OpenVisionCapsules

Aotu

Mar 18, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is a Capsule?

A capsule is a single file with a `.cap` file extension. It contains the code, metadata, model files, and any other files the capsule needs to operate.

Capsules take a frame and information from other capsules as input, run some kind of analysis, and provide metadata about the frame as output. For example, a person detection capsule would take a frame as input and output person detections in that frame. A gender classifier capsule would take this frame and each person detection as input, and output a male or female classification for that detection.

Capsules provide metadata describing these inputs and outputs alongside other information on the capsule. Applications that are compatible OpenVisionCapsules use this metadata to know when to run the capsule and with what input.

CHAPTER

TWO

FILE STRUCTURE

All capsules start off their life as unpackaged capsules. An unpackaged capsule is simply a directory containing all files that will be packaged into the capsule. This directory must contain, at minimum, a meta.conf file and a capsule.py file.

```
detector_person
├── meta.conf
└── capsule.py
```

The meta.conf file is a simple configuration file which specifies the major and minor version of OpenVisionCapsules that this capsule requires. Applications use this information to decide if your capsule is compatible with the version of OpenVisionCapsules the application uses. A capsule with a compatibility version of 0.1 are expected to be compatible with applications that use OpenVisionCapsules version 0.1 through 0.x, but not 1.x or 2.x.

```
[about]
api_compatibility_version = 0.1
```

The capsule.py file is the meat of the capsule. It contains the actual behavior of the capsule. We will talk more about the contents of this file in a later section.

If your capsule uses other files for its operation, like a model file, it should be included in this directory as well. All files in the capsule's directory will be included and made accessible once it's packaged.

```
person_detection_capsule
├── meta.conf
├── capsule.py
└── frozen_inference_graph.pb
```

CHAPTER

THREE

RUNTIME ENVIRONMENT

3.1 Loading

When a capsule is loaded, the `capsule.py` file is imported as a module and an instance of the `Capsule` class defined in that module is instantiated. Then, for each compatible device, an instance of the capsule's `Backend` class is created using the provided `backend_loader` function.

3.2 Importing

Capsules have access to a number of helpful libraries, including:

- The entirety of the Python standard library
- Numpy (`import numpy`)
- OpenCV (`import cv2`)
- Tensorflow (`import tensorflow`)
- Scikit Learn (`import sklearn`)
- OpenVino (`import openvino`)

Applications may provide more libraries in addition to these. Please see that application's documentation for more information.

3.2.1 Importing From Other Files in the Capsule

In order to allow for more complex capsules that have code reuse within them, capsules may consist of multiple Python files. These files are made available through relative imports.

For example, with the following directory structure:

```
capsule_dir/
├── capsule.py
├── backend.py
└── utils/
    ├── img_utils.py
    └── ml_utils.py
```

The `capsule.py` file may import the other Python files like so:

```
from . import backend
from .utils import img_utils, ml_utils
```

Note that non-relative imports to these files will *not* work:

```
import backend
from utils import img_utils, ml_utils
```

3.2.2 Limiting GPU memory Growth

By default, OpenVisionCapsules maps all available memory of all visible CUDA configured GPUs. To prevent this, use the following Environment flag while using Tensorflow.

```
TF_FORCE_GPU_ALLOW_GROWTH=True
```

- This Environment variable is only applicable to Tensorflow.

For proper reference, visit Tensorflow: https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth

CHAPTER

FOUR

THE CAPSULE CLASS

4.1 Introduction

The Capsule class provides information to the application about what a capsule is and how it should be run. Every capsule defines a Capsule class that extends BaseCapsule in its `capsule.py` file.

```
from vcap import (
    BaseCapsule,
    NodeDescription,
    options
)

class Capsule(BaseCapsule):
    name = "detector_person"
    version = 1
    stream_state = StreamState
    input_type = NodeDescription(size=NodeDescription.Size.NONE),
    output_type = NodeDescription(
        size=NodeDescription.Size.ALL,
        detections=["person"])
    backend_loader = backend_loader_func
    options = {
        "threshold": options.FloatOption(
            default=0.5,
            min_val=0.1,
            max_val=1.0)
    }
```

class BaseCapsule(*capsule_files: Dict[str, bytes]*, *inference_mode=True*)

An abstract base class that all capsules must subclass. Defines the interface that capsules are expected to implement.

