dp-agent Documentation

Release v0.1-alpha

mipt

Jun 04, 2020

Overview

DeepPavlov Agent is a framework for development of scalable and production ready multi-skill virtual assistants, complex dialogue systems and chatbots.

CHAPTER 1

Architecture Overview

Modern virtual assistants such as Amazon Alexa and Google assistants integrate and orchestrate different conversational skills to address a wide spectrum of user's tasks. **DeepPavlov Agent** is a framework for development of scalable and production ready *multi-skill virtual assistants*, complex dialogue systems and chatbots.

Key features:

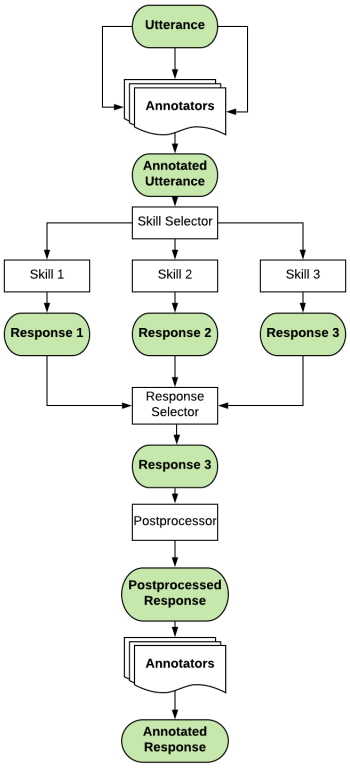
- scalability and reliability in highload environment due to micro-service architecture
- ease of adding and orchestrating conversational skills
- shared memory of dialog state and NLP annotations accessible to all skills

Core concepts of DeepPavlov Agent architecture:

- `Utterance` is a single message produced by a human or a bot;
- `Service` is a NLP model or any other external service that supports a REST API.

  DeepPavlov Agent orchestrates following types of services:

  - `Annotator` is a service for NLP preprocessing of an utterance. It can implement some basic text processing like spell correction, named entity recognition, etc.;
  - `Skill` is a service producing a conversational response for a current dialogue state;
  - `Skill Selector` is a service that selects a subset of available skills for producing candidate responses;
  - `Response Selector` is a service selecting out of available candidates a response to be sent to the user;
  - `Postprocessor` is a service postprocessing a response utterance. It can make some basic things like adding a user name, inserting emojis, etc.
- `Postprocessed Response` is a final postprocessed conversational agent utterance that is shown to the user.
- `State` stores current dialogs between users and a conversational agent as well as other infromation serialized in a **json** format. State is used to share information across the services and stores all required information about the current dialogs. Dialogue state is documented here.


Utterance
Annotators
Annotated Utterance
Skill Selector
Skill 1
Skill 2
Skill 3
Response 1
Response 2
Response 3
Response Selector
Response 3
Postprocessor
Postprocessed Response
Annotators
Annotated Response

CHAPTER 2

Installation

Deeppavlov agent requires python >= 3.7 and can be installed from pip.

```
pip install deeppavlov_agent
```

CHAPTER 3

Running the Agent

Agent can be run inside a container or on a local machine. The default Agent port is **4242**. To launch the agent enter:

```
python -m deeppavlov_agent.run -ch http_client -p 4242 -pl pipeline_conf.
↪json -db db_conf.json -rl -d
```

Command parameters are:

- -ch - output channel for agent. Could be either `http_client` or `cmd_client`
- -p - port for http_client, default value is 4242
- -pl - pipeline config path, you can use multiple pipeline configs at the time, next one will update previous
- -d - database config path
- -rl - include response logger
- -d - launch in debug mode (additional data in http output)

3.1 HTTP api server

1. **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 \
   ```

(continues on next page)

(continued from previous page)

