Artificial Neural Networks

Overview

The aim of this practical is to construct a neural system capable of suggesting a travel destination. The assignment consists of three core parts and a fourth one for extensions. The system was developed in Python, it achieves all basic requirements and the following extensions. The system can accept ambiguous answers (‘probably’, ‘maybe not’, ‘I don’t know’, and variations). The system can record the learning progression of a neural network in a file. Additionally, these output files were used to create charts, so as to visually compare the effects of changing the learning rate, the momentum, and the number of hidden units.

Design

The submission consists of three Python packages that contain the source code for the three core parts of the assignment. The extensions are implemented within these parts. Each part can be executed separately, without affecting the others. The system has three modes of execution – basic, extra, and user. Instructions on how to execute each part in any of the three modes are provided at the end of this report.

Datasets

The system supports two datasets – ‘training’ and ‘testing’. They are used to train the neural network and test its performance. The given data is not large enough to justify creating a ‘verification’ set. As per part 3 of the assignment, the data can be increased, but it needs many executions to reach the size needed for a ‘verification’ set. The ratio between the sizes of the ‘training’ and ‘testing’ sets is as recommended in the lectures – 80% training and 20% testing. This allows the neural network to find a good fitting function while still being able to generalise well. The sets are saved in files under “assets/data/<mode of execution>”. These folders also contain other files needed by the system for each of its modes of execution.

The format of the set files is as follows. Each line in a file represents a sequence of answers to questions and ends with the name of a travel destination. The answers are single digit binary values, where 1 represents “yes” and 0 represents “no”. The name of the destination is case sensitive, so there is a difference between “Argentina” and “argentina”. The system can be improved to handle destinations in a case-insensitive manner. They would be stored as all-lower-case strings but when printed, the first character would be capitalized. However, such an extension would not directly contribute to the goal of the practical. The focus was on the objectives described in the Overview section of this report.

Modes of execution

The modes of execution determine what initial data the system uses. Additionally, the “user” mode allows for expanding that data. In “basic” mode, the system uses only the data provided with this

assignment. The file “trip.csv” has 90 rows which are split into 80% * 90 = 72 rows for training and 20% * 90 = 18 rows for testing. The “basic” neural network associated with this mode has a mean squared error of less than one percent. In other words, only one predicted output pattern is off from the actual pattern and the difference between the two is very small.

In “extra” mode, the system uses the data provided with this assignment, and the ‘extra’ data, added as required by part 1, step 9. The difference is very small in that only one destination (“Italy”) and one feature (“Architecture”) are added. Both training and testing sets are slightly expanded. The “extra” neural network associated with this mode has a mean squared error of less than two percent.

In “user” mode, the system uses the data provided with the assignment, the ‘extra’ data from part 1, step 9, and all data acquired as part of the learning process in part 3, step 1. The two datasets ‘training’ and ‘testing’ grow a bit with each new destination and feature learned.

Part 1 – Network

In this part, the system creates and trains a classification neural network. The “datasets” module is used to load the data sets into global data structures, so that other modules can reference them without the overhead of file IO. This module also stores the mode of execution as a pair of boolean values. Various functions throughout the system reference these two, so as to determine which folders and files should be accessed.

The functions in the module “datasets” complete the first two steps in part 1, namely – the encoding of inputs and outputs. When parsing a dataset file (the format is as described in the “Datasets” subsection of this report), for each line, the system stores all integers in a list, and all non-integers in a string. These are then stored in a dictionary that maps a destination name to a list of input patterns. After the whole file has been parsed, the destinations are encoded as tuples of single digit binary numbers. The one-hot encoding has been used because it allows increasing the number of destinations without adding to the complexity of the algorithm. The alternative approach of binary number encoding has higher computational complexity (converting the index of every destination to binaries of the same length). It also introduces an additional edge case in decoding the output of the neural network (discussed later in this section).
The codes are stored twice in two different dictionaries – one used to get the code from a destination name, and the other used to get the destination name from a code. Finally, the ‘training’ and ‘destination’ dictionaries, map input patterns to expected output patterns (destination codes).

The module “network” handles the training and testing of the neural networks used in the three different modes of execution. When accessing a dataset, a conversion function is used to create a pair of lists. The first list contains all the input patterns (as lists) and the second list contains all the expected output patterns. The reason for this is that the scikit learn uses lists but the system stores the data as tuples, so as to allow dictionaries to hash them. The functions “store_training_data”, “write_first_column”, and “write_next_column” are use to store the learning progression of a neural network in a file. These functions are executed only if the command line argument ‘chart’ is supplied. In that case, during training, the standard output is redirected to a file and the scikit classifier’s “fit” function writes lines of the following format “Iteration n, loss = e” where n is the iteration number and e is the error at that iteration. After training is complete, the storing functions

access this file, and copy across the errors to a CSV. Each column of the output file shows how a neural network gradually reduces the error. The differences between columns is the number of hidden units. Those numbers appear as the first line (a header) in the csv. The name of the file gives information about the learning rate, the momentum, and the maximum number of iterations. For example, “user_0.1_0.4_1000000.csv” contains the result of training a neural network in “user” mode, with a learning rate of 0.1, momentum of 0.4, and a maximum number of iterations of 1000000. The header of the CSV contains the numbers from 15 to 2, that means the network was trained 14 times with the 14 different amounts of hidden units. There were a lot of unexpected hurdles when implementing this functionality. The final version has been tested on large datasets and works. However, it uses two separate functions for storing the first column (which needs no commas), and for storing subsequent columns (which add commas to the end of every row and repeat the last entry if the new column is of a different length). A more elegant solution would have one function that uses if branches to distinguish between these special cases.

