alternat

Kepler

Jan 22, 2021

CONTENTS

alternat is a collection of open-source toolsets with the ambition of lowering the barrier of adopting accessibility solutions**. alternat** helps to generate default intelligible alternative text for images in websites.

Based on our experience, just adding alt-text is not a complete solution but collecting images to be annotated is a big part of the task. Keeping in mind above two requirements the current version of alternat offers two features:

- Collect images from a website
- Generate recommended alternative text

CHAPTER

ONE

WHY ALTERNAT

70% of the sites are inaccessible and the inaccessibility is causing a loss of 7 billion every year. In spite of availability of accessibility standards for long and tools to point out where you are falling short, there is a shortage of solutions which allow you to implement them with ease. One of these areas is – **Alternate Text (alt text)**.

The alt-text attribute of image tag in html is supposed to make images in websites accessible. But in practice, one doesn't see it meaningfully implemented. Someone has to go through all the images in question and craft a corresponding alt text based on context. The investment to do it can become high and it could take time to author the content.

What if there is a library, which can be integrated in your projects, that provides a recommended alt text for a given image, which can be either passed on as is or as a recommendation to a reviewer? **alternat** just does that.

1.1 Installing alternat

1.1.1 Installing alternat macOS

Install using pypi (macOS)

1. Install Node (>=v.12)
2. Install Python (>=3.8)
3. Install alternat:

```
pip install alternat
```

4. Install apify

```
mkdir -p ~/.alternat && cd ~/.alternat && npm install apify && cd -
```

Install from source (macOS)

1. Install git
2. Install Node (>=v.12)
3. Install Python (>=3.8)
4. Open terminal and clone the repo:

```
git clone https://github.com/keplerlab/alternat.git
```

5. Change the directory to the directory where you have cloned your repo

```
$cd path_to_the_folder_repo_cloned
```

6. Install apify

```
mkdir -p ~/.alternat && cd ~/.alternat && npm install apify && cd -
```

7. Install alternat using setup.py

```
python setup.py install
```

Installation using Anaconda python (macOS)

1. Install git
2. Install Node (>=v.12)
3. Install Python (>=3.8)
4. Open terminal and clone the repo:

```
git clone https://github.com/keplerlab/alternat.git
```

5. Change the directory to the directory where you have cloned your repo

```
$cd path_to_the_folder_repo_cloned
```

6. Create conda environment and install dependencies using alternat.yml file

```
cd setup_scripts
conda env create --name alternat --file=alternat.yml
```

7. Activate newly created environment:

```
conda activate alternat
```

8. Install apify

```
mkdir -p ~/.alternat && cd ~/.alternat && npm install apify && cd -
```

Installation using Docker (macOS)

1. Download and Install Docker Desktop for Mac using link: https://docs.docker.com/docker-for-mac/install/
2. Clone this repo
3. Change your directory to the cloned repo.
4. Open terminal and run following commands:

```
cd <path-to-repo> //you need to be in your repo folder
docker-compose up
```

5. In a new terminal window open terminal and enter into alternat docker container using command:

```
docker-compose exec alternat bash
```

1.1.2 Installing alternat ubuntu

Install using pypi (ubuntu)

1. Install Node (>=v.12)
2. Install Python (>=3.8)
3. Install alternat:

```
pip install alternat
```

4. Install apify by first downloading install_apify_ubuntu.sh located at setup_scripts folder in alternat Repo link and then executing downloaded script

```
sudo sh install_apify_ubuntu.sh
```

Install from source (ubuntu)

1. Install git
2. Install Node (>=v.12)
3. Install Python (>=3.8)
4. Open terminal and clone the repo

   git clone https://github.com/keplerlab/alternat.git

5. Change the directory to the directory where you have cloned your repo

```
$cd path_to_the_folder_repo_cloned
```

6. Install apify by executing given script

```
cd setup_scripts
sudo sh install_apify_ubuntu.sh
```

7. Install alternat using setup.py

```
python setup.py install
```

Installation using Anaconda python (ubuntu)

1. Install git
2. Install Node (>=v.12)
3. Install Python (>=3.8)
4. Open terminal and clone the repo:

```
git clone https://github.com/keplerlab/alternat.git
```

5. Change the directory to the directory where you have cloned your repo

```
$cd path_to_the_folder_repo_cloned
```

6. Create conda environment and install dependencies using alternat.yml file

```
cd setup_scripts
conda env create --name alternat --file=alternat.yml
```

7. Activate newly created environment:

```
conda activate alternat
```

8. Install apify by executing given script

```
cd setup_scripts
sudo sh install_apify_ubuntu.sh
```

Installation using Docker (ubuntu)

1. Download and Install Docker Desktop for Mac using link: https://docs.docker.com/docker-for-mac/install/
2. Clone this repo
3. Change your directory to the cloned repo.
4. Open terminal and run following commands:

```
cd <path-to-repo> //you need to be in your repo folder
docker-compose up
```

5. In a new terminal window open terminal and enter into alternat docker container using command:

```
docker-compose exec alternat bash
```

1.1.3 Installing alternat windows

Install using pypi (Windows)

1. Install Node (>=v.12)
2. Install Python (>=3.8)

3. Install apify by first downloading install_from_pypi_windows.bat script located at setup_scripts folder in alternat repo [link](https://github.com/keplerlab/alternat/blob/main/setup_scripts/install_from_pypi_windows.bat) and then executing downloaded script inside new windows powershell prompt:

```
.\install_from_pypi_windows.bat
```

Install from source (Windows)

1. Install git
2. Install Node (>=v.12)
3. Install Python (>=3.8)
4. Open terminal and clone the repo

   git clone https://github.com/keplerlab/alternat.git

5. Change the directory to the directory where you have cloned your repo

```
$cd path_to_the_folder_repo_cloned
```

6. Install apify by executing given script inside windows powershell prompt:

```
cd setup_scripts
.\install_from_source_windows.bat
```

Installation using Anaconda python (Windows)

1. Install git
2. Install Node (>=v.12)
3. Install Python (>=3.8)
4. Open terminal and clone the repo inside windows powershell prompt:

```
git clone https://github.com/keplerlab/alternat.git
```

5. Change the directory to the directory where you have cloned your repo

```
$cd path_to_the_folder_repo_cloned
```

6. Create conda environment and install dependencies using enviorment_windows.yml file

```
cd setup_scripts
conda env create --name alternat --file=alternat.yml
```

7. Activate newly created environment:

```
conda activate alternat
```

8. Install apify by executing given script inside windows powershell promp:

```
cd setup_scripts
.\install_apify_windows.bat
```

Installation using Docker (Windows)

1. Download and Install Docker Desktop for Mac using link: https://docs.docker.com/docker-for-mac/install/
2. Clone this repo
3. Change your directory to the cloned repo.
4. Start docker container using this command inside windows powershell or cmd promp:

```
cd <path-to-repo> //you need to be in your repo folder
docker-compose up
```

5. In a new windows powershell or cmd window open terminal and enter into alternat docker container using command:

```
docker-compose exec alternat bash
```

1.2 Running alternat in 5 minutes

Alternat can run in the following mode:

1. Application Mode: In application mode, users use Command Line Interface (CLI) to run the alternat. We have created a sample app.py which will run the application via CLI command.
2. Library Mode: In Library mode, users install alternat from pip (python package installer) and can use in new or existing application via the library.
3. Service Mode: In Service mode, REST API endpoint is exposed where a POST request can be submitted with a JSON request to get the alt-text generated by alternat.

Below is the 5-minute guide to run alternat in the modes described above:

1. Application Mode:

   Collection:

   Use case : collect and store images from a URL and store them in a folder

```
python app.py collect --url="https://page_url" --output-dir-path="sample/
↪images/test"
```

   Generation:

   Use case: generate alt-text for images in input folder and save result in a directory)

```
python app.py generate --input-dir-path="sample/images_with_text" --
↪output-dir-path="results"
```

   Use case: generate alt-text for single image and save result json in directory

```
python app.py generate --input-image-file-path="sample/images_with_text/
↪sample1.png" --output-dir-path="results"
```

2. Library Mode:

   Collection

```
# import the alternat library
from alternat.collection import Collector

# instantiate the collector
collector = Collector()

# Download images from url and saves image files in  output_dir_path
# Optional parameters, download_recursive if True crawls whole site␣
↪mentioned in
# url by visiting each link recursively and downloads images
# collect_using_apify in future more crawlers will be supported this␣
↪parameter
# ensures that apify crawler is used.
collector.process(url, output_dir_path, download_recursive, collect_using_
↪apify)
```

   Generation:

```
# import the alternat library
from alternat.generation import Generator

# instantiate the generator
generator = Generator()

# generate alt text from file (file at location sample/images_with_text/
↪sample1.png
# and results saved at location folder results)
generator.generate_alt_text_from_file("sample/images_with_text/sample1.png
↪", "results")

# OR

# generate alt text from base64 image
generator.generate_alt_text_from_base64(base64_image_string)
```

3. Service Mode:

In this mode, alternat exposes web API to generate alt-text for an image. Alternat use python based API framework - fastAPI to create APIs. fastAPI comes with a lightweight python server uvicorn which is used to expose the API. To start the server :

```
# Go to api folder
cd api

# run this command to start the service
uvicorn message_processor:app --port 8080 --host 0.0.0.0 --reload
```

The following web APIs are available:

```
# send a post request with base64 image to the Web Server
URL: http://localhost:8080/generate_text_base64
body: { base64: "base64_image_str"}

# send a post request with URL of the image to the Web Server
URL: http://localhost:8080/generate_text_url
body: { url: "url_of_the_image"}
```

1.3 Understanding alternat

alternat features are centered around tasks. Following table features break up across each task:

Task	Description	Options	Details
Collection	Scans the website and downloads images	Uses puppeteer to crawl the web page.	We are using apify - A puppeteer scrapper that crawls website using the headless chrome https://apify.com
Generate	Generates alt-text using, image captioning, OCR and images labels	Azure ML API	Use azure CV API for caption & OCR
		Google ML API based.	Use Google vision OCR and Image Labelling
		Open source based.	Use pytorch based model for OCR (EasyOCR) as well as image captioning

Library offers the flexibility of choosing either or both tasks and selecting suitable options from each task. Options are called drivers in alternat lingo. So, if you want to use azure for alt-text generation then you initialize the generator with azure driver. Same goes for google and "opensource" driver. Read the options as drivers.