A class that subclasses from this class is expected to be defined in a capsule.py file in a capsule.

Parameters

- **capsule_files** – A dict of {"file_name": FILE_BYTES} of the files that were found and loaded in the capsule
- **inference_mode** – If True, the model will be loaded and the backends will start for it. If False, the capsule will never be able to run inference, but it will still have it's various readable attributes.

abstract static backend_loader(*capsule_files: Dict[str, bytes]*, *device: str*) → vcap.backend.BaseBackend

A function that creates a backend for this capsule.

Parameters

- **capsule_files** – Provides access to all files in the capsule. The keys are file names and the values are `bytes`.
- **device** – A string specifying the device that this backend should use. For example, the first GPU device is specified as "GPU:0".

property description
A human-readable description of what the capsule does.

property device_mapper
A device mapper contains a single field, `filter_func`, which is a function that takes in a list of all available device strings and returns a list of device strings that are compatible with this capsule.

property input_type
Describes the types of DetectionNodes that this capsule takes in as input.

property name
The name of the capsule. This value uniquely defines the capsule and cannot be shared by other capsules.

By convention, the capsule's name is prefixed by some short description of the role it plays ("detector", "recognizer", etc) followed by the kind of data it relates to ("person", "face", etc) and, if necessary, some differentiating factor ("fast", "close_up", "retail", etc). For example, a face detector that is optimized for quick inference would be named "detector_face_fast".

property options
A list of zero or more options that can be configured at runtime.

property output_type
Describes the types of DetectionNodes that this capsule produces as output.

property stream_state
(Optional) An instance of this object will be created for every new video stream that a capsule is run on, and de-initialized when that stream is deleted. It is intended to be overridden by capsules that have stateful operations across a single stream.

property version
The version of the capsule.

When should you bump the version of a capsule? When:

- You've changed the usage of existing capsule options
- You've changed the model or algorithm
- You've changed the input/output node descriptions

When shouldn't you bump the version of a capsule? When:

- You only did code restructuring
- You've updated the capsule to work with a newer API version
- You've added (but not removed or changed previous) capsule options

In summary, the version is most useful for differentiating a capsule from its previous versions.

CHAPTER

FIVE

BACKENDS

5.1 Introduction

A backend is what provides the low-level analysis on a video frame. For machine learning, this is the place where the frame would be fed into the model and the results would be returned. Every capsule must define a backend class that subclasses the BaseBackend class.

The application will create an instance of the backend class for each device string returned by the capsule's device mapper.

5.2 Required Methods

All backends must subclass the BaseBackend abstract base class, meaning that there are a couple methods that the backend must implement.

class BaseBackend

An object that provides low-level prediction functionality for batches of frames.

close() → None

De-initializes the backend. This is called when the capsule is being unloaded. This method should be overridden by any Backend that needs to release resources or close other threads.

The backend will stop receiving frames before this method is called, and will not receive frames again.

abstract process_frame(*frame: numpy.ndarray*, *detection_node: Union[None, vcap.detection_node.DetectionNode, List[vcap.detection_node.DetectionNode]]*, *options: Dict[str, Union[int, float, bool, str]]*, *state: vcap.stream_state.BaseStreamState*) → Union[None, vcap.detection_node.DetectionNode, List[vcap.detection_node.DetectionNode]]

A method that does the pre-processing, inference, and postprocessing work for a frame.

If the capsule uses an algorithm that benefits from batching, this method may call `self.send_to_batch`, which will asynchronously send work out for batching. Doing so requires that the `batch_predict` method is overridden.

