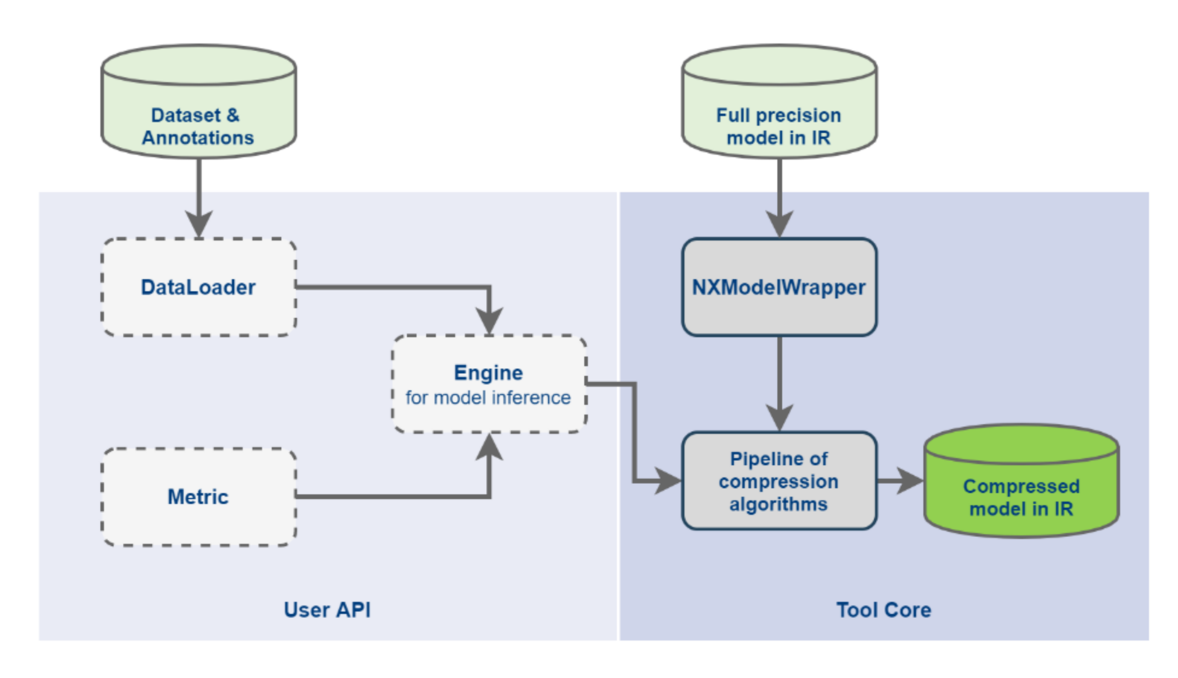
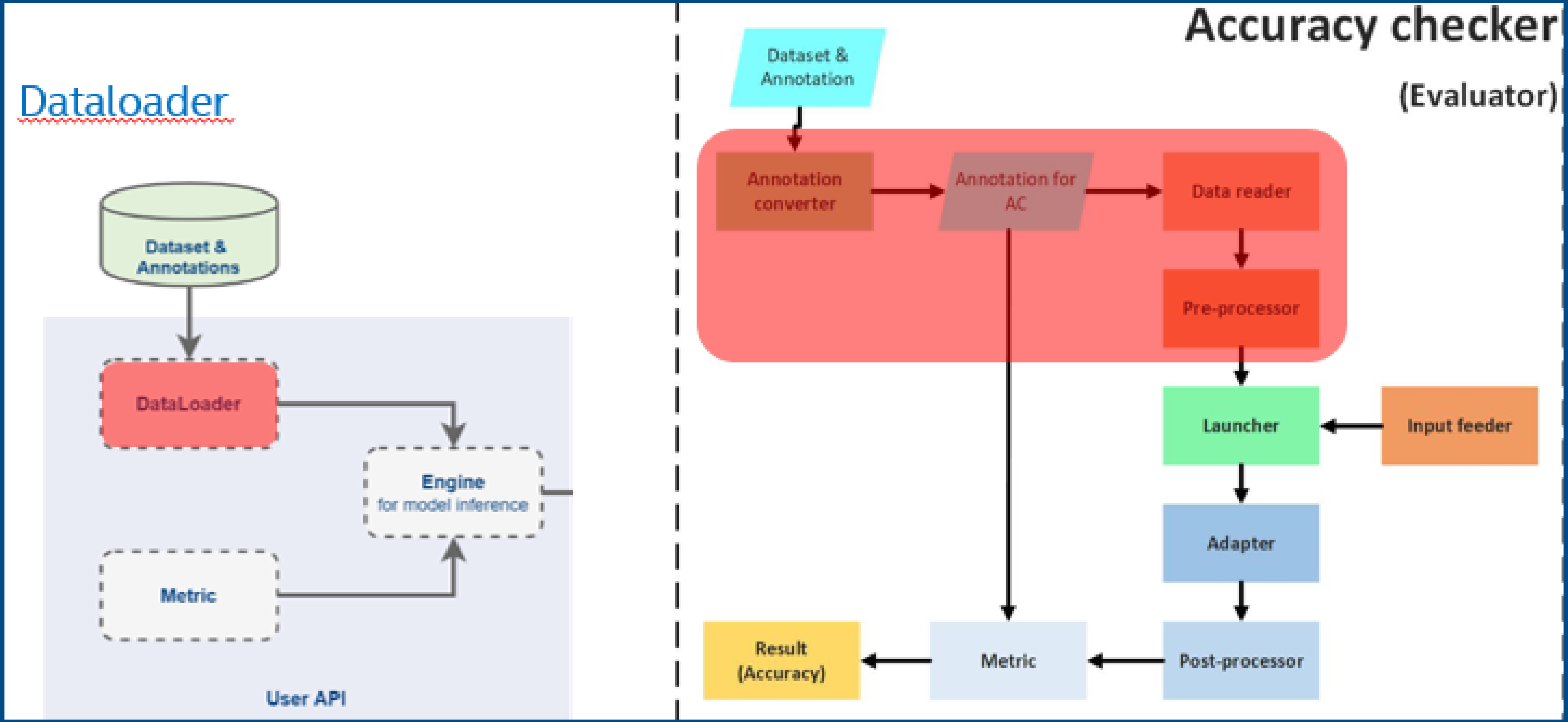
Using the Post Optimization Toolkit API to convert models to INT8