```
    --data '{"user_id":"xyz","payload":"hello"}' \
    http://localhost:4242
```

Agent returns 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.

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

   If you want to send anything to the Agent, 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 in the Agents `state` into the `attributes` field of a `HumanUtterance`. Dialogue state is documented here

3. **Retrieve dialogs from the database through GET requests**

   Dialogs' history is returned in json format which can be easily prettifyed with various browser extensions.

   Logs can be accessed at (examples are shown for the case when the agent is running on http://localhost:4242):

   - http://localhost:4242/api/dialogs/ list of `dialog_id`
   - http://localhost:4242/api/dialogs/<dialog_id> - provides exact dialog (`dialog_id` can be seen on /dialogs page)
   - http://localhost:4242/api/user/<user_id> - provides all dialogs by user_id

4. **Load analytics**

   Number of processing tasks and average response time for both the agent and separate services are provided in a real time on the page http://localhost:4242/debug/current_load .

CHAPTER 4

Analyzing the data

History of the agent's `state` for all dialogues is stored to a Mongo DB. The state includes utterences from user with corresponding responses. It also includes all the additional data generated by agent's services. Following Mongo collections can be dumped separately:

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

These types of dialog services can be connected to the agent's conversational pipeline:

- **Annotator**
- **Skill Selector**
- **Skills**
- **Response Selector**
- **Postprocessor**

CHAPTER 5

Input Format

All services should accept an input in an agent `state` format. This format is described here. If an input format of a service differs from the agent state format then a **formatter** function should be implemented. This formatter function receives a request in agent state format and returns a request in format supported by the service.

CHAPTER 6

Output Format

All services should provide an output in an agent `state` format. This format is described here. To use the same formatter for input and output set the `mode=='out'` flag.

CHAPTER 7

Annotator

Annotator service returns a free-form response.

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

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

Sentiment annotator can return a list of labels:

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

Also, Sentiment annotator can return just a string:

```
"neutral"
```

CHAPTER 8

Skill Selector

Skill Selector service should return a list of names for skills selected to generate a candidate response for a dialog.

For example:

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

CHAPTER 9

Skill

Skill service should return a **list of dicts** where each dict corresponds to a single candidate response. Each candidate response entry requires `text` and `confidence` keys. The Skill can update **Human** or **Bot** profile. To do this, 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 in agent state. 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 agent state documentation for 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 10

Response Selector

Unlike Skill Selector, Response Selector service should select a *single* skill as a source of the final version of response. The service returns a name of the selected skill, text (might be overwritten from the original skill response) and confidence (also might 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 11

Postprocessor

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

If a response was modified by Postprocessor then a new version goes the `text` field of the final utterance and shown to the user, and the utterance selected by Response Selector goes to the `orig_text` field.

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

CHAPTER 12

User State API

Each utterance in a **Dialog state** is generated by some **User** either **Human** or **Bot**. The `user.user_type` field stores reference to source of the utterance:

```
{"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 13

/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.

CHAPTER 14

Defining the formatters

There are two main formatter types: which extracts data from dict representation of dialogs and formats it to service accessible form (dialog formatter), and which extracts data from service response and formats it prior adding to state (response formatter, this is optional one)

Dialog formatters

This functions should accept a single parameter: dialog (in dict form), and return a list of tasks for service processing. Each task should be in a format, which is correct for associated service. From a dict form of a dialog you can extract data on:

- Human - `dialog['human']`
- Bot - `dialog['bot']`
- List of all utterances - `dialog['utterances']`
- List of only human utterances - `dialog['human_utterances']`
- List of only bot utterances - `dialog['bot_utterances']`

Each utterance (both bot and human) have some amount of same parameters:

- Text - `utterance['text']`
- Annotations - `utterance['annotations']`
- User (human or bot, depending on type of utterance) - `utterance['user']`

Human utterance have an additional parameters:

- List of hypotheses - `utterance['hypotheses']`
- Additional attributes - `utterance['attributes']`

Bot utterance also have additional attributes:

- Active skill name (skill, which provided actual response) - `utterance['active_skill']`
- Response confidence - `utterance['confidence']`
- Original response text (not modified by postprocessors) - `utterance['orig_text']`

Response formatters

This functions should accept one sample of skill response, and re-format it, making further processing available. This formatters are optional.

CHAPTER 15

Agent Configuration

Configuration of pipeline and database for the **Agent** can be defined in `json` or `yml` file.

CHAPTER 16

Database Config Description

Database configuration parameters are provided via `db_conf` file. Currently, agent supports Mongo DB.

