use anyhow::{anyhow, Result};

use arrayvec::ArrayVec;

use itertools::Itertools;

use log::{error, info};

use std::fmt::{Debug, Display};

use std::string::String;

use std::sync::Arc;

use std::time::Duration;

use tokio::process::Command;

use tokio::sync::mpsc::error::TryRecvError;

use tokio::sync::mpsc::{Receiver, Sender};

use tokio::sync::{mpsc, oneshot};

use tokio::time::{Instant, sleep};

use warp::Filter;

use crate::batch::BatchPredictor;

use crate::bootstrap::TensorInput;

use crate::{MAX\_NUM\_MODELS, MAX\_VERSIONS\_PER\_MODEL, META\_INFO, metrics, ModelFactory, PredictMessage, PredictResult, TensorReturnEnum, utils};

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

use crate::cores::validator::validatior::cli\_validator;

use crate::metrics::MPSC\_CHANNEL\_SIZE;

use serde\_json::{self, Value};

pub trait Model: Send + Sync + Display + Debug + 'static {

fn warmup(&self) -> Result<()>;

//TODO: refactor this to return vec<vec<TensorScores>>, i.e.

//we have the underlying runtime impl to split the response to each client.

//It will eliminate some inefficient memory copy in onnx\_model.rs as well as simplify code

fn do\_predict(

&self,

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

total\_len: u64,

) -> (Vec<TensorReturnEnum>, Vec<Vec<usize>>);

fn model\_idx(&self) -> usize;

fn version(&self) -> i64;

}

#[derive(Debug)]

pub struct PredictService<T: Model> {

tx: Sender<PredictMessage<T>>,

}

impl<T: Model> PredictService<T> {

pub async fn init(model\_factory: ModelFactory<T>) -> Self {

cli\_validator::validate\_ps\_model\_args();

let (tx, rx) = mpsc::channel(32\_000);

tokio::spawn(PredictService::tf\_queue\_manager(rx));

tokio::spawn(PredictService::model\_watcher\_latest(

model\_factory,

tx.clone(),

));

let metrics\_route = warp::path!("metrics").and\_then(metrics::metrics\_handler);

let metric\_server = warp::serve(metrics\_route).run(([0, 0, 0, 0], ARGS.prometheus\_port));

tokio::spawn(metric\_server);

PredictService { tx }

}

#[inline(always)]

pub async fn predict(

&self,

idx: usize,

version: Option<i64>,

val: Vec<TensorInput>,

ts: Instant,

) -> Result<PredictResult> {

let (tx, rx) = oneshot::channel();

if let Err(e) = self

.tx

.clone()

.send(PredictMessage::Predict(idx, version, val, tx, ts))

.await

{

error!("mpsc send error:{}", e);

Err(anyhow!(e))

} else {

MPSC\_CHANNEL\_SIZE.inc();

rx.await.map\_err(anyhow::Error::msg)

}

}

async fn load\_latest\_model\_from\_model\_dir(

model\_factory: ModelFactory<T>,

model\_config: &Value,

tx: Sender<PredictMessage<T>>,

idx: usize,

max\_version: String,

latest\_version: &mut String,

) {

match model\_factory(idx, max\_version.clone(), model\_config) {

Ok(tf\_model) => tx

.send(PredictMessage::UpsertModel(tf\_model))

.await

.map\_or\_else(

|e| error!("send UpsertModel error: {}", e),

|\_| \*latest\_version = max\_version,

),

Err(e) => {

error!("skip loading model due to failure: {:?}", e);

}

}

}

async fn scan\_load\_latest\_model\_from\_model\_dir(

model\_factory: ModelFactory<T>,

model\_config: &Value,

tx: Sender<PredictMessage<T>>,

model\_idx: usize,

cur\_version: &mut String,

) -> Result<()> {

let model\_dir = &ARGS.model\_dir[model\_idx];

let next\_version = utils::get\_config\_or\_else(model\_config, "version", || {

info!("no version found, hence use max version");

std::fs::read\_dir(model\_dir)