API Description

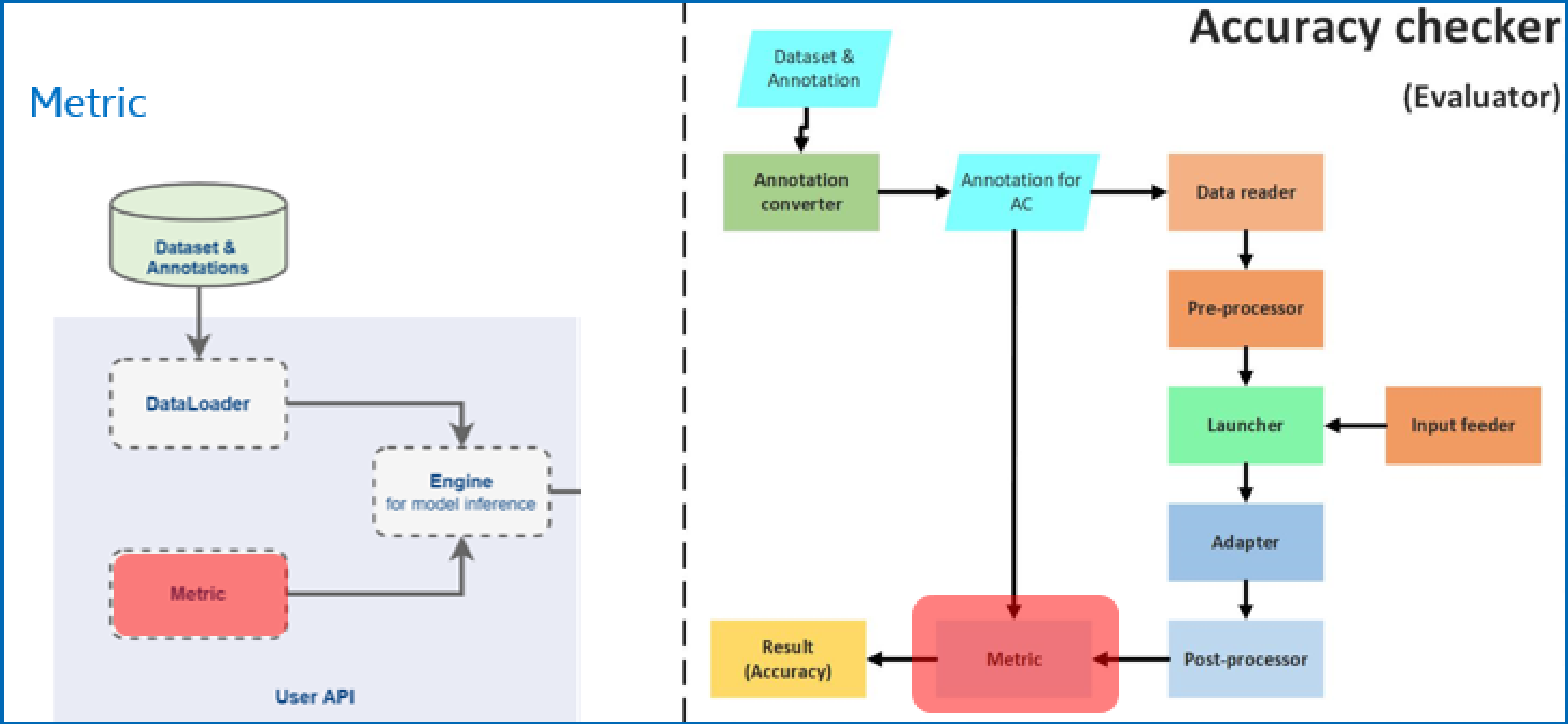
- DataLoader
  - Loads data from a dataset and applies pre-processing to them which provides access to the pre-processed data by index.
- Metric
  - Evaluates the result
