TOMAAT Documentation

Release 0.1

Fausto Milletari et al

Oct 12, 2018

Contents:

TOMAAT allows you to make deep learning models public by providing them with an interface to communicate with the "outside" world. This interface is built on top of HTTP 1.1 through a framework called Klein.

Here you can find documentation about TOMAAT that will help you get started with it.

You can find pre-built docker images, running services through TOMAAT, on DockerHub.

CHAPTER 1

TOMAAT

TOMAAT allows you to serve deep learning apps over the cloud. If you have a trained deep learning model, you can use TOMAAT to create an app and a service for your algorithm. An app is a combination of pre-processing, inference and post-processing. A service is a way to expose your app to the outside world via a standardized yet flexible inteface.

1.1 Getting started

- Install TOMAAT through `pip install tomaat`
- Refer to the documentation and examples

Note: depending on your workflow TOMAAT may require additional packages to be installed. Dependencies such as deep learning frameworks and software such as VTK are not installed by running `pip install`.

To install Python VTK on Ubuntu run `sudo apt-get install python-vtk` (help for other platforms here). TO install a deep learning framework such as Tensorflow or Pytorch, refer to the appropriate documentation published online.

1.1.1 Service Docker Registry

You can also get started by running services that have been already created and exposed through TOMAAT. Docker images corresponding to these services are hosted on DockerHub You just need to have docker and nvidia-docker 2 installed, and execute the command

- `nvidia-docker run -it -p 9001:9001 --rm tomaat/services:tag python serve.py start_service`

where tag is one of the tags shown here.

This will start the service in your current system, into a docker container.

1.2 Disclaimer

This software is provided "as-it-is" without any guarantee of correct functionality or guarantee of quality. No formal support for this software will be given to users. It is possible to report issues on GitHub though. This repository and any other part of the TOMAAT project should not be used for medical purposes. In particular this software should not be used to make, support, gain evidence on and aid medical decisions, interventions or diagnoses. The privacy of the data is not guardanteed when TOMAAT is used and we should not be held responsible for data mis-use following transfer. Although we reccommend to users of TOMAAT to organize their services such that no data is permanently stored or held when users request predictions, we cannot guardantee that any data transferred to those remote services is not going to be misused, stored or manipulated. Use TOMAAT responsibly.

1.3 Architecture

In this repository you find TOMAAT. TOMAAT is written in python and contains three parts.

- Server
- Client
- Announcement Service

The Server provides the main functionalities that are used to by TOMAAT to wrap deep learning models and make them available over the cloud, through HTTP protocol, using Klein. To facilitate developement by users, we also include examples of how to use TOMAAT to instantiate prediction services. We also propose framework-specific implementations of the prediction function, which can be used to create a TomaatApp.

TODO The Client implements functionalities that can be used by client machines to query and obtain predictions from services created using TOMAAT. We include a CLI that allows simple interaction with remote services.

The Announcement Service implements the model endpoint announcement service, which is included in TOMAAT to allow users to run their own endpoint announcement services (as an alternative to the official endpoint announcement service), if they wish.

A summary of the current architecture of TOMAAT is shown below: All communications between local and remote machines – for service discovery and inference – happen through HTTP 1.1 protocol. Services are discovered by a GET request to the appropriate URL while images are segmented through a POST request containing JSON data. Each service has its own interface that is defined by the server administrator as a collection of standard elements. User can trasfer data to services in order to get predictions.

1.4 Using TOMAAT

Making an algorithm available on the cloud through TOMAAT is a combination of two steps:

1. Creating an APP
2. Creating a Service that runs and exposes the app

Some examples of services published through TOMAAT using different Deep Learning frameworks can be found here.

