use arrayvec::ArrayVec;

use itertools::Itertools;

use log::info;

use std::sync::Arc;

use tokio::sync::oneshot::Sender;

use tokio::time::Instant;

use crate::bootstrap::{TensorInput, TensorInputEnum};

use crate::cli\_args::{ARGS, MODEL\_SPECS};

use crate::{Callback, MAX\_NUM\_INPUTS, PredictResult};

use crate::metrics::{

BATCH\_SIZE, BATCH\_SIZE\_BY\_MODEL, BLOCKING\_REQUEST\_NUM, MODEL\_INFERENCE\_TIME\_COLLECTOR,

NUM\_BATCH\_PREDICTION, NUM\_BATCH\_PREDICTION\_BY\_MODEL, NUM\_BATCHES\_DROPPED,

NUM\_BATCHES\_DROPPED\_BY\_MODEL, NUM\_PREDICTION\_BY\_MODEL, NUM\_REQUESTS\_DROPPED,

NUM\_REQUESTS\_DROPPED\_BY\_MODEL,

};

use crate::predict\_service::Model;

use crate::tf\_proto::tensorflow\_serving::model\_spec::VersionChoice;

use crate::tf\_proto::tensorflow\_serving::PredictRequest;

use crate::tf\_proto::DataType;

#[derive(Debug)]

pub struct BatchPredictor<T: Model> {

pub model: Arc<T>,

pub input\_tensors: Vec<Vec<TensorInput>>,

pub callbacks: Vec<Callback>,

pub cur\_batch\_size: usize,

pub max\_batch\_size: usize,

pub batch\_time\_out\_millis: u64,

pub queue\_reset\_ts: Instant,

pub queue\_earliest\_rq\_ts: Instant,

}

impl PredictRequest {

#[inline(always)]

pub fn take\_input\_vals(

&mut self,

inputs: &ArrayVec<String, MAX\_NUM\_INPUTS>,

) -> Vec<TensorInput> {

let mut model\_inputs = Vec::<TensorInput>::new();

for input\_name in inputs.as\_slice() {

let input\_tensor = self

.inputs

.get\_mut(input\_name)

.unwrap\_or\_else(|| panic!("can't find {:?}", input\_name));

let dims = match &input\_tensor.tensor\_shape {

None => None,

Some(data) => Some(data.dim.iter().map(|d| d.size).collect\_vec()),

};

match input\_tensor.dtype() {

DataType::DtFloat => model\_inputs.push(TensorInput::new(

TensorInputEnum::Float(std::mem::take(&mut input\_tensor.float\_val)),

input\_name.to\_string(),

dims,

)),

DataType::DtDouble => model\_inputs.push(TensorInput::new(

TensorInputEnum::Double(std::mem::take(&mut input\_tensor.double\_val)),

input\_name.to\_string(),

dims,

)),

DataType::DtInt32 => model\_inputs.push(TensorInput::new(

TensorInputEnum::Int(std::mem::take(&mut input\_tensor.int\_val)),

input\_name.to\_string(),

dims,

)),

DataType::DtString => model\_inputs.push(TensorInput::new(

TensorInputEnum::String(std::mem::take(&mut input\_tensor.string\_val)),

input\_name.to\_string(),

dims,

)),

DataType::DtInt64 => model\_inputs.push(TensorInput::new(

TensorInputEnum::Int64(std::mem::take(&mut input\_tensor.int64\_val)),

input\_name.to\_string(),

dims,

)),

DataType::DtBool => model\_inputs.push(TensorInput::new(

TensorInputEnum::Boolean(std::mem::take(&mut input\_tensor.bool\_val)),

input\_name.to\_string(),

dims,

)),

\_ => panic!("unsupport input tensor type {:?}", input\_tensor.dtype()),

}

}

model\_inputs

}

#[inline(always)]

pub fn take\_model\_spec(&mut self) -> (String, Option<i64>) {

let model\_spec = self.model\_spec.as\_mut().unwrap();

let version = model\_spec

.version\_choice

.as\_ref()

.and\_then(|choice| match choice {

VersionChoice::Version(version) => Some(\*version),

\_ => None,

});

(std::mem::take(&mut model\_spec.name), version)

}

}

impl<T: Model> Drop for BatchPredictor<T> {

fn drop(&mut self) {

info!(

"drop old batch predictor for:{:}, queue:{}",

self.model,

self.input\_tensors.len()

);

if !self.input\_tensors.is\_empty() {

info!("now flush old predictor queue:{}", self.input\_tensors.len());

self.batch\_predict();