- IEEngine
  - Provides model inference, collects statistics for activations, and calculates accuracy metrics for dataset



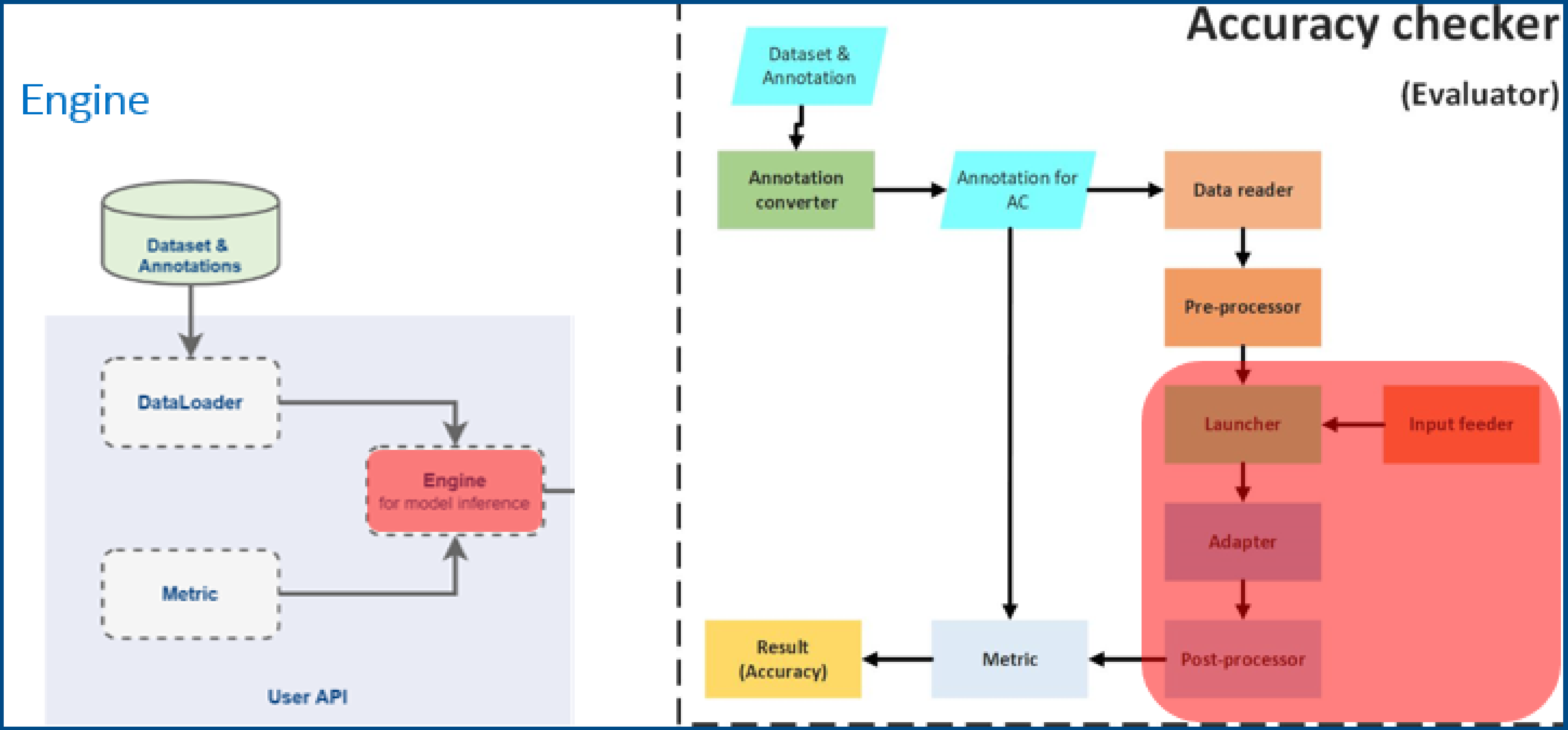
Comparison of the POT API and Accuracy Checker