Shifting the focus of this report back to the steps of part 1, the “create_classifier” function instantiates a feed forward neural network with one hidden layer. The number of hidden units is specified by the caller function. That happens in the “train_new” function. It creates and trains (using backpropagation) multiple neural networks with different amounts of hidden units. The rules of thumb in the lectures have been taken into consideration when determining the amounts. Specifically, the rule “#hidden = 2/3 * (#input + #output)” is what drives the implementation. The system calculates that number and then increases it by a quarter of itself. The first network created has that many hidden units. Each following network has one unit less than the one before. The last network has one hidden unit. The rules of thumb do not provide a strict minimum. The number of output units can be used as such, but there are instances where a network performs well, even if it goes against that rule. The recommended maximum of twice the input units is softly enforced. The starting amount is practically (5/4)*(2/3)*(#input + #output) = (5/6)*(#input + #output). Due to the nature of the data, the number of output units is always less than the number of input units. Thus the sum above is always less than twice the input units.

Each network is evaluated via the testing set. If the error is less than a predefined maximum, than the network is good enough and can be used. At that point, the function terminates and no other networks are trained. The algorithm keeps track of the network that has produced the least error. If no network achieves the desired error rate, then the one with the least error is returned. Before returning, the function saves the network in a file, according to the mode of execution.

As required, networks used by part 2 are not affected by networks trained in part 1. The “network” module saves the networks to “assets/neuralnetworks/<mode of execution>.net”. Part 2 reads from “assets/data/<mode of execution>/network.net”. This way, if a user creates a new network that has a high error rate, the previous one is still saved. In order to use a network for part 2, the user has to manually move the newly created network to the appropriate folder. Since there are two locations where a neural net can be found, the modules provides two functions to access them.

The rest of the functions in this module deal with producing output. The MLPClassifier “predict” and “predict_proba” functions are used in that case. The “predict” function returns a list of integers which represents an encoding of a destination. If there is only a single 1 and all other integers are 0s, then the code can be mapped to a destination. Mapping will fail if there are only 0s, or multiple

1s. In that case, the function “predict_proba” is used to return probabilities for each integer. The highest probability is treated as a 1, all others become 0s and this is the guess that the network makes. This efficient solution is possible due to the one-hot encoding. If binary encoding was used instead, then multiple high probabilities would need to be considered and a more complex algorithm to be developed. Thus, the trade-off between one-hot and binary is speed vs storage. One-hot is fast requiring linear time to create codes and linear time to find the highest probability in a list, but *n* output units for *n* destinations. Binary is somewhat slower requiring at least *n.logn* complexity to create codes and find the highest probabilities in a list, but *logn* output units for *n* destinations.

So far, steps 1 through 8 have been covered in this section, as the report discusses the construction, training, evaluation of a neural network, as well as selecting an appropriate amount of hidden units, and saving it to a file. Step 9 was achieved manually. The dataset was expanded with destination Italy, and feature Architecture. For each previously known destination, an unambiguous answer was added to the training and testing data. The new ‘extra’ network has one more input unit, one more output unit, and by extension one more hidden unit, as in that case it produced a sufficiently good error. The ‘extra’ execution mode was created, so that a user can run the system with these datasets.

Note: The library ‘scikit-learn’ uses a deprecated module ‘imp’. This does not affect the system described in this report. A warning is sent to standard error output when importing the library. To hide the warning, the error stream is temporarily redirected during the import.

Part 2 – AI Travel Genie

This part lets a user interact with the previously created neural network. The system asks questions about the user’s dream destination and converts the answers to input patterns. The neural network makes a prediction and returns a possible destination.

When part 2 is executed, the system first loads the questions from the file that contains it. For each mode of execution, there is a file “assets/data/<mode of execution>/questions.txt” that contains all the questions that the genie will ask. After that, the function “ask_questions” is called to iterate through the questions, parse responses from the user, and make guesses. At each iteration, the system accepts any one of the following responses: “yes”, “no”, “probably”, “probably not”, “maybe”, “maybe not”, “I don’t know”, “Idk”, and their spelling variations. Any other input is treated as invalid, in which case the system waits for a new answer. The answers are converted to 0s and 1s and are stored in a list ‘answers’ of input patterns. Each input pattern here is a list, so the structure is practically a two-dimensional array. It starts off with only a single input pattern. When “yes” or “no” is parsed, all input patterns are extended with a 1 or a 0, respectively. All other answers double the size of the ‘answers’ list. When a “maybe not” or a “probably not” is parsed, the system duplicates all input patterns inside the ‘answers’ list. The first half of the new total gets a 0 appended at the end. The second half of the new total gets a 1 appended at the end. The responses “probably”, “maybe”, “I don’t know”, etc. follow the same principle, except they append 1 to the first half and 0 to the second half. After that, the input patterns are sorted based on their distance from the first pattern in the ‘answers’ list. “Maybe not” puts a 0 in the first input pattern, and “maybe” puts a 1. This way the first input pattern most closely reflects the thoughts and assumptions of the user. Sorting the ‘answers’ two-dimensional array means that input patterns with lower indexes are better assumptions.

So far, this description covers steps 1 and 2 from part 2. With regards to step 3, the mode of execution specifies the neural network to be used (as well as the question to be asked). The input patterns described above, and a reference to the network loaded from a file are given to the "network" module which makes predictions and returns a list of possible dream destinations. The occurrences of each destination are counted. The most occurring one is the best guess, as it fits most input patterns. Ties are broken by taking the destination with a lower index. That destination is a better guess because it is closer to the user's assumptions.

