A. Pretrained Models

We specify details about all the pretrained models used, as well as the code-generation large language model:

- **GLIP** [32]. We use the implementation from the official GitHub repository[3]. In our experiments we use the GLIP-L (large) version. In order to adapt to new versions of PyTorch, we had to modify the CUDA implementation of some functions, as the repository relies on old versions of PyTorch. We provide our updated version of GLIP in our code.
- **MiDaS** [45]. We use the implementation from PyTorch hub[4], and use the "DPT_Large" version.
- **BLIP-2** [31]. We tried both the implementation from the official repository[5] and the Huggingface one[6], with little difference between the two, being the former slightly more performant and the latter faster. In both cases, we used the Flan-T5 XXL version.
- **X-VLM** [67]. We used the official implementation[7], specifically the version finetuned for retrieval on MSCOCO.
- **GPT-3 for `llm_query`**. The GPT-3 model we use for the LLM query function is the `text-davinci-003` one. We use the official OpenAI Python API[8].
- **Codex**. The GPT-3 model we use for code generation is the `code-davinci-002` one.

See the code for more detailed implementation details.

B. API

We provide the full API next, in Listing 1:

```python
class ImagePatch:
    """A Python class containing a crop of an image centered around a particular object, as well as relevant information.
    Attributes
    ----------
    cropped_image : array_like
        An array-like of the cropped image taken from the original image.
    left : int
        An int describing the position of the left border of the crop's bounding box in the original image.
    lower : int
        An int describing the position of the bottom border of the crop's bounding box in the original image.
    right : int
        An int describing the position of the right border of the crop's bounding box in the original image.
    upper : int
        An int describing the position of the top border of the crop's bounding box in the original image.

    Methods
    -------
    find(object_name: str)->List[ImagePatch]
        Returns a list of new ImagePatch objects containing crops of the image centered around any objects found in the
        image matching the object_name.
    exists(object_name: str)->bool
        Returns True if the object specified by object_name is found in the image, and False otherwise.
    verify_property(property: str)->bool
        Returns True if the property is met, and False otherwise.
    best_text_match(option_list: List[str], prefix: str)->str
        Returns the string that best matches the image.
    simple_query(question: str=None)->str
        Returns the answer to a basic question asked about the image. If no question is provided, returns the answer
        to "What is this?".
    compute_depth()->float
        Returns the median depth of the image crop.
    crop(left: int, lower: int, right: int, upper: int)->ImagePatch
        Returns a new ImagePatch object containing a crop of the image at the given coordinates.
    """

    def __init__(self, image, left: int=None, lower: int=None, right: int=None, upper: int=None):
        """Initializes an ImagePatch object by cropping the image at the given coordinates and stores the coordinates as attributes.
```