Comparison of the POT API and Accuracy Checker Cont.



Comparison of the POT API and Accuracy Checker Cont.



Implementing Dataloader

- Override 3 functions
  - __init__() -> parse configs, ex: dataset directory.
  - __len__() -> determine the dataset size.
  - __getitem__() -> provide the data which processed by reader and preprocess functions and annotation file for **Engine** by index.
- Implement date reader function
  - Ex: image reader
- Implement preprocess function
  - Ex: resize

Implementing Dataloader Example

- Dataloader for libri speech dataset
  - Audio file -> .wav file -> **audio reader**
  - Annotation file -> .txt file -> **text reader**
- Audio preprocess function
  - MFCC feature for audio file

Implementing Dataloader Cont.

```
class LibriSpeechDataLoader(DataLoader):                          # -> New Dataloader for librispeech dataset

    def __init__(self, config):                                   # -> config is from main function which include the arguments
    if not isinstance(config, Dict):
            config = Dict(config)
        super().__init__(config)
        self._audio_files = sorted(os.listdir(self.config.audio_source))       # -> parse config, and sort the data

    def __len__(self):                                            # -> check the dataset size
        return len(self._audio_files)

    def __getitem__(self, index):                                 # -> get data by index
        if index >= len(self):
            raise IndexError
                                                                  # -> get the path of data and annotation file, and reader the file
        audio_path = os.path.join(self.config.audio_source, self._audio_files[index])
        annotation_path = os.path.join(self.config.annotation_source, self._audio_files[index][:-3] + 'txt')

        annotation = (index, self._text_reader(annotation_path))  # -> functions for text and audio reader, and audio preprocess
        audio_ftr = self._preprocess_audio(self._audio_reader(audio_path))
        return annotation, audio_ftr
```

Implementing Metric

- Override 3 properties
  - value -> returns the accuracy metric value for the last model output.
  - avg_value -> returns the average accuracy metric value for all model outputs.
  - get_attributes -> returns a dictionary of metric attributes
- Override 3 functions
  - __init__() -> parse arguments, ex: threshold
  - update -> calculates and updates the accuracy metric value using last model output and annotation.
  - reset -> resets collected accuracy metric.

