CS5011: Assignment 1 – Artificial Neural Networks

170024030 Xingzhi Yue

List of the implemented parts and extensions

Part 1:

1. Coding the input (0 for no and 1 for yes) and outputs (one-hot encoding).
2. Construct a training table with the encoding scheme mentioned in a) for the input/output of the dataset.
3. A multilayer feedforward neural network is trained, with one hidden layer and sigmoid activation function. Input and output units are fixed as seven and five for this part.
4. The number of hidden units is determined to be three.
5. Train the network using backpropagation, with learning rate as 0.3, learning momentum as 0.3, and error threshold as 0.01.
6. Save the trained network.
7. Test the network by a set of inputs and compare the output of the training set.
8. Try different numbers of hidden units and analyze the results.

Part 2:

1. A system is designed to ask a series of questions corresponding to the features expecting a yes/no answer.
2. Use the saved network in Part 1 to compute the output of the given input.
3. Map the calculated output to a corresponding character, also considering the case when the user gives a pattern that does not exist

in the training set.

4. Show the guess after all the questions. Also, to make early guess if possible.
5. Add a new character and examine the changed parameters for the network.

Part 3:

1. The case in which the answers are yes/no/maybe is considered.
2. The system is modified to allow the user to add a new character from the user's given pattern. Training table is modified accordingly and the network is retrained to let the user play another game with updated dataset.

Literature review

Artificial neural networks (ANN), a parallel, distributed structure to process information (Hecht-Nielsen, 1988), is one of the classifier systems that are widely used nowadays (Byvatov, et al, 2003) and dates back to the late 1950's (Rosenblatt, 1958). A standard ANN consists an input layer, an output layer, and one or more hidden layers, with many neurons or nodes that trigger input activations to influence connected neurons in the next layer (Schmidhuber, 2015), and the learning is essential to find weights that connect neuron to make the ANN do desire task. And backpropagation is the most widely applied approach to train the ANN (Hecht-Nielsen, 1988) that back-propagates the observed error in the output layer, calculates how each node is "responsible" for error and then adjust the weights given a fixed structure (Russel, et al, 1995). ANNs can be divided into feedforward and recurrent classes according to the way nodes are connected (Yao, 1999). The ANN we applied to solve this "Guess Who" problem is feedforward since all of the connections are one-way, from the input-side to the output-side. There are mainly four general learning

algorithms: supervised learning, reinforcement learning, unsupervised learning and semi-supervised learning. This “Guess Who” game is basically a supervised-learning classification problem and its implementation is stated as followed.

Part 1

Design

1. Input and output coding

In the first part, designing and training the neural network to learn the classification patterns, there are seven features for each character and five different characters.

For the input, yes = 1, no = 0 are applied as the coding scheme which is quite natural for such true/false description.

As for the output, since the activation function is a sigmoid function with output in the range between -1 and 1, the coding scheme can only be chosen between one-hot coding and binary coding that both have their pros and cons. For the binary coding scheme, its strength is to deal with massive amount of data that has better space efficiency. On the other hand, it is more convenient to set the threshold for the one-hot output data. Especially in this ‘Guess Who?’ game, it can also give a rank of the likelihood for each possible character.

2. Construct training table

I split the training table into an input table and an output table, and manually put them into the code instead of reading from a coded CSV file, because the relative path for java will be quite messy once it is exported as jar file.

For the purpose of reusability, I store the input table, output table, questions and character’s names in four array list so that it can be updated and resized.

3. Construct the feedforward neural network

The numbers of input units and output units are set to be seven and five

respectively, corresponding to the number of features and the number of characters. A single hidden layer is also included in the network.

4. Hidden units and error threshold

The number of hidden units in related to the complexity of the neural network which depends on the problem. In this project, the error threshold is set to be 1%. The following parts of this section use 3 as the number of hidden units. The method to decide the number of hidden units is discussed in 8.

5. Training the network

To find the appropriate learning rate and momentum, a series of parameter pairs are tested (Figure 1).