For now, let's start assuming that you, the developer, have defined three functions for pre-processing, inference and post processing. This is, most of the time, a good way to split the test-time functionality of your DL algorithms (More complex cases can still be addressed, but are beyond the scope of this introduction).

```
from .my_project import pre_processing, inference, post_processing

# pre_processing function takes in data and returns pre-processed data.
# inference function takes in data and runs inference on it. For example it runs sess.
↪run(outputs, feed_dict=...) on data and returns results.
# post_processing function takes in inference results and returns post-processed data.
```

1.4.1 Creating an APP

At this point we can define the APP:

```
from tomaat.server import TomaatApp

my_app = TomaatApp
    (
        preprocess_fun=pre_processing,
        inference_fun=inference,
        postprocess_fun=post_processing
    )
```

1.4.2 Specifying Service configuration and interfaces

To define a service we need:

1. Service configuration, that can be for example loaded from a user-defined JSON file.
2. Input interface
3. Output interface

The service configuration has a few mandatory fields which are used to enable communication with the announcement service and to define a few things such as the port at which the service will be available etc. More info can be found at **TODO***. Here you can find an example of service configuration:

```
config =
    {
        "name": "Example TOMAAT app with tensorflow",
        "modality": "Example Modality",
        "task": "Example Task",
        "anatomy": "Example Anatomy",
        "description":"Example Description",
        "port": 9001,
        "announce": false,
        "api_key": "",
    }
```

The input interface can be defined according to what we have already explained below in section "Service input interface". Nevertheless we provide an example:

```
input_interface = \
    [
        {'type': 'volume', 'destination': 'images'},
        {'type': 'slider', 'destination': 'threshold', 'minimum': 0, 'maximum': 1},
        {'type': 'checkbox', 'destination': 'RAS', 'text': 'use slicer coordinate␣
↪conventions'},
```

(continues on next page)

(continued from previous page)

```
        {'type': 'radiobutton', 'destination': 'spacing_metric', 'text': 'choose:',
↪'options': ['mm', 'm']},
    ]
```

The output interface has already been explained below, in section "Service output interface". We nevertheless provide an example here:

```
output_interface = \
    [
        {'type': 'LabelVolume', 'field': 'images'}
    ]
```

1.4.3 Creating a Service

At this point we can specify a Service:

```
from tomaat.server import TomaatService

my_service = TomaatService
    (
        config=config,
        app=my_app,
        input_interface=input_interface,
        output_interface=output_interface
    )
```

and we can run the service through:

`my_service.run()`

which will make it available on the network.

1.4.4 Assumptions about data

TOMAAT is designed to feed `data` to the APP using a python **dictionary**. Data will have some fields, that are named after the content of the 'destination' field of the input interface. For example, if the input interface specified for the current app is

```
input_interface = \
    [
        {'type': 'volume', 'destination': 'images'},
    ]
```

data will be a `dict()` having one field `data['images']` which contains a volume in SimpleITK format. Again refer to the examples to understand more.

1.4.5 Service input interface

Input interfaces are specified making use of standardize data elements. These are the currently supported input interface elements.

- `{'type': 'volume', 'destination': field}`: instucts the client to build its interface such that the user can choose a volume, in MHA format, and place it in the field `field` of the POST request.

- `{'type': 'slider', 'destination': field, 'minimum': a, 'maximum': b}`: instucts the client to build its such that the user can choose a value from a fixed interval [a, b] which will be expected to be in the field `field` of the POST request.
- `{'type': 'checkbox', 'destination': field, 'text': UI_text }`: instructs the client to build an interface widget similar to a checkbox, to allow the user to pass a on/off type of variable which is expected to be in the field `field` of the POST request.
- `{'type': 'radiobutton', 'destination': field, 'text': UI_text , 'options': ['a', 'b']}`: instructs the client to spawn a UI element similar to a radio button which allows the user to choose among multiple options, which will be passed to the server in the POST field `field`. These elements can be combined into a list as in this example: .. code-block:: guess

  input_interface =

  [{'type': 'volume', 'destination': 'images'}, {'type': 'slider', 'destination': 'threshold', 'minimum': 0, 'maximum': 1}, {'type': 'checkbox', 'destination': 'switch', 'text': 'on'}, {'type': 'radiobutton', 'destination': 'pick', 'text': 'choose:', 'options': ['a', 'b']},

