loaders to show in UI

- the disclaimers is to be same for all subtasks.

audio

```
DISCLAIMER:

This dataset loader script is provided as an example.
It assumes a specific directory structure, file formats and
preprocessing steps.
You will likely need to adjust it for your specific use case.

Typical items you may customize include:

- Directory names (e.g., "audio", "input", "target")
- File formats and extensions (e.g., .wav, .mp3)
- Labeling scheme and logic
- Audio sampling rates, durations, and processing steps
- Transforms passed via the transforms parameter
- Return structure of data: "dict", "tuple", or "raw"

Make sure to install and import the necessary packages
such as torchaudio, torch, torchvision, etc. before running.

Always verify the loader behavior with your own dataset structure and
preprocess needs.
"""
```

1. Audio Classification

PROF

```
class AudioClassificationDataset(Dataset):
    def __init__(self, root, split="train", duration=5.0,
label_type="folder-name",
                 label_map=None, return_format="dict", transforms=None):
        self.root = Path(root) / split
        self.duration = duration
        self.return_format = return_format
        self.label_type = label_type
        self.transforms = transforms if transforms is not None else []

        self.sample_rate = 16000
        self.clip_samples = int(self.duration * self.sample_rate)

        if label_map is None:
            self.label_map = self._infer_label_map()
        else:
```

```
            self.label_map = label_map

        self.data = self._load_files()

    def _infer_label_map(self):
        classes = sorted([d.name for d in self.root.iterdir() if
d.is_dir()])
        return {cls: idx for idx, cls in enumerate(classes)}

    def _load_files(self):
        data = []
        for cls in self.label_map:
            class_files = (self.root / cls).glob('*.wav')
            data.extend([(file, self.label_map[cls]) for file in
class_files])
        return data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        path, label = self.data[idx]
        waveform, sr = torchaudio.load(path)

        # Resample if necessary
        if sr != self.sample_rate:
            waveform = torchaudio.functional.resample(waveform, sr,
self.sample_rate)

        # Trim or pad waveform
        if waveform.shape[1] < self.clip_samples:
            waveform = torch.nn.functional.pad(waveform, (0,
self.clip_samples - waveform.shape[1]))
        else:
            waveform = waveform[:, :self.clip_samples]

        # Apply custom transforms
        for transform in self.transforms:
            waveform = transform(waveform)

        if self.return_format == 'tuple':
            return waveform, label
        elif self.return_format == 'raw':
            return waveform
        else:
            return {'audio': waveform, 'label': label}

def get_audio_classification_loader(root, batch_size=16, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = AudioClassificationDataset(root=root, split=split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

2. Audio Conversion

```
class AudioConversionDataset(Dataset):
    def __init__(self, root, split="train", duration=5.0,
return_format="dict", transforms=None):
        self.root = Path(root) / split
        self.sample_rate = 16000
        self.clip_samples = int(duration * self.sample_rate)
        self.return_format = return_format
        self.transforms = transforms if transforms is not None else []

        self.input_files = sorted((self.root / 'input').glob('*.wav'))

    def __len__(self):
        return len(self.input_files)

    def __getitem__(self, idx):
        input_path = self.input_files[idx]
        target_path = Path(str(input_path).replace('input', 'target'))

        input_waveform, sr = torchaudio.load(input_path)
        target_waveform, target_sr = torchaudio.load(target_path)

        if sr != self.sample_rate:
            input_waveform =
torchaudio.functional.resample(input_waveform, sr, self.sample_rate)
        if target_sr != self.sample_rate:
            target_waveform =
torchaudio.functional.resample(target_waveform, target_sr,
self.sample_rate)

        input_waveform = self._fix_waveform_length(input_waveform)
        target_waveform = self._fix_waveform_length(target_waveform)

        for transform in self.transforms:
            input_waveform = transform(input_waveform)
            target_waveform = transform(target_waveform)

        if self.return_format == 'tuple':
            return input_waveform, target_waveform
        elif self.return_format == 'raw':
            return input_waveform
        else:
            return {'input': input_waveform, 'target': target_waveform}

    def _fix_waveform_length(self, waveform):
        if waveform.shape[1] < self.clip_samples:
            return torch.nn.functional.pad(waveform, (0,
self.clip_samples - waveform.shape[1]))
        else:
```

PROF

```
            return waveform[:, :self.clip_samples]

def get_audio_conversion_loader(root, batch_size=4, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = AudioConversionDataset(root=root, split=split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