In figure 1a, the curve shows an undesirable fluctuation at the beginning which is because the learning rate is too high and the errors are oscillating. In figure 1b, due to the low momentum, the error approaches the threshold quite slowly after 26 steps. By adding 0.3 momentum to it (Figure 1c), the network takes 20 steps less than that in figure 1b without error fluctuation. Hence, we use 0.3 and 0.3 for rate and momentum in the following implementation.

6. Save the network

After training, the network is saved for the usage in the parts that follows

7. Test the network

Once the training is finished, a set of example inputs with known corresponded output is used to test the network.

8. Appropriate number of hidden units

To start with, I use the rule of thumbs for 1 hidden layer that the number of hidden units should be in the range between the numbers of input units and output units. Therefore, I started from 6 hidden units, and it converges quite swiftly. Then I prune the hidden units one by one to see how few it takes to ensure the error to converge. And the result is when the hidden units are less than 2, the error can never converge. Therefore, I set 3 as the number of hidden

units. Detail is discussed in the evaluation section.

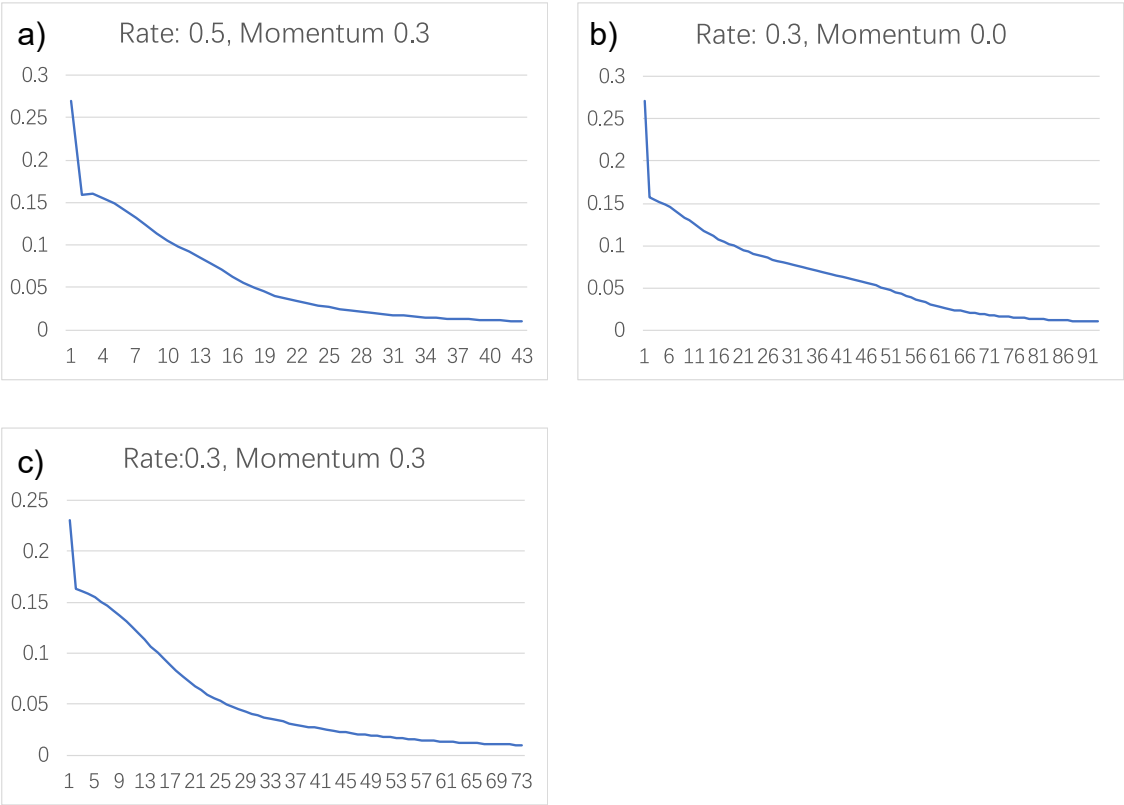

Figure 1. Training curve for this problem. The X axis labels the step counts and the Y axis labels the error. a) the learning rate is 0.5 and the momentum is 0.3; b) the learning rate is 0.3 and the momentum is 0; c) the learning rate is 0.3 and the momentum is 0.3.