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

Example of a database config:

```
{
    "env": false,
    "host": "mongo",
    "port": 27017,
    "name": "dp_agent"
}
```

- **env**
  - If set to **false** (or not mentioned), specified parameters' values will be used for db initialisation. Otherwise, agent will try to get an environmental variable by name, associated with parameter.
- **host**
  - A database host, or env variable, where database host name is stored.
- **port**
  - A database port, or env variable, where database port is stored.
- **name**
  - An name of the database, or env variable, where name of the database is stored.

CHAPTER 17

Pipeline Config Description

Pipeline configuration parameters are specified in `pipeline_conf` file. There are two different sections in `pipeline_conf` to configure Connectors and Services.

17.1 Services Config

Service is a single node of pipeline graph, or a single step in processing of user message. In `pipeline_conf` all services are grouped under `service` key.

Example of a service config:

```
{"group_name": {
        "service_label": {
            "dialog_formatter": "dialog formatter",
            "response_formatter": "response formatter",
            "connector": "used connector",
            "previous_services": "list of previous services",
            "required_previous_services": "list of previous services",
            "state_manager_method": "associated state manager method",
            "tags": "list of tags"
        }
    }
}
```

- **group name**
  - This is an optional key. If it is specified then services can be referenced by their *group name* in `previous_services` and `required_previous_services`.
  - If *group name* is specified then the service name is `<group name>.<service label>`.
- **service_label**
  - Label of the service. Used as a unique service name, if service is not grouped.
  - Passed to a state manager method, associated with the service. So,‘‘service_label‘‘ is saved in state.

- **dialog_formatter**
  - Generates list of tasks for services from a dialog state.
  - Can be configured as `<python module name>:<function name>`.
  - Formatter can generate several tasks from the same dialog, for example, if you want to annotate all hypotheses.
  - Each generated task corresponds to a single valid request payload to be processed by service without further formatting.
- **response_formatter**
  - Maps a service response to the format of dialog state.
  - Can be configured as `<python module name>:<function name>`.
  - Optional parameter. If not specified then unformatted service output is sent to state manager method.
- **connector**
  - Specifies a connector to a service. Can be configured here, or in *connectors* section.
  - Can be configured as `<python module name>:<connector's class name>`.
- **previous_services**
  - List of services to be executed (or skipped, or respond with an error) before sending data to the service.
  - Should contain either group names or service names.
- **required_previous_services**
  - List of services to be completed correctly before the service, because it depends on their output.
  - If at least one of the `required_previous_services` is skipped or finished with an error, the service is not executed.
  - Should contain either group names or service names.
- **state_manager_method**
  - Name of a `StateManager` class method to be executed after the service response.
- **tags**
  - Tags, associated with the service to indicate a specific behaviour.
  - **selector** - corresponds to skill selector service. This service returns a list of skills selected for response generation.
  - **timeout** - corresponds to timeout service. This service is called when processing time exceeds specified limit.
  - **last_chance** - corresponds to last chance service. This service is called if other services in pipeline have returned an error, and further processing is impossible.

17.2 Connectors config

Connector represents a function, where tasks are sent in order to process. Can be implementation of some data transfer protocol or model implemented in python. Since agent is based on asynchronous execution, and can be slowed down by blocking synchronous parts, it is strongly advised to implement computational heavy services separate from agent, and use some protocols (like http) for data transfer.

There are several possibilities, to configure connector:

1. *Built-in HTTP*

```
{"connector name": {
        "protocol": "http",
        "url": "connector url",
        "batch_size": "batch size for the service"
    }
}
```

   - **connector name**
     - A name of the connector. Used in *services* part of the config, in order to associate service with the connector
   - **protocol**
     - http
   - **url**
     - Actual url, where an external service api is accessible. Should be in format `http://<host>:<port>/<path>`
   - **batch_size**
     - Represents a maximum task count, which will be sent to a service in a batch. If not specified is interpreted as 1
     - If the value is 1, an HTTPConnector class is used.
     - If the value is more than one, agent will use AioQueueConnector. That connector sends data to asyncio queue. Same time, worker QueueListenerBatchifyer, which collects data from queue, assembles batches and sends them to a service.