}

}

}

impl<T: Model> BatchPredictor<T> {

#[inline(always)]

pub fn push(&mut self, val: Vec<TensorInput>, resp: Sender<PredictResult>, ts: Instant) {

if self.input\_tensors.is\_empty() {

//only when queue is empty then we update ts to represent first request time

self.queue\_reset\_ts = Instant::now();

self.queue\_earliest\_rq\_ts = ts;

}

self.cur\_batch\_size += 1;

self.input\_tensors.push(val);

self.callbacks.push(Callback(resp, self.cur\_batch\_size));

}

#[inline(always)]

pub fn batch\_predict(&mut self) {

BATCH\_SIZE\_BY\_MODEL

.with\_label\_values(&[&MODEL\_SPECS[self.model.model\_idx()]])

.add(self.cur\_batch\_size as i64);

BATCH\_SIZE.add(self.cur\_batch\_size as i64);

let mut batch\_input\_tensors = Vec::with\_capacity(self.max\_batch\_size);

let mut batch\_callbacks = Vec::with\_capacity(self.max\_batch\_size);

let mut batch\_size = 0;

//now we swap so we can take two queues to the blocking-send thread and reset current queues

std::mem::swap(&mut self.input\_tensors, &mut batch\_input\_tensors);

std::mem::swap(&mut self.callbacks, &mut batch\_callbacks);

std::mem::swap(&mut self.cur\_batch\_size, &mut batch\_size);

let model = self.model.clone();

let batch\_earliest\_rq\_ts = self.queue\_earliest\_rq\_ts;

//info!("batch predict for model:{}, size:{}", self.tf\_model.export\_dir, vals0.len());

BLOCKING\_REQUEST\_NUM.inc();

tokio::task::spawn\_blocking(move || {

//proactively drop stale batches, we drop the entire batch

//as long as one request in that batch is stale. We may drop more than we could this way

//but this should work fairly decently well

if (batch\_earliest\_rq\_ts.elapsed().as\_millis() as u64) < ARGS.batch\_drop\_millis {

let model\_inference\_time\_start = Instant::now();

let (tensor\_outs, batch\_ends) =

model.do\_predict(batch\_input\_tensors, batch\_size as u64);

MODEL\_INFERENCE\_TIME\_COLLECTOR

.with\_label\_values(&[&MODEL\_SPECS[model.model\_idx()]])

.observe(model\_inference\_time\_start.elapsed().as\_millis() as f64);

let mut batch\_starts = vec![0; tensor\_outs.len()];

for (i, Callback(resp, \_)) in batch\_callbacks.into\_iter().enumerate() {

let mut tensors\_send\_back = vec![];

for (j, tensor\_out) in tensor\_outs.iter().enumerate() {

tensors\_send\_back.push(tensor\_out.slice(batch\_starts[j], batch\_ends[j][i]));

batch\_starts[j] = batch\_ends[j][i];

}

if resp

.send(PredictResult::Ok(tensors\_send\_back, model.version()))

.is\_err()

{

//use dropped metrics here as this is expected under high load

NUM\_REQUESTS\_DROPPED.inc();

NUM\_REQUESTS\_DROPPED\_BY\_MODEL

.with\_label\_values(&[&MODEL\_SPECS[model.model\_idx()]])

.inc();

}

}

} else {

for Callback(resp, \_) in batch\_callbacks.into\_iter() {

if resp.send(PredictResult::DropDueToOverload).is\_err() {

NUM\_REQUESTS\_DROPPED.inc();

NUM\_REQUESTS\_DROPPED\_BY\_MODEL

.with\_label\_values(&[&MODEL\_SPECS[model.model\_idx()]])

.inc();

}

}

NUM\_BATCHES\_DROPPED.inc();

NUM\_BATCHES\_DROPPED\_BY\_MODEL

.with\_label\_values(&[&MODEL\_SPECS[model.model\_idx()]])

.inc();

}

BLOCKING\_REQUEST\_NUM.dec();

});

NUM\_BATCH\_PREDICTION.inc();

NUM\_BATCH\_PREDICTION\_BY\_MODEL

.with\_label\_values(&[&MODEL\_SPECS[self.model.model\_idx()]])

.inc();

// Note:

// self.cur\_batch\_size is swapped with batch\_size above

// Use the local variable batch\_size here

NUM\_PREDICTION\_BY\_MODEL

.with\_label\_values(&[&MODEL\_SPECS[self.model.model\_idx()]])

.inc\_by(batch\_size as u64);

}

#[inline(always)]

pub fn duration\_past(&self, millis: u64) -> bool {

self.queue\_reset\_ts.elapsed().as\_millis() as u64 >= millis

}

}