Examples and Testing

Once the network is trained, a sample set of input-data {0,1,0,1,0,0,0}, marking the feature of Alex {1,0,0,0,0} in the training table, is used to test the network. And the result is `{0.8172081022671263, 0.10964401227299723, 0.12738367757569322, 0.0011459249542406164, 0.06006023226367826}`, where the first unit is significantly greater than the rest.

To extend the test of the generated network, the data is divided into two parts: first 40% as the testing set, and the rest as the training set.

“Learning1_60.java” ” in the “Learning1” project use the last 21 pairs of

input-output data as training set and network generated by this is named by “network_60.eg”. Then, “Test.java” test the saved network with first 14 input-output pairs as the testing set.

The classification accuracy for this network tested by the other 40% as the testing set is also 100%. Therefore, it is safe to conclude that the training for this part is implemented properly.

Evaluation

The most important problem during training is to decide how many units should be put in the hidden layer. For some unknown reasons, the built-in “PruneIncremental” method does not work with one-hot coding. Therefore, I started from 6 units and manually prune the hidden units (Table 1).

For each number of hidden units, I train the network five times to minimize the bias caused by different starting weights. The minimum number of hidden units is three, however, the number of steps to converge seems to increase with less hidden units. This gives rise to another question: are the extra hidden units simply redundancy? Or they are still beneficial for the network.

Table 1. The number of steps that is needed to converge with different hidden units with same training table. When the number of hidden units is less than three, the error won’ t converge.

#HIDDEN UNITS	6	5	4	3
STEPS TO CONVERGE	33	29	44	52
	32	39	40	43
	36	34	37	48
	36	29	38	53
	33	35	34	58
AVERAGE STEPS	34	33.2	38.6	50.8

Learning rate = 0.5, Momentum = 0.3

One way to consider this problem is to adopt the Ockham’s razor (Russel, et al, 1995) which prefer the simplest possible hypothesis. Although extra

hidden units seem to promote the training progress, it may also cause overfitting that ruins the generalization.

Further analysis can focus on comparing the difference between one-hot and binary coding scheme, and how it affects the required number of hidden units.

Running

Part 1 includes three jar files.

To train the network with all of the data, use

"java -jar Learning1.jar",

which saves the network as "my_network.eg".

To test the training method by training the network with 60% of the data, use

"java -jar Learning1_60.jar",

which saves the network as "network_60.eg".

And then use

"java -jar Test.jar",

which gives the classification accuracy.

Part 2

Design

1. Ask the questions

"Learning2.java" implements a series of methods that ask a set of questions to get the features of the characters. The user can answer each of the questions by input "yes" or "no" in the terminal. The agent will transform these inputs into

input-coding as is stated in part 1, and then store them.

2. Use the neural network

With input coming in, the agent then loads the neural network that is trained in part 1 and uses it to compute the output of the given input pattern.

3. Map the outputs to the characters and new pattern

The outputs calculated by the neural network are five numbers, all between 0 and 1. To make sense of these outputs, the agent also implements a method to map the outputs to a character. This method follows several rules:

I. If there is only one output unit that is greater than 0.5, set it to be 1 and others to be 0;

II. If there are more than one output unit that are greater than 0.5, set the greatest one to be 1 and others to be 0;

III. If all of the output units are less than 0.5, set the greatest one to be 1 and others to be 0.

Then match these five numbers to a specific character following the one-hot output coding scheme.

If the user gives a pattern that does not exist in the training data, the system will still guess the character which is most possible. Detail is discussed in the evaluation section.

4. Early guess

The final guess is shown by the agent after all the questions are answered by the user.

Early guess making is also implemented in “Learning2.java”. For example, in table 2, first four out of seven question is answered by the user, whereas the other three is not. To make an early guess, every time the user answers one of the questions, the agent completes the input with a positive assumption and a negative assumption (all yes and all no for the questions to be answered). Afterwards, two sets of outputs are computed by the two assumed sets of inputs to compare with each other. If the two sets of outputs point to the same character, make early guess.

Table 2. Early guess inputs with four solid answers and three positive or negative expectation.