[3]https://github.com/microsoft/GLIP
[4]https://pytorch.org/hub/intelisl_midas_v2/
[5]https://github.com/salesforce/LAVIS/tree/main/projects/blip2
[6]https://huggingface.co/Salesforce/blip2-flan-t5-xxl
[7]https://github.com/zengyan-97/X-VLM
[8]https://openai.com/blog/openai-api

```python
        If no coordinates are provided, the image is left unmodified, and the coordinates are set to the dimensions of the image.
        Parameters
        -------
        image : array_like
            An array-like of the original image.
        left : int
            An int describing the position of the left border of the crop's bounding box in the original image.
        lower : int
            An int describing the position of the bottom border of the crop's bounding box in the original image.
        right : int
            An int describing the position of the right border of the crop's bounding box in the original image.
        upper : int
            An int describing the position of the top border of the crop's bounding box in the original image.

        """
        if left is None and right is None and upper is None and lower is None:
            self.cropped_image = image
            self.left = 0
            self.lower = 0
            self.right = image.shape[2]  # width
            self.upper = image.shape[1]  # height
        else:
            self.cropped_image = image[:, lower:upper, left:right]
            self.left = left
            self.upper = upper
            self.right = right
            self.lower = lower

        self.width = self.cropped_image.shape[2]
        self.height = self.cropped_image.shape[1]

        self.horizontal_center = (self.left + self.right) / 2
        self.vertical_center = (self.lower + self.upper) / 2

    def find(self, object_name: str) -> List[ImagePatch]:
        """Returns a list of ImagePatch objects matching object_name contained in the crop if any are found.
        Otherwise, returns an empty list.
        Parameters
        ----------
        object_name : str
            the name of the object to be found

        Returns
        -------
        List[ImagePatch]
            a list of ImagePatch objects matching object_name contained in the crop

        Examples
        --------
        >>> # return the children
        >>> def execute_command(image) -> List[ImagePatch]:
        >>>     image_patch = ImagePatch(image)
        >>>     children = image_patch.find("child")
        >>>     return children
        """

    def exists(self, object_name: str) -> bool:
        """Returns True if the object specified by object_name is found in the image, and False otherwise.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.

        Examples
        -------
        >>> # Are there both cakes and gummy bears in the photo?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     is_cake = image_patch.exists("cake")
        >>>     is_gummy_bear = image_patch.exists("gummy bear")
        >>>     return bool_to_yesno(is_cake and is_gummy_bear)
        """
        return len(self.find(object_name)) > 0

    def verify_property(self, object_name: str, property: str) -> bool:
        """Returns True if the object possesses the property, and False otherwise.
        Differs from 'exists' in that it presupposes the existence of the object specified by object_name, instead checking whether the object possesses the property.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        property : str
            A string describing the property to be checked.

        Examples
        -------
        >>> # Do the letters have blue color?
        >>> def execute_command(image) -> str:
        >>>     image_patch = ImagePatch(image)
        >>>     letters_patches = image_patch.find("letters")
        >>>     # Question assumes only one letter patch
        >>>     if len(letters_patches) == 0:
        >>>         # If no letters are found, query the image directly
        >>>         return image_patch.simple_query("Do the letters have blue color?")
        >>>     return bool_to_yesno(letters_patches[0].verify_property("letters", "blue"))
        """
        return verify_property(self.cropped_image, object_name, property)

    def best_text_match(self, option_list: List[str]) -> str:
        """Returns the string that best matches the image.
        Parameters
        -------
        option_list : str
            A list with the names of the different options
        prefix : str
            A string with the prefixes to append to the options

        Examples
        -------
        >>> # Is the cap gold or white?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     cap_patches = image_patch.find("cap")
        >>>     # Question assumes one cap patch
        >>>     if len(cap_patches) == 0:
        >>>         # If no cap is found, query the image directly
        >>>         return image_patch.simple_query("Is the cap gold or white?")
        >>>     return cap_patches[0].best_text_match(["gold", "white"])
        """
        return best_text_match(self.cropped_image, option_list)

    def simple_query(self, question: str = None) -> str:
        """Returns the answer to a basic question asked about the image. If no question is provided, returns the answer to "What is this?".
        Parameters
        -------
        question : str
            A string describing the question to be asked.

        Examples
        -------

        >>> # Which kind of animal is not eating?
        >>> def execute_command(image) -> str:
        >>>     image_patch = ImagePatch(image)
        >>>     animal_patches = image_patch.find("animal")
        >>>     for animal_patch in animal_patches:
        >>>         if not animal_patch.verify_property("animal", "eating"):
        >>>             return animal_patch.simple_query("What kind of animal is eating?") # crop would include eating so keep it in the query
        >>>     # If no animal is not eating, query the image directly
        >>>     return image_patch.simple_query("Which kind of animal is not eating?")

        >>> # What is in front of the horse?
        >>> # contains a relation (around, next to, on, near, on top of, in front of, behind, etc), so ask directly
        >>> return image_patch.simple_query("What is in front of the horse?")
        >>>
        """
        return simple_qa(self.cropped_image, question)

    def compute_depth(self):
        """Returns the median depth of the image crop
        Parameters
        ----------
        Returns
        -------
        float
            the median depth of the image crop

        Examples
        --------
        >>> # the person furthest away
        >>> def execute_command(image)->ImagePatch:
        >>>     image_patch = ImagePatch(image)
        >>>     person_patches = image_patch.find("person")
        >>>     person_patches.sort(key=lambda person: person.compute_depth())
        >>>     return person_patches[-1]
        """
        depth_map = compute_depth(self.cropped_image)
        return depth_map.median()

    def crop(self, left: int, lower: int, right: int, upper: int) -> ImagePatch:
        """Returns a new ImagePatch cropped from the current ImagePatch.
        Parameters
        -------
        left : int
            The leftmost pixel of the cropped image.
        lower : int
            The lowest pixel of the cropped image.
        right : int
            The rightmost pixel of the cropped image.
        upper : int
