Hugger

Ben Jones  
15-Aug-2024

The purpose of this set of programs is to try out different Machine Learning Models in [https://huggingface.co](https://huggingface.co/) and evaluate them for functionality and for their interaction with the network. They can be downloaded from <https://github.com/BenMJonesGit/hugger>.

# Hugger Program

The main program is run as follows:

**python** **-m** **hugger** *taskName* *arguments* **repeat** *r*

Set a model grouping *taskName*. Specify *arguments as follows:*

***m*** – A non-negative number selecting from an array of model names.

***m1,m2,m3,***… – A series of numbers indicating model names to be tried in sequence on the inputs indicated by the remaining arguments. If a given number is specified as ***x***-***y***, it means models ***x*** through ***y*** inclusive.

**all** – If this is the first argument after *taskName*, it means do all models in sequence.

***name*** – The name of a file in the **inputs/*taskName***  folder.

**all**– If this is the second argument after taskName, it means do all files in the inputs/taskName folder in alphabetical order.

**ds*=d*:*i1,i2,i3,…*** – Select inputs from dataset ***d***, indexing into it in sequence i1, i2, i3, …. If a given number is specified a ***x***-***y***, it means inputs ***x*** through ***y*** inclusive. If it is specified as **all**, it mean go though all the indexes in sequence.

**repeat** – As the last argument, it means to repeat the argument list indefinitely. If an ***r*** follows **repeat**, it means to repeat the argument list *r* more times.

So here is the idea: Each *taskName* corresponds to a grouping of models within **HuggingFace**. In the **task** folder, there are ***taskName*.py** modules that define the interface to a set of models, including a list of **models** that we wish to reference in the argument list. There is also a list of **datasets** that we can reference in the **ds=** argument. There is a folder called **inputs/*taskName*** that contains input files that may be run against the models. The desire is to put a model or set of models through some exercises to see how they respond, and to evaluate how they interact with the Internet to see if anything nefarious is going on.

If no parameters are given after *taskName*, the program prompts first for “Model:”, and then for “Input:” until **quit** is given.

If the environment variable **EXECUTE** is defined, it overrides the parameters following *taskName*. If the environment variable **TASK** is defined, it overrides the ***taskName*** parameter. The main reason for these variables is to control the action of a docker container running this program, to override the defaults given in the **compose.yml** file.

## Examples

So, for example:

**python -m hugger summarization 0 input0 input1 input2**

Use model 0 from **summarization** and use **input0.txt**, **input1.txt**, and **input2.txt** from the **inputs/summarization** folder.

**python -m hugger summarization 0,2,4-7 ds=0:1,3,5-7,10**

Use inputs 1, 3, 5, 6, 7, and 10 from dataset 0 on models 0, 2, 4, 5, 6, and 7.

**python -m hugger summarization all input0 repeat 5**

Try **input0.txt** on each model referenced in **summarization.py**. Repeat this sequence 5 times after the initial sequence.

## Tasks

The reason for the grouping by ***taskName*** is that different classes of models require different types of inputs. Here are the ***taskName***’s that have been defined so far:

**summarization**

A **summarization** model takes as input, a set of sentences. The result is a summary of those sentences. The input files are .txt files. The datasets associated with this contain sets of sentences.

**automatic\_speech\_recognition**

An **automatic\_speech\_recognition** model takes as input, an audio clip. The result is a text rendering of that audio clip. The input files are .flac files. The datasets associated with this contain references to audio files stored in HuggingFace.

**translation**

A **translation** model takes as input, a set of sentences. The result is a rendering of those sentences in another language. Currently, the set of models defined here translate from English to French. The input files are .txt files containing sentences to be translated. The datasets associated with this contain sets of sentences to be translated.

**question\_answering**

A **question\_answering** model takes as input, a json object containing a question and a context. The question searches the context from the answer to that question, which is then returned as a result. The input files are .json files contain name value pairs for **question** and **context**. The datasets associated with this contain objects that include name value pairs for **question** and **context**.

## Templates

The **template.py** module shows what items needs to be defined for any given *taskName*:

**task**=”taskName”

Define the “taskName” to be given to the HuggingFace pipeline. If this contains multiple words, they are separated by dashes whereas the taskname has them separated by underscores.

