System documentation Back-end

Library Watson

Introduction

In this document the architecture and design choices of the back-end of the Library Watson chatbot project are explained in different sections based on the folder structure of the project. The files in these folders are grouped on related functionality and is thus a natural way of explaining the components of the project. A short summary of the entire back-end functionality is first provided.

Summary

The back-end can roughly be divided into two major components: The chatbot algorithm which determines the answer of the chatbot and the local server for communication with the front-end.

Chatbot algorithm summary

The chatbot is designed with as core component a deep neural network. This neural network is trained on anonymised questions posed to the liveChat of the Radboud. The network can classify its input by outputting the probability of a number of tags, which each have a predetermined answer. The training data processed in different phases using natural language processing to create a better representation for training the network.

The network is trained using Adam optimization with a standard forward pass, backward pass training structure. It is furthermore possible to use validation sets and accuracy measurement to get more insights into training. There is also the possibility of training multiple models in succession while varying a single variable between each model for quick model comparison.

For testing, aside from validation and training accuracy, a quick test with some pre-selected questions can be done for multiple networks at once, as well as a simulated chat with the chatbot through the console.

All the training and testing functionality can be accessed through a main class. Training and testing parameters can also be manipulated here.

Local server summary

The back-end communicates with the front-end by hosting a local server with python that listens to a specific port. Javascript can make a request to send information to the back-end (the user question) or a request to receive information (The chatbot answer).

When the user receives an answer from the chatbot in the front-end, they can choose whether they deem the answer as correct. The data about the question answer and tag the correctness of the answer are stored in a separate file which could be used for future training, but has currently no specific application.

Programming languages

The primary programming language used for the development of the back-end is python. This choice was made because of the team’s experience with Python and because of the powerful machine learning libraries accessible in Python, primarily Pytorch. This library is a primary component in the architecture design of the chatbot.

Since the chatbot needs to interact with the user, there are some systems in place to communicate with the front-end javascript. Some local server related functionality is thus also implemented in javascript.

Use of existing systems

In agreement with the client it was chosen to implement the chatbot from scratch, instead of using an existing framework. The client was interested in receiving a chatbot from scratch and the team was willing to try and implement it. In the development the team did make use of existing libraries, which, along with their required version, are as follows:
pandas>=0.23.4
torch
Flask_Cors>=3.0.9
Flask>=1.0.2
numpy>=1.16.4
spellchecker>=0.4
pyspellchecker
nltk>=3.5
language-detector
matplotlib
This exact same list is also provided in the requirements.txt file for quickly installing these libraries.

Chatbot algorithm

Data

Representation

The training data is represented as a pattern, tag and response within an Excel file.
This Excel file is exported as .csv and read using the Pandas library and stored in a list of dictionaries containing the pattern-tag-response combinations.
We found this to be the most common way to store natural language as training data.

patterns: ["hey, how are you?", “hello”]
tag: "greeting"
response: "great, how are you?"

Data files

There are four data files:

1. trainingdata.xlsx:
   This is where all the training data is added or changed.
2. trainingdata.csv:
   This is the training data exported as a .csv file for the program to read.
3. nn_feedback_data.csv:
   This is where new “approved” training data is stored.
4. testQuestions.csv:
   Contains a few test questions for the quick_test() function.

Saving the model

The importExport.py file was created as a way to store and open a trained nn model.
After training, the network parameters and relevant data will be stored in a file, automatically set to “data.pth”.
The parameters stored are:

- trained model weights.
- input layer size.
- output layer size.
- hidden layer size.
- all unique word(-pairs).
- all unique tags
- raw word data
- nlp class settings
- output class settings

Data Processing

Natural language processing

We’ve created an NLP class which is responsible for handling language.
It’s got four main functionalities:

1. *Tokenizing*: chopping a sentence into single words.
2. *Stemming*: bringing words to their root form (“eating” -> “eat”).
3. *Checking synonyms*: get a word’s synonym list
4. *Form ngrams*: creating tuples of words (“The dog eats” -> {The dog}, {dog eats})

Initializing the training data

1. The training patterns are tokenized and stemmed in WordData.py. It creates a list of all words that occurred in the patterns, a list of all unique tags as well as a list of pattern-tag combinations called xy.
Ngrams are formed of size n (using the NLP class). We suggest using size n == 1 unless you have a very large dataset, because bigger ngrams means more unique word-pairs.

```
all_words: [("hey",), ("how",), ("are",), ("you",), ("hello",), ("bye",)]
tags:      ["greeting", "ending"]
xy:        [([("hey",), ("how",), ("are",), ("you",)], “greeting),  ([(“hello”,)], “greeting”), ...]
```