Once a guess is made, it is displayed to the user (as per step 5 of part 2). If the user enters 'yes' (or a variation of it), than the travel genie has completed its job and the system terminates. Any other input is treated as a negative. The system can make multiple guesses as per step 6 of this part. An early negative answer simply means that the genie must continue asking questions. If with its last guess, the genie still does not succeed, it admits that it cannot guess.

Early guesses are implemented by counting how many questions have been asked, and making a guess only upon reaching a threshold. When a threshold is reached, the system creates a copy of the 'answers' list and fills it as if the user has randomly answered "maybe" and "maybe not". This ensures that all possible permutations of the remaining questions are inserted into the list. The input patterns are then of length appropriate for the neural network. Predictions are calculated and a guess is made. On a negative answer, the system moves the threshold further and continues asking questions. The threshold is the index of the question after which a guess is to be made. The initial value of the threshold is the middle of the list of questions, when half of them have been asked. The threshold is then moved to the middle of the remaining half (aka one quarter further in the total). The next threshold is the middle of the remaining quarter (aka one eight further in the total).

Formally, the value of the threshold is $\left(1-\frac{1}{2^{n+1}}\right)*Q$. Where *n* is the number of guesses made so far and *Q* is the total number of questions. The system calculates the threshold by dividing the number of remaining questions by two every time. This is more accurate when the number of questions is not a power of two. When the calculation says that the threshold should be increased by 1 (in other words, wait for one more question before guessing again), the system assumes that it is nearing the end of questions and sets the threshold to the last index in the questions list.
This method allows the genie to quickly guess destinations that have unique input patterns. The first questions about penguins and islands quickly reduce the number of possibilities. In a way, this approach is similar to binary search. The total search space is divided in two at every step. With each guess, there are twice as less questions, so twice as less random input bits.

Finally, the system keeps track of what guesses it has made, so it does not make the same guess twice. This presents the problem of what happens, when the neural network returns predictions that contain only guessed destinations. In that case, the genie gives up early, as it cannot improve. The guesses made have included all the permutations of any remaining questions. Thus, it does not matter what else the user enters, the network will still produce the same outputs. It should be noted, that this can only occur if there is a big difference between the training data and the user input (such as bad training data, or if the user is thinking of a destination that the genie does not know about).

Note: The basic specification requires that only "yes" and "no" are acceptable. Accepting ambiguous answers such as "probably" and "maybe not" is an extension.

Note: step 4 of part 2 is handled in the “network” module where probabilities are used to find a destination for an output pattern that does not directly map to a known destination.

Part 3 – Learning AI Travel Genie

The third Python package consists of two modules. The module “learning_genie” completes the first step of part 3, and “reporting” completes the second step.

The “learning_genie” uses the “genie” module from part 2 to ask questions and make guesses as regular. The functions in this module deal with user input and editing files. The function “get_binary_input” parses a “yes” or a “no” from the user and returns True or False. The caller may allow this function to return None if an answer is not strictly required. The function “user_will_not_teach” asks the user if they wish to teach the Learning Genie about a new travel destination and feature. The function “obtain_question” prompts the user to enter a destination, a feature, and a question for that feature. The function returns early if the destination or feature are recognized.

The “learn” function is the one that calls “obtain_question” and then it asks the user to answer the question for each destination known by the genie. This creates enough data so as to update the training and testing sets. If no question was returned (either because the destination or feature are known), the system makes a small update to the training set. The “user” network is then retrained to account for the increase in input and output units. This training updates the network under “assets/neuralnets/user.net”. However, when guessing, the system will access the network under “assets/data/user/network.net”. After learning about a new destination, the system should be able to use it when guessing. Thus, upon retraining a network, “learning_genie” copies it and replaces the one that will get loaded on the next execution. This does not break the aforementioned mechanism of protecting the neural networks. If part1 is executed, it will not affect the following parts (part2 and part3). If part2 is executed it will also not affect the other parts. The update in part3 only affects part2 and part3 when they are running in “user” mode. It is necessary to show that the system works correctly.

When asking the new question for each destination, the system wraps the answer inside a lambda function. If the answer was “yes”, the lambda will always return 1. If the answer was “no”, the lambda will always return 0. If the user has quit, the lambda will return a 1 or a 0 at random every time it is executed. A dictionary then maps each destination to a lambda function. The training and testing sets are updated by iterating through the lines and adding a 1 or a 0 as the last integer before the name of the destination. The value added is the result of executing the lambda function. If the user did know how to answer their question with regards to the destination, then the lambda will consistently and correctly return the same value. If the user didn’t know, then the lambda will randomly fill a 1 or a 0. This is important because when the network is retrained, this new integer will represent the input for a node. Destinations with consistent answers will be strongly affected by this node, whereas destinations that have random answers will not be affected by this node. This simulates the fact that the network cannot make a strong judgement given that it does not have the necessary knowledge.

In a single execution in “user” mode, the system either successfully guesses, fails to guess and the user responds with “no” when asked to teach the genie, or learns new input pattern(s) and retrains the network.