Parameters

- **frame** – A numpy array representing a frame. It is of shape (height, width, num_channels) and the frames come in BGR order.
- **detection_node** – The detection_node type as specified by the `input_type`

- **options** – A dictionary of key (string) value pairs. The key is the name of a capsule option, and the value is its configured value at the time of processing. Capsule options are specified using the `options` field in the Capsule class.
- **state** – This will be a StreamState object of the type specified by the `stream_state` attribute on the Capsule class. If no StreamState object was specified, a simple BaseStreamState object will be passed in. The StreamState will be the same object for all frames in the same video stream.

5.3 Batching Methods

Batching refers to the process of collecting more than one video frame into a "batch" and sending them all out for processing at once. Certain algorithms see performance improvements when batching is used, because doing so decreases the amount of round-trips the video frames take between devices.

If you wish to use batching in your capsule, you may call the `send_to_batch` method in `process_frame` instead of doing analysis in that method directly. The `send_to_batch` method sends the input to a `BatchExecutor` which collects inference requests for this capsule from different streams. Then, the `BatchExecutor` routinely calls your backend's `batch_predict` method with a list of the collected inputs. As a result, users of `send_to_batch` must override the `batch_predict` method in addition to the other required methods.

The `send_to_batch` method is asynchronous. Instead of immediately returning analysis results, it returns a `concurrent.futures.Future` where the result will be provided. Simple batching capsules may call `send_to_batch`, then immediately call `result` to block for the result.

```
result = self.send_to_batch(frame).result()
```

An argument of any type may be provided to `send_to_batch`, as the argument will be passed in a list to `batch_predict` without modification. In many cases only the video frame needs to be provided, but additional metadata may be included as necessary to fit your algorithm's needs.

class BaseBackend

An object that provides low-level prediction functionality for batches of frames.

batch_predict(*input_data_list: List[Any]*) → List[Any]

This method takes in a batch as input and provides a list of result objects of any type as output. What the result objects are will depend on the algorithm being defined, but the number of prediction objects returned _must_ match the number of video frames provided as input.

Parameters **input_data_list** – A list of objects. Whatever the model requires for each frame.

CHAPTER

SIX

INPUTS AND OUTPUTS

6.1 Introduction

Capsules are defined by the data they take as input and the information they give as output. Applications use this information to connect capsules to each other and schedule their execution. These inputs and outputs are *defined* by NodeDescription objects and *realized* by DetectionNode objects.

class DetectionNode(*, *name: str*, *coords: List[List[Union[int, float]]]*, *attributes: Dict[str, str] = None*, *children: List[DetectionNode] = None*, *encoding: Optional[numpy.ndarray] = None*, *track_id: Optional[uuid.UUID] = None*, *extra_data: Dict[str, object] = None*)

Capsules use DetectionNode objects to communicate results to other capsules and the application itself. A DetectionNode contains information on a detection in the current frame. Capsules that detect objects in a frame create new DetectionNodes. Capsules that discover attributes about detections add data to existing DetectionNodes.

Parameters

- **name** – The detection class name. This describes what the detection is. A detection of a person would have a name="person".
- **coords** – A list of coordinates defining the detection as a polygon in-frame. Comes in the format `[[x,y], [x,y]...]`.
- **attributes** – A key-value store where the key is the type of attribute being described and the value is the attribute's value. For instance, a capsule that detects gender might add a "gender" key to this dict, with a value of either "masculine" or "feminine".
- **children** – Child DetectionNodes that are a "part" of the parent, for instance, a head DetectionNode might be a child of a person DetectionNode
- **encoding** – An array of float values that represent an encoding of the detection. This can be used to recognize specific instances of a class. For instance, given a picture of person's face, the encoding of that face and the encodings of future faces can be compared to find that person in the future.
- **track_id** – If this object is tracked, this is the unique identifier for this detection node that ties it to other detection nodes in future and past frames within the same stream.
- **extra_data** – A dict of miscellaneous data. This data is provided directly to clients without modification, so it's a good way to pass extra information from a capsule to other applications.

class NodeDescription(*, *size: vcap.node_description.NodeDescription.Size*, *detections: List[str] = None*, *attributes: Dict[str, List[str]] = None*, *encoded: bool = False*, *tracked: bool = False*, *extra_data: List[str] = None*)

Capsules use NodeDescriptions to describe the kinds of DetectionNodes they take in as input and produce as output.

