#[cfg(feature = "torch")]

pub mod torch {

use std::fmt;

use std::fmt::Display;

use std::string::String;

use crate::TensorReturnEnum;

use crate::SerializedInput;

use crate::bootstrap::TensorInput;

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

use crate::metrics;

use crate::metrics::{

INFERENCE\_FAILED\_REQUESTS\_BY\_MODEL, NUM\_REQUESTS\_FAILED, NUM\_REQUESTS\_FAILED\_BY\_MODEL,

};

use crate::predict\_service::Model;

use anyhow::Result;

use dr\_transform::converter::BatchPredictionRequestToTorchTensorConverter;

use dr\_transform::converter::Converter;

use serde\_json::Value;

use tch::Tensor;

use tch::{kind, CModule, IValue};

#[derive(Debug)]

pub struct TorchModel {

pub model\_idx: usize,

pub version: i64,

pub module: CModule,

pub export\_dir: String,

// FIXME: make this Box<Option<..>> so input converter can be optional.

// Also consider adding output\_converter.

pub input\_converter: Box<dyn Converter>,

}

impl Display for TorchModel {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(

f,

"idx: {}, torch model\_name:{}, version:{}",

self.model\_idx, MODEL\_SPECS[self.model\_idx], self.version

)

}

}

impl TorchModel {

pub fn new(idx: usize, version: String, \_model\_config: &Value) -> Result<TorchModel> {

let export\_dir = format!("{}/{}/model.pt", ARGS.model\_dir[idx], version);

let model = CModule::load(&export\_dir).unwrap();

let torch\_model = TorchModel {

model\_idx: idx,

version: Args::version\_str\_to\_epoch(&version)?,

module: model,

export\_dir,

//TODO: move converter lookup in a registry.

input\_converter: Box::new(BatchPredictionRequestToTorchTensorConverter::new(

&ARGS.model\_dir[idx].as\_str(),

version.as\_str(),

vec![],

Some(&metrics::register\_dynamic\_metrics),

)),

};

torch\_model.warmup()?;

Ok(torch\_model)

}

#[inline(always)]

pub fn decode\_to\_inputs(bytes: SerializedInput) -> Vec<Tensor> {

//FIXME: for now we generate 4 random tensors as inputs to unblock end to end testing

//when Shajan's decoder is ready we will swap

let row = bytes.len() as i64;

let t1 = Tensor::randn(&[row, 5293], kind::FLOAT\_CPU); //continuous

let t2 = Tensor::randint(10, &[row, 149], kind::INT64\_CPU); //binary

let t3 = Tensor::randint(10, &[row, 320], kind::INT64\_CPU); //discrete

let t4 = Tensor::randn(&[row, 200], kind::FLOAT\_CPU); //user\_embedding

let t5 = Tensor::randn(&[row, 200], kind::FLOAT\_CPU); //user\_eng\_embedding

let t6 = Tensor::randn(&[row, 200], kind::FLOAT\_CPU); //author\_embedding

vec![t1, t2, t3, t4, t5, t6]

}

#[inline(always)]

pub fn output\_to\_vec(res: IValue, dst: &mut Vec<f32>) {

match res {

IValue::Tensor(tensor) => TorchModel::tensors\_to\_vec(&[tensor], dst),

IValue::Tuple(ivalues) => {

TorchModel::tensors\_to\_vec(&TorchModel::ivalues\_to\_tensors(ivalues), dst)

}

\_ => panic!("we only support output as a single tensor or a vec of tensors"),

}

}

#[inline(always)]

pub fn tensor\_flatten\_size(t: &Tensor) -> usize {

t.size().into\_iter().fold(1, |acc, x| acc \* x) as usize

}

#[inline(always)]

pub fn tensor\_to\_vec<T: kind::Element>(res: &Tensor) -> Vec<T> {

let size = TorchModel::tensor\_flatten\_size(res);

let mut res\_f32: Vec<T> = Vec::with\_capacity(size);

unsafe {

res\_f32.set\_len(size);

}

res.copy\_data(res\_f32.as\_mut\_slice(), size);

// println!("Copied tensor:{}, {:?}", res\_f32.len(), res\_f32);

res\_f32

}

#[inline(always)]

pub fn tensors\_to\_vec(tensors: &[Tensor], dst: &mut Vec<f32>) {

let mut offset = dst.len();

tensors.iter().for\_each(|t| {

let size = TorchModel::tensor\_flatten\_size(t);

let next\_size = offset + size;

unsafe {

dst.set\_len(next\_size);

}

t.copy\_data(&mut dst[offset..], size);

offset = next\_size;

});

}

pub fn ivalues\_to\_tensors(ivalues: Vec<IValue>) -> Vec<Tensor> {

ivalues

.into\_iter()

.map(|t| {

if let IValue::Tensor(vanilla\_t) = t {

vanilla\_t

} else {

panic!("not a tensor")

}

})

.collect::<Vec<Tensor>>()

}

}

impl Model for TorchModel {

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

Ok(())

}

//TODO: torch runtime needs some refactor to make it a generic interface

#[inline(always)]

fn do\_predict(

&self,

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

total\_len: u64,

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

let mut buf: Vec<f32> = Vec::with\_capacity(10\_000);

let mut batch\_ends = vec![0usize; input\_tensors.len()];

for (i, batch\_bytes\_in\_request) in input\_tensors.into\_iter().enumerate() {

for \_ in batch\_bytes\_in\_request.into\_iter() {

//FIXME: for now use some hack

let model\_input = TorchModel::decode\_to\_inputs(vec![0u8; 30]); //self.input\_converter.convert(bytes);

let input\_batch\_tensors = model\_input

.into\_iter()

.map(|t| IValue::Tensor(t))

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

// match self.module.forward\_is(&input\_batch\_tensors) {

match self.module.method\_is("forward\_serve", &input\_batch\_tensors) {

Ok(res) => TorchModel::output\_to\_vec(res, &mut buf),

Err(e) => {

NUM\_REQUESTS\_FAILED.inc\_by(total\_len);

NUM\_REQUESTS\_FAILED\_BY\_MODEL

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

.inc\_by(total\_len);

INFERENCE\_FAILED\_REQUESTS\_BY\_MODEL

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

.inc\_by(total\_len);

panic!("{model}: {e:?}", model = MODEL\_SPECS[self.model\_idx], e = e);

}

}

}

batch\_ends[i] = buf.len();

}

(

vec![TensorReturnEnum::FloatTensorReturn(Box::new(buf))],

vec![batch\_ends],

)

}

#[inline(always)]

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

self.model\_idx

}

#[inline(always)]

fn version(&self) -> i64 {

self.version

}

}

}