There are few reasons for providing 3 drivers:

- Azure and google gives ready to use API, essentially lowering the barrier to get started.
- Most of the organizations don't have the data to train their own model for OCR and image captioning.
- Open source is a free alternative but can be little less accurate in few situations.

The tradeoff here is between cost and accuracy.

The OCR function is responsible for reading text from images. However, most of the ML API for OCR would treat single line as one text blob and might lead to unexpected out-of-order OCR text. For this reason, alternat comes with its own clustering implementation for OCR. alternat by default applies a clustering algorithm to create nearby data as a single text blob and combines them into a single line thereby generating more in-order human friendly OCR text.

1.4 Configuring alternat

Alternat can be configured at a global generator level or at the driver level with settings related to individual driver. We discuss configuration for both the generator and the driver for Application as well as Library mode below:

1.4.1 Configure Generator

Following configuration parameters are available for generator:

1. **DEBUG:** Setting the debug value to true will generate confidence level value for each of the OCR line detected by the system. In the debug output, it gives the line height and its ratio compared to the image height. This information is useful if you want to tune confidence level threshold value for drivers and OCR line height to image height ratio for filtering out insignificant text in the image.
2. **ENABLE_OCR_CLUSTERING:** Alternat comes with its own clustering rule which clusters (blobs) OCR data to create final OCR text from it. This is enabled by default and can be disabled by setting this value to false.

Application Mode:

You can find sample configuration for all the three drivers namely: opensource, azure, and google under "path-to-repo/sample/generator_driver_conf/<drivername>.json". Inside the JSON file you will find the following configuration parameter: GENERATOR: {DEBUG: false, ENABLE_OCR_CLUSTERING: true}

Here is an example to use the generator configuration for driver with name <drivername>:

```
python app.py generate --output-dir-path="results" --input-image-file-path="sample/
↪images_with_text/sample1.png" --driver-config-file-path="sample/generator_driver_
↪conf/<drivername>.json"
```

Where <drivername> is the name of the driver (opensource, azure or google)

Based on the setting for DEBUG and ENABLE_OCR_CLUSTERING in the sample/generator_driver_conf/<drivername>.json file the above command will generate the result and dump it inside "results" folder.

Library Mode:

In Library mode, you can directly interact with alternat library API to set the generator level config via a json object. Following is an example that will walk you through the same:

Once you have setup the alternat using "pip install alternat" you can open the python shell and run these commands to set generator config

```
# import the Generator library
from alternat.generation import Generator

# instantiate the Generator for opensource driver (you can pass "azure" or
```

(continues on next page)

(continued from previous page)

```
# "google" when instantiating to the let the library know the driver you want
# to use.
generator = Generator()

# get the current generator settings
# This will return the existing configuration
# {'DEBUG': False, 'ENABLE_OCR_CLUSTERING': True}
generator.get_config()

# set debug to true
generator_config = {"DEBUG": True}
generator.set_config(generator_config)

# or disable OCR clustering
generator_config = {"ENABLE_OCR_CLUSTERING": False}
generator.set_config(generator_config)

# or set values for both the parameters in one go
generator_config = {"DEBUG": True, "ENABLE_OCR_CLUSTERING": False}
generator.set_config(generator_config)

# run generator over an image and dump the output inside "results" folder
# this will run with DEBUG=true.
generator.generate_alt_text_from_file("sample/images_with_text/sample1.png", "results
↪")
```

1.4.2 Configure Driver

Generator comes with 3 drivers:

1. **Opensource:** Currently opensource drivers uses a pytorch based trained model for image captioning based on this repo It also uses EasyOCR for generating OCR text in case image have text in it. This is the default driver for generator, does not require any kind of registration and is free to use.
2. **Azure:** Uses Azure computer vision API to describe an image, generate OCR and also provide labels to the image. To use this driver, you need to register for computer vision API from Microsoft which will give you the SUBSCRIPTION_KEY and ENDPOINT URL to access the API.
3. **Google:** Uses Google computer vision API to generate OCR and provide labels to the image. To use this driver, you need to register for google computer vision API, download the google cloud service credentials file on your system and set the path to it in the driver configuration parameter ABSOLUTE_PATH_TO_CREDENTIAL_FILE (will be discussed below)

The following generator driver settings are available:

1. **CAPTION_CONFIDENCE_THRESHOLD:** Decimal based threshold to filter out caption data. For example, if you only want captions with confidence level above say 70%, then set this value to 0.70. This is most useful when using "azure" as driver as Microsoft compute vision API has support for describing an image. This option is also used in opensource driver.
2. **OCR_CONFIDENCE_THRESHOLD:** Decimal based threshold to filter out OCR data. For example, if you want OCR text with confidence level about say 50%, then set this value to 0.50.

3. **LABEL_CONFIDENCE_THRESHOLD:** Decimal based threshold to filter out label data. For example, if you want labels with confidence level about say 80%, then set this value to 0.80. This is useful when using google and azure driver as both the APIs have support for labelling image.
4. **OCR_HEIGHT_RATIO_TO_IMAGE_THRESHOLD:** Decimal based threshold to filter out OCR text which does not occupy a major portion of image and is practically irrelevant even if detected by the system. This threshold considers the ratio of the height of the text and the image to decide whether the text needs to be filtered out or not. For example, if you want OCR data only when the line height is greater than let's say 1.5% then set this value to 0.015 in the config.
5. **SUBSCRIPTION_KEY:** This is the subscription key for azure computer vision API, and is only required when using **azure** as the driver.
6. **ENDPOINT:** This is the API endpoint URL for azure computer vision API, and is only required when using **azure** as the driver.
7. **AZURE_RATE_LIMIT_ON:** This enables rate limiting when using azure driver in free account. Azure has a limit of 30 requests / minute in free tier account and when running alternat over a set of images this limit can hit very quickly. Alternat avoids this by sleeping for 30 sec by default and trying again. This setting is enabled by default. This setting is only required when using **azure** as the driver.
8. **AZURE_RATE_LIMIT_TIME_IN_SEC:** This is the rate limit time in sec. Alternat will sleep for these many seconds (30 by default) when azure rate limiting is reached in free tier account. To increase the sleep timer from 30 to say 40 seconds, set the value of this parameter to 40. This setting is only required when using **azure** as the driver.
9. **ABSOLUTE_PATH_TO_CREDENTIAL_FILE:** This setting holds the absolute path to the google credentials file (required to access the Google cloud services and computer vision API). This setting is only required when using **google** as the driver.

Let's see how to configure the above parameters in both the application and library mode.

Application Mode:

You can find sample configuration for all the three drivers namely: opensource, azure, and google under "path-to-repo/sample/generator_driver_conf/<drivername>.json". Inside the configuration file, you find all the parameters above with default values already set. To change these values and run generator use the following command:

```
python app.py generate --output-dir-path="results" --input-image-file-path="sample/
↪images_with_text/sample1.png" --driver-config-file-path="sample/generator_driver_
↪conf/<drivername>.json"
```

Where <drivername> is the name of the driver (opensource, azure or google)

Library Mode:

Once you have setup the alternat using "pip install alternat" you can open the python shell and run these commands to set generator config:

```
# import the Generator library
from alternat.generation import Generator

# instantiate the Generator for opensource driver (you can pass "azure" or
# "google" when instantiating to the let the library know the driver you want
# to use.

# for opensource
generator = Generator()

# or for azure
generator = Generator("azure")

# or for google
generator = Generator("google")

# get the current generator driver settings
# This will return the existing configuration based on the driver
generator.get_driver_config()

# set threshold value for caption, OCR and label
generator_driver_config = {"CAPTION_CONFIDENCE_THRESHOLD": 0.2, "OCR_CONFIDENCE_
↪THRESHOLD": 0.3, "LABEL_CONFIDENCE_THRESHOLD":0.75}
generator. generator.set_driver_config (generator_driver_config)

# or set OCR_HEIGHT_RATIO_TO_IMAGE_THRESHOLD
generator_driver_config = {"OCR_HEIGHT_RATIO_TO_IMAGE_THRESHOLD":0.015}
generator. generator.set_driver_config (generator_driver_config)

# or set subscription key and endpoint URL for azure
generator_driver_config = {"SUBSCRIPTION_KEY": "yoursubscriptionkey", "ENDPOINT":
↪"https://<ENTER_PROJECT_NAME>.cognitiveservices.azure.com/"}
generator. generator.set_driver_config (generator_driver_config)

# run generator over an image and dump the output inside "results" folder
# this will run with DEBUG=true.
generator.generate_alt_text_from_file("sample/images_with_text/sample1.png", "results
↪")
```

1.4.3 Configure Web API

Web API use opensource driver by default. Both application mode and Web API internally rely on the alternat library. To configure Web API for different driver and configuration the following changes are required:

1. Navigate to **api** folder.
2. Locate file **message_processor.py**. Here you will see the Generator being instantiated (just like in library mode).
3. Use the samples from **Library Mode** section under *Configure Driver* to configure web API using alternat library.