Features	**Curly hair**	**Blonde**	**Red cheek**	**Moustache**	**Beard**	**Ear ring**	**Female**
Status	Answered				Not answered		
Positive	yes	no	no	yes	yes	yes	yes
Negative	yes	no	no	yes	no	no	no

5. Add more characters and features

In “Learning2_addChara.java”, a new character named Thomas is added in the name list and give all negative answers, with input coded as {0,0,0,0,0,0,0}. Consequently, the number of output units is increased by one, and all the original outputs are extended by a “0” unit. The output coding for this new person is {0,0,0,0,0,1} since he is the sixth character in the data set. Therefore, the number of input units is unchanged if no feature is added; the number of output units should match the total number of characters according to the one-hot coding scheme; as for the hidden units, three hidden units is enough to handle this problem so it should also remain unchanged. Furthermore, I test different situations with up to six more characters added to the training table, and it doesn’t require more hidden units to converge.

In “Learning2_addFeat.java”, a new feature “left-handed” is added with corresponded questions to the question list. The number of input units should increase by one to match the total number of questions (features); the number of output units remains unchanged, as well as the number of hidden units.

To sum up, the number of inputs increase with extra features, and the number of outputs and extra characters alike. The hidden units can remain unaltered with small amounts of extra features and characters, which needs further study that how many inputs and outputs it takes to increase the complexity of this problem.

Examples and testing

In order to test the performance of this system, we used the input patterns in table 3, to see if it can make appropriate early guesses. As is shown, the average required questions for the system to make early guess giving the training data is around 4 and 5 out of 7 questions.

Table 3. Testing the early guess with first five patterns. Gray area marks the needed features to give a valid early guess.

Name	*Curly hair*	*Blonde*	*Red Cheecks*	*Moustache*	*Beard*	*ear rings*	*female*	*Correctness*
Alex	No	Yes	No	Yes	No	No	No	✓
Alfred	No	No	No	No	Yes	No	No	✓
Anita	No	Yes	Yes	No	No	No	Yes	✓
Anne	Yes	No	No	No	No	Yes	No	✓
Bernard	No	No	Yes	No	No	No	No	✓

Evaluation

The system basically uses positive and negative assumptions to look for valid early guess according to the threshold listed in 3. For patterns that exist in the training data, the early guess will work properly and find the character with less than 6 known features.

However, if the pattern does not exist in the training data, for example, an all-no pattern, the output is {0.02789449297922685, 0.04416097601316746, 0.08277379722707116, 0.1513262723757427, 0.1773909385624779}. Since none of the output units are greater than 0.5, the system will not give a valid guess. To tackle this problem, the system will rank the possibilities for each character by the value of output units. For this all-no pattern, the last output unit gives the greatest value, therefore the system will guess the last character Bernard and let the user judge if it's correct. If it's not, guess the character represented by the next biggest one in the rest of output values, so that the

system can somewhat act humanly as one of the definitions of AI.

For the game with an extra character, even if the given pattern exactly matches that of the new character, it is still very difficult to make the correct guess. My hypothesis is that only one set of patterns is learned for the new characters whereas each of the other five characters have seven patterns. Therefore, I tried to duplicate 5 all-no pattern for in both input and output data, then the system can recognize this new character properly.

The system can be improved by recognizing absolute input judgments other than “yes” or “no”, such as “yeah”, “sure thing”, “negative” and so on. Also, sometimes the system give incorrect early guess, depending on the network. The threshold for whether to make early guess or not also needs to be further analyzed.

Running

Part 2 includes three jar files.

To play the game with saved network “my_network.eg”, use

“java -jar Learning2.jar”,

To play the game with an extra character, use

“java -jar Learning2_addChara.jar”,

To play the game with an extra feature, use

“java -jar Learning2_addFeat.jar”.

Part 3

Design

1. Consider “maybe” answers

Methods are implemented to deal with user’s answer, “maybe”. An array-list is

used to store these “maybe” features. During the early guess, these “maybe” features are treated somewhat like the not-answered features to generate inputs with positive and negative assumptions. For example, if the user answers the first four questions as “yes”, “maybe”, “no” and “maybe”, the assumed inputs for early guess is generated as shown in table 4.