            The uppermost pixel of the cropped image.
        -------
        """
        return ImagePatch(self.cropped_image, left, lower, right, upper)

    def overlaps_with(self, left, lower, right, upper):
        """Returns True if a crop with the given coordinates overlaps with this one,
        else False.
        Parameters
        ----------
        left : int
            the left border of the crop to be checked
        lower : int
            the lower border of the crop to be checked
        right : int
            the right border of the crop to be checked
        upper : int
            the upper border of the crop to be checked

        Returns
        -------
        bool
            True if a crop with the given coordinates overlaps with this one, else False

        Examples
        --------
        >>> # black cup on top of the table
        >>> def execute_command(image) -> ImagePatch:
        >>>     image_patch = ImagePatch(image)
        >>>     table_patches = image_patch.find("table")
        >>>     if len(table_patches) == 0:
        >>>         table_patches = [image_patch]  # If no table found, assume the whole image is a table
        >>>     table_patch = table_patches[0]
        >>>     cup_patches = image_patch.find("black cup")
        >>>     for cup in cup_patches:
        >>>         if cup.vertical_center > table_patch.vertical_center
        >>>             return cup
        >>>     return cup_patches[0]  # If no cup found on top of the table, return the first cup found
        """
        return self.left <= right and self.right >= left and self.lower <= upper and self.upper >= lower


def best_image_match(list_patches: List[ImagePatch], content: List[str], return_index=False) -> Union[ImagePatch, int]:
    """Returns the patch most likely to contain the content.
    Parameters
    ----------
    list_patches : List[ImagePatch]
    content : List[str]
        the object of interest
    return_index : bool
        if True, returns the index of the patch most likely to contain the object

    Returns
    -------
    int
        Patch most likely to contain the object
```

```python

    Examples
    --------
    >>> # Return the man with the hat
    >>> def execute_command(image):
    >>>     image_patch = ImagePatch(image)
    >>>     man_patches = image_patch.find("man")
    >>>     if len(man_patches) == 0:
    >>>         return image_patch
    >>>     hat_man = best_image_match(list_patches=man_patches, content=["hat"])
    >>>     return hat_man

    >>> # Return the woman with the pink scarf and blue pants
    >>> def execute_command(image):
    >>>     image_patch = ImagePatch(image)
    >>>     woman_patches = image_patch.find("woman")
    >>>     if len(woman_patches) == 0:
    >>>         return image_patch
    >>>     woman_most = best_image_match(list_patches=woman_patches, content=["pink scarf", "blue pants"])
    >>>     return woman_most
    """
    return best_image_match(list_patches, content, return_index)


def distance(patch_a: ImagePatch, patch_b: ImagePatch) -> float:
    """
    Returns the distance between the edges of two ImagePatches. If the patches overlap, it returns a negative distance
    corresponding to the negative intersection over union.
    """
    return distance(patch_a, patch_b)


def bool_to_yesno(bool_answer: bool) -> str:
    return "yes" if bool_answer else "no"


def llm_query(question: str) -> str:
    '''Answers a text question using GPT-3. The input question is always a formatted string with a variable in it.

    Parameters
    ----------
    question: str
        the text question to ask. Must not contain any reference to 'the image' or 'the photo', etc.
    '''
    return llm_query(question)


class VideoSegment:
    """A Python class containing a set of frames represented as ImagePatch objects, as well as relevant information.
    Attributes
    ----------
    video : torch.Tensor
        A tensor of the original video.
    start : int
        An int describing the starting frame in this video segment with respect to the original video.
    end : int
        An int describing the ending frame in this video segment with respect to the original video.
    num_frames->int
        An int containing the number of frames in the video segment.

    Methods
    -------
    frame_iterator->Iterator[ImagePatch]
    trim(start, end)->VideoSegment
        Returns a new VideoSegment containing a trimmed version of the original video at the [start, end] segment.
    select_answer(info, question, options)->str
        Returns the answer to the question given the options and additional information.
    """

    def __init__(self, video: torch.Tensor, start: int = None, end: int = None, parent_start=0, queues=None):
        """Initializes a VideoSegment object by trimming the video at the given [start, end] times and stores the
        start and end times as attributes. If no times are provided, the video is left unmodified, and the times are
        set to the beginning and end of the video.

        Parameters
        -------
        video : torch.Tensor
            A tensor of the original video.
        start : int
```

```python
            An int describing the starting frame in this video segment with respect to the original video.
        end : int
            An int describing the ending frame in this video segment with respect to the original video.
        """

        if start is None and end is None:
            self.trimmed_video = video
            self.start = 0
            self.end = video.shape[0]  # duration
        else:
            self.trimmed_video = video[start:end]
            if start is None:
                start = 0
            if end is None:
                end = video.shape[0]
            self.start = start + parent_start
            self.end = end + parent_start

        self.num_frames = self.trimmed_video.shape[0]

    def frame_iterator(self) -> Iterator[ImagePatch]:
        """Returns an iterator over the frames in the video segment."""
        for i in range(self.num_frames):
            yield ImagePatch(self.trimmed_video[i], self.start + i)

    def trim(self, start: Union[int, None] = None, end: Union[int, None] = None) -> VideoSegment:
        """Returns a new VideoSegment containing a trimmed version of the original video at the [start, end]
        segment.

        Parameters
        ----------
        start : Union[int, None]
            An int describing the starting frame in this video segment with respect to the original video.
        end : Union[int, None]
            An int describing the ending frame in this video segment with respect to the original video.

        Examples
        --------
        >>> # Return the second half of the video
        >>> def execute_command(video):
        >>>     video_segment = VideoSegment(video)
        >>>     video_second_half = video_segment.trim(video_segment.num_frames // 2, video_segment.num_frames)
        >>>     return video_second_half
        """
        if start is not None:
            start = max(start, 0)
        if end is not None:
            end = min(end, self.num_frames)

        return VideoSegment(self.trimmed_video, start, end, self.start)

    def select_answer(self, info: dict, question: str, options: List[str]) -> str:
        return select_answer(self.trimmed_video, info, question, options)

    def __repr__(self):
        return "VideoSegment({}, {})".format(self.start, self.end)
```