2. TrainingData.py converts these patterns to binary word occurrence arrays stored in data matrix X. When converting the word ngrams to binary data, the program checks for synonyms using the wordnet from the nltk module.
Each row in X has the same length as the input layer of the Neural network; len(all_words).
The tags are stored as Integers in array y with as range {0, len(all unique tags)}.

```
all_words: [("hey",), ("how",), ("are",), ("you",), ("hello",), ("bye",)]
xy[index]:      ([("hey",), ("how",), ("are",), ("you",)], “greeting)
X[index] =>     [1, 1, 1, 1, 0, 0]
y[index] =>     0
```

User input

The user input is handled the same way as the training data: Tokenized, Stemmed, stored as ngrams and then converted to a binary array x.
If a word(-pair) from the input sentence is not known from the training data, the information will be lost.

```
all_words:          [("hey",), ("how",), ("are",), ("you",), ("hello",), ("bye",)]
input sentence:   “How are you doing?”
processed input: [(“how”,), (“are”,), (“you”,), (“doing”,)]
x =>                [0, 1, 1, 1, 0, 0]
```

Algorithm

Neural network structure

The network is a feedforward neural network and consists of three layers:

1. Input layer with size len(all unique training words)
2. Hidden layer of size hidden_size (chosen in main.py)
3. Output layer of size len(all unique training tags)

After the computation of L1 -> L2 there’s a ReLU activation.
This is not the case for L2 -> L3 in order to preserve the output probabilities.

Output

The network outputs a list (Tensor) of probabilities for each of the tags.
The predicted_tags() function from output.py calculates the predicted tags and their probabilities, which are used to determine the response of the chatbot.

There are three different types of responses:

1. None response.
   This happens when there’s no tag with probability above the respond and guess thresholds. The bot will give a standard answer.

2. Normal response.
   When a tag has a probability > respond_threshold it’ll give the response that matches the predicted tag.

3. Guess response.

When respond_threshold > probability > guess_threshold. One or more tags will be grouped together based on the distance of their probabilities. To determine if another tag will be grouped the program checks if:

(prob[i] - prob[i+1])/prob[i+1] < guess_gap_perc_threshold

The bot will respond by saying it's unsure with each of the chosen tags as a clickable option.

Testing

Test chat

This is what you'd expect when you think of testing: you can ask a question and the bot response. It'll give you some insight by showing the predicted probability for each of the possible tags.

Quick test

An easy way to check if everything is still going right. It'll test the algorithm against the data from test_questions.csv. It'll show how many of the tags were correctly predicted and what the average predicted percentage was for the corresponding tags.

Training

Because the neural network is implemented with pytorch, it is possible to use tools from pytorch in training the model. It was decided to use a standard gradient descent approach to train, using the Adam optimizer. Aside from a standard training system, the team also developed a file to quickly test multiple models called trainingOptimizer.py. Here we give an overview of the two systems:

Single model

1. Uses Adam optimizer
    a. This yields faster training results compared to standard gradient descent and stochastic gradient descent with on par performance.
2. basic training with gradient descent
    a. Training runs for a certain number of iterations or epochs
    b. The data is randomly divided into subsections called batches, which are looped through in each epoch. In each batch the following happens:
        i. forward pass -> The network predicts the output based on the input.
        ii. backward pass -> adjust parameter values with the Adam optimizer in direction of the correct answers, based on the loss as determined by

the optimizer which uses the prediction in the forward pass and the true answers.

3. When training the following systems can be used:
    a. training and validation sets
        i. By using a validation set you can see when the model starts to overfit.
    b. training and validation accuracy tests
    c. training progression plots for loss and accuracy
        i. The progression of loss and accuracy for both the training and validation sets can be visualized. This makes it easier as a developer to determine training speed and critical points such as the number of epochs it takes before overfitting starts to happen.
4. There are many adjustable parameters -> This makes it easy to adjust the model, this makes it more flexible in future.

Multiple models

1. Many models can be tested in quick succession, while varying only one variable between the models.
    a. There is no consensus in science on the optimal parameters for a deep neural network. A system that quickly allows you to test multiple models allows you to more quickly reach the optimal parameters for your application.
    b. Parameters of the models are stored separate from the “real” model and can be tested and compared later.
2. The ultimate goal of this system is letting it determine the best model by itself, but this ended up being outside the scope of the project.
3. The parameters for the model can easily be accessed through the main class.