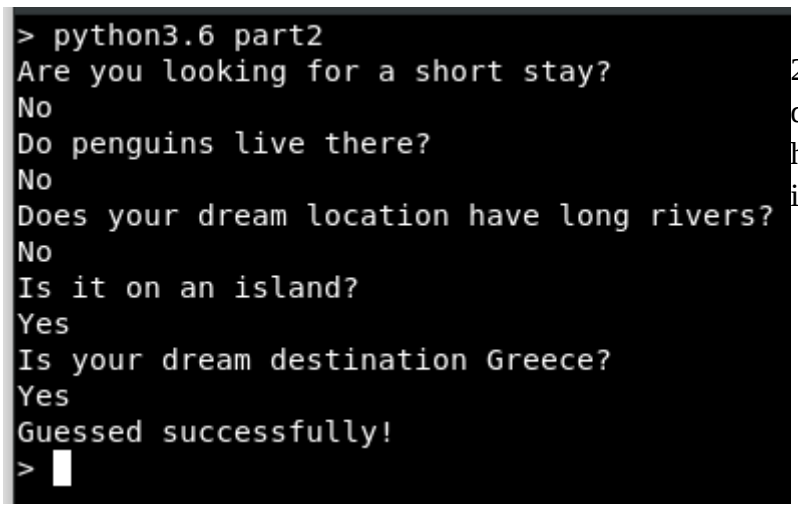
The “reporting” module provides three short functions to store information that might be useful for the agency. This module is only used by the “learning_genie” when executing in “user” mode. The function “log_learning” stores the destinations and features that the system learns about. The format is *destination feature answer*. The last two are not necessary. This achieves the first half of step 2. The function “count_dream” keeps track of how many times each destination was the dream destination. The format is *destination count.* A travel agency can use this information to see which is the most popular destination. The counting can be reset every season. This will allow the agency to make decisions such as “Greece is very popular in the summer. We should organise more trips there.” or “Argentina is not popular in the winter. We should cut down our spendings at that time.”. The function “count_answers” keeps count of how many times a question was answered “yes” and how many times - “no”. It will ignore cases when the answer was ambiguous. The respective file stores only integers – two values per line. The left one is how many times the answer was “yes” and the right one is how many times the answer was “no”. The line number is used as a relation between questions and answer counts. The counts for question 1 are in line 1. The counts for question 2 are in line 2, and so forth. This information is useful when a question is consistently getting the same answer. Here’s an example decision: “People always answer with ‘no’ to the question about penguins. Therefore, there is no need to ask that question.” This feature can be extended to count ambiguous answers, example benefit - “Everyone answers ‘maybe’ to the question about short stay. Therefore, we can should give a specific definition of ‘short’ in this case.”.

Examples and Testing

Extensive testing has been performed on the system to validate that it works as expected. The networks have been trained and retrained numerous times. Recordings of their trainings can be found under “assets/charts”. Note that the file “user_0.1_0.4_1000000.ods” is very big and takes several minutes to open on the lab machines.

Performance check have been run for every network. The output of that lets the user compare the actual destinations and the predicted destinations. The Running section provides instructions on how to run the performance checks. The rest of this section contains screenshot examples of the system running.

Performance check in all three modes produces no error. This check is done by running the saved neural network from “assets/neuralnets/<mode of execution>.net” on the testing set “assets/data/<mode of execution>/testing.set”.