3. Audio Generation

```
class AudioGenerationDataset(Dataset):
    def __init__(
        self,
        root,
        split="train",
        duration=5.0,
        include_prompt=True,
        include_target=True,
        audio_pair=False,
        text_pair=False,
        return_format="dict",
        transforms=None
    ):
        self.root = Path(root) / split
        self.include_prompt = include_prompt
        self.include_target = include_target
        self.audio_pair = audio_pair
        self.text_pair = text_pair
        self.return_format = return_format
        self.transforms = transforms if transforms is not None else []

        self.sample_rate = 16000
        self.clip_samples = int(duration * self.sample_rate)

        self.data = self._load_data()

    def _load_data(self):
        if self.audio_pair:
            prompt_files = sorted((self.root / "input").glob("*.wav"))
        elif self.text_pair:
            prompt_files = sorted((self.root / "text").glob("*.txt"))
        else:
            prompt_files = sorted((self.root / "prompt").glob("*.wav"))
        return prompt_files

    def __len__(self):
        return len(self.data)

    def _load_audio(self, path):
        waveform, sr = torchaudio.load(path)
```

PROF

```
        if sr != self.sample_rate:
            waveform = torchaudio.functional.resample(waveform, sr,
self.sample_rate)
        if waveform.shape[1] < self.clip_samples:
            waveform = torch.nn.functional.pad(waveform, (0,
self.clip_samples - waveform.shape[1]))
        else:
            waveform = waveform[:, :self.clip_samples]

        for transform in self.transforms:
            waveform = transform(waveform)
        return waveform

    def __getitem__(self, idx):
        item = {}
        if self.audio_pair:
            prompt_path = self.data[idx]
            target_path = Path(str(prompt_path).replace("input",
"target"))
            if self.include_prompt:
                item["prompt"] = self._load_audio(prompt_path)
            if self.include_target:
                item["target"] = self._load_audio(target_path)

        elif self.text_pair:
            prompt_path = self.data[idx]
            target_path = Path(str(prompt_path).replace("text",
"target").replace('.txt', '.wav'))
            if self.include_prompt:
                item["prompt"] = prompt_path.read_text().strip()
            if self.include_target:
                item["target"] = self._load_audio(target_path)

        else:  # Default audio prompt only
            audio_path = self.data[idx]
            if self.include_prompt:
                item["prompt"] = self._load_audio(audio_path)

        if self.return_format == 'tuple':
            # Return tuple with ordering: (prompt, target) if target is
included.
            output = tuple(item[key] for key in ('prompt', 'target') if
key in item)
            return output[0] if len(output) == 1 else output
        elif self.return_format == 'raw':
            return item.get('prompt', None)
        else:
            return item

def get_audio_generation_loader(root, batch_size=8, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = AudioGenerationDataset(root=root, split=split,
```

```
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

image

```
DISCLAIMER:

This dataset loader script is a starting template and likely needs
adjustment to exactly match your dataset requirements.

Common scenarios needing your attention:

- Folder and file naming (currently assumes 'images', 'masks', 'prompt',
'target', etc.).
- Image formats (.jpg, .png, etc.).
- Image dimensions and resizing behavior.
- Custom transformations, passed as a list in the "transforms" argument.
- Label format (annotations, class mappings, etc.).
- Output data format ('tuple', 'dict', 'raw').

Ensure all necessary packages such as torch, torchvision, PIL, etc.
are installed and imported.

Always thoroughly verify data loading and preprocessing for correctness
before actual training or evaluation.
"""
```

1. Image Classification

PROF

```
class ImageClassificationDataset(Dataset):
    def __init__(self, root, split="train", return_format="dict",
label_map=None, transforms=None):
        self.root = Path(root) / split
        self.return_format = return_format
        self.transforms = transforms if transforms else [
            transforms.Resize((224, 224)),
            transforms.ToTensor()]

        self.label_map = label_map or self._infer_label_map()
        self.data = self._load_data()

    def _infer_label_map(self):
        classes = sorted([d.name for d in self.root.iterdir() if
d.is_dir()])
        return {cls: idx for idx, cls in enumerate(classes)}

    def _load_data(self):
        items = []
```

```
        for cls in self.label_map:
            for img_path in (self.root / cls).glob('*.jpg'):
                items.append((img_path, self.label_map[cls]))
        return items

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_path, label = self.data[idx]
        image = Image.open(img_path).convert("RGB")

        for transform in self.transforms:
            image = transform(image)

        if self.return_format == 'tuple':
            return image, label
        elif self.return_format == 'raw':
            return image
        else:
            return {'image': image, 'label': label}

def get_image_classification_loader(root, batch_size=32, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = ImageClassificationDataset(root=root, split=split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