**ext**=”extension”

Define the extension (i.e. “txt”) for files found in the **inputs/*taskName*** folders.

**mode**=”rt”

This is the mode for opening files found in the **inputs/*taskName*** folders.

**loadModel(model\_name)**

Given a “***model\_name***” (which is usually of the form “*folder*/*subfolder*”, which is relative to the [**https://huggingface.co**](https://huggingface.co) URL), return an object representing the *model*. Please note that if ***model\_name*** is a path to a file on the file system, a local copy of the model is used. Note also that once a ***model\_name*** has been reference within the HuggingFace URL, it is cached in:

**~/.cache/huggingface/hub/***folder*--*subfolder*

**execModel(model, input)**

Given the *model* object returned by loadModel, process an *input*. An input may be a string of text to be processed, a string defining a JSON object, or a binary blob (such as the content of an audio or video file). The result is a string or object that may be printed.

**models = [ “*model\_name0*”, “*model\_name1*”, … ]**

This is an array of names of models to be passed to **loadModel**. The command line interface for hugger indexes into this array to find the models.

**datasets = [ {*object0*}, {*object1*}, … ]**

This is an array of descriptors for datasets. The following fields may be defined in each object:

**name** – The name of the dataset on HuggingFace.

**split** – The name of the split to be used (i.e. “train”, “test”, …)

**input** – The name of the subset to be used for input (i.e. “document”). If this is names separated by commas, the input will be a JSON string containing those names and their content. This mode is used in **question\_answering**.

**take** – The number of items to be extracted from the dataset (0 if all). This is necessary when the items themselves are quite large, such as audio or video clips.

Please note that the datasets array was added later. Originally, the program was set up to use input files stored in the **inputs/*taskName*** folders.

Also, note that there is a **models** subfolder in **hugger**. The idea here was that models could be copied from HuggingFace into folders under models, using the same folder structure. Thus for example “**facebook/bart-large-cnn**” referenced from <https://huggingface.co> could be copied down to **models/facebook/bart-large-cnn**. The program checks to see if a valid filesystem path exists to any given **models[*n*]** under the models folder and passes that path to pipeline. Otherwise just the string in **models[*n*]** is passed.

## Output

When the hugger program executes various output lines are generated to stdout:

**MODEL[*n*]: *folder*/*subfolder***

Show what model is currently loaded.

**INPUT: inputs/*taskName*/*filename*: *text of file***

Display the input file, unless it is binary.

**DATASET[*d*]: *folder*/*subfolder***

Show which dataset is being used.

**INPUT[*m*]: *text of dataset item***

Show the text of the dataset item being used.

**OUTPUT: *text of output***

Show the output generated when running the input through the current model.

Note that error messages being generated by the execution of the model and also messages indicating the downloading of models and/or datasets may be displayed in between the above messages. You may suppress these by piping **stderr** to null:

**python -m hugger summarization all input0 2> /dev/null**

This however suppresses prompts for model and input when parameters after the *taskName* are not provided. It will still work but you just won’t see the prompts.

# Docker

The **Dockerfile** script is set up to create a docker container for the programs in hugger and also pre-copy certain models into the container.

## Building and Running Docker

A container may be built as follows:

**docker build -t *name* .**

Build a docker container called “*name”.* Using**--build-arg *argName*=*value*** as a switch to the docker build can set things up using the following arguments:

**task** – This string defines the taskName

**models** – This string contains a list of space-separated numbers indicating the models to be pre-loaded into the container.

**save** – This string indicates the name of a python module used to do the downloading of the module (see next section).

**arguments** – This string contains the parameters to be given to the python -m hugger command to be the default command used to run this container.

Please note that **Dockerfile** uses **requirements.txt** to preload using pip various modules needed by the python program in order to run the hugger.py program.

**docker run -it *name***

Run the previously built docker container using the *name* given in the **-t** switch in the build. This will use the hugger.py script with the *taskName* and *arguments* given by the **--build-arg** arguments given in the build.

**docker run -it *name* bash**

This runs the container using the bash shell, which prompts the user. Here, you may give the command **python -m hugger *taskName* *arguments****.*

Note that the environment variables TASK and EXECUTE may be set up when running docker to change how the hugger module is run:

**TASK=translation EXECUTE=”0,1,2 ds=0:0-3” docker run -it *name***

This would run with the translation task, executing models 0, 1, and 2 using inputs from dataset 0, items 0, 1, 2, and 3.