  ]

1.4.6 Service output interface

Output interfaces can be specified using standardize output data elements. A full list of the supported output elements is shown here:

- `{'type': 'LabelVolume', 'field': 'data_dict_field'}`: instructs the reponse creation function that the content of the `data` dictionary in correspondence of the field 'data_dict_field' contains a label volume that needs to be sent to the client.
- `{'type': 'VTKMesh', 'field': 'data_dict_field'}`: instructs the reponse creation function that the content of the `data` dictionary in correspondence of the field 'data_dict_field' contains a VTK Mesh that needs to be sent to the client.
- `{'type': 'PlainText', 'field': 'data_dict_field'}`: instructs the reponse creation function that the content of the `data` dictionary in correspondence of the field 'data_dict_field' contains plain text that needs to be sent to the client. .. code-block:: guess

  output_interface =

  [{'type': 'LabelVolume', 'field': 'labels'}, # data['labels'] contains a label volume in SimpleITK format {'type': 'VTKMesh', 'field': 'mesh'}, # data['mesh'] contains a mesh in vtk format {'type': 'PlainText', 'field': 'text'}, # data['text'] contains plain text in form of a string

  ]

1.5 Interfaces in TOMAAT

We detail here the interfaces used for communication between the various components of TOMAAT and between TOMAAT and client code.

1.5.1 Publishing a service

ToDo

1.5.2 Discovering public services

A **GET** request can be made to the service discovery server url (for example http://tomaat.cloud:8001/discover). A JSON message is received. It contains a **list** of dictionaries containing the following fields:

- 'interface_url': the URL that the user can GET from to obtain the description of the server interface
- 'prediction_url': the URL that the user can POST to in order to obtain predictions
- 'modality': the medical imaging modality
- 'anatomy': the anatomy
- 'task': the task
- 'name': the name of the prediction service
- 'description': a description of the prediction service
- 'SID': an unique identifier for a server which allows specific services to be included in workflows
- 'creation_time': the time of last service announcement to the announcement service
- 'api_key': empty string

1.5.3 Requesting the input interface details

The input interface description can be obtained by a **GET** request to the `interface_url` hosted by the prediction server. The response to the GET request is a list of dictionaries containing an arbitrary combination of standardized data elements which follow the conventions described above in the "Service input interface" section.

1.5.4 Requesting a prediction

A **POST** request to a service is used to trigger prediction. The POST requests needs to contain all the fields required by the service, as specified in the input interface (see above "Service input interface"). Each service requires different arguments and data to be supplied by the client. The type and field (in the POST request) of these arguments is specified in the interface description. The POST request needs to be done using the `prediction_url` of the service.

1.6 Mechanics

1.6.1 Prediction

In this section we want to give some insight about what happens when a POST request to a server exposing a particular interface is made. The POST request must contain data in correspondence to the fields expected by the remote server. The request should be made to the URL in the `prediction_url` field of the server description dictionary obtained from the announcement service. A service that expects the interface

```
[
    {'type': 'volume', 'destination': 'image'},
    {'type': 'radiobutton', 'destination': 'type', 'text': 'MRI sequence' , 'options
↪': ['T1', 'T2']}
]
```

will expect POST requests having fields `image` and `type`. The `image` field will need to be populated the content of a MHA file and the `type` field will need to contain either the string `T1` or the string `T2`. POST request should be multipart. An example of client can be found at the URL https://github.com/faustomilletari/TOMAAT-Slicer

1.7 Endpoint announcement service

ToDo

1.8 Framework specific examples/solutions

CHAPTER 2

Indices and tables

- genindex
- modindex
- search