dp-agent Documentation

Release v0.1-alpha

mipt

Oct 17, 2019

Overview

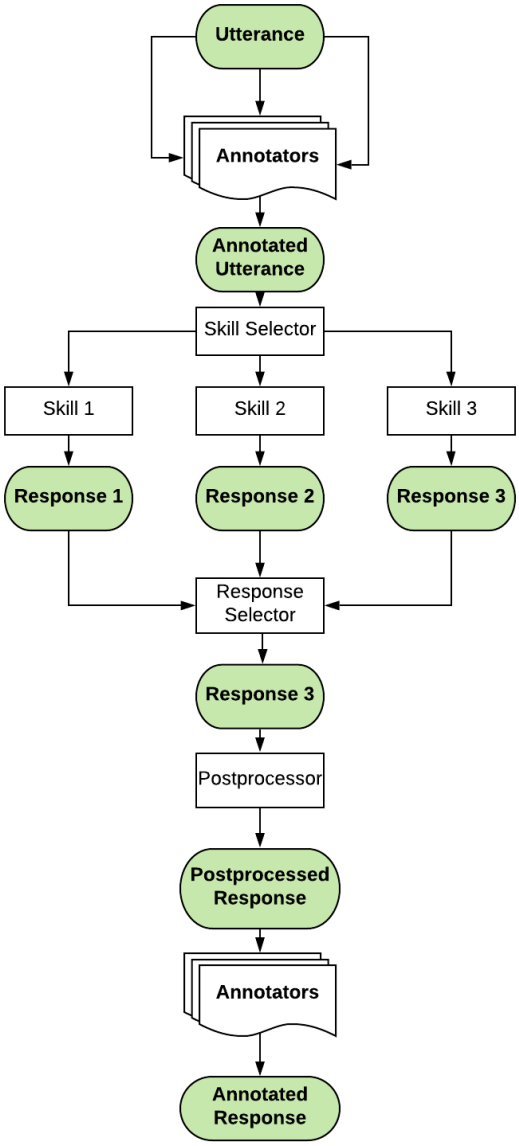
DeepPavlov Agent is a platform for creating multi-skill chatbots.

CHAPTER 1

Architecture Overview

DeepPavlov Agent helps production chatbot developers to organize multiple NLP models in a single pipeline. Modern chatbots usually solve different tasks (like chitchat, goal-oriented, question answering) simultaneously, so the platform should have the following characteristics:

- be stable at highload environment
- save and pass the chatbot state across all the connected models



Key concepts in DeepPavlov Agent architecture:

- `Utterance` is a single message produced by a human or a bot;
- `Service` is any NLP model that can be inferred as a REST service.

  There are different types of services:

  - `Annotator` is a service for utterance preprocessing. It can be some basic text preprocessing like coreference resolution, named entity recognition, spell correction, etc.;
  - `Skill` is a service producing a bot reply to a user utterance;
  - `Skill Selector` is a service choosing which bunch of available skills should be responsible for producing possible bot replies;
  - `Response Selector` is a service choosing a single bot reply from the available replies;
  - `Postprocessor` is a service postprocessing a bot utterance. It can make some basic things like adding a user name to the reply, inserting emojis, etc.;

- `Postprocessed Response` is a final postprocessed bot utterance that is shown to the user.
- `State` is current dialogs between users and a bot serialized as **json**. State is used to pass information across the services and contains all possibly needed information about the current dialogs. It has separate documentation.

CHAPTER 2

Ready Agent from the box

To demonstrate the abilities of the platform, we included in this repo some basic skills and selectors made on DeepPavlov. Currently all these services are made only for **Russian language**.

CHAPTER 3

Services Configuration

You can configure services at the Agent config file.

Config Description

Database

All default values are taken from Mongo DB documentation. Please refer to these docs if you need to change anything.

- **DB_NAME**
  - An name of the database. Default name is **"test"**
- **DB_HOST**
  - A database host, **"127.0.0.1"** by default
- **DB_PORT**
  - A database port, **"27017"** by default
- **DB_PATH**
  - A database data path. Default path is **"/data/db"**.

    Please make sure that this path exists on your machine and has valid permissions.

Services

- **name**
  - An arbitrary and unique name of the service
- **protocol**
  - A web protocol, **"http"** by default
- **host**
  - A service host machine name, **"127.0.0.1"** by default
- **port**
  - A port on a service host machine

- **endpoint**
  - A service URL endpoint, **"/skill"** by default
- **url (optional)**
  - A service url. By default it is generated from **protocol + host + port + endpoint**
- **path**
  - A path to the agent service config file, currently valid only for DeepPavlov skills
- **env**
  - Environment variables dictionary
- **external (optional)**
  - If the service is running from the **dp-agent** repo. **False** by default.
- **dockerfile (optional)**
  - Specify a dockerfile name available inside the Agent repo. **"dockerfile_skill_cpu"** by default.