Here is an example to say change the driver to azure. In **message_processor.py**,

```
# find the following statement
generator = Generator()

# for azure, change the statement to this
generator = Generator("azure")
```

(continues on next page)

(continued from previous page)

```
# following statements change the driver specific configuration
# add this to set subscription key and endpoint URL for azure
generator_driver_config = {"SUBSCRIPTION_KEY": "yoursubscriptionkey", "ENDPOINT":
↪"https://<ENTER_PROJECT_NAME>.cognitiveservices.azure.com/"}

# add this to update the threshold value for caption, OCR and label
generator_driver_config = {"CAPTION_CONFIDENCE_THRESHOLD": 0.2, "OCR_CONFIDENCE_
↪THRESHOLD": 0.3, "LABEL_CONFIDENCE_THRESHOLD":0.75}

#  add this to update OCR_HEIGHT_RATIO_TO_IMAGE_THRESHOLD
generator_driver_config = {"OCR_HEIGHT_RATIO_TO_IMAGE_THRESHOLD":0.015}

# add this to set the configuration
generator.set_driver_config(generator_driver_config)
```

1.5 Using alternat

1.5.1 Application Mode via CLI (Command Line Interface)

Alternat comes with a python-based CLI app **app.py** which provides commands to run collection and generation task. Below we give some example on how to use this app:

Collection:

1. **Collect and store images from a URL and store them in a folder sample/images/test**

```
python app.py collect --url="https://page_url" --output-dir-path="sample/
↪images/test"
```

2. **Collect and store the images from a URL recursively and store them in a folder sample/images/test**

```
python app.py collect --url="https://page_url" --output-dir-path="sample/
↪images/test" --download-recursively=true
```

Generation

1. **Generate alt-text for the images in a directory name sample/images_with_text and save data in directory structure results**

```
python app.py generate --input-dir-path="sample/images_with_text" --output-
↪dir-path="results"
```

2. **Generate alt-text for a single image in a folder sample/images_with_text and save its result in a directory inside results:**

```
python app.py generate --input-image-file-path=./sample/images_with_text/
↪sample1.png --output-dir-path=./results
```

3. **Generate alt-text based on user defined (custom) config for driver azure :**

```
python app.py generate --input-image-file-path=./sample/images/sample1.jpg --
↪output-dir-path=./results --driver-config-file-path=./sample/generator_
↪driver_conf/azure.json
```

The above command can be changed based on the driver by using the driver sample files under sample/generator_driver_conf. For example, to use google driver change the –driver-config-file-path to "sample/generator_driver_conf/google.json".

1.5.2 Library Mode

With library mode, users can integrate alternat in their existing applications as well. In library mode the package is installed via pip and can be import into python applications directly. Below are some examples on using the library mode for collection and generation tasks:

Collection:

Download the image from a site given its URL to specified folder location:

```
# import the alternat library
from alternat.collection import Collector

# instantiate the collector
collector = Collector()

# Download images from url and saves image files in  output_dir_path
# Optional parameters, download_recursive if True crawls whole site␣
↪mentioned in
# url by visiting each link recursively and downloads images
# collect_using_apify in future more crawlers will be supported this␣
↪parameter
# ensures that apify crawler is used.
collector.process(url, output_dir_path, download_recursive, collect_using_
↪apify
```

Generator:

1. **Generate alt-text for a single image in a folder "results" and save its result in a directory inside result/test:**

```
# import the Generator
from alternat.generation import Generator

# instantiate the generator (uses opensource driver by default)
generator = Generator()

# to use a specific driver pass the driver name when instantiating. For e.g,␣
↪to use
# azure driver use
generator = Generator("azure")

# generate the alt text
generator.generate_alt_text_from_file("sample/images_with_text/sample1.png",
↪"results")
```

(continues on next page)

(continued from previous page)

2. **Generate alt-text for a single image in base64 image:**

```
# import the Generator
from alternat.generation import Generator

# instantiate the generator (uses opensource driver by default)
generator = Generator()

# generate the alt text
base64_image_str = "base64-image-data-here"
generator.generate_alt_text_from_base64(base64_image_str)
```

1.5.3 Service Mode

In this mode, alternat exposes web API to generate alt-text for an image. Alternat use python based API framework - fastAPI to create APIs. fastAPI comes with a lightweight python server uvicorn which is used to expose the API. To start the server :

```
# Go to api folder
cd api

# run this command to start the service
uvicorn message_processor:app --port 8080 --host 0.0.0.0 --reload
```

The following web APIs are available:

```
# send a post request with base64 image to the Web Server
URL: http://localhost:8080/generate_text_base64
body: { base64: "base64_image_str"}

# send a post request with URL of the image to the Web Server
URL: http://localhost:8080/generate_text_url
body: { url: "url_of_the_image"}
```

1.6 Extending alternat

1.6.1 Adding new generator driver

To add a new driver, use the existing driver architecture. Alternat currently supports 3 drivers

1. google
2. azure
3. opensource

Note:

All the drivers need to output JSON data and adhere to the schema here : alternat/generation/data/analyzer_output.json. **Failing to do so would impact proper functioning of the driver**.

Follow the steps to add a new driver:

1. Create a new folder with name custom inside alternat/generation

```
cd alternat/generation
mkdir custom
```

2. Create the same file structure as in the existing drivers

```
# move inside **custom** folder
cd custom

# create the following files inside **custom** folder
# command will differ on windows shell
touch analyzer.py
touch config.py
```

3. Copy paste the contents of config file from opensource/config.py
4. Copy paste the contents of analyzer file from opensource/analyzer.py
5. Edit the following methods in custom/analyzer.py to add your own functionality.
    1. open analyzer.py in custom/analyzer.py
    2. overwrite **describe_image** method to add your custom implementation of image captioning.

```
# overwrite this method to extract caption

  def describe_image(self, image: PIL_Image):
  """Describe image using your custom solution.

  :param image: PIL Image object
  :type image: PIL_Image
  """

      # add the extracted caption data here instead of empty dictionary
      # the data needs to adhere to the sample JSON data at alternat/
↪data/analyzer_output.json
      self.data[self.actions.DESCRIBE] = {}
```

    3. overwrite the **extract_labels** method to add your custom implementation of getting label data.

```
def extract_labels(self, image: PIL_Image):
"""Extract labels of image using open source solution.

:param image: PIL Image object.
:type image: PIL_Image
"""

    # add the extracted label data here instead of empty dictionary
    # the data needs to adhere to the sample JSON data at alternat/
↪data/analyzer_output.json
    self.data[self.actions.LABELS] = {}
```

    4. overwrite the **ocr_analysis** method to add your custom implementation for ocr extraction.

```
def ocr_analysis(self, image: PIL_Image):
"""Does OCR Analysis using EasyOCR.

:param image: PIL Image object.
```

(continues on next page)

(continued from previous page)

```
:type image: PIL_Image
"""

    # add the ocr extracted data here instead of empty dictionary
    # the data needs to adhere to the sample JSON data at alternat/
↪data/analyzer_output.json
    self.data[self.actions.OCR] = {}
```

6. Expose the driver to the generator library so it is available across the application. Following are the steps to the same:

    1. open alternat/generation/generator.py (This is the library for alternat)
    2. Import the Analyzer & Config class of your custom driver.

```
from alternat.generation.custom.config import Config as␣
↪CustomAnalyzerConfig
from alternat.generation.custom.analyze import AnalyzeImage as␣
↪CustomAnalyzer
```

    2. find the **Drivers** class and add your custom driver there.

```
class Drivers:
"""Driver name for alternat Library.
"""
    OPEN = "opensource"
    MICROSOFT = "azure"
    GOOGLE = "google"

    # custom driver added here
    CUSTOM = "custom"
```

    3. modify **_set_current_driver** method and add your custom driver in if-elif-else statements.

```
# TODO: This behavior will be changed later one so no method␣
↪modification is required.

def _set_current_driver(self):
"""Sets the current driver internally within the application.

:raises InvalidGeneratorDriver: Driver name is invalid or not␣
↪implemented.
"""
    if self.CURRENT_DRIVER == Drivers.OPEN:
        setattr(Config, Config.CURRENT_ANALYZER.__name__,␣
↪OpenAnalyzer)
    elif self.CURRENT_DRIVER == Drivers.MICROSOFT:
        setattr(Config, Config.CURRENT_ANALYZER.__name__,␣
↪MicrosoftAnalyzer)
    elif self.CURRENT_DRIVER == Drivers.GOOGLE:
        setattr(Config, Config.CURRENT_ANALYZER.__name__,␣
↪GoogleAnalyzer)

    # custom driver added
    elif self.CURRENT_DRIVER == Drivers.CUSTOM:
        setattr(Config, Config.CURRENT_ANALYZER.__name__,␣
↪CustomAnalyzer)
```

(continues on next page)

(continued from previous page)

```
    else:
        raise InvalidGeneratorDriver(self.ALLOWED_DRIVERS)
```

4. modify **_get_current_driver** method and add your custom driver in if-elif-else statements.

```
def _get_current_driver_conf_cls(self):
"""Retreives the driver configuration class based on the␣
↪currently driver

:raises InvalidGeneratorDriver: Driver name is invalid or not␣
↪implemented.
:return: [description]
:rtype: [type]
"""
    current_driver_cls = None
    if self.CURRENT_DRIVER == Drivers.OPEN:
        current_driver_cls = OpenAnalyzerConfig
    elif self.CURRENT_DRIVER == Drivers.MICROSOFT:
        current_driver_cls = MicrosoftAnalyzerConfig
    elif self.CURRENT_DRIVER == Drivers.GOOGLE:
        current_driver_cls = GoogleAnalyzerConfig

    # custom driver added
    elif self.CURRENT_DRIVER == Drivers.CUSTOM:
        current_driver_cls = CustomAnalyzerConfig
    else:
        raise InvalidGeneratorDriver(self.ALLOWED_DRIVERS)

    return current_driver_cls
```

7. The new custom driver will be available for use now.

1.6.2 Training image caption model for alternat

Alternat uses pytorch based image caption method based on repo by Sagar Vinodababu Follow these steps derived from *Sagar Vinodababu* repo to understand, train and evaluate caption model to be used for alternat:

This repo implements the Show, Attend, and Tell. paper. This is by no means the current state-of-the-art, but is still pretty darn amazing. The authors' original implementation can be found here.

This model learns _where_ to look.

As you generate a caption, word by word, you can see the model's gaze shifting across the image.

This is possible because of its _Attention_ mechanism, which allows it to focus on the part of the image most relevant to the word it is going to utter next.