In this example, the genie from part 2 has successfully guessed the dream destination Greece, and has done so halfway through the questions. This is the first early guess.

```
> python3.6 part2 extra debug
Are you looking for a short stay?
maybe
Do penguins live there?
no
Does your dream location have long rivers?
probably
Is it on an island?
yes
Is it known for its seaside?
yes
potentials = [[1, 0, 1, 1, 1, 0, 1, 1, 0], [0, 0, 1, 1, 1, 0, 1, 1, 0], [1, 0, 0, 1, 1, 0, 1, 1, 0], [1,
0, 1, 1, 1, 1, 1, 1, 0], [1, 0, 1, 1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 0, 1, 0, 0], [1, 0, 1, 1, 1, 0, 1,
1, 1], [0, 0, 0, 1, 1, 0, 1, 1, 0], [0, 0, 1, 1, 1, 1, 1, 1, 0], [1, 0, 0, 1, 1, 1, 1, 1, 0], [0, 0, 1, 1
, 1, 0, 0, 1, 0], [1, 0, 0, 1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 1, 0, 1, 0], [0, 0, 1, 1, 1, 0, 1, 0, 0],
[1, 0, 0, 1, 1, 0, 1, 0, 0], [1, 0, 1, 1, 1, 1, 1, 0, 0], [1, 0, 1, 1, 1, 0, 0, 0, 0], [0, 0, 1, 1, 1, 0,
 1, 1, 1], [1, 0, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 1, 1, 1, 1, 1], [1, 0, 1, 1, 1, 0, 0, 1, 1], [1, 0,
1, 1, 1, 0, 1, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1, 0], [0, 0, 0, 1, 1, 0, 0, 1, 0], [0, 0, 1, 1, 1, 1, 0, 1,
0], [1, 0, 0, 1, 1, 1, 1, 0, 1, 0], [0, 0, 0, 1, 1, 1, 0, 1, 0, 0], [0, 0, 1, 1, 1, 1, 1, 1, 0, 0], [1, 0, 0, 1, 1
, 1, 1, 0, 0], [0, 0, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0, 0], [1, 0, 1, 1, 1, 1, 0, 0, 0], [0,
 0, 0, 1, 1, 0, 1, 1, 1], [0, 0, 1, 1, 1, 1, 1, 1, 1], [1, 0, 0, 1, 1, 1, 1, 1, 1], [0, 0, 1, 1, 1, 0, 0,
 1, 1], [1, 0, 0, 1, 1, 0, 0, 1, 1], [1, 0, 1, 1, 1, 1, 0, 1, 1], [0, 0, 1, 1, 1, 0, 1, 0, 1], [1, 0, 0,
1, 1, 0, 1, 0, 1], [1, 0, 1, 1, 1, 1, 1, 0, 1], [1, 0, 1, 1, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 0, 1, 0],
 [0, 0, 0, 1, 1, 1, 1, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0, 0], [0, 0, 1, 1, 1, 1, 0, 0, 0], [1, 0, 0, 1, 1, 1
, 0, 0, 0], [0, 0, 0, 1, 1, 1, 1, 1, 1], [0, 0, 0, 1, 1, 0, 0, 1, 1], [0, 0, 1, 1, 1, 1, 0, 1, 1], [1, 0,
 0, 1, 1, 0, 1, 1], [0, 0, 0, 1, 1, 0, 1, 1, 0], [0, 0, 1, 1, 1, 1, 0, 1, 1], [1, 0, 0, 1, 1, 1, 1, 0, 1],
 [0, 0, 1, 1, 1, 0, 0, 0, 1], [1, 0, 0, 1, 1, 0, 0, 0, 1], [1, 0, 1, 1, 1, 1, 0, 0, 1], [0, 0, 0, 1,
1, 1, 0, 0, 0], [0, 0, 0, 1, 1, 1, 0, 1, 1], [0, 0, 0, 1, 1, 1, 1, 0, 1], [0, 0, 0, 1, 1, 0, 0, 0, 1], [0
, 0, 1, 1, 1, 1, 0, 0, 1], [1, 0, 0, 1, 1, 1, 0, 0, 1], [0, 0, 0, 1, 1, 1, 0, 0, 1]]
occurrences = {'Argentina': 14, 'Spain': 7, 'Italy': 8, 'Australia': 10, 'Greece': 25}
Is your dream destination Greece?
no
Is your dream location historically famous?
dont know
Do people there speak Spanish?
no
potentials = [[1, 0, 1, 1, 1, 1, 0, 1, 1], [0, 0, 1, 1, 1, 1, 0, 1, 1], [1, 0, 0, 1, 1, 1, 0, 1, 1], [1,
0, 1, 1, 1, 0, 0, 1, 1], [1, 0, 1, 1, 1, 1, 0, 0, 1], [1, 0, 1, 1, 1, 1, 0, 1, 0], [0, 0, 0, 1, 1, 1, 0,
1, 1], [0, 0, 1, 1, 1, 0, 0, 1, 1], [1, 0, 0, 1, 1, 0, 0, 1, 1], [0, 0, 1, 1, 1, 1, 0, 0, 1], [1, 0, 0, 1
, 1, 1, 0, 0, 1], [1, 0, 1, 1, 1, 0, 0, 1, 1], [0, 0, 1, 1, 1, 1, 0, 1, 0], [1, 0, 0, 1, 1, 1, 0, 1, 0],
[1, 0, 1, 1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 1, 1, 0, 0], [0, 0, 0, 1, 1, 1, 0, 0, 1], [0, 0, 0, 1, 1, 1,
 0, 0, 1], [0, 0, 1, 1, 1, 0, 0, 0, 1], [1, 0, 0, 1, 1, 0, 0, 0, 1], [0, 0, 1, 1, 1, 1, 0, 1, 0], [0, 0,
1, 1, 1, 0, 0, 1, 0], [1, 0, 0, 1, 1, 0, 0, 1, 0], [0, 0, 1, 1, 1, 1, 0, 0, 0], [1, 0, 0, 1, 1, 1, 0, 0,
0], [1, 0, 1, 1, 1, 0, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 0, 0, 1, 0], [0, 0, 0, 1, 1
, 1, 0, 0, 0], [0, 0, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0, 0]]
occurrences = {'Italy': 8, 'Spain': 2, 'Australia': 10}
Is your dream destination Australia?
yes
Guessed successfully!
>
```

This example shows how the genie from part 2 can make early guesses, and handle ambiguous answers (extension). The ‘debug’ argument has been added to force the system to print some of its variables. The ‘potentials’ array contains all possible input patterns. It is a temporary copy of the ‘answers’ list described in the subsection “Part 2 – AI Travel Genie” under the “Design” section of this report. In this example, at the time of the first guess, the ‘potentials’ array is large because it contains all possible permutations of answers to the remaining questions. After a second unambiguous answer (“no” to “Do people there speak Spanish?”), the size of ‘potentials’ is halved, as possible permutations are eliminated. Additionally, the occurrences map no longer keeps count of how many times the neural network outputted “Greece” because that guess has already been attempted.

A close examination reveals that input patterns in ‘potentials’ are ordered by their distance from the first element in the list.
Looking at the second value of the list. The first element is “[1, 0, 1, 1, 1, 1, 0, 1, 1]”. The following elements differ from that by one value. Then the elements differ by two values, and so on.

```
> python3.6 part2 user debug
Are you looking for a short stay?
No
Do penguins live there?
no
Does your dream location have long rivers?
probably
Is it on an island?
no
Is it known for its seaside?
probably not
Is your dream location historically famous?
yes
Do people there speak Spanish?
probably
potentials = [[0, 0, 1, 0, 0, 1, 1, 0, 0, 0,
```

Many potential values…….

```
1, 0, 0]]
occurrences = {'USA': 159, 'Spain': 89, 'Argentina': 11, 'Japan': 88, 'Egypt': 29, 'Italy': 60, 'Greece': 76}
Is your dream destination USA?
yes
Guessed successfully!
>
```