2. Image Generation

```
class ImageGenerationDataset(Dataset):
    def __init__(self, root, split="train", return_format="dict",
text_pair=True, transforms=None):
        self.root = Path(root) / split
        self.return_format = return_format
        self.text_pair = text_pair
        self.transforms = transforms if transforms else [
            transforms.Resize((256, 256)),
            transforms.ToTensor()
        ]

        self.data = self._load_data()

    def _load_data(self):
        if self.text_pair:
            prompt_files = sorted((self.root / 'prompt').glob('*.txt'))
            return [(p, Path(str(p).replace('prompt',
'target').replace('.txt', '.jpg'))) for p in prompt_files]
        else:
            return list((self.root / 'target').glob('*.jpg'))
```

PROF

```
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = {}
        if self.text_pair:
            prompt_path, img_path = self.data[idx]
            prompt = prompt_path.read_text().strip()
            item['prompt'] = prompt
        else:
            img_path = self.data[idx]

        image = Image.open(img_path).convert("RGB")
        for transform in self.transforms:
            image = transform(image)

        item['image'] = image

        if self.return_format == 'tuple':
            return (item['prompt'], item['image']) if 'prompt' in item
else (item['image'],)
        elif self.return_format == 'raw':
            return item['image']
        else:
            return item

def get_image_generation_loader(root, batch_size=16, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = ImageGenerationDataset(root=root, split=split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

3. Object Detection

PROF

```
class ObjectDetectionDataset(Dataset):
    def __init__(self, root, split="train", return_format="dict",
label_map=None, transforms=None):
        self.root = Path(root) / split
        self.return_format = return_format
        self.transforms = transforms if transforms else [
            transforms.Resize((512, 512)),
            transforms.ToTensor()
        ]

        self.data = list((self.root / 'images').glob('*.jpg'))
        self.label_map = label_map or self._infer_label_map()

    def _infer_label_map(self):
        labels = set()
```

```
        for annotation in (self.root / 'labels').glob('*.json'):
            anns = json.loads(annotation.read_text())
            labels.update(ann['label'] for ann in anns)
        return {label: idx for idx, label in enumerate(sorted(labels))}

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_path = self.data[idx]
        ann_path = Path(str(img_path).replace("images",
"labels").replace(".jpg", ".json"))

        anns = json.loads(ann_path.read_text())
        boxes = torch.tensor([ann['bbox'] for ann in anns],
dtype=torch.float32)
        labels = torch.tensor([self.label_map[ann['label']] for ann in
anns], dtype=torch.int64)

        image = Image.open(img_path).convert("RGB")
        for transform in self.transforms:
            image = transform(image)

        target = {'boxes': boxes, 'labels': labels}

        if self.return_format == 'tuple':
            return image, target
        elif self.return_format == 'raw':
            return image
        else:
            return {'image': image, **target}

def get_object_detection_loader(root, batch_size=4, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = ObjectDetectionDataset(root=root, split=split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle,
                      collate_fn=lambda x: tuple(zip(*x)))
```

PROF

4. Segmentation

```
class SegmentationDataset(Dataset):
    def __init__(self, root, split="train", return_format="dict",
transforms=None):
        self.root = Path(root) / split
        self.return_format = return_format

        self.transforms = transforms if transforms else [
            transforms.Resize((256, 256)),
            transforms.ToTensor()
```

```
        ]
        self.mask_transform = transforms.Compose([
            transforms.Resize((256, 256), interpolation=Image.NEAREST),
            transforms.PILToTensor()
        ])

        self.image_paths = sorted((self.root / 'images').glob('*.jpg'))
        self.mask_paths = [Path(str(p).replace('images',
'masks').replace('.jpg', '.png')) for p in self.image_paths]

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image = Image.open(self.image_paths[idx]).convert("RGB")
        mask = Image.open(self.mask_paths[idx])

        for transform in self.transforms:
            image = transform(image)
        mask = self.mask_transform(mask)[0].long()

        if self.return_format == 'tuple':
            return image, mask
        elif self.return_format == 'raw':
            return image
        else:
            return {'image': image, 'mask': mask}

def get_segmentation_loader(root, batch_size=8, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = SegmentationDataset(root=root, split=split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

PROF

text

```
DISCLAIMER:

This text dataset loader script serves as a starting template.
It is likely you will need to adjust it specifically for your dataset
and workflow.

Areas requiring special attention are usually:

- Directory paths (currently assumed structures like "prompt", "target",
"source", "document", "summary" etc.)
- File formats and extensions (.txt, .json, etc.)
- Tokenizer choice and configurations (Hugging Face tokenizer models or
custom)
- Maximum text lengths, padding rules, and truncation strategies
```

```
- Additional preprocessing or data transformations via the "transforms"
parameter.

Always test thoroughly with your specific dataset.
"""
```