Utils

Auxiliary functions

1. directoryFinder -> reliably find parent directories (only known to work on windows), this was used multiple times throughout the code because of different file structures. and is more easily adjusted this way.
2. shortcutMaker -> Creates a shortcut for running the local server and the index.html for the front-end. This could potentially create shortcuts of more files, but this was beyond the scope of the project.

Local server

The user and the chatbot can interact through a python-based local server that serves as a bridge between Python and Javascript. Information in the form of questions by the user and an answer by the Chatbot are first sent to the local server and are then directed to either the chatbot or the user.

The server is built using the Flask Python library. With this library it is very simple to create web-based (Python) applications, but for the purpose of this project only a local application was built. When this is required however it should be fairly easy to connect this application to the web. Researching and implementing security measures for online hosting or how hosting is otherwise done at the Radboud is beyond the scope of this project.

Using Flask, the local Python server can send information from the back-end to the front-end by receiving POST requests from Javascript and can receive information from the front-end by receiving a GET request from Javascript. Information is stored in a JSON format when it is sent between the two languages.

Now follows an explanation of the question-and answer loop between the user and the chatbot that is provided by the local server:

1. The user poses a question in the chat window shown to the user through the Front-end.
2. The question is passed as a String to server.js

    a. This handles making POST and GET requests to the server by posting the request to the appropriate page at the port that the local server is currently listening to.
3. Javascript with server.js makes a Post request to the local server to send the user question.
4. The question is received by the local server implemented in chatbotLocalServer.py.
5. This calls the file answerData.py, which starts the processing pipeline of the chatbot up to and including the prediction of the answer after the probabilities of the neural network are output.
6. When answerData.py receives the output of the neural network, it is stored in its chatbotAnswerQuestion class in the following parameters:
    a. question -> String question of the user.
    b. answer -> String answer that the user will see.
    c. response_type -> type of answer provided by the chatbot pipeline. The answer can be provided in different ways depending on the probability of each answer as determined by the neural network. The type of answer determines which value of these three parameters is stored:
        i. “single” -> Only one answer has the highest probability.
        ii. “multiple” -> Low probability and multiple answers have similar values.
        iii. “none” -> No answer has high enough probability to be considered by the network.
        iv. The probabilities at which a type of answer is returned can be adjusted in main.py.
    d. response
        i. String single answer -> Single answer corresponding to “single”
        ii. List of String answers -> Multiple answers corresponding to “multiple”
        iii. None -> No Answer, corresponding to “none”.
    e. tags
        i. String tag of a single answer.
        ii. List of tags corresponding to possible answers in the response parameter.
        iii. No answer.
    f. The reasoning behind the choice of these parameters is explained further below in the “feedback handling” section.
7. When the answer is fully processed the POST request is done, Javascript does a GET request to the local server through server.py.
    a. Promises are used in Javascript in the form of .then() statements to make the GET command wait until the POST command is done processing. The rest of the code can still continue to function this way, but the answer will only be shown to the user after the chatbot has the answer ready.
8. The local server sends the answer data from chatbotAnswerQuestion to javaScript as a dictionary for easy access in javascript.
9. Javascript receives the answer data and the answer field is displayed to the user.

Feedback handling

If the chatbot is not sure of the answer (too low probability for any answer), it can return multiple options, or none if the probability for all answers is too low. In the answer data received by javascript the answer types are stored in the “response_type” field, the answers themselves in “response” field, the tags corresponding the the answers in the “tags” field. The “answer” field is what will be displayed to the user.

1. multiple answers:
    a. chatbot displays clickable buttons which when one is selected give the answer corresponding to the button, this calls the process for a single answer.
2. single answer:
    a. After the answer is displayed, a “was this answer helpful” message is displayed with yes/no buttons.
3. no answer:
    a. The chabot replies that it doesn’t understand the question.

4. After yes/no is clicked for a single answer, Javascript selects the question, response, tag and valid (1=yes,0=no) fields corresponding to this answer as a dictionary, and does a POST request to the local server via the Javascript server.js.
5. In the local server the answer data dictionary is passed to the answerData file and stored in a dictionary for this session, and written to an excel file through the databaseWrtiter called in the answerStorer class in answerData.py.