In this example, the system has performed many predictions in the matter of milliseconds. It has quickly determined that “USA” is the most probable answer. This example also demonstrates the increased knowledge of the system under “user” mode. It is aware of destinations such as USA and Japan, which were added when testing the implementation of part 3 of this assignment.

```
> python3.6 part3 user debug
Are you looking for a short stay?
no
Do penguins live there?
no
Does your dream location have long rivers?
no
Is it on an island?
no
Is it known for its seaside?
no
Is your dream location historically famous?
no
Do people there speak Spanish?
no
potentials = [[0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0,
 1, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1, 0
, [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1
0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1],
0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0], [0, 0
0, 0, 1, 0, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0
1, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1], [0, 0, 0, 0, 0, 0
0, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 1
0, 0], [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
, 0, 0, 0, 1, 1, 1, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1], [0, 0,
, 0, 0, 1, 1, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0], [0, 0, 0, 0,
occurrences = {'Japan': 18, 'Spain': 31, 'Italy': 3, 'USA': 8, 'Greece': 4}
Is your dream destination Spain?
no
Is it famous for its cuisine?
no
Does your dream location have peculiar architecture?
no
Is your dream destination know for its geographical shape?
no
potentials = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0,
 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
occurrences = {}
Does your dream destination contain an international wonder?
no
Does your dream destination cover a large territory?
no
Is your dream destination the birthplace of anime?
no
potentials = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
occurrences = {}
You have defeated me. Congratulations!
Would you tell me your destination? [yes/no]
no
>
```

This example shows how part 3 makes use of part 2 and, upon failing to guess the user's destination, the system asks if they would share it. It is important to note that at the second and third threshold, when the 'potentials' list is computed, the genie does not make a guess. That is because all destinations that the network outputs have already been attempted as guesses. The initial implementation halted the system once there were no options. However, that hindered the learning process. When the system cannot compute a destination for an input patter, it should use this input pattern for its learning. Therefore, it is best for it to gather as many answers as it can.

```
> python3.6 part3 user
Are you looking for a short stay?
probably
Do penguins live there?
no
Does your dream location have long rivers?
maybe
Is it on an island?
no
Is it known for its seaside?
no
Is your dream location historically famous?
no
Do people there speak Spanish?
no
Is your dream destination Spain?
no
Is it famous for its cuisine?
no
Does your dream location have peculiar architecture?
no
Is your dream destination know for its geographical shape?
no
Is your dream destination USA?
no
Does your dream destination contain an international wonder?
no
Does your dream destination cover a large territory?
yes
Is your dream destination the birthplace of anime?
no
You have defeated me. Congratulations!
Would you tell me your destination? [yes/no]
yes
What is the name of your destination?
```

In this example, the system is running part 3 in “user” mode. It fails to guess the destination and the user agrees to share their knowledge. The system asks the user for a feature and a question to ask. It then prompts the user to enter “yes”/“no” for all known destinations. These answers are appended to the training sets. The input pattern(s), derived from the answers to the questions in the first image, are added to the training set as well and they map to the new destination.

```
What is the name of your destination?
Canada
What is its distinguishing feature?
Hockey
What question should I ask about it?
Do people there play hockey
What is the answer to "Do people there play hockey?" if your dream location was Spain?
no
What is the answer to "Do people there play hockey?" if your dream location was Greece?
no
What is the answer to "Do people there play hockey?" if your dream location was Argentina?
no
What is the answer to "Do people there play hockey?" if your dream location was Egypt?
no
What is the answer to "Do people there play hockey?" if your dream location was Australia?
yes
What is the answer to "Do people there play hockey?" if your dream location was Italy?
yes
What is the answer to "Do people there play hockey?" if your dream location was South Africa?
no
What is the answer to "Do people there play hockey?" if your dream location was USA?
yeah
Please enter "yes" or "no". Alternatively you can "quit".
yes
What is the answer to "Do people there play hockey?" if your dream location was Japan?
yes
What is the answer to "Do people there play hockey?" if your dream location was Canada?
yes
20 hidden neurons produced an error of 0.00 which is sufficiently good.
>
```

Evaluation

Overall, the system achieves all basic requirements, and extensions that allow it to record learning data, and to accept ambiguous answers (“maybe”, “probably not”, “I don’t know”). A network can be tested without the need to re-train it first. Interaction with the genie is well supported, as it allows the user to enter various spellings of recognized inputs.

The system can be improved by altering the method for expanding the training set in part 3. Currently it prompts the user to enter all the data, which is infeasible for a large array of destinations. The method was preferred to setting all values to “yes” or all to “no” (or randomly assigning them). The classification would be perfect if there is a 1 to 1 mapping between a feature and a destination. However, that seems unrealistic and takes away from the value of asking broader questions. The genie is more impressive if it guesses the destination by asking about features shared by many destinations. A better algorithm would be able to put consistent answers to questions about such features without asking the user multiple times.

In addition to this evaluation, the training of a network has also been observed. The results and conclusions are described in the appendix to this report.

Running

The system can be run in many ways. It was developed using Python3.6. The library ‘scikit-learn’ was used for creating and backpropagation training of a neural network. The installation on the lab machines is version 19.2, which is missing some functionality described in the lectures. If there are problems in running the system, the version of ‘scikit-learn’ should be checked first with “pip3.6 freeze | grep scikit”. The updated version 20.0 can be installed via "pip3.6 install -U scikit-learn –user". This will fix the error “n_iter_no_change is not a recognized parameter”.

The usual commands for running the system are “python3.6 part1”, “python3.6 part2”, “python3.6 part3 user”. All modules, except for ‘reporting’, can be run from the command line. The command template is “python3.6 partX/module.py [command line arguments]”.

The three packages contain “__main__.py” files which makes the packages themselves executable. The command template is “python3.6 partX”. This will run the “__main__” module inside the package. That module is simply a wrapper for one of the other modules. It makes commands shorter.
“python3.6 part1 [args]” acts the same way as “python3.6 part1/network.py [args]”
“python3.6 part2 [args]” acts the same way as “python3.6 part2/genie.py [args]”
“python3.6 part3 [args]” acts the same way as “python3.6 part3/learning_genie.py [args]”

The packages also include “__init__.py” files which are used by the Python interpreter to recognize the folders as Python packages. In this system, the “__init__” modules are empty, as no setup is needed.