Implementing Metric Example

- Metric for word error rate
  - Function for determine the word error rate between predict result and annotation
  - threshold argument for word error rate

Implementing Metric Cont.

```
class WordErrorRate(Metric):                                        # -> new Metric for word error rate

    def __init__(self, threshold = 20.0):
        super().__init__()
        self._threshold = threshold                                 # -> threshold for word error rate
        self._name = 'word_error_rate'
        self._overall_metric = []                                   # -> a list to save metric value of each data

    @property
    def value(self):                                                # -> return the accuracy of last output
        return {self._name: [np.mean(self._overall_metric[-1])]}

    @property
    def avg_value(self):                                            # -> return the mean accuracy
        return {self._name: np.mean(self._overall_metric)}
    def reset(self):                                                # -> reset the metric list
        self._overall_metric = []
    def get_attributes(self):                                       # -> set the data type and tuning direction of metric,
        return {self._name: {'direction': 'higher-better', 'type': 'dice_index'}}  direction means the metric is higher better or higher worse
    def update(self, output, target):                               # -> compute the word error rate of output
        _WER = word_error_rate(output, target)                      # -> static method to compute the word error rate
        self._overall_metric.append(int(_WER <= self._threshold))
```

Implementing IE Engine

- Assign the Dataloader and Metric to Engine for model inference
- Override the static method (if required)
  - postprocess_output -> process output data, ex CTC
- Override the blow function (if required)
  - _fill_input -> override it when you have multi-input node (default support 1 and 2 input nodes)
  - _update_metrics -> override it when you have multi-output node
  - _process_dataset -> override it when infer process is specific
  - _process_dataset_async -> override it when async infer process is specific

Implementing IE Engine Cont.

- Example:
- Inference engine of Deep speech
  - 3 input nodes -> **override _fill_input**
  - 3 output nodes and only 1 node provide for result -> **override _update_metric**
  - Iterate the model to support different input length -> **override _process_dataset**
  - Post process -> CTC decoder -> **override postprocess_output**

Implementing IE Engine Cont.

```
class SpeechEngine(IEEngine):                                    # -> New IEEngine for deep speech

    def _fill_input(self, audio_batch):
        if len(self._model.inputs) == 3:                         # -> Check the model  has three input nodes
            stack_size = len(audio_batch)
            hidden_state_init = np.zeros((stack_size, 1, 2048))  # -> Initialize hidden state

            return {'input_node': np.stack(audio_batch, axis=0), # -> the data for inference
                    'previous_state_c/read/placeholder_port_0': hidden_state_init,
                    'previous_state_h/read/placeholder_port_0': hidden_state_init}

            raise Exception('Unsupported number of inputs')

    @staticmethod
    def postprocess_output(output, metadata):# -> run post process, just fit the interface, no mata data is fine
        text = CTC_decoder(output)                               # -> static function CTC_decoder
        return text
```

Implementing IE Engine Cont.

```
def _update_metrics(self, outputs, annotations, need_metrics_per_sample=False):
    """ Updates metrics.
    :param outputs: layer outputs
    :param annotations: a list of annotations for metrics collection [(img_id, annotation)]
    :param need_metrics_per_sample: whether to collect metrics for each batch
    """
    # TODO: Create some kind of an order for the correct metric calculation
    logits = outputs['Softmax']  # output_layers are in a random order      # -> specific output node for result
    _, batch_annotations = map(list, zip(*annotations))
    annotations_are_valid = all(a is not None for a in batch_annotations)

        if self._metric and annotations_are_valid:
        self._metric.update(logits, batch_annotations)
        if need_metrics_per_sample:
            batch_metrics = self._metric.value
            for metric_name, metric_value in batch_metrics.items():
                for i, annotation in enumerate(annotations):
                    self._per_sample_metrics.append({'sample_id': annotation[0],
                                                     'metric_name': metric_name,
                                                     'result': metric_value[i]})
```