Concepts

- **Image captioning**.
- **Encoder-Decoder architecture**. Typically, a model that generates sequences will use an Encoder to encode the input into a fixed form and a Decoder to decode it, word by word, into a sequence.
- **Attention**. The use of Attention networks is widespread in deep learning, and with good reason. This is a way for a model to choose only those parts of the encoding that it thinks is relevant to the task at hand. The same mechanism you see employed here can be used in any model where the Encoder's output has multiple points in space or time. In image captioning, you consider some pixels more important than others. In sequence to sequence tasks like machine translation, you consider some words more important than others.
- **Transfer Learning**. This is when you borrow from an existing model by using parts of it in a new model. This is almost always better than training a new model from scratch (i.e., knowing nothing). As you will see, you can always fine-tune this second-hand knowledge to the specific task at hand. Using pretrained word embeddings is a dumb but valid example. For our image captioning problem, we will use a pretrained Encoder, and then fine-tune it as needed.
- **Beam Search**. This is where you don't let your Decoder be lazy and simply choose the words with the _best_ score at each decode-step. Beam Search is useful for any language modeling problem because it finds the most optimal sequence.

Overview

In this section, we will present an overview of this model. If you're already familiar with it, you can skip straight to the Implementation section or the commented code.

Encoder

The Encoder **encodes the input image with 3 color channels into a smaller image with "learned" channels**.

This smaller encoded image is a summary representation of all that's useful in the original image.

Since we want to encode images, we use Convolutional Neural Networks (CNNs).

We don't need to train an encoder from scratch. Why? Because there are already CNNs trained to represent images.

For years, people have been building models that are extraordinarily good at classifying an image into one of a thousand categories. It stands to reason that these models capture the essence of an image very well.

We have chosen to use the **101 layered Residual Network trained on the ImageNet classification task**, already available in PyTorch. As stated earlier, this is an example of Transfer Learning. You have the option of fine-tuning it to improve performance.

These models progressively create smaller and smaller representations of the original image, and each subsequent representation is more "learned", with a greater number of channels. The final encoding produced by our ResNet-101 encoder has a size of 14x14 with 2048 channels, i.e., a *2048, 14, 14* size tensor.

I encourage you to experiment with other pre-trained architectures. The paper uses a VGGnet, also pretrained on ImageNet, but without fine-tuning. Either way, modifications are necessary. Since the last layer or two of these models are linear layers coupled with softmax activation for classification, we strip them away.

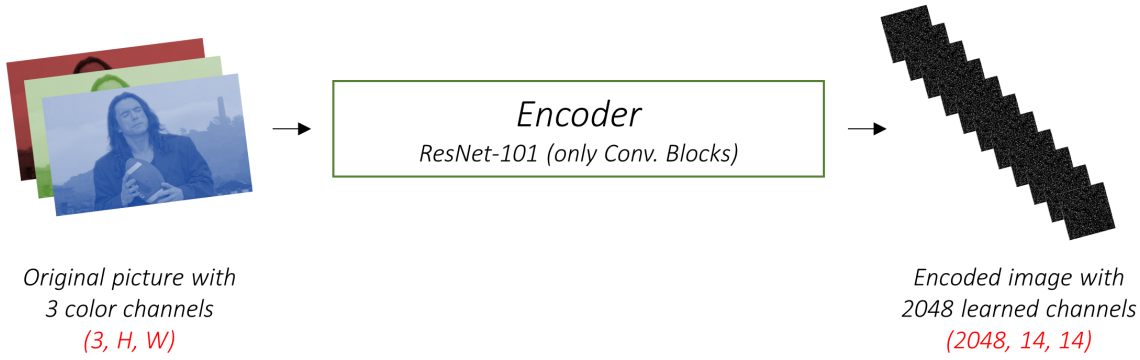


Fig. 1: ResNet Encoder

Decoder

The Decoder's job is to **look at the encoded image and generate a caption word by word**.

Since it's generating a sequence, it would need to be a Recurrent Neural Network (RNN). We will use an LSTM.

In a typical setting without Attention, you could simply average the encoded image across all pixels. You could then feed this, with or without a linear transformation, into the Decoder as its first hidden state and generate the caption. Each predicted word is used to generate the next word.

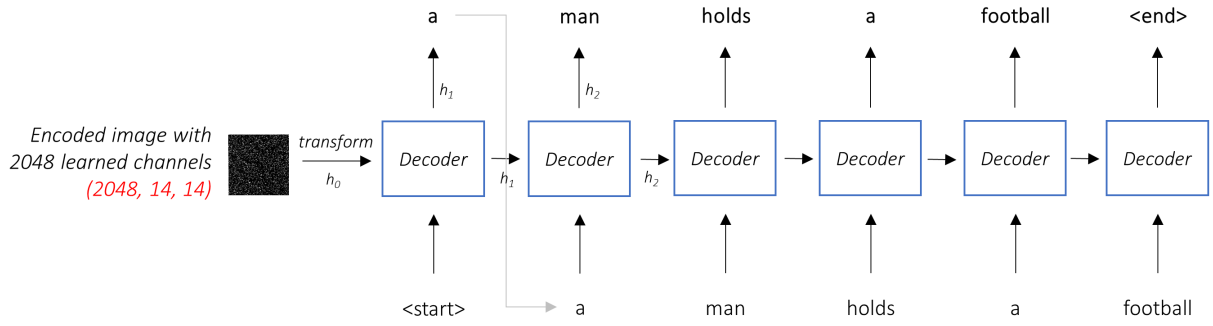


Fig. 2: Decoder without Attention

In a setting _with_ Attention, we want the Decoder to be able to **look at different parts of the image at different points in the sequence**. For example, while generating the word *football* in *a man holds a football*, the Decoder would know to focus on – you guessed it – the football!

Instead of the simple average, we use the _weighted_ average across all pixels, with the weights of the important pixels being greater. This weighted representation of the image can be concatenated with the previously generated word at each step to generate the next word.

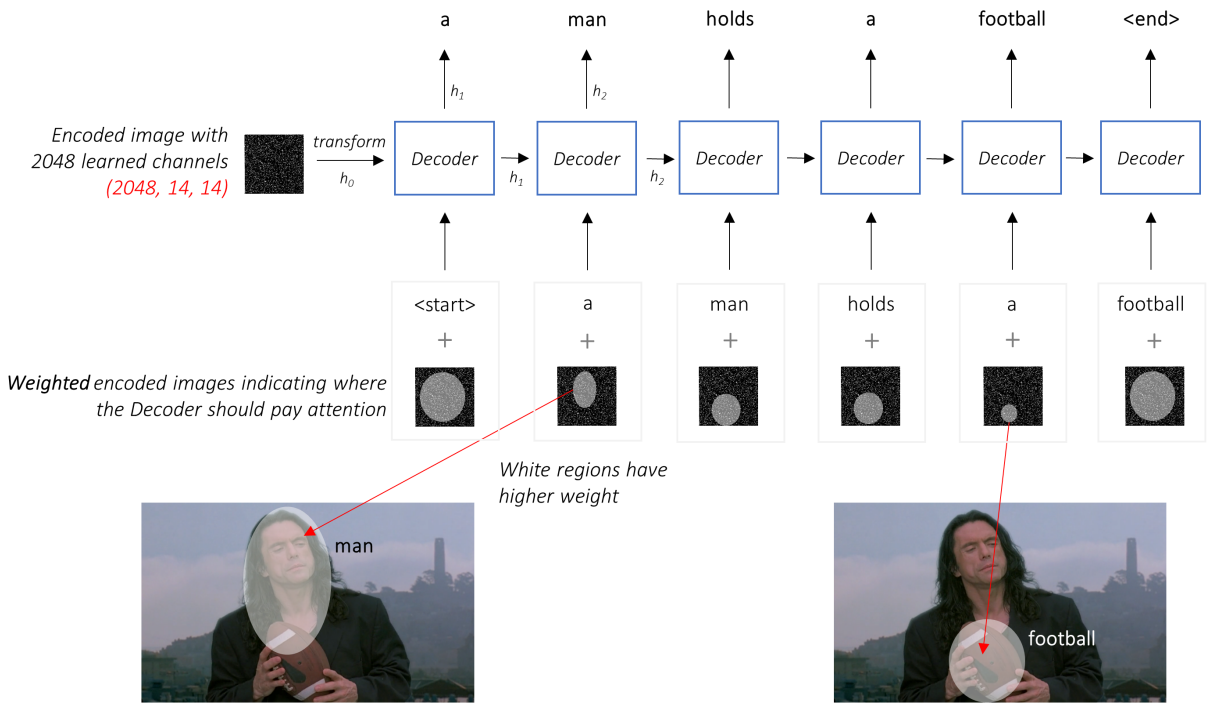


Fig. 3: Decoder with Attention

Attention

The Attention network **computes these weights**.

Intuitively, how would you estimate the importance of a certain part of an image? You would need to be aware of the sequence you have generated *so far*, so you can look at the image and decide what needs describing next. For example, after you mention *a man*, it is logical to declare that he is *holding a football*.

This is exactly what the Attention mechanism does – it considers the sequence generated thus far, and _attends_ to the part of the image that needs describing next.

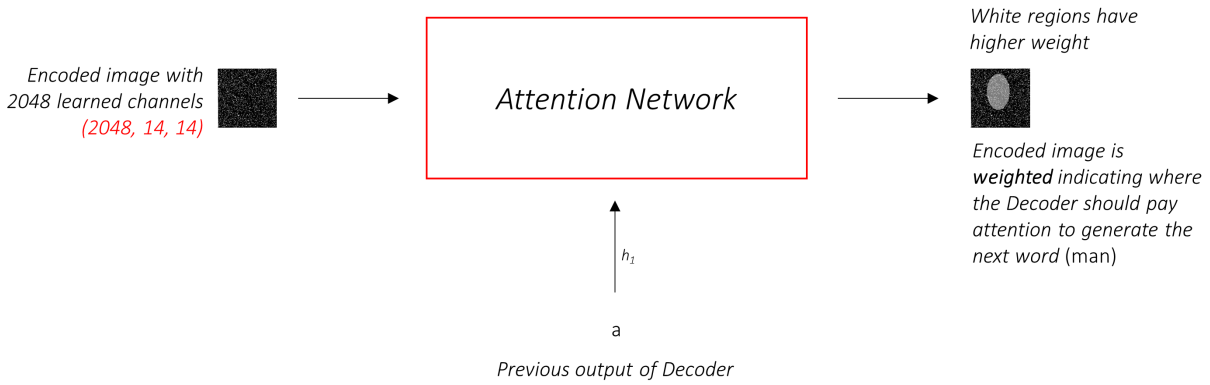


Fig. 4: Attention

We will use _soft_ Attention, where the weights of the pixels add up to 1. If there are P pixels in our encoded image,

then at each timestep t –

$$\sum_{p}^{P} \alpha_{p,t} = 1$$

Fig. 5: weights

You could interpret this entire process as computing the **probability that a pixel is _the_ place to look to generate the next word**.