The most common command line arguments are “extra” and “user” to run the system in the respective mode of execution. If neither is supplied, the system runs in “basic” mode. If mode are

typed, the system will prioritize “user” mode. Only the “network” module can accept additional command line arguments. Other modules will ignore them. Below are all possible commands:
“python3.6 part1/datasets.py” – load and display the data sets for “basic” mode
“python3.6 part1/datasets.py extra” – load and display the data sets for “extra” mode
“python3.6 part1/datasets.py user” – load and display the data sets for “user” mode
“python3.6 part1/network.py” – run a performance check on the “basic” network
“python3.6 part1/network.py extra” – run a performance check on the “extra” network
“python3.6 part1/network.py user” – run a performance check on the “user” network
“python3.6 part1/network.py train” – train the “basic” network and then check performance
“python3.6 part1/network.py train extra” – train the “extra” network and then check performance
“python3.6 part1/network.py train user” – train the “user” network and then check performance
“python3.6 part1/network.py train chart” – train the “basic” network while saving learning progress to a file and then check performance
“python3.6 part1/network.py train extra chart” – train the “extra” network while saving learning progress to a file and then check performance
“python3.6 part1/network.py train user chart” – train the “user” network while saving learning progress to a file and then check performance
“python3.6 part2/genie.py” - play the guessing game with the genie in “basic” mode
“python3.6 part2/genie.py extra” - play the guessing game with the genie in “extra” mode
“python3.6 part2/genie.py user” - play the guessing game with the genie in “user” mode
“python3.6 part3/learning_genie.py” - is equivalent to “python3.6 part2/genie.py” because learning genie only learns in user mode
“python3.6 part3/learning_genie.py extra” - is equivalent to “python3.6 part2/genie.py extra” because learning genie only learns in user mode
“python3.6 part3/learning_genie.py user” - play the guessing game with the genie in “user” mode then, optionally, teach it about a new destination and/or feature

The “debug” command line argument can be added when executing part 2 or 3, to print some of the variables to the screen. For example, “python3.6 part2 user debug” will run the system in “user” mode and cause a couple of variables to be printed when making guesses.

Note: all commands should be executed from within the submission directory, so that the system has access to “assets/” and the Python packages (“part1”, “part2”, “part3”).

Literature Review

Artificial Neural Networks are applicable to wide arrays of problems. ANNs model the brain[1], they can be trained to simulate mapping functions between input domains and output domains. Thus, they are a reasonable solution to any problem that involves predicting data based on past data. With a well designed network, it is even possible to solve the infamous Travelling Salesman Problem[1].

The authors of [1] examine multiple types of ANNs and their applications. Pattern Recognition is the most popular. It encompasses image recognition, voice recognition, and even this assignment is a patter recognition to some extent.

Pattern Recognition is also the area explored by [3]. A type of ANN called Cellular Neural Network combines neural networks and cellular automaton to achieve high speed image processing.

In chemistry, neural networks are used to predict how different elements will interact with each other[2]. A well designed and trained network can predict interactions between complex compound molecules. This can be used in designing new types of medicines.

Despite being a somewhat old concept, neural networks are still improving. This [4] article discusses a recent breakthrough which resulted in the creation of a quantum neural network. It has *"exponential advantage over classical model"* which makes it much faster and more robust than classical neural networks.

Word count to here: 5743.

Bibliography

[1] Theory and Applications of Neural Networks for Industrial Control Systems
Toshio Fukuda, Takanori Shibata
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=170966

[2] Applications of Neural Networks in Quantitative Structure – Activity Relationships of Dihydrofolate Reductase Inhibitors
T. A. Andrea, Hooshmand Kalayeh
https://pubs.acs.org/doi/pdf/10.1021/jm00113a022

[3] Cellular Neural Networks: Applications
Leon O. Chua, Lin Yang
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7601

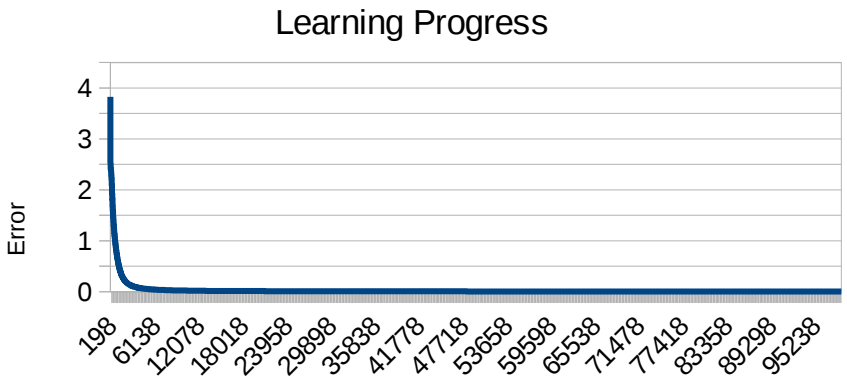
[4] Breakthrough neural network paves the way for quantum AI
Tristan Greene
https://thenextweb.com/artificial-intelligence/2018/11/19/breakthrough-neural-network-paves-the-way-for-quantum-ai/