.map\_err(|e| format!("read dir error:{}", e))

.and\_then(|paths| {

paths

.into\_iter()

.flat\_map(|p| {

p.map\_err(|e| error!("dir entry error: {}", e))

.and\_then(|dir| {

dir.file\_name()

.into\_string()

.map\_err(|e| error!("osstring error: {:?}", e))

})

.ok()

})

.filter(|f| !f.to\_lowercase().contains(&META\_INFO.to\_lowercase()))

.max()

.ok\_or\_else(|| "no dir found hence no max".to\_owned())

})

.unwrap\_or\_else(|e| {

error!(

"can't get the max version hence return cur\_version, error is: {}",

e

);

cur\_version.to\_string()

})

});

//as long as next version doesn't match cur version maintained we reload

if next\_version.ne(cur\_version) {

info!("reload the version: {}->{}", cur\_version, next\_version);

PredictService::load\_latest\_model\_from\_model\_dir(

model\_factory,

model\_config,

tx,

model\_idx,

next\_version,

cur\_version,

)

.await;

}

Ok(())

}

async fn model\_watcher\_latest(model\_factory: ModelFactory<T>, tx: Sender<PredictMessage<T>>) {

async fn call\_external\_modelsync(cli: &str, cur\_versions: &Vec<String>) -> Result<()> {

let mut args = cli.split\_whitespace();

let mut cmd = Command::new(args.next().ok\_or(anyhow!("model sync cli empty"))?);

let extr\_args = MODEL\_SPECS

.iter()

.zip(cur\_versions)

.flat\_map(|(spec, version)| vec!["--model-spec", spec, "--cur-version", version])

.collect\_vec();

info!("run model sync: {} with extra args: {:?}", cli, extr\_args);

let output = cmd.args(args).args(extr\_args).output().await?;

info!("model sync stdout:{}", String::from\_utf8(output.stdout)?);

info!("model sync stderr:{}", String::from\_utf8(output.stderr)?);

if output.status.success() {

Ok(())

} else {

Err(anyhow!(

"model sync failed with status: {:?}!",

output.status

))

}

}

let meta\_dir = utils::get\_meta\_dir();

let meta\_file = format!("{}{}", meta\_dir, META\_INFO);

//initialize the latest version array

let mut cur\_versions = vec!["".to\_owned(); MODEL\_SPECS.len()];

loop {

info!("\*\*\*polling for models\*\*\*"); //nice deliminter

if let Some(ref cli) = ARGS.modelsync\_cli {

if let Err(e) = call\_external\_modelsync(cli, &cur\_versions).await {

error!("model sync cli running error:{}", e)

}

}

let config = utils::read\_config(&meta\_file).unwrap\_or\_else(|e| {

info!("config file {} not found due to: {}", meta\_file, e);

Value::Null

});

info!("config:{}", config);

for (idx, cur\_version) in cur\_versions.iter\_mut().enumerate() {

let model\_dir = &ARGS.model\_dir[idx];

PredictService::scan\_load\_latest\_model\_from\_model\_dir(

model\_factory,

&config[&MODEL\_SPECS[idx]],

tx.clone(),

idx,

cur\_version,

)

.await

.map\_or\_else(

|e| error!("scanned {}, error {:?}", model\_dir, e),

|\_| info!("scanned {}, latest\_version: {}", model\_dir, cur\_version),

);

}

sleep(Duration::from\_secs(ARGS.model\_check\_interval\_secs)).await;

}

}

async fn tf\_queue\_manager(mut rx: Receiver<PredictMessage<T>>) {

// Start receiving messages

info!("setting up queue manager");

let max\_batch\_size = ARGS

.max\_batch\_size

.iter()

.map(|b| b.parse().unwrap())