1. Text Classification

```
class TextClassificationDataset(Dataset):
    def __init__(self, root, split="train", tokenizer_name="bert-base-
uncased",
                 max_length=512, return_format="dict", label_map=None,
transforms=None):
        self.root = Path(root) / split
        self.tokenizer = transforms if transforms else
AutoTokenizer.from_pretrained(tokenizer_name)
        self.max_length = max_length
        self.return_format = return_format
        self.label_map = label_map or self._infer_label_map()
        self.data = self._load_data()

    def _infer_label_map(self):
        classes = sorted([d.name for d in self.root.iterdir() if
d.is_dir()])
        return {cls:idx for idx,cls in enumerate(classes)}

    def _load_data(self):
        return [(file.read_text(), self.label_map[cls.name])
                for cls in self.root.iterdir() if cls.is_dir()
                for file in cls.glob('*.txt')]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        text, label = self.data[idx]
        enc = self.tokenizer(text, truncation=True,
padding="max_length",
                             max_length=self.max_length,
return_tensors="pt")
        item = {'input_ids': enc['input_ids'].squeeze(0),
'attention_mask': enc['attention_mask'].squeeze(0), 'label':label}
        if self.return_format == 'tuple':
            return item['input_ids'], item['label']
        elif self.return_format == 'raw':
            return item['input_ids']
        else:
            return item

def get_text_classification_loader(root, batch_size=32, split='train',
```

```
shuffle=True, transforms=None, **kwargs):
    dataset = TextClassificationDataset(root, split,
transforms=transforms, **kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

2. Text Generation

```
class TextGenerationDataset(Dataset):
    def __init__(self, root, split="train", tokenizer_name="gpt2",
                 max_length=512, return_format="dict", text_pair=False,
transforms=None):
        self.root = Path(root) / split
        self.tokenizer = transforms if transforms else
AutoTokenizer.from_pretrained(tokenizer_name)
        self.max_length = max_length
        self.return_format = return_format
        self.text_pair = text_pair
        self.data = self._load_data()

    def _load_data(self):
        prompts = sorted((self.root/'prompt').glob('*.txt'))
        if self.text_pair:
            return [(p.read_text().strip(),
Path(str(p).replace('prompt', 'target')).read_text().strip()) for p in
prompts]
        return [(p.read_text().strip(), None) for p in prompts]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        prompt, target = self.data[idx]
        text_input = prompt if not target else prompt +
self.tokenizer.eos_token + target
        enc = self.tokenizer(text_input, truncation=True,
padding="max_length",
                             max_length=self.max_length,
return_tensors="pt")
        item = {'input_ids': enc.input_ids.squeeze(0), 'attention_mask':
enc.attention_mask.squeeze(0)}
        if target:
            item['target'] = target
        format_handlers = {
            'tuple': lambda x: (x['input_ids'], x.get('target')),
            'raw': lambda x: x['input_ids']
        }
        return format_handlers.get(self.return_format, lambda x:x)(item)

def get_text_generation_loader(root, batch_size=8, split='train',
shuffle=True, transforms=None, **kwargs):
```

```
    dataset = TextGenerationDataset(root, split, transforms=transforms,
**kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

3. Text Pair (Summarization/Translation)

```
class TextPairDataset(Dataset):
    def __init__(self, root, split="train", tokenizer_name="xlm-roberta-
base",
                 max_length=512, return_format="dict",
src_folder="source", tgt_folder="target", transforms=None):
        self.root = Path(root) / split
        self.tokenizer = transforms if transforms else
AutoTokenizer.from_pretrained(tokenizer_name)
        self.max_length = max_length
        self.return_format = return_format
        self.data = self._load_data(src_folder, tgt_folder)

    def _load_data(self, src_folder, tgt_folder):
        src_paths = sorted((self.root/src_folder).glob('*.txt'))
        tgt_paths = [Path(str(p).replace(src_folder, tgt_folder)) for p
in src_paths]
        return [(sp.read_text().strip(), tp.read_text().strip()) for
sp,tp in zip(src_paths,tgt_paths)]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        src, tgt = self.data[idx]
        enc = self.tokenizer(src, truncation=True, padding="max_length",
                             max_length=self.max_length,
return_tensors="pt")
        item = {'input_ids':enc.input_ids.squeeze(0),
'attention_mask':enc.attention_mask.squeeze(0), 'target':tgt}
        format_handlers = {
            'tuple': lambda x:(x['input_ids'], x['target']),
            'raw': lambda x:x['input_ids']
        }
        return format_handlers.get(self.return_format, lambda x:x)(item)

def get_text_pair_loader(root, batch_size=8, split='train',
shuffle=True, transforms=None, **kwargs):
    dataset = TextPairDataset(root, split, transforms=transforms,
**kwargs)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

PROF