Putting it all together

It might be clear by now what our combined network looks like.

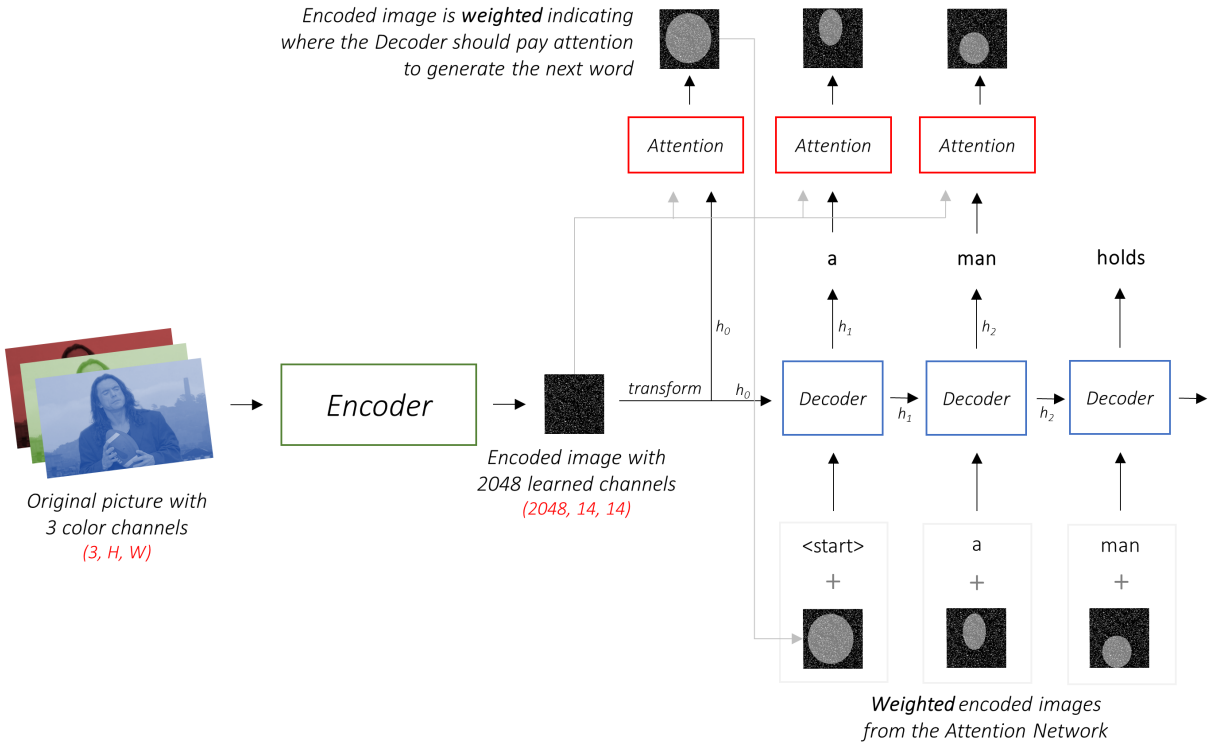


Fig. 6: Putting it all together

- Once the Encoder generates the encoded image, we transform the encoding to create the initial hidden state h (and cell state C) for the LSTM Decoder.
- At each decode step, - the encoded image and the previous hidden state is used to generate weights for each pixel in the Attention network. - the previously generated word and the weighted average of the encoding are fed to the LSTM Decoder to generate the next word.

Beam Search

We use a linear layer to transform the Decoder's output into a score for each word in the vocabulary.

The straightforward – and greedy – option would be to choose the word with the highest score and use it to predict the next word. But this is not optimal because the rest of the sequence hinges on that first word you choose. If that choice isn't the best, everything that follows is sub-optimal. And it's not just the first word – each word in the sequence has consequences for the ones that succeed it.

It might very well happen that if you'd chosen the _third_ best word at that first step, and the _second_ best word at the second step, and so on… _that_ would be the best sequence you could generate.

It would be best if we could somehow _not_ decide until we've finished decoding completely, and **choose the sequence that has the highest _overall_ score from a basket of candidate sequences**.

Beam Search does exactly this.

- At the first decode step, consider the top k candidates.
- Generate k second words for each of these k first words.
- Choose the top k [first word, second word] combinations considering additive scores.
- For each of these k second words, choose k third words, choose the top k [first word, second word, third word] combinations.
- Repeat at each decode step.
- After k sequences terminate, choose the sequence with the best overall score.

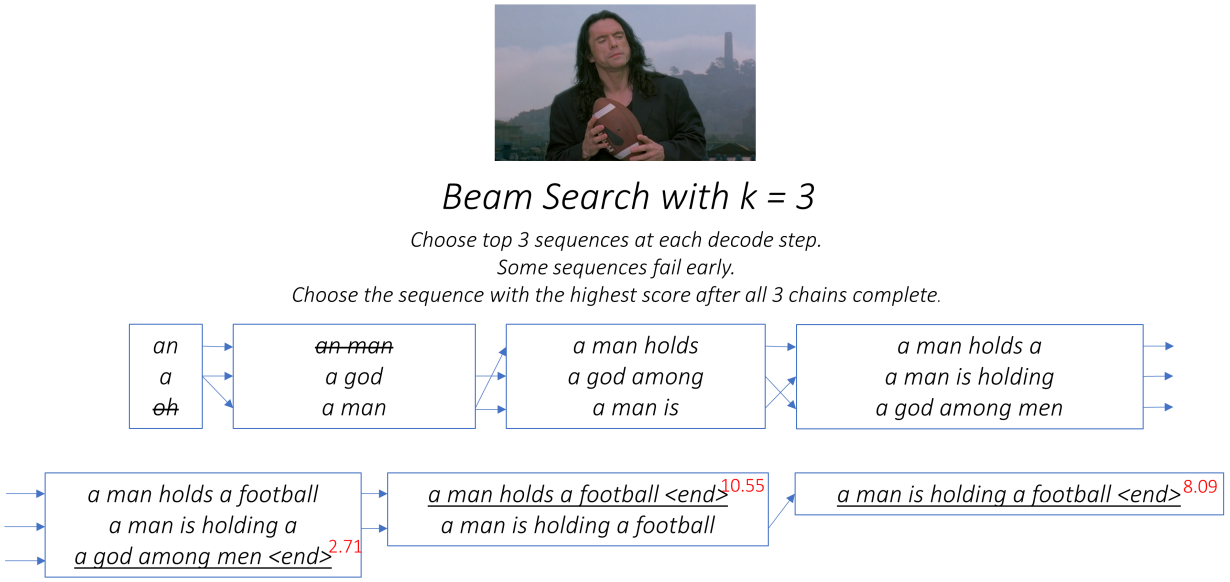


Fig. 7: Beam Search example

As you can see, some sequences (striked out) may fail early, as they don't make it to the top k at the next step. Once k sequences (underlined) generate the <*end*> token, we choose the one with the highest score.

Implementation

The sections below briefly describe the implementation.

They are meant to provide some context, but **details are best understood directly from the code**, which is quite heavily commented.

Dataset

I'm using the MSCOCO '14 Dataset. You'd need to download the Training (13GB) and Validation (6GB) images.

We will use Andrej Karpathy's training, validation, and test splits . This zip file contain the captions. You will also find splits and captions for the Flicker8k and Flicker30k datasets, so feel free to use these instead of MSCOCO if the latter is too large for your computer.

Inputs to model

We will need three inputs.

Images

Since we're using a pretrained Encoder, we would need to process the images into the form this pretrained Encoder is accustomed to.

Pretrained ImageNet models available as part of PyTorch's *torchvision* module. This page details the preprocessing or transformation we need to perform – pixel values must be in the range [0,1] and we must then normalize the image by the mean and standard deviation of the ImageNet images' RGB channels.:

```
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
```

Also, PyTorch follows the NCHW convention, which means the channels dimension (C) must precede the size dimensions.

We will resize all MSCOCO images to 256x256 for uniformity.

Therefore, **images fed to the model must be a `Float` tensor of dimension `N, 3, 256, 256`**, and must be normalized by the aforesaid mean and standard deviation. *N* is the batch size.

Captions

Captions are both the target and the inputs of the Decoder as each word is used to generate the next word.

To generate the first word, however, we need a *zeroth* word, <*start*>.

At the last word, we should predict <*end*> the Decoder must learn to predict the end of a caption. This is necessary because we need to know when to stop decoding during inference.

<*start*> *a man holds a football* <*end*>

Since we pass the captions around as fixed size Tensors, we need to pad captions (which are naturally of varying length) to the same length with <*pad*> tokens.

<*start*> *a man holds a football* <*end*> <*pad*> <*pad*> <*pad*>. . . .

Furthermore, we create a *word_map* which is an index mapping for each word in the corpus, including the <*start*>,`<end>`, and <*pad*> tokens. PyTorch, like other libraries, needs words encoded as indices to look up embeddings for them or to identify their place in the predicted word scores.

9876 1 5 120 1 5406 9877 9878 9878 9878....

Therefore, **captions fed to the model must be an `Int` tensor of dimension `N, L`** where *L* is the padded length.