.collect::<Vec<usize>>();

let batch\_time\_out\_millis = ARGS

.batch\_time\_out\_millis

.iter()

.map(|b| b.parse().unwrap())

.collect::<Vec<u64>>();

let no\_msg\_wait\_millis = \*batch\_time\_out\_millis.iter().min().unwrap();

let mut all\_model\_predictors: ArrayVec::<ArrayVec<BatchPredictor<T>, MAX\_VERSIONS\_PER\_MODEL>, MAX\_NUM\_MODELS> =

(0 ..MAX\_NUM\_MODELS).map( |\_| ArrayVec::<BatchPredictor<T>, MAX\_VERSIONS\_PER\_MODEL>::new()).collect();

loop {

let msg = rx.try\_recv();

let no\_more\_msg = match msg {

Ok(PredictMessage::Predict(model\_spec\_at, version, val, resp, ts)) => {

if let Some(model\_predictors) = all\_model\_predictors.get\_mut(model\_spec\_at) {

if model\_predictors.is\_empty() {

resp.send(PredictResult::ModelNotReady(model\_spec\_at))

.unwrap\_or\_else(|e| error!("cannot send back model not ready error: {:?}", e));

}

else {

match version {

None => model\_predictors[0].push(val, resp, ts),

Some(the\_version) => match model\_predictors

.iter\_mut()

.find(|x| x.model.version() == the\_version)

{

None => resp

.send(PredictResult::ModelVersionNotFound(

model\_spec\_at,

the\_version,

))

.unwrap\_or\_else(|e| {

error!("cannot send back version error: {:?}", e)

}),

Some(predictor) => predictor.push(val, resp, ts),

},

}

}

} else {

resp.send(PredictResult::ModelNotFound(model\_spec\_at))

.unwrap\_or\_else(|e| error!("cannot send back model not found error: {:?}", e))

}

MPSC\_CHANNEL\_SIZE.dec();

false

}

Ok(PredictMessage::UpsertModel(tf\_model)) => {

let idx = tf\_model.model\_idx();

let predictor = BatchPredictor {

model: Arc::new(tf\_model),

input\_tensors: Vec::with\_capacity(max\_batch\_size[idx]),

callbacks: Vec::with\_capacity(max\_batch\_size[idx]),

cur\_batch\_size: 0,

max\_batch\_size: max\_batch\_size[idx],

batch\_time\_out\_millis: batch\_time\_out\_millis[idx],

//initialize to be current time

queue\_reset\_ts: Instant::now(),

queue\_earliest\_rq\_ts: Instant::now(),

};

assert!(idx < all\_model\_predictors.len());

metrics::NEW\_MODEL\_SNAPSHOT

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

.inc();

//we can do this since the vector is small

let predictors = &mut all\_model\_predictors[idx];

if predictors.len() == 0 {

info!("now we serve new model: {}", predictor.model);

}

else {

info!("now we serve updated model: {}", predictor.model);

}

if predictors.len() == ARGS.versions\_per\_model {

predictors.remove(predictors.len() - 1);

}

predictors.insert(0, predictor);

false

}

Err(TryRecvError::Empty) => true,

Err(TryRecvError::Disconnected) => true,

};

for predictor in all\_model\_predictors.iter\_mut().flatten() {

//if predictor batch queue not empty and times out or no more msg in the queue, flush

if (!predictor.input\_tensors.is\_empty() && (predictor.duration\_past(predictor.batch\_time\_out\_millis) || no\_more\_msg))

//if batch queue reaches limit, flush

|| predictor.cur\_batch\_size >= predictor.max\_batch\_size

{

predictor.batch\_predict();

}

}

if no\_more\_msg {

sleep(Duration::from\_millis(no\_msg\_wait\_millis)).await;

}

}

}

#[inline(always)]

pub fn get\_model\_index(model\_spec: &str) -> Option<usize> {

MODEL\_SPECS.iter().position(|m| m == model\_spec)

}

}