2. *Python class*

```
{"connector name": {
        "protocol": "python",
        "class_name": "class name in 'python module name:class name'␣
↪format",
        "other parameter 1": "",
        "other parameter 2": ""
    }
}
```

   - **connector name**
     - Same as in HTTP connector case
   - **protocol**
     - python
   - **class_name**
     - **Path to the connector's class in `<python module name>:<class name>` format**

       * Connector's class should implement asynchronous `send(self, payload: Dict, callback: Callable)` method

    * `payload represents` a single task, provided by a dialog formatter, associated with service, alongside with `task_id`: `{'task_id': some_uuid, 'payload': dialog_formatter_task_data}`
    * `callback` is an asynchronous function process. You should call that with service response and task_id after processing

- **other parameters**
  - Any json compatible parameters, which will be passed to the connector class initialisation as `**kwargs`

CHAPTER 18

Built-in StateManager

Built-in StateManager is responsible for all database read and write operations, and it's working with MongoDB database. You can assign it's methods to services in your pipeline in order to properly save their responses to dialogs state. You can read more on the pipeline configuration in *Services Config*

CHAPTER 19

Available methods

Each of the methods have a following input parameters, which are filled automatically by agent during message processing.

- `dialog` - dialog object, which will be updated
- `payload` - response of the service with output formatter applied
- `label` - label of the service
- `kwargs` - minor arguments which are also provided by agent

You can use several state manager methods in your pipeline:

1. **add_annotation**
    - Adds a record to `annotations` section of the last utterance in dialog
    - `label` is used as a key
    - `payload` is used as a value
2. **add_annotation_prev_bot_utt**
    - Adds a record to `annotations` section of the second utterance from the end of the dialog
    - Only works if that utterance is bot utterance
    - Suitable for annotating last bot utterance on the next dialog round
    - `label` is used as a key
    - `payload` is used as a value
3. **add_hypothesis**
    - Adds a record to `hypotheses` section of the last utterance in dialog
    - Works only for human utterance, since bot utterance doesn't have such section
    - Accepts list of hypotheses dicts, provided by service
    - Two new keys are added to each hypothesis: `service_name` and `annotations`

- `label` is used as a value for `service_name` key
- Empty dict is used as a value for `annotations` key

4. **add_hypothesis_annotation**
    - Adds an annotation to a single element of the `hypotheses` section of the last utterance in dialog under `annotations` key
    - In order to identify a certain hypothesis, it's index is used and stored in agent
    - `label` is used as a key
    - `payload` is used as a value
5. **add_text**
    - Adds a value to `text` field of the last utterance in dialog
    - Suitable for modifying a response in a bot utterance (original text can be found in `orig_text` field)
    - `payload` us used as a value
6. **add_bot_utterance**
    - This method is intended to be associated with response selector service
    - Adds a new bot utterance to the dialog
    - Modifies associated user and bot objects
    - We consider, that payload will be a single hypothesis, which was chosen as a bot response. So it will be parsed to different fields of bot utterance
    - `text` and `orig_text` fields of new bot utterance are filled with `text` value from payload
    - `active_skill` field is filled with `skill_name` value from payload
    - `confidence` field is filled with `confidence` value from payload
    - `annotations` from payload are copyed to `annotations` field of bot utterance
    - We expect, that skills will return `text` and `confidence` fields at least. `skill_name` and `annotations` are created within `add_hypothesis` method
7. **add_bot_utterance_last_chance**
    - This method is intended to be associated with a failure processing service, like timeout or last chance responder
    - It is very similar in processing to `add_bot_utterance`, but it performs an additional check on the type of a last utterance in dialog
    - If the last utterance is a human utterance the method acts as an `add_bot_utterance` one
    - Otherwise, it will skip a stage with creating a new bot utterance and inserting it at the end of the dialog

There are two additional state manager methods, which are automatically assigned during agent's initialisation.

1. **add_human_utterance**
    - This method is assigned to an input service, which is created automatically during agent's initialisation process
    - Adds a new human utterance to the dialog
    - `payload` is used for `text` field of the new human utterance

2. **save_dialog**

    - This method is assigned to a responder service, which is created automatically during agent's initialisation process
    - It just saves a dialog to database

CHAPTER 20

Built-in connectors

Generally, connector is a python class with a method `send`. It can be either a model, nn or rule based, or implementation of some transport protocols. Although, we strongly recommend to implement nn models as an external services.