  Available options:
  - **"dockerfile_skill_cpu"**
  - **"dockerfile_skill_gpu"**
- **formatter**
  - The name of a function that converts the Agent state into a service input format and converts a service output format into the Agent state
- **batch_size (optional)** A size of input batch for the services. By default it's always 1, but for neural services it is usually makes more sense to increase it for better performance.

Notice that you can leave **SKILL_SELECTORS** and **RESPONSE_SELECTORS** empty. If you do so, all skills are selected at each user utterance and the final response is selected by the skills' confidence.

Also you can include in the Agent configuration any external service running on some other machine.

CHAPTER 4

Services Deployment

1. Create a new **Python 3.7.4** virtual environment.
2. Install requirements for Docker config generator:

   ```
   pip -r install gen_requirements.txt
   ```

3. Install and configure Docker (version 19.03.2 or later) and Docker-compose (version 1.19.0 or later).
4. (optional) Install nvidia-docker if you wish to run some services on GPU.

   To be able to run GPU-based docker files please make sure about two things on your host system:

   - Your nvidia driver has to support the CUDA version installed in the GPU-based docker file.
   - Please notice that `docker-compose.yml` of **3.7** version doesn't officially support *runtime: nvidia* option anymore, so you have to manually edit `/etc/docker/daemon.json` on your system. Read in the nvidia-container-runtime documentation how to do it.

5. Create a directory for storing downloaded data, such as pre-trained models. It should be located outside the agent project's home directory.
6. Setup an **EXTERNAL_FOLDER** variable with the path to data directory. This pat will be used by Agent to download models' data:

   ```
   EXTERNAL_FOLDER=<path to data directory>
   ```

7. (optional) If you want to communicate with the Agent via Telegram, setup the following environment variables:

   ```
   TELEGRAM_TOKEN=<token>
   TELEGRAM_PROXY=socks5://<user>:<password>@<path:port>
   ```

   Here's an example of values:

   ```
   TELEGRAM_TOKEN=123456789:AAGCiO0QFb_I-GXL-CbJDw7--JQbHkiQyYA
   TELEGRAM_PROXY=socks5://tgproxy:tgproxy_pwd@123.45.67.89:1447
   ```

If you run the Agent via docker, put this variables into a file and configure it's path under `AGENT_ENV_FILE` variable in the config file. This file name is automatically picked up when the docker-compose file is being generated.

8. Configure all skills, skill selectors, response selectors, annotators and database connection in the config file.

   If you want a minimal configuration, you need only one skill.

9. Generate a Docker environment configuration by running the command:

   ```
   python generate_composefile.py
   ```

   This configuration represents the choice of skills from the previous step. Re-generate it every time you change config file.

10. Run the Docker environment with:

    ```
    docker-compose up --build
    ```

    Now you have a working environment with the following services:

    - DeepPavlov Agent (`agent`)
    - MongoDB (`mongo`)
    - A service for each skill, selector or other component.

    In this shell you will now see the logs from all working services.

CHAPTER 5

Running the Agent

Agent can run both from container and from a local machine. The default Agent port is **4242**.

5.1 Container

1. Connect to agent's container:

   ```
   docker exec -it agent /bin/bash
   ```

   (more information on docker-exec)

2. Start communicating with the chatbot from the agent's container console:

   ```
   python -m core.run
   ```

5.2 Local machine

1. (optional) Please consider setting your locale according your input language to avoid decoding errors while communicating agent via command line. For example:

   ```
   export LANG="C.UTF-8"
   export LC_ALL="C.UTF-8"
   ```

2. Setup **DPA_LAUNCHING_ENV** environment variable:

   ```
   export DPA_LAUNCHING_ENV="local"
   ```

3. Install Agent requirements:

   ```
   pip -r install gen_requirements.txt
   ```

4. Start communicating with the chatbot from the console:

```
python -m core.run
```

or via the Telegram:

```
python -m core.run -ch telegram
```

5.3 HTTP api server

1. **Run the agent api server from both container and local environment**

```
python -m core.run -ch http_client [-p 4242]
```

In both cases api will be accessible on your localhost

2. **Web server accepts POST requests with application/json content-type**

Request should be in form:

```
{
  "user_id": "unique id of user",
  "payload": "phrase, which should be processed by agent"
}
```

Example of running request with curl:

```
curl --header "Content-Type: application/json" \
     --request POST \
     --data '{"user_id":"xyz","payload":"hello"}' \
     http://localhost:4242
```

Agent will return a json response:

```
{
  "user_id": "same user id as in request",
  "response": "phrase, which were generated by skills in order to respond"
}
```

In case of wrong format, HTTP errors will be returned.