A capsule may take a DetectionNode as input and produce zero or more DetectionNodes as output. Capsules define what information inputted DetectionNodes must have and what information outputted detection nodes will have using NodeDescriptions.

For example, a capsule that encodes people and face detections would use NodeDescriptions to define its inputs and outputs like so:

```
>>> input_type = NodeDescription(
...     detections=["person", "face"])
>>> output_type = NodeDescription(
...     detections=["person", "face"],
...     encoded=True)
```

A capsule that uses a car's encoding to classify the color of a car would look like this.

```
>>> input_type = NodeDescription(
...     detections=["car"],
...     encoded=True)
>>> output_type = NodeDescription(
...     detections=["car"],
...     attributes={"color": ["blue", "yellow", "green"]},
...     encoded=True)
```

A capsule that detects dogs and takes no existing input would look like this.

```
>>> input_type = NodeDescription(size=NodeDescription.Size.NONE)
>>> output_type = NodeDescription(
>>>                 size=NodeDescription.Size.ALL,
>>>                 detections=["dog"])
```

Parameters

- **size** – The number of DetectionNodes that this capsule either takes as input or provides as output
- **detections** – A list of acceptable detection class names. A node that meets this description must have a class name that is present in this list
- **attributes** – A dict whose key is the classification type and whose value is a list of possible attributes. A node that meets this description must have a classification for each classification type.
- **encoded** – If true, the DetectionNode must be encoded to meet this description
- **tracked** – If true, the DetectionNode is being tracked
- **extra_data** – A list of keys in a NodeDescription's extra_data. A DetectionNode that meets this description must have extra data for each name listed here.

6.2 Examples

6.2.1 detections

A capsule that can encode cars or trucks would use a NodeDescription like this as its `input_type`:

```
NodeDescription(detections=["car", "truck"])
```

A capsule that can detect people and dogs would use a NodeDescription like this as its `output_type`:

```
NodeDescription(detections=["person", "dog"])
```

6.2.2 attributes

A capsule that operates on detections that have been classified for gender use a NodeDescription like this as its `input_type`:

```
NodeDescription(
    attributes={
        "gender": ["male", "female"],
        "color": ["red", "blue", "green"]
    })
```

A capsule that can classify people's gender as either male or female would have the following NodeDescription as its `output_type`:

```
NodeDescription(
    detections=["person"],
    attributes={
        "gender": ["male", "female"]
    })
```

6.2.3 encoded

A capsule that operates on detections of cars that have been encoded use a NodeDescription like this as its `input_type`:

```
NodeDescription(
    detections=["car"],
    encoded=True)
```

A capsule that encodes people would use a NodeDescription like this as its `output_type`:

```
NodeDescription(
    detections=["person"],
    encoded=True)
```

6.2.4 tracked

A capsule that operates on person detections that have been tracked would use a NodeDescription like this as its `input_type`.

```
NodeDescription(
    detections=["person"],
    tracked=True)
```

A capsule that tracks people would use a NodeDescription like this as its `output_type`:

```
NodeDescription(
    detections=["person"],
    tracked=True)
```

6.2.5 extra_data

A capsule that operates on people detections with a "process_extra_fast" `extra_data` field would use a NodeDescription like this as its `input_type`:

```
NodeDescription(
    detections=["person"],
    extra_data=["process_extra_fast"])
```

A capsule that adds an "is_special" `extra_data` field to its person-detected output would use a NodeDescription like this as its `output_type`:

```
NodeDescription(
    detections=["person"],
    extra_data=["is_special"])
```

CHAPTER

SEVEN

OPTIONS

7.1 Introduction

Capsules can provide runtime configuration options that change the way the capsule operates. These options will appear on the client and can also be changed in the UI. Options have a type and constraints that define what values are valid.

class FloatOption(*, *default: float*, *min_val: Optional[float]*, *max_val: Optional[float]*, *description: Optional[str] = None*)

A capsule option that holds a floating point value with defined boundaries.