Table 4. Early guess generated from inputs with four answers including two maybes.

Features	**Curly hair**	**Blonde**	**Red cheek**	**Moustache**	**Beard**	**Ear ring**	**Female**
Status	Answered				Not answered		
	yes/no	maybe	yes/no	maybe			
Actual	yes	maybe	no	maybe			
Positive	yes	yes	no	yes	yes	yes	yes
Negative	yes	no	no	no	no	no	no

However, if the early guess part does not work out, the agent will set inputs, corresponding to “maybe” features, to be 0.5 to calculate a possible character.

2. Add new characters including “maybe”

The system also implements methods to add a new character when the agent cannot guess this character. The user will be asked to input this character’s name, and then the features that the user answered formerly will be added into the dataset pairing the given name just as discussed in part 2.5.

However, the problem becomes slightly more complicated if we are adding a new character with some “maybe” features. There are two ways to do this. The first approach is done by following these procedures:

I. Add the name of this new character to the end of the name list.
II. Count the number of “maybe” features of this character (N).
III. Extend all of the current output data by adding a “0” unit to their end. For example, {0,0,0,0,1} should be altered to be {0,0,0,0,1,0}.
IV. Add 2^n sets of output data representing this new character. For example, if this character is the sixth character, the output coding

should be {0,0,0,0,0,1}, according to the one-hot coding scheme.

V. Generate 2^n sets of input data that includes all the possibilities to give either yes or no for “maybe” features. Then add them to the current input data-set.

The second approach simply uses 0.5 to represent “maybe” in the training table. In this system, the first method is implemented.

In “Learning3.java”, the system will ask if the user want to add a new feature or character after each game.

In “Learning3_2.java”, the system will only attempt to add a new character when the pattern cannot be recognized. New character will be automatically initialized with the pattern entered by the user during the game.

Note that, for both versions, do not give the new character more than 2 “maybe” features, or the error may not converge due to high complexity.

Example and testing

Several patterns that include “maybe” is used to test this system (Table 5). All of the test examples gives correct characters even with 3 “maybe”s out of 7 features for Bernard.

Table 5. Testing the system with last five patterns in the learning set (some of the features is altered by maybe). Gray area marks the needed features to give a valid early guess.

Name	*Curly hair*	*Blonde*	*Red Cheecks*	*Moustache*	*Beard*	*ear rings*	*female*	*Correctness*
Alex	No	Maybe	No	Yes	Yes	No	No	✓
Alfred	Maybe	No	No	Maybe	Yes	Yes	No	✓
Anita	No	Maybe	Yes	No	No	Maybe	Yes	✓
Anne	Maybe	No	No	No	No	Yes	Maybe	✓
Bernard	No	Maybe	Yes	Maybe	No	Maybe	No	✓

Evaluation

The system performs very well with patterns that include “maybe” (Table 5).

As for the two methods to add a character with “maybe” features, the first method has a problem that it may increase the size of the training table enormously, however, it makes more sense to allow multiple inputs to give the same output as in the original training table. For the second method, the problem is that networks generated by this method may not generalize properly given the binary yes-or-no input for this person. Therefore, the first method is implemented in this part.

Further improvements for this system include giving access for the user to directly altering the database, automatically enhance the learning for the added cases, as is discussed in part 2.

Running

Part 3 includes two jar file.

To play the “Guess Who 1.0” game, use

“java -jar Learning3.jar”,

To play the “Guess Who 2.0” game, use

“java -jar Learning3_2.jar”.

(Word count: 3498)

Reference:

Byvatov, E., Fechner, U., Sadowski, J., & Schneider, G. (2003). Comparison of support vector machine and artificial neural network systems for drug/nondrug classification. *Journal of chemical information and computer sciences*, *43*(6), 1882-1889.

Hecht-Nielsen, R. (1988). Theory of the backpropagation neural network. *Neural*

Networks, *1*(Supplement-1), 445-448.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Russell, S., Norvig, P., & Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, *25*, 27.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, *61*, 85-117.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423-1447.