We have two different connectors for HTTP protocol as a built-in ones. Single sample and batchifying. Of course you can send a batch of samples to your model using single sample connector, but in this case you should form the batch with proper dialog formatter. Batchifying connector will form batch from samples, available at the time, but can't guarantee actual batch size, only it's maximum size.

There are three more connectors, which can be used for different purposes. Each of them can be configurend as a *python* connector with it's name You can read more on the connectors configuration in *Connectors config*.

CHAPTER 21

Built-in python connectors

21.1 ConfidenceResponseSelectorConnector

This connector provides a simple response selection functionality. It chooses a best hypothesis based on its `confidence` parameter. In order to use it, you should consider a few things:

- You don't need to define a dialog formatter (if you use built-in state manager)
- You need to ensure, that all of your skills (or services with assighed `add_hypothesis` SM method) provides a `confidence` value somehow
- It returns a chosen hypothesis, so you don't need to define output formatter as well
- No special configuration parameters are needed

So the basic configuration for it is very simple:

```
{"response_selector": {
    "connector": {
        "protocol": "python",
        "class_name": "ConfidenceResponseSelectorConnector"
    },
    "state_manager_method": "add_bot_utterance",
    "previous_services": ["place previous skill names here"]
}}
```

21.2 PredefinedTextConnector

This connector can be used in order to provide a simple way to answer in time, or in case of errors in your pipeline. It returns a basic parameters, which can be used to form a proper bot utterance.

- `text` parameter will be a body of a bot utterance

- Additionally, you can provide an `annotations` parameter, in case if you need to have a certain annotations for further dialog
- There is no need to configure a dialog and response formatters

This example configuration represents simple last chance service:

```
{"last_chance_service": {
    "connector": {
        "protocol": "python",
        "class_name": "PredefinedTextConnector",
        "response_text": "Sorry, something went wrong inside. Please tell me, what␣
↪did you say."
        "annotations": {"ner": "place your annotations here"}
    },
    "state_manager_method": "add_bot_utterance_last_chance",
    "tags": ["last_chance"]
}}
```

More on last chance and timeout service configuration here:

21.3 PredefinedOutputConnector

This connector is quite similar to PredefinedTextConnector. It returns a predefined values, but instead of fixed `text` and `annotations` keys, it can be configured to return any arbitrary json compatible data structure. The main purpose of this connector class is testing of pipeline routing, formatting or outputs. You can make a dummy service, which will imitate (in terms of structure) the response of desired model. This connector have only one initialisation parameter:

- `output` - list or dict, which will be passed to agent's callback as payload

This example configuration represents a dummy service, representing skill:

```
{"skill": {
    "connector": {
        "protocol": "python",
        "class_name": "PredefinedOutputConnector",
        "output": [{"text": "Hypotheses1", "confidence": 1}]
    },
    "dialog_formatter": "place your dialog formatter here",
    "response_formatter": "place your response formatter here",
    "state_manager_method": "add_hypothesis",
    "previous_services": ["list of the previous_services"]
}}
```

But you can imitate any skill type with this connector.

CHAPTER 22

Writing your own connectors

In order to define your own connector, you should follow these requirements:

- It should be a python class
- You can pass initialisation parameters to it via *Connectors config* python class
- You need to implement an asynchronous method `send(self, payload: Dict, callback: Callable)`
- It should return a result to agent using `callback` function
- `payload` input parameter is a dict of following structure:

```
{
    "task_id": "unique identifyer of processing task",
    "payload": "single task output, of the associated dialog formatter"
}
```

So basically, your connector should look like this:

```
class MyConnector:
    def __init__(self, **kwargs):
        # Your code here

    async def send(self, payload: Dict, callback: Callable):
        try:
            # Write processing part here
            await callback(
                task_id=payload['task_id'],
                response=response  # Supposing that result of the processing is␣
↪stored in a variable named "response"
            )
        except Exception as e:
            # That part allows agent to correctly process service internal errors
            # and call a "last chane" service without stopping the ongoing dialogs
            response = e
```

(continues on next page)

(continued from previous page)

```
            await callback(
                task_id=payload['task_id'],
                response=response
            )
```