Parameters

- **default** – The default value of this option
- **min_val** – The minimum allowed value for this option, inclusive, or None for no lower limit
- **max_val** – The maximum allowed value for this option, inclusive, or None for no upper limit
- **description** – The description for this option

class IntOption(*, *default: int*, *min_val: Optional[int]*, *max_val: Optional[int]*, *description: Optional[str] = None*)

A capsule option that holds an integer value.

Parameters

- **default** – The default value of this option
- **min_val** – The minimum allowed value for this option, inclusive, or None for no lower limit
- **max_val** – The maximum allowed value for this option, inclusive, or None for no upper limit
- **description** – The description for this option

class EnumOption(*, *default: str*, *choices: List[str]*, *description: Optional[str] = None*)

A capsule option that holds a choice from a discrete set of string values.

Parameters

- **default** – The default value of this option
- **choices** – A list of all valid values for this option
- **description** – The description for this option

class BoolOption(*, *default: bool*, *description: Optional[str] = None*)

A capsule option that holds an boolean value.

Parameters

- **default** – The default value of this option
- **description** – The description for this option

CHAPTER

EIGHT

CREATING A CAPSULE

For application developers, OpenVisionCapsules provides a function to package up unpackaged capsules. The optional `key` field encrypts the capsule with AES.

```
from vcap import package_capsule

package_capsule(Path("detector_person"),
                Path("capsules", "detector_person.cap"),
                key="[AES Key]")
```

For capsule developers for an application, it is the job of the application to provide a way to package capsules. Please see the documentation for the application you are using for more information.

8.1 Creating an Object Detector Capsule with Supervisely

If you've trained tensorflow-object-detection-API object detector using Supervisely, you can follow the following steps to deploy your model as a capsule:

8.1.1 Set up the TF Object Detection API

First, set up the Tensorflow Object Detection API on your machine by cloning the `tensorflow/models` repository and following the object detection API installation instructions. Make sure the tests pass before continuing- otherwise, you might have forgotten to set up certain environment variables!

8.1.2 Freeze your trained Supervisely model

Next, you must download your trained Supervisely model and extract it. Inside you should see the following directory structure:

```
<DOWNLOADED MODEL DIR>
├── config.json
├── model.config
└── model_weights
    ├── checkpoint
    ├── model.ckpt.data-00000-of-00001
    ├── model.ckpt.index
    └── model.ckpt.meta
```

Now, you must simply freeze the model to get the `frozen_inference_graph.pb`. To do that, run `models/research/object_detection/export_inference_graph.py` script inside of your downloaded model directory.

```
python PATH/TO/export_inference_graph.py \
    --input_type image_tensor \
    --pipeline_config_path model.config \
    --trained_checkpoint_prefix model_weights/model.ckpt \
    --output_directory .
```

There should now be a frozen_inference_graph.pb in the current directory. This is the model file that has been optimized for inference, and is much more portable for production use.

CHAPTER

NINE

STREAM STATE

It is sometimes desirable to carry state throughout the lifetime of an entire video stream, rather than on a frame-by-frame basis. This is where StreamState comes in.

If the `stream_state` field of the capsule's Capsule class is set, the `process_frame` method for your capsule's backend will be passed an instance of the provided class. Any state that should exist for the duration of the videostream may be saved here.

This is commonly used by capsules that track objects between video frames. Information on previous detections can be stored in the StreamState object and read when new detections are found. This can also be useful for result smoothing, for caching frames (think RNN models), and many other use cases.

A capsule's StreamState class does not need to implement any methods and has no functional purpose outside of the capsule.

This is a guide on how to encapsulate an algorithm using OpenVisionCapsules. Capsules are discrete components that define new ways to analyze video streams.

This guide discusses the Open Vision Capsule system generally, not any specific information on how to write a Capsule of a certain type or with certain technology. Example capsules are available under `vcap/examples` that show how to encapsulate models from various popular machine learning frameworks.

INDEX