Implementing IE Engine Cont.

```
def _process_dataset(self, stats_layout, sampler, print_progress=False, need_metrics_per_sample=False):

    for batch_id, batch in iter(enumerate(sampler)):
        batch_annotations, audio_batch, none_meta = self._process_batch(batch)

        all_inputs = self._fill_input(audio_batch)
        b, p, s, c, i = all_inputs['input_node'].shape

        output_sentance = np.empty((0, 1, 29))
        for _b in range(b):                                                    # -> based on the batch size to run inference
            state_c = all_inputs['previous_state_c/read/placeholder_port_0'][_b]
            state_h = all_inputs['previous_state_h/read/placeholder_port_0'][_b]
            for _p in range(p):                                                # -> based on the package size to iterate model
                predictions = self._exec_model.infer(inputs={'previous_state_c/read/placeholder_port_0': state_c,
                                                   'previous_state_h/read/placeholder_port_0': state_h,
                                                   'input_node': [all_inputs['input_node'][_b][_p]]})

                state_c = predictions['lstm_fused_cell/BlockLSTM/TensorIterator.2']   # -> iterate hidden state and save output text
                state_h = predictions['lstm_fused_cell/BlockLSTM/TensorIterator.1']
                output_sentance = np.concatenate((output_sentance, predictions['Softmax']))
        output_sentance = np.expand_dims(output_sentance, axis=0)
        self._process_infer_output(stats_layout, {'Softmax': output_sentance}, batch_annotations, none_meta,
                                   need_metrics_per_sample)                    # -> pass the output for post process
```

Implementing Main

- Configs
  - model_config
  - engine_config
  - dataset_config
  - quantize algorithms
- Quantize pipeline
  - Follow the implemented Dataloader, Metric, IEEngine to assign the configs and pass it to pipeline

Implementing Main Cont.

```
def main():
    parser = get_common_argparser()
    parser.add_argument(                       # -> add the new argument
        '--label-dir',                         # -> which are not default
        help= "Path to the annotation files' directory“,
        required=True                          # -> default argument is
    )                                          # -> --model and --dataset
    args = parser.parse_args()
    if not args.weights:
        args.weights = '{}.bin'.format(os.path.splitext(args.model)[0])

    model_config = Dict({
        'model_name': 'deep-speech',
        'model': os.path.expanduser(args.model),
        'weights': os.path.expanduser(args.weights)
    })
    engine_config = Dict({                     # -> For inference
        'device': 'CPU',                       # -> run POT on which device
        'stat_requests_number': 1,             # -> number for infer_request
        'eval_requests_number': 1              # -> 1 means do sync
inference
    })
```

```
dataset_config = Dict({
        'audio_source': os.path.expanduser(args.dataset),
        'annotation_source': os.path.expanduser(args.lab
el_dir),
    })

    algorithms = [                             # -> algorithm for POT
        {
            'name': 'DefaultQuantization',
            'params': {
                'target_device': 'CPU',
                'preset': 'performance',
                'stat_subset_size': 1
            }
        }
    ]                 # -> In the list, algorithms, you can follow the
json
                      # -> file to set up the algorithm and parameter
to
                      # -> to run the POT.
```

Implementing Main Cont.

```
# Step 1: Load the model.
    model = load_model(model_config)

    # Step 2: Initialize the data loader.
    data_loader = LibriSpeechDataLoader(dataset_config)

    # Step 3 (Optional. Required for AccuracyAwareQuantization): Initialize the metric.
    metric = WordErrorRate(threshold=20.0)

    # Step 4: Initialize the engine for metric calculation and statistics collection.
    engine = SpeechEngine(config=engine_config,
                          data_loader=data_loader,
                          metric=metric)

    # Step 5: Create a pipeline of compression algorithms.
    pipeline = create_pipeline(algorithms, engine)
```

Implementing Main Cont.

```
# Step 6: Execute the pipeline.
compressed_model = pipeline.run(model)

# Step 7 (Optional):  Compress model weights to quantized precision
# in order to reduce the size of final .bin file.
compress_model_weights(compressed_model)

# Step 8: Save the compressed model to the desired path.
save_model(compressed_model, os.path.join(os.path.curdir, 'optimized'))

# Step 9 (Optional): Evaluate the compressed model. Print the results.
metric_results = pipeline.evaluate(compressed_model)
if metric_results:
    for name, value in metric_results.items():
        print('{: <27s}: {}'.format(name, value))
```

Execute

- python3 {$PYTHON_FILE} –m ${MODLE} –d ${DATASET} –added_augument
- Example:
  - python3 deep_speech_pot.py –m ds_0.7.4.xml
    –d dev-clean/wav
    --label_dir dev-clean/annotation

deep_speech.py