3. **Arbitrary input format of the Agent Server**

If you want to pass anything except `user_id` and `payload`, just pass it as an additional key-value item, for example:

```
curl --header "Content-Type: application/json" \
     --request POST \
     --data '{"user_id":"xyz","payload":"hello", "my_custom_dialog_id":␣
↪111}' \
     http://localhost:4242
```

All additional items will be stored into the `attributes` field of a `HumanUtterance`.

4. **Modify the default response format of the Agent server**

If you need the Agent server to return something different than `user_id` and `reponse`, try the *output formatters*.

5. **View dialogs in the database through GET requests**

   The result is returned in json format which can be easily prettifyed with various browser extensions.

   Three main web pages are provided (examples are shown for the case when agent is running on http://localhost:4242):

   - http://localhost:4242/dialogs - provides list of all dialogs (without utterances)
   - http://localhost:4242/dialogs/all - provides list of all dialogs (with utterances)
   - http://localhost:4242/dialogs/<dialog_id> - provides exact dialog (dialog_id can be seen on /dialogs page)

CHAPTER 6

Analyzing the data

All conversations with the Agent are stored to a Mongo DB. When they are dumped, they have the same format as the Agent's state. Someone may need to dump and analyze the whole dialogs, or users, or annotations. For now, the following Mongo collections are available and can be dumped separately:

- Human
- Bot
- User (Human & Bot)
- HumanUtterance
- BotUtterance
- Utterance (HumanUtterance & BotUtterance)
- Dialog

To dump a DB collection, make sure that `mongoengine` is installed:

```
pip install mongoengine==0.17.0
```

Then run:

```
python -m utils.get_db_data [collections]
```

For example:

```
python -m utils.get_db_data Dialog User
```

CHAPTER 7

Testing HTTP API and automatic processing of predefined dialogs

In order to process predefined dialogs or generate a random one from predefined list of phrases you can use *utils/http_api_script.py* script.

Make sure that `aiohttp` is installed:

```
pip install aiohttp==3.5.4
```

Processing a predefined dialog

In this mode the script will pass predefined dialogs from the file `-df` to the agent's API.

1. Create a JSON file with a dialog. You can find an example in `utils/ru_test_dialogs.json`:

   ```
   {
       "uuid1": ["phrase1.1", "phrase1.2", "..."],
       "uuid2": ["phrase2.1", "phrase2.2", "..."],
       "uuid3": ["phrase3.1", "phrase3.2", "..."],
   }
   ```

2. Run:

   ```
   python utils/http_api_test.py -u <api url> -df <dialogs file path>
   ```

3. The command line arguments are:

   - -u –url - url address of the agent's API
   - -df –datafile - path to a file with predefined dialogs

Processing a random dialog from predefined phrases

In this mode the script will generate `-uc` dialogs with `-pc` phrases in each. Phrases will be selected randomly from the phrase file `-pf` and passed to the agent's API.

1. Create a file with sample phrases. This is a simple text file with one phrase per line.

   You can find an example in `utils/ru_test_phrases.txt`:

2. Run:

```
python utils/http_api_test.py -u <api url> -pf <phrases file path> -uc
↪<user count> -pc <phrase per dialog count>
```

3. The command line arguments are:

   - -u –url - url address of the agent's API
   - -pf –phrasefile - path to a file with predefined sample phrases
   - -uc –usercount - number of users taking part in the dialogs
   - -pc –phrasecount - number of phrases in each dialog

CHAPTER 8

Testing Agent in a batch mode

To test how the Agent replies if it receives a list of utterances, use `utils/agent_test.py`. Pass a file with a list of utterances as input. Use the existing `utils/ru_test_phrases.py` or create your own file:

```
python utils/agent_batch_test.py utils/ru_test_phrases.py
```

There are 5 types of dialog services that can be connected to the Agent's dialog pipeline:

- **Annotators**
- **Skill Selector**
- **Skills**
- **Response Selector**
- **Postprocessor**

CHAPTER 9

Input Format

All services get a standardized Agent State as input. The input format is described here.

To reformat Agent State format into your service's input format, you need to write a **formatter** function and specify it's name into the Agent's config file. You can use our DeepPavlov formatters as example.

CHAPTER 10

Output Format

All services have it's own specified output format. If you need to reformat your service's response, you should use the same formatter function that you used for the input format, just use the `mode=='out'` flag.

CHAPTER 11

Annotator

Annotator should return a free-form response.

For example, the NER annotator may return a dictionary with `tokens` and `tags` keys:

```
{"tokens": ["Paris"], "tags": ["I-LOC"]}
```

For example, a Sentiment annotator can return a list of labels:

```
["neutral", "speech"]
```

Also a Sentiment annotator can return just a string:

```
"neutral"
```

CHAPTER 12

Skill Selector

Skill Selector should return a list of selected skill names.