Caption Lengths

Since the captions are padded, we would need to keep track of the lengths of each caption. This is the actual length + 2 (for the <*start*> and <*end*> tokens).

Caption lengths are also important because you can build dynamic graphs with PyTorch. We only process a sequence upto its length and don't waste compute on the *<pad>*s.

Therefore, **caption lengths fed to the model must be an `Int` tensor of dimension `N`**.

Data pipeline

See *create_input_files()* in utils.py.

This reads the data downloaded and saves the following files –

- An **HDF5 file containing images for each split in an `I, 3, 256, 256` tensor**, where *I* is the number of images in the split. Pixel values are still in the range [0, 255], and are stored as unsigned 8-bit `Int`s.
- A **JSON file for each split with a list of `N_c` * `I` encoded captions**, where *N_c* is the number of captions sampled per image. These captions are in the same order as the images in the HDF5 file. Therefore, the `i`th caption will correspond to the `i // N_c`th image.
- A **JSON file for each split with a list of `N_c` * `I` caption lengths**. The `i`th value is the length of the `i`th caption, which corresponds to the `i // N_c`th image.
- A **JSON file which contains the `word_map`**, the word-to-index dictionary.

Before we save these files, we have the option to only use captions that are shorter than a threshold, and to bin less frequent words into an <*unk*> token.

We use HDF5 files for the images because we will read them directly from disk during training / validation. They're simply too large to fit into RAM all at once. But we do load all captions and their lengths into memory.

See *CaptionDataset* in datasets.py.

This is a subclass of PyTorch Dataset . It needs a ***len*** method defined, which returns the size of the dataset, and a ***getitem*** method which returns the `i`th image, caption, and caption length.

We read images from disk, convert pixels to [0,255], and normalize them inside this class.

The *Dataset* will be used by a PyTorch DataLoader in *train.py* to create and feed batches of data to the model for training or validation.

Encoder

See *Encoder* in models.py.

We use a pretrained ResNet-101 already available in PyTorch's *torchvision* module. Discard the last two layers (pooling and linear layers), since we only need to encode the image, and not classify it.

We do add an *AdaptiveAvgPool2d()* layer to **resize the encoding to a fixed size**. This makes it possible to feed images of variable size to the Encoder. (We did, however, resize our input images to *256, 256* because we had to store them together as a single tensor.)

Since we may want to fine-tune the Encoder, we add a *fine_tune()* method which enables or disables the calculation of gradients for the Encoder's parameters. We **only fine-tune convolutional blocks 2 through 4 in the ResNet**, because the first convolutional block would have usually learned something very fundamental to image processing, such as detecting lines, edges, curves, etc. We don't mess with the foundations.

Attention

See *Attention* in models.py.

The Attention network is simple – it's composed of only linear layers and a couple of activations.

Separate linear layers **transform both the encoded image (flattened to `N, 14 * 14, 2048`) and the hidden state (output) from the Decoder to the same dimension**, viz. the Attention size. They are then added and ReLU activated. A third linear layer **transforms this result to a dimension of 1**, whereupon we **apply the softmax to generate the weights** *alpha*.

Decoder

See *DecoderWithAttention* in models.py.

The output of the Encoder is received here and flattened to dimensions *N, 14 * 14, 2048*. This is just convenient and prevents having to reshape the tensor multiple times.

We **initialize the hidden and cell state of the LSTM** using the encoded image with the *init_hidden_state()* method, which uses two separate linear layers.

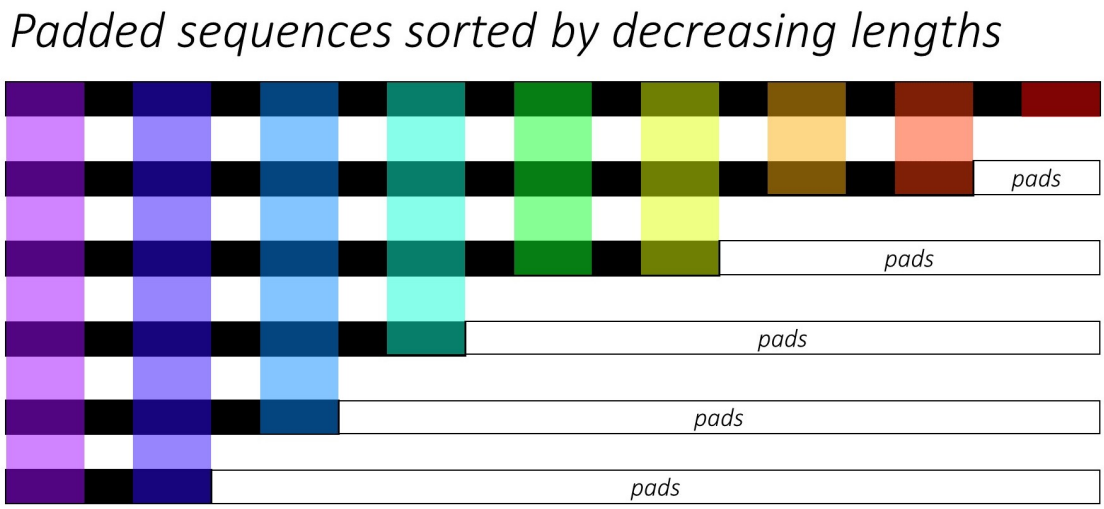
At the very outset, we **sort the `N` images and captions by decreasing caption lengths**. This is so that we can process only _valid_ timesteps, i.e., not process the `<pad>`s.

We can iterate over each timestep, processing only the colored regions, which are the **_effective_ batch size** *N_t* at that timestep. The sorting allows the top *N_t* at any timestep to align with the outputs from the previous step. At the third timestep, for example, we process only the top 5 images, using the top 5 outputs from the previous step.

This **iteration is performed _manually_ in a `for` loop** with a PyTorch LSTMCell instead of iterating automatically without a loop with a PyTorch LSTM. This is because we need to execute the Attention mechanism between each decode step. An *LSTMCell* is a single timestep operation, whereas an *LSTM* would iterate over multiple timesteps continously and provide all outputs at once.

We **compute the weights and attention-weighted encoding** at each timestep with the Attention network. In section *4.2.1* of the paper, they recommend **passing the attention-weighted encoding through a filter or gate**. This gate is a sigmoid activated linear transform of the Decoder's previous hidden state. The authors state that this helps the Attention network put more emphasis on the objects in the image.

We **concatenate this filtered attention-weighted encoding with the embedding of the previous word** (*<start>* to begin), and run the *LSTMCell* to **generate the new hidden state (or output)**. A linear layer **transforms this new hidden state into scores for each word in the vocabulary**, which is stored.



We also store the weights returned by the Attention network at each timestep. You will see why soon enough.

Training

Before you begin, make sure to save the required data files for training, validation, and testing. To do this, run the contents of create_input_files.py after pointing it to the the Karpathy JSON file and the image folder containing the extracted *train2014* and *val2014* folders from your downloaded data.

See train.py.

The parameters for the model (and training it) are at the beginning of the file, so you can easily check or modify them should you wish to.

To **train your model from scratch**, simply run this file –

python train.py

To **resume training at a checkpoint**, point to the corresponding file with the *checkpoint* parameter at the beginning of the code.

Note that we perform validation at the end of every training epoch.

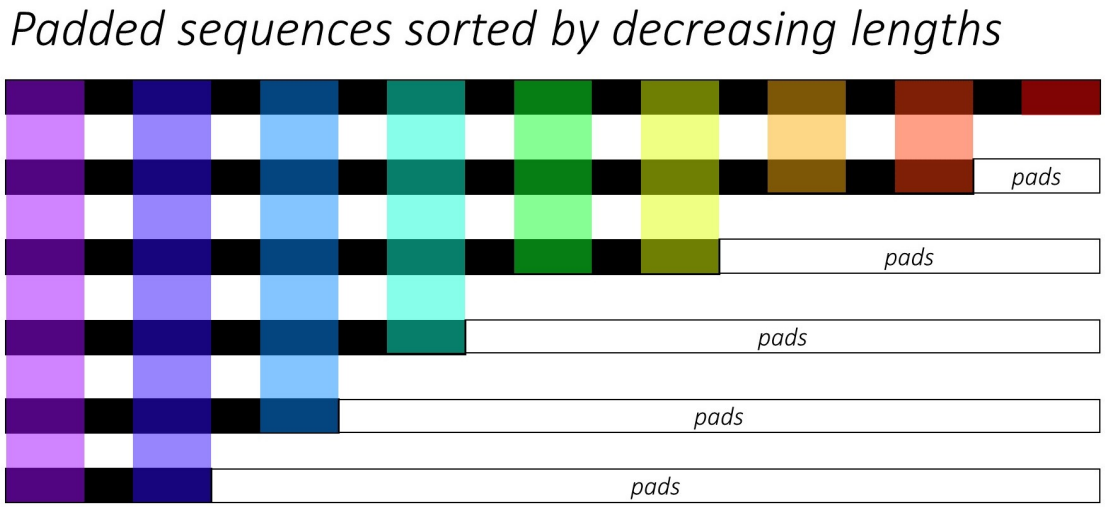
Loss Function

Since we're generating a sequence of words, we use CrossEntropyLoss. You only need to submit the raw scores from the final layer in the Decoder, and the loss function will perform the softmax and log operations.

The authors of the paper recommend using a second loss – a "**doubly stochastic regularization**". We know the weights sum to 1 at a given timestep. But we also encourage the weights at a single pixel p to sum to 1 across _all_ timesteps T –

This means we want the model to attend to every pixel over the course of generating the entire sequence. Therefore, we try to **minimize the difference between 1 and the sum of a pixel's weights across all timesteps**.

We do not compute losses over the padded regions. An easy way to do get rid of the pads is to use PyTorch's pack_padded_sequence(), which flattens the tensor by timestep while ignoring the padded regions. You can now aggregate the loss over this flattened tensor.