Listing 1. **Full API.**

Not all methods are used in all the benchmarks. Next we describe in more detail what content is used for the API specifications for every benchmark.

- **RefCOCO and RefCOCO+**. We use all the methods from the `ImagePatch` class except for `best_text_match` and `simple_query`. We also use the `best_text_match` and `distance` functions. Additionally we add `ImagePatch` usage examples in the API definition that are representative of the RefCOCO dataset, and look like the following:

```python
# chair at the front
def execute_command(image) -> ImagePatch:
    # Return the chair
    image_patch = ImagePatch(image)
    chair_patches = image_patch.find("chair")
    chair_patches.sort(key=lambda chair: chair.compute_depth())
    chair_patch = chair_patches[0]
    # Remember: return the chair
    return chair_patch
```

Listing 2. **RefCOCO example.**

- **GQA**. The GQA API contains all the contents in the API from Listing 1 up until the `llm_query` function, which is not used. The `ImagePatch` usage examples look like the following:

```python
# Is there a backpack to the right of the man?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    man_patches = image_patch.find("man")
    # Question assumes one man patch
    if len(man_patches) == 0:
        # If no man is found, query the image directly
        return image_patch.simple_query("Is there a backpack to the right of the man?")
    man_patch = man_patches[0]
    backpack_patches = image_patch.find("backpack")
    # Question assumes one backpack patch
    if len(backpack_patches) == 0:
        return "no"
    for backpack_patch in backpack_patches:
        if backpack_patch.horizontal_center > man_patch.horizontal_center:
            return "yes"
    return "no"
```

Listing 3. **GQA example.**

- **OK-VQA**. The API only uses the `simple_query` method from `ImagePatch`. It additionally uses the `llm_query` function. The `ImagePatch` usage examples look like the following:

```python

# Who is famous for allegedly doing this in a lightning storm?
def execute_command(image)->str:
    # The question is not direct perception, so we need to ask the image for more information
    # Salient information: what is being done?
    image = ImagePatch(image)
    guesses = []
    action = image.simple_query("What is being done?")
    external_knowledge_query = "Who is famous for allegedly {} in a lightning storm?".format(action)
    step_by_step_guess = llm_query(external_knowledge_query)
    guesses.append("what is being done is {}".format(action) + ", so " + step_by_step_guess)
    direct_guess = image.simple_query("Who is famous for allegedly doing this in a lightning storm?")
    guesses.append(direct_guess)
    return process_guesses("Who is famous for allegedly doing this in a lightning storm?", guesses)
```

Listing 4. **OK-VQA example.**

- **NeXT-QA**. The `VideoSegment` class is added to the API definition, and the available `ImagePatch` methods are `find`, `exists`, `best_text_match` and `simple_query`. The function `best_image_match` is also used. The `ImagePatch` usage examples look like:

```python
# why does the man with a red hat put his arm down at the end of the video
# possible answers: ['watching television', 'searching for food', 'move its head', 'looking over cardboard box', 'looks at the camera']
def execute_command(video, possible_answers, question)->[str, dict]:
    # Reason every step
    video_segment = VideoSegment(video)
    # Caption last frame of the video (end of video)
    last_frame = ImagePatch(video_segment, -1)
    last_caption = last_frame.simple_query("What is this?")
    men = last_frame.find("man")
    if len(men) == 0:
        men = [last_frame]
    man = men[0]
    man_action = man.simple_query("What is the man doing?")
    # Answer the question. Remember to create the info dictionary
    info = {
        "Caption of last frame": last_caption,
        "Man looks like he is doing": man_action
    }
    answer = video_segment.select_answer(info, question, possible_answers)
    return answer, info
```

Listing 5. **NeXT-QA example.**

- **Beyond benchmarks**. For the examples in Figure 1 we use the same API as the one used for the benchmarks, and the usage examples are taken from the benchmark APIs, combining them to have more generality. We do not add any other example, `ViperGPT` generalizes to the complex cases shown in Figure 1 just based on the provided API.

Note that in some of the examples we added comments, as well as error handling. The generated code also contains similar lines. We removed those for clarity in the figures shown in the main paper.