not forget that derivatives f'_{hid} and f'_{out} need to be modified according to f_{hid} and f_{out}.

Obligatory try different configurations of hidden layer (number of neurons, activation function) and different values of learning rate.
Other modifications that you can try to improve accuracy:

- more hidden layers
- *momentum* to skip through local minima
- *learning rate schedule* for dynamic adjustment of learning rate
- different methods of weights initialization
- *early-stopping*

Program:

You can use the code from 9th exercise (perceptron) as a template. You can also use *utils.py* which is included in the project assignment. It contains functions for plotting graphs and data visualisation.

Assignment (15p + bonus max 2p according to model accuracy):

Program the network, so that it learns to approximate the given function. Try different combinations of parameters (obligatory number of hidden neurons, activation function and learning rate) and other network properties to achieve the best possible accuracy (on testing data). After choosing the best set of hyper-parameters, you can train the network on all data available to you. Submit the final version of your code in a state, that when we run it, the training starts (and we do not have to tweak the code for 15 minutes to make the training process start).

Also implement functions:

- *evaluate(file_path):* loads testing data from given file, computes network output and returns average error
  $= average((d-y)^2)$
- *save_weights(file_path):* saves trained weights in a file
- *load_weights(file_path):* loads saved weights from file

You can also attach a file with exported weights of your best model - during evaluation we will load it using *load_weights()* and evaluate the accuracy of your top model on testing data using the function *evaluate()*.

Grading:

First round: we run the training, calculate the average error on the training data (all the data you had), compute the error on the testing data (our data, which you don't have access to). Then we calculate the average of these errors in the ratio of 30:70 (the testing error has greater weight).
Second round: if you attach a file with exported trained weights, we also run the second phase of testing. We calculate the accuracy of training and testing data, again in the ratio of 30:70.
Results of both rounds are combined in the same ratio (70:30), the network which is trained during evaluation has bigger importance.

Points are computed as $p = 15 - 65 \cdot err$, where err is the average error, computed as described above. Another three points are assigned for your approach finding the best set of hyper-parameters. Modular and easy-to-read code and final report are worth 2 more points. So you can get 100% (15 points) if you carefully choose the best model, your code looks neat and the average error is 1/13 (cca 0.075) or below. Bonus points are awarded for accuracy, if the average error of your model is less than 1/13.

Time limit for training (during first round of evaluation) is approximately 5 minutes. However, the network, whose weights you export and submit can be trained as long as you wish.
In case you have any questions regarding this project please contact I. Bečková
(email: iveta.beckova@fmph.uniba.sk or via MS Teams).
Please note: using libraries as TensorFlow, Theano, Keras, Lasagne, etc. is forbidden.