Note – This function is actually used to perform the same dynamic batching (i.e., processing only the effective batch size at each timestep) we performed in our Decoder, when using an *RNN* or *LSTM* in PyTorch. In this case, PyTorch handles the dynamic variable-length graphs internally. You can see an example in dynamic_rnn.py in my other tutorial on sequence labeling. We would have used this function along with an *LSTM* in our Decoder if we weren't manually iterating because of the Attention network.

Early stopping with BLEU

To evaluate the model's performance on the validation set, we will use the automated BiLingual Evaluation Understudy (BLEU) evaluation metric. This evaluates a generated caption against reference caption(s). For each generated caption, we will use all *N_c* captions available for that image as the reference captions.

The authors of the **Show, Attend and Tell** paper observe that correlation between the loss and the BLEU score breaks down after a point, so they recommend to stop training early on when the BLEU score begins to degrade, even if the loss continues to decrease.

I used the BLEU tool available in the NLTK module.

Note that there is considerable criticism of the BLEU score because it doesn't always correlate well with human judgment. The authors also report the METEOR scores for this reason, but I haven't implemented this metric.

Remarks

I recommend you train in stages.

I first trained only the Decoder, i.e. without fine-tuning the Encoder, with a batch size of *80*. I trained for 20 epochs, and the BLEU-4 score peaked at about *23.25* at the 13th epoch. I used the Adam() optimizer with an initial learning rate of *4e-4*.

I continued from the 13th epoch checkpoint allowing fine-tuning of the Encoder with a batch size of *32*. The smaller batch size is because the model is now larger because it contains the Encoder's gradients. With fine-tuning, the score rose to *24.29* in just about 3 epochs. Continuing training would probably have pushed the score slightly higher but I had to commit my GPU elsewhere.

An important distinction to make here is that I'm still supplying the ground-truth as the input at each decode-step during validation, regardless of the word last generated. This is called **Teacher Forcing**. While this is commonly used during training to speed-up the process, as we are doing, conditions during validation must mimic real inference conditions as much as possible. I haven't implemented batched inference yet – where each word in the caption is generated from the previously generated word, and terminates upon hitting the <*end*> token.

Since I'm teacher-forcing during validation, the BLEU score measured above on the resulting captions _does **not**_ reflect real performance. In fact, the BLEU score is a metric designed for comparing naturally generated captions to ground-truth captions of differing length. Once batched inference is implemented, i.e. no Teacher Forcing, early-stopping with the BLEU score will be truly 'proper'.

With this in mind, I used eval.py to compute the correct BLEU-4 scores of this model checkpoint on the validation and test sets _without_ Teacher Forcing, at different beam sizes –

Beam Size | Validation BLEU-4 | Test BLEU-4 | :—: | :—: | :—: | 1 | 29.98 | 30.28 | 3 | 32.95 | 33.06 | 5 | 33.17 | 33.29 |

The test score is higher than the result in the paper, and could be because of how our BLEU calculators are parameterized, the fact that I used a ResNet encoder, and actually fine-tuned the encoder – even if just a little.

Also, remember – when fine-tuning during Transfer Learning, it's always better to use a learning rate considerably smaller than what was originally used to train the borrowed model. This is because the model is already quite optimized, and we don't want to change anything too quickly. I used *Adam()* for the Encoder as well, but with a learning rate of *1e-4*, which is a tenth of the default value for this optimizer.

On a Titan X (Pascal), it took 55 minutes per epoch without fine-tuning, and 2.5 hours with fine-tuning at the stated batch sizes.

Model Checkpoint

You can download this pretrained model and the corresponding *word_map* here and here respectively.

Note that this checkpoint should be loaded directly with PyTorch, or passed to caption.py – see below.

Inference

See caption.py.

During inference, we _cannot_ directly use the *forward()* method in the Decoder because it uses Teacher Forcing. Rather, we would actually need to **feed the previously generated word to the LSTM at each timestep**.

caption_image_beam_search() reads an image, encodes it, and applies the layers in the Decoder in the correct order, while using the previously generated word as the input to the LSTM at each timestep. It also incorporates Beam Search.

visualize_att() can be used to visualize the generated caption along with the weights at each timestep as seen in the examples.

To **caption an image** from the command line, point to the image, model checkpoint, word map (and optionally, the beam size) as follows –

python caption.py –img='path/to/image.jpeg' –model='path/to/BEST_checkpoint_coco_5_cap_per_img_5_min_word_freq.pth.tar' –word_map='path/to/WORDMAP_coco_5_cap_per_img_5_min_word_freq.json' –beam_size=5

Alternatively, use the functions in the file as needed.

Also see eval.py, which implements this process for calculating the BLEU score on the validation set, with or without Beam Search.

FAQs

You said soft attention. Is there, um, a hard attention?

Yes, the **Show, Attend and Tell** paper uses both variants, and the Decoder with "hard" attention performs marginally better.

In *soft* attention, which we use here, you're computing the weights *alpha* and using the weighted average of the features across all pixels. This is a deterministic, differentiable operation.

In *hard* attention, you are choosing to just sample some pixels from a distribution defined by *alpha*. Note that any such probabilistic sampling is non-deterministic or _stochastic_, i.e. a specific input will not always produce the same output. But since gradient descent presupposes that the network is deterministic (and therefore differentiable), the sampling is reworked to remove its stochasticity. My knowledge of this is fairly superficial at this point – I will update this answer when I have a more detailed understanding.

How do I use an attention network for an NLP task like a sequence to sequence model?

Much like you use a CNN to generate an encoding with features at each pixel, you would use an RNN to generate encoded features at each timestep i.e. word position in the input.

Without attention, you would use the Encoder's output at the last timestep as the encoding for the entire sentence, since it would also contain information from prior timesteps. The Encoder's last output now bears the burden of having to encode the entire sentence meaningfully, which is not easy, especially for longer sentences.

With attention, you would attend over the timesteps in the Encoder's output, generating weights for each timestep/word, and take the weighted average to represent the sentence. In a sequence to sequence task like machine translation, you would attend to the relevant words in the input as you generate each word in the output.

You could also use Attention without a Decoder. For example, if you want to classify text, you can attend to the important words in the input just once to perform the classification.

Can we use Beam Search during training?

Not with the current loss function, but yes. This is not common at all.

What is Teacher Forcing?

Teacher Forcing is when we use the ground truth captions as the input to the Decoder at each timestep, and not the word it generated in the previous timestep. It's common to teacher-force during training since it could mean faster convergence of the model. But it can also learn to depend on being told the correct answer, and exhibit some instability in practice.

It would be ideal to train using Teacher Forcing only some of the time, based on a probability. This is called Scheduled Sampling.

(I plan to add the option).

Can I use pretrained word embeddings (GloVe, CBOW, skipgram, etc.) instead of learning them from scratch?

Yes, you could, with the *load_pretrained_embeddings()* method in the *Decoder* class. You could also choose to fine-tune (or not) with the *fine_tune_embeddings()* method.

After creating the Decoder in train.py, you should provide the pretrained vectors to *load_pretrained_embeddings()* stacked in the same order as in the *word_map*. For words that you don't have pretrained vectors for, like <*start*>, you can initialize embeddings randomly like we did in *init_weights()*. I recommend fine-tuning to learn more meaningful vectors for these randomly initialized vectors.:

```
decoder = DecoderWithAttention(attention_dim=attention_dim,
                               embed_dim=emb_dim,
                               decoder_dim=decoder_dim,
                               vocab_size=len(word_map),
                               dropout=dropout)
decoder.load_pretrained_embeddings(pretrained_embeddings)  # pretrained_embeddings␣
↪should be of dimensions (len(word_map), emb_dim)
decoder.fine_tune_embeddings(True)  # or False
```

Also make sure to change the *emb_dim* parameter from its current value of *512* to the size of your pre-trained embeddings. This should automatically adjust the input size of the decoder LSTM to accomodate them.

1.7 Troubleshooting and FAQ

1. If you get error like **Error: spawn wmic.exe ENOENT** while running collect command (using apify) in alternat on **Microsoft Windows** This indicates that the wmic utility's directory is not found on your PATH. Open the advanced System Properties window (you can open the System page with Windows+Pause/Break) and on the Advanced tab, click Environment Variables. In the section for system variables, find PATH (or any capitalization thereof). Add this entry to it:

```
%SystemRoot%\System32\Wbem
```

Note that entries are delimited by semicolons.

2. In some cases with running collect command on windows you might get error: Chrome is downloaded but fails to launch on Node.js 14 If you get an error that looks like this when trying to launch Chromium:

(node:15505) UnhandledPromiseRejectionWarning: Error: Failed to launch the browser process! spawn /Users/. . . /node_modules/puppeteer/.local-chromium/mac-756035/chrome-mac/Chromium.app/Contents/MacOS/Chromium ENOENT This means that the browser was downloaded but failed to be extracted correctly. The most common cause is a bug in Node.js v14.0.0 which broke extract-zip, the module Puppeteer uses to extract browser downloads into the right place. The bug was fixed in Node.js v14.1.0, so please make sure you're running that version or higher. Alternatively, if you cannot upgrade, you could downgrade to Node.js v12, but we recommend upgrading when possible.

1.8 Alternat Reference

The alternat reference guide will walk through the implementation for both Generation and Collection, the alternat library, and the app.

1.8.1 Generation - Analyzer

Google Analyzer

class alternat.generation.google.analyze.**AnalyzeImage**
Bases: alternat.generation.base.analyzer.AnalyzeImageBase

Google Analyzer driver class.

Parameters **AnalyzeImageBase** (*[type]*) – Driver base class.

describe_image(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/l packages/PIL/Image.py'>*)
Describe image (used for captioning) - Not availble in Google Computer Vision API

Parameters **image** (*PIL_IMAGE*) – [description]

extract_labels(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/l packages/PIL/Image.py'>*)
Extract labels of image using Google Computer Vision API.

Parameters **image** (*PIL_IMAGE*) – PIL Image object.

Raises **Exception** – Google Cloud specific error messages based on request.

handle(*image_path: str = None*, *base64_image: str = None*, *actions: list = None*) → dict
Entry point for the driver. Implements all the action and generates data for rule engine.