## Save Methods

Several different modules are available to try out different methods of saving models into a docker image:

**python -m save*Method* *taskName m0 m1 m2* …**

This downloads models indicated by indexes ***m0***, ***m1***, etc. named in the models array in ***taskName***. The folder structure indicated in ***taskName*** is duplicated in the **models** subfolder.

Various **save*Method*.py** files are provided:

**saveNone.py**

Do nothing (default). That is nothing will be loaded into the **models** folder.

**saveModels.py**

This using **huggingface-cli** to download models into the **models** folder.

**savePretrained.py**

This uses a built-in function in the model returned by the **loadModel** function called **save\_pretrained**. This saves the model into a local folder.

**saveCache.py**

This uses what we know about how the models are cached to copy information from **~/.cache/huggingface/hub/models--*folder*--*subfolder*/snapshot**

Note that the various modules may be run stand-alone to store models into the **models** folder. One curious thing is that when these modules are run within a docker build, not only the models folder is loaded, which is copied into the docker image but also the cache which is copied to **/root/.cache** in the docker image. Because of this the Dockerfile removes the cache in the process of building the docker image.

We can also manually copy models into the **models** folder before we do the docker buildTo get rid of models already copied, do the following:

**rm -rf models/\***

## File System Comparison

As we run **docker** in one terminal window, we may examine the file system of the docker container using another terminal window. For this purpose, we need to know the name of the docker instance created when it is run. This can be done in one of two ways.

We can check the name of the instance by running docker ps. This will list out the names of running docker containers.

We can assign a name in the docker run command using **--name**:

**docker run --name *imageName* -it *name***

The only trouble with this method is that if we rerun with the same *imageName*, we have to remove the docker *imageName* first:

**docker rm *imageName***

Given the *imageName*, we can then use the following script:

**python -m compare *imageName***

This uses **docker diff** to see what has changed in ***imageName*** since it was first run. The only trouble with **docker diff** is that it always lists all the changes since the container was first run. The **compare.py** script saves the results of the first docker diff in the **cmp** folder and then on the next run compares with the previous results. It therefore only prints the files and folders that have changed since the last time this script was run.

So if we are running hugger in interactive mode (**EXECUTE=””**), after each model is loaded or each input is run, we can check to see what has changed. What we have found is that if a certain model was preloaded, nothing changes when that particular model is loaded. However, if we select a different model that was not preloaded, there are a bunch of folders and files that are added to the docker image when this different model is first loaded.

# Network Activity

We ran experiments to see what kind of network activity happens when a docker container is run. These use docker compose to set up a docker container running **tcpdump** and another docker container running **hugger** both referencing a network bridge. This is built as follows:

**docker compose build**

It can then be run as follows:

**docker compose up --abort-on-container-exit**

The result is that output activity from **tcpdump** and **hugger** is displayed as the two containers run. Outputs from **tcpdump** are prefaced by **tcpdump-1** while outputs from hugger are prefixed by hugger-1. Note that if in another window you want to monitor file activity using the **compare.py** script, the *imageName* is **hug-hugger-1**. Note that when using compose, we didn’t have to worry about the imageName’s being removed. That happened automatically when compose was done.

To make this easier, we have a script to do the same thing:

**source compose.sh**

What we found was kind of odd. We expected if the program went out to HuggingFace to download models that were not preloaded, we would see network activity involving gigabytes worth of data being downloaded but **tcpdump** did not report this activity but only much smaller amounts of activity each time a model or dataset was loaded for the first time.

## Simplified Docker Image

Another method was suggested for preloading and running models:

**docker build --file Dockerfile-simple -t *imageName* .**

This copies a model directly from the **model** subfolder and uses inputs provided in the **input** subfolder.

Thus, we could use the various **save*Method*** modules to copy something into models and then copy from **models/*folder*/*subfolder*** into model. Or just copy something else into model. This would use the minimum amount of data to be loaded into docker.

This did not make a bit of difference in the network activity detected when using **compose-simple.sh** to run the whole thing.