For example:

```
["chitchat", "hello_skill"]
```

CHAPTER 13

Skill

Skill should return a **list of dicts** where each dict ia a single hypothesis. Each dict requires `text` and `confidence` keys. If a skill wants to update either **Human** or **Bot** profile, it should pack these attributes into `human_attributes` and `bot_attributes` keys.

All attributes in `human_attributes` and `bot_attributes` will overwrite current **Human** and **Bot** attribute values accordingly. And if there are no such attributes, they will be stored under `attributes` key inside **Human** or **Bot**.

The minimum required response of a skill is a 2-key dictionary:

```
[{"text": "hello", "confidence": 0.33}]
```

But it's possible to extend it with `human_attributes` and `bot_attributes` keys:

```
[{"text": "hello", "confidence": 0.33, "human_attributes": {"name": "Vasily"}
↪,
"bot_attributes": {"persona": ["I like swimming.", "I have a nice swimming␣
↪suit."]}}]
```

Everything sent to `human_attributes` and `bot_attributes` keys will update *user* field in the same utterance for the human and in the next utterance for the bot. Please refer to user_state_api to find more information about the **User** object updates.

Also it's possible for a skill to send any additional key to the state:

```
[{"text": "hello", "confidence": 0.33, "any_key": "any_value"}]
```

CHAPTER 14

Response Selector

Unlike Skill Selector, Response Selector should select a *single* skill responsible for generation of the final response shown to the user. The expected result is a name of the selected skill, text (may be overwritten from the original skill response) and confidence (also may be overwritten):

```
{"skill_name": "chitchat", "text": "Hello, Joe!", "confidence": 0.3}
```

Also it's possible for a Response Selector to overwrite any `human` or `bot` attributes:

```
{"skill_name": "chitchat", "text": "Hello, Joe!", "confidence": 0.3, "human_
↪attributes": {"name": "Ivan"}}
```

CHAPTER 15

Postprocessor

Postprocessor has a power to rewrite a final bot answer selected by the Response Selector. For example, it can take a user's name from the state and add it to the final answer.

It simply should return a rewritten answer. The rewritten answer will go the `text` field of the final utterance shown to the user, and the original skill answer will go to the `orig_text` field.

```
"Goodbye, Joe!"
```

CHAPTER 16

User State API

Each utterance in a **Dialog** is generated either by a **Human** or by a **Bot**. To understand, which of two has generated the utterance, refer to the `user.user_type` field:

```
"utterances": [{"user": {"user_type": "human"}}]
```

A Skill can update any fields in **User** (**Human** or **Bot**) objects. If a **Skill** updates a **Human**, the **Human** fields will be changed in this utterance accordingly. If a **Skill** updates a **Bot**, the **Bot** fields will be changed in the *next* (generated by the bot) utterance.

Each new dialog starts with a new **Bot** with all default fields. However, the **Human** object is updated permanently, and when a **Human** starts a new dialog, the object is retrieved from a database with all updated fields.

The history of all changes made by skills to users can be looked up at the list of possible responses in the `hypotheses` field of a human utterance:

```
"utterances": [{"user": {"user_type": "human"}, "hypotheses": []}]
```

CHAPTER 17

/start

To start a new dialog send **"/start"** utterance to the bot.

Formatters are the functions that allow converting the input and output API of services into Agent's API. In the provided example configuraton file you can find that each service has its own formatter function:

```
{
"name": "odqa",
"formatter": odqa_formatter
}
```

CHAPTER 18

DeepPavlov Formatters

The pre-built DeepPavlov formatters exist for demonstration purposes. These formatters have three attributes:

- **payload**

  If `mode==in` **payload** is a batch of input states (dialogs) to the model. If the model can't accept a list of dialog dictionaries, they can be formatted into something else, for example into a batch of last utterances from each dialog.

  If `mode==out` **payload** is the result returned by the DeepPavlov model. Unlike `in` mode, here we format a *a single element* of batch results returned by the model. The same formatting is applied to all other elements of the batch. Here the result should be formatted according to the Agent's *Services HTTP API*.

- **model_args_names**

  Should be the same names as the particular DeepPavlov model config accepts.

- **mode**

  Can be `in` or `out`. In the `in` mode we format everything that goes from the Agent to the service. In the `out` mode we format everything that goes from the service to the Agent.

CHAPTER 19

Output Formatters

Output Formatters allows regularization of what the Agent's HTTP server can return. By default the Agent's server returns only a bot utterance and a user id:

```
{
  "user_id": "same user id as in request",
  "response": "phrase, which were generated by skills in order to respond"
}
```

But if you need the server to return some additional information, for example the name of the active skill, you can do the following:

- Edit the code in the `state_formatters/http_debug_output_formatter()` method. It accepts the whole Agent's state as `payload` argument, so anything available in the state can be extracted from `payload`.
- Run `run.py` with `debug==True` option.