Parameters

- **image_path** (*str, optional*) – Path to image on disk, defaults to None
- **base64_image** (*str, optional*) – Base64 image string, defaults to None
- **actions** (*list, optional*) – list of actions to run, defaults to None (all actions execute)

Returns [description]

Return type dict

ocr_analysis(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/lib/p packages/PIL/Image.py'>*)
Does OCR analysis using Google Computer Vision API. Also runs the alternat clustering rule if app is configured for it.

Parameters **image** (*PIL_IMAGE*) – PIL Image object.

set_environment_variables()
Sets environment variable GOOGLE_APPLICATION_CREDENTIALS based on config.

Azure Analyzer

class alternat.generation.microsoft.analyze.**AnalyzeImage**
Bases: alternat.generation.base.analyzer.AnalyzeImageBase

Azure / Microsoft Analyzer driver class.

Parameters **AnalyzeImageBase** (*[type]*) – Driver base class.

describe_image(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/l packages/PIL/Image.py'>*)
Describe image using Azure Vision API.

Parameters **image** (*PIL_Image*) – PIL Image object

extract_labels(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/l packages/PIL/Image.py'>*)
Extract labels of image using Azure Vision API.

Parameters **image** (*PIL_Image*) – PIL Image object.

handle(*image_path: str = None*, *base64_image: str = None*, *actions: list = None*) → dict
Entry point for the driver. Implements all the action and generates data for rule engine.

Parameters

- **image_path** (*str, optional*) – Path to image on disk, defaults to None
- **base64_image** (*str, optional*) – Base64 image string, defaults to None
- **actions** (*list, optional*) – list of actions to run, defaults to None (all actions execute)

Returns [description]

Return type dict

is_clean(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/lib/python packages/PIL/Image.py'>*) → bool
Check if the image has proper resolution, and is clean.

:param image:PIL Image object. :type image: PIL_Image :return: [description] :rtype: bool

modifyBoundingBoxData(*bounding_box: list*)
Transform bounding box data as per the convention. Azure API return bounding box info in the format [left, top, right, top, right, bottom, left, bottom] which is transformed to format [{x: left, y: top}, {x: right, y: top}, {x: right, y: bottom}, {x: left, y: bototm}].

Parameters **bounding_box** (*list*) – Bounding box data form Azure API.

Returns [description]

Return type [type]

ocr_analysis(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/lib/p packages/PIL/Image.py'>*)
Does OCR Analysis using Azure Vision API.

Parameters **image** (*PIL_Image*) – PIL Image object.

resize_image(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/lib/p packages/PIL/Image.py'>*)
Resize image (maintaining aspect ratio) if width / height > 5000 pixels (API constrain from Azure)

Parameters **image** (*PIL_Image*) – [description]

Opensource Analyzer

class alternat.generation.opensource.analyze.**AnalyzeImage**

Bases: alternat.generation.base.analyzer.AnalyzeImageBase

Opensource driver class.

Parameters **AnalyzeImageBase** (*[type]*) – Driver base class.

describe_image(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/l packages/PIL/Image.py'>*)

Describe image using open source solution. Not implemented right now.

Parameters **image** (*PIL_Image*) – PIL Image object

extract_labels(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/l packages/PIL/Image.py'>*)

Extract labels of image using open source solution. Not implemented righ now.

Parameters **image** (*PIL_Image*) – PIL Image object.

handle(*image_path: str = None*, *base64_image: str = None*, *actions: list = None*) → dict

Entry point for the driver. Implements all the action and generates data for rule engine.

Parameters

- **image_path** (*str, optional*) – Path to image on disk, defaults to None
- **base64_image** (*str, optional*) – Base64 image string, defaults to None
- **actions** (*list, optional*) – list of actions to run, defaults to None (all actions execute)

Returns [description]

Return type dict

modifyBoundingBoxData(*bounding_box: list*)

Transform bounding box data as per the convention. EasyOCR return bounding box info in the format [left, top, right, top, right, bottom, left, bottom] which is transformed to format [{x: left, y: top}, {x: right, y: top}, {x: right, y: bottom}, {x: left, y: bototm}].

Parameters **bounding_box** (*list*) – Bounding box data form EasyOCR.

Returns [description]

Return type [type]

ocr_analysis(*image: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/alternat/envs/main/lib/p packages/PIL/Image.py'>*)

Does OCR Analysis using EasyOCR.

Parameters **image** (*PIL_Image*) – PIL Image object.

1.8.2 Generation - Rule Engine

OCR Rule

class alternat.generation.rules.ocr_handler.**OCRDataHandler**(*input_data*, *confidence_threshold: float = None*, *ocr_filter_threshold: float = None*)

Bases: alternat.generation.base.action_data_handler.ActionDataHandler

Rule for processing OCR data from driver.

Parameters **ActionDataHandler** (*[type]*) – Base class for rule.

Initialize the handler with input data and confidence threshold (if available)

Parameters

- **input_data** (*[type]*) – Data from driver.
- **confidence_threshold**(*float, optional*) – Confidence threshold to filter OCR with low threshold, defaults to None (Driver config default)
- **ocr_filter_threshold**(*float, optional*) – Confidence threshold to filter OCR data based on line height ratio to image height.

apply(*interim_result: dict*) → dict

Process intermin result from previous rules in the chain and run OCR rule.

Parameters **interim_result** (*dict*) – Intermediate results from previous rules in the chain.

Returns [description]

Return type dict

has_data() → bool

Checks whethere OCR data is avalible in the input data.

Returns [description]

Return type bool

process_ocr() → dict

Process the OCR data from the driver and filter it on the basis of line confidence threshold value and the ratio of line height to image height. Based on the configuration also invokes alternat clustering implementation (default to True)

Returns [description]

Return type dict

Caption Rule

class alternat.generation.rules.caption_handler.**CaptionDataHandler**(*input_data: dict*, *confidence_threshold: float = None*)

Bases: alternat.generation.base.action_data_handler.ActionDataHandler

Rule for processing caption data from driver.

Parameters **ActionDataHandler** (*[type]*) – Base class for rule.

Initialize the handler with input data and confidence threshold (if available)

Parameters

- **input_data** (*dict*) – Data from driver.
- **confidence_threshold** (*float, optional*) – Confidence threshold to filter captions with low threshold, defaults to None (Driver config default)

apply(*interim_result: dict*) → dict

Process intermin result from previous rules in the chain and run caption rule.

Parameters **interim_result** (*dict*) – Intermediate results from previous rules in the chain.

Returns [description]

Return type dict

has_data() → bool

Checks whethere caption data is avalible in the input data.

Returns [description]

Return type bool

Label Rule

class alternat.generation.rules.label_handler.**LabelDataHandler**(*input_data*, *confidence_threshold: float = None*)

Bases: alternat.generation.base.action_data_handler.ActionDataHandler

Rule for processing label data from driver.

Parameters **ActionDataHandler** (*[type]*) – Base class for rule.

Initialize the handler with input data and confidence threshold (if available)

Parameters

- **input_data** (*[type]*) – Data from driver.
- **confidence_threshold** (*float, optional*) – Confidence threshold to filter labels with low threshold, defaults to None (Driver config default)

apply(*interim_result: dict*) → dict

Process intermin result from previous rules in the chain and run label rule.

Parameters **interim_result** (*dict*) – Intermediate results from previous rules in the chain.

Returns [description]

Return type dict

has_data() → bool
Checks whethere label data is avalible in the input data.

Returns [description]

Return type bool

1.8.3 Library

class alternat.generation.generator.**Drivers**
Bases: object

Driver name for alternat Library.

GOOGLE = 'google'

MICROSOFT = 'azure'

OPEN = 'opensource'

class alternat.generation.generator.**Generator**(*driver_name: str = None*)
Bases: object

Generator class to implement alternat Library.

Raises

- **InvalidGeneratorDriver** – Driver Invalid
- **InvalidGeneratorDriver** – Driver Invalid
- **InvalidGeneratorDriver** – Driver Invalid
- **OutputDirPathNotGiven** – Output director path is not given.

Returns [description]

Return type [type]

Initializes generator with driver to use.

Parameters **driver_name** (*str, optional*) – Name of the driver], defaults to None (open-source)

Raises **InvalidGeneratorDriver** – Driver name is invalid or not implemented.

ALLOWED_DRIVERS = ['opensource', 'azure', 'google']

DEFAULT_DRIVER = 'opensource'

generate_alt_text_from_base64(*base64_image: str*)
Generates alt-text from base64 image string.

Parameters **base64_image** (*str*) – base64 image string

Returns [description]

Return type [type]

generate_alt_text_from_file(*input_image_path: str*, *output_dir_path: str*)
Generates alt-text from file on disk.

Parameters

- **input_image_path** (*str*) – Path to image to be processed.
- **output_dir_path** (*str*) – Path to directory where the results needs to be saved.

Returns [description]

Return type [type]

get_config()
Get the generator level config in the form of JSON.

Returns [description]

Return type [type]

get_current_driver()
Get the current driver.

Returns [description]

Return type [type]

get_driver_config()
Get the driver config in the form of JSON. Retreives public members [name: value] pair from the driver config class.

Returns [description]

Return type [type]

set_config(*conf*)
Sets the generator level configuration parameters passed via JSON.

Parameters **conf** (*[type]*) – Generator configuration parameters with values.

set_driver_config(*conf: dict*)
Sets the driver config parameters using the JSON passed. There is one-to-one mapping between key in json and driver class public members.

Parameters **conf** (*dict*) – Configuration JSON to set the driver configuration.

class alternat.collection.collector.**Collector**
Bases: object

process(*url: str*, *output_dir_path: str*, *download_recursive: bool = False*, *collect_using_apify: bool = False*)
Collects image from the url into the output directory

Parameters

- **url** (*str*) – [description]
- **output_dir_path** (*str*) – [description]
- **download_recursive** (*bool, optional*) – [description], defaults to False
- **collect_using_apify** (*bool, optional*) – [description], defaults to False