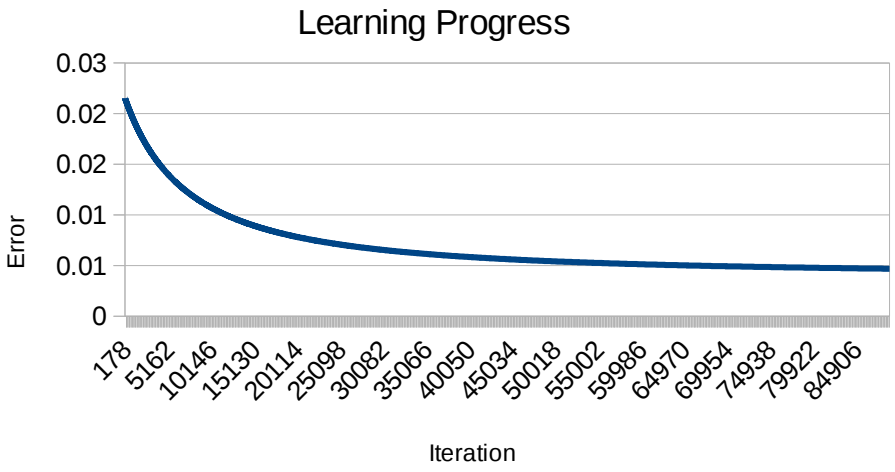
Appendix

As an extension to the evaluation of this assignment, the output during training of a neural network was closely examined. As expected, the initial learning rate and the momentum are the key elements to how quickly the network converges. It was also discovered that networks with fewer hidden neurons converge faster. This seemed counter intuitive but, in fact, fewer neurons mean fewer connections and less weights between neurons. So, there is less computation happening at each training iteration.

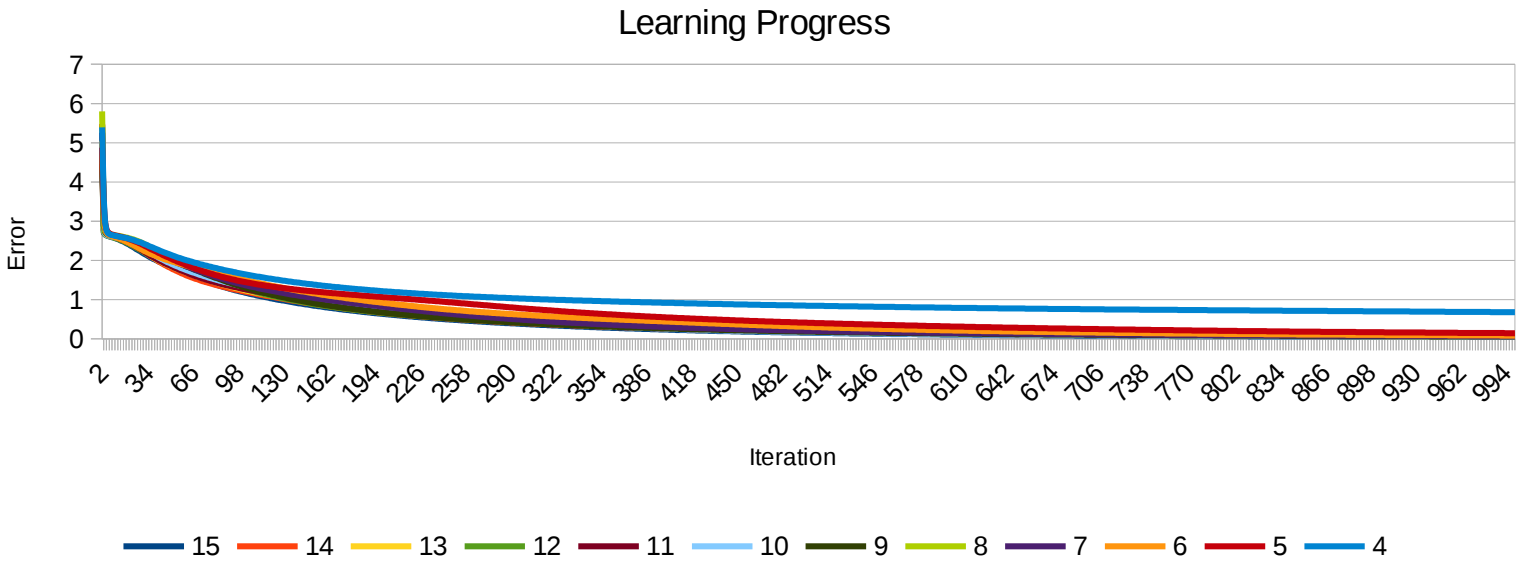
It was also discovered that training with an iteration limit of more than 100 000 is unnecessary. Every time that boundary is exceeded, the training fails to produce an error lower than needed. The file “assets/charts/user_0.1_0.4_1000000.ods” shows the results of training 14 networks with different amounts of hidden neurons (from 2 to 15) and after 1 million iterations they have failed to converge with a low error. Note that this file is very large and may consume a lot of computer resources to open. The experiments with the “basic” and “extra” neural networks trainings can be repeated to similar results. The training data for “user” has changed, so new experiments will likely produce different results.

Below are some chart and conclusions.





These two charts are from basic_0.1_0.4_200000. Training has completed after 98 677 iterations. They show that the fist 10 000 iterations have the highest impact on the network. Afterwards, training slows. The zoomed-in section from 10 000 to 98 677 shows that progress slows even more until it is barely making any in the last few hundred iterations.



This chart (“user_0.8_0.2_1000”) shows that a high learning rate produces a different curve. At start, there is a bump where the training produces more errors in iteration counts 40-50 than in iterations 25-35. It does eventually smooth out thanks to the momentum.

This chart also shows that networks with low amounts of hidden units (4) often get stuck at a high error rate and cannot find a good fit. This happens when there are many input and output patterns.