#[cfg(feature = "tf")]

pub mod tf {

use arrayvec::ArrayVec;

use itertools::Itertools;

use log::{debug, error, info, warn};

use prost::Message;

use std::fmt;

use std::fmt::Display;

use std::string::String;

use tensorflow::io::{RecordReader, RecordReadError};

use tensorflow::Operation;

use tensorflow::SavedModelBundle;

use tensorflow::SessionOptions;

use tensorflow::SessionRunArgs;

use tensorflow::Tensor;

use tensorflow::{DataType, FetchToken, Graph, TensorInfo, TensorType};

use std::thread::sleep;

use std::time::Duration;

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

use crate::tf\_proto::tensorflow\_serving::prediction\_log::LogType;

use crate::tf\_proto::tensorflow\_serving::{PredictionLog, PredictLog};

use crate::tf\_proto::ConfigProto;

use anyhow::{Context, Result};

use serde\_json::Value;

use crate::TensorReturnEnum;

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

use crate::metrics::{

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

};

use crate::predict\_service::Model;

use crate::{MAX\_NUM\_INPUTS, utils};

#[derive(Debug)]

pub enum TFTensorEnum {

String(Tensor<String>),

Int(Tensor<i32>),

Int64(Tensor<i64>),

Float(Tensor<f32>),

Double(Tensor<f64>),

Boolean(Tensor<bool>),

}

#[derive(Debug)]

pub struct TFModel {

pub model\_idx: usize,

pub bundle: SavedModelBundle,

pub input\_names: ArrayVec<String, MAX\_NUM\_INPUTS>,

pub input\_info: Vec<TensorInfo>,

pub input\_ops: Vec<Operation>,

pub output\_names: Vec<String>,

pub output\_info: Vec<TensorInfo>,

pub output\_ops: Vec<Operation>,

pub export\_dir: String,

pub version: i64,

pub inter\_op: i32,

pub intra\_op: i32,

}

impl Display for TFModel {

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

write!(

f,

"idx: {}, tensorflow model\_name:{}, export\_dir:{}, version:{}, inter:{}, intra:{}",

self.model\_idx,

MODEL\_SPECS[self.model\_idx],

self.export\_dir,

self.version,

self.inter\_op,

self.intra\_op

)

}

}

impl TFModel {

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

// Create input variables for our addition

let config = ConfigProto {

intra\_op\_parallelism\_threads: utils::get\_config\_or(

model\_config,

"intra\_op\_parallelism",

&ARGS.intra\_op\_parallelism[idx],

)

.parse()?,

inter\_op\_parallelism\_threads: utils::get\_config\_or(

model\_config,

"inter\_op\_parallelism",

&ARGS.inter\_op\_parallelism[idx],

)

.parse()?,

..Default::default()

};

let mut buf = Vec::new();

buf.reserve(config.encoded\_len());

config.encode(&mut buf).unwrap();

let mut opts = SessionOptions::new();

opts.set\_config(&buf)?;

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

let mut graph = Graph::new();

let bundle = SavedModelBundle::load(&opts, ["serve"], &mut graph, &export\_dir)

.context("error load model")?;

let signature = bundle

.meta\_graph\_def()

.get\_signature(&ARGS.serving\_sig[idx])

.context("error finding signature")?;

let input\_names = INPUTS[idx]

.get\_or\_init(|| {

let input\_spec = signature

.inputs()

.iter()

.map(|p| p.0.clone())

.collect::<ArrayVec<String, MAX\_NUM\_INPUTS>>();

info!(

"input not set from cli, now we set from model metadata:{:?}",

input\_spec

);

input\_spec

})

.clone();

let input\_info = input\_names

.iter()

.map(|i| {

signature

.get\_input(i)

.context("error finding input op info")

.unwrap()

.clone()

})

.collect\_vec();

let input\_ops = input\_info

.iter()

.map(|i| {

graph

.operation\_by\_name\_required(&i.name().name)

.context("error finding input op")

.unwrap()

})

.collect\_vec();

info!("Model Input size: {}", input\_info.len());

let output\_names = OUTPUTS[idx].to\_vec().clone();

let output\_info = output\_names

.iter()

.map(|o| {

signature

.get\_output(o)

.context("error finding output op info")

.unwrap()

.clone()

})

.collect\_vec();

let output\_ops = output\_info

.iter()

.map(|o| {

graph

.operation\_by\_name\_required(&o.name().name)

.context("error finding output op")

.unwrap()

})

.collect\_vec();

let tf\_model = TFModel {

model\_idx: idx,

bundle,

input\_names,

input\_info,

input\_ops,

output\_names,

output\_info,

output\_ops,

export\_dir,

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

inter\_op: config.inter\_op\_parallelism\_threads,

intra\_op: config.intra\_op\_parallelism\_threads,

};

tf\_model.warmup()?;

Ok(tf\_model)

}

#[inline(always)]

fn get\_tftensor\_dimensions<T>(

t: &[T],

input\_size: u64,

batch\_size: u64,

input\_dims: Option<Vec<i64>>,

) -> Vec<u64> {

// if input size is 1, we just specify a single dimension to outgoing tensor matching the

// size of the input tensor. This is for backwards compatiblity with existing Navi clients

// which specify input as a single string tensor (like tfexample) and use batching support.

let mut dims = vec![];

if input\_size > 1 {

if batch\_size == 1 && input\_dims.is\_some() {

// client side batching is enabled?

input\_dims

.unwrap()

.iter()

.for\_each(|axis| dims.push(\*axis as u64));

} else {

dims.push(batch\_size);

dims.push(t.len() as u64 / batch\_size);

}

} else {

dims.push(t.len() as u64);

}

dims

}

fn convert\_to\_tftensor\_enum(

input: TensorInput,

input\_size: u64,

batch\_size: u64,

) -> TFTensorEnum {

match input.tensor\_data {

TensorInputEnum::String(t) => {

let strings = t

.into\_iter()

.map(|x| unsafe { String::from\_utf8\_unchecked(x) })

.collect\_vec();

TFTensorEnum::String(

Tensor::new(&TFModel::get\_tftensor\_dimensions(

strings.as\_slice(),

input\_size,

batch\_size,

input.dims,

))

.with\_values(strings.as\_slice())

.unwrap(),

)

}

TensorInputEnum::Int(t) => TFTensorEnum::Int(

Tensor::new(&TFModel::get\_tftensor\_dimensions(

t.as\_slice(),

input\_size,

batch\_size,

input.dims,

))

.with\_values(t.as\_slice())

.unwrap(),

),

TensorInputEnum::Int64(t) => TFTensorEnum::Int64(

Tensor::new(&TFModel::get\_tftensor\_dimensions(

t.as\_slice(),

input\_size,

batch\_size,

input.dims,

))

.with\_values(t.as\_slice())

.unwrap(),

),

TensorInputEnum::Float(t) => TFTensorEnum::Float(

Tensor::new(&TFModel::get\_tftensor\_dimensions(

t.as\_slice(),

input\_size,

batch\_size,

input.dims,

))

.with\_values(t.as\_slice())

.unwrap(),

),

TensorInputEnum::Double(t) => TFTensorEnum::Double(

Tensor::new(&TFModel::get\_tftensor\_dimensions(

t.as\_slice(),

input\_size,

batch\_size,

input.dims,

))

.with\_values(t.as\_slice())

.unwrap(),

),

TensorInputEnum::Boolean(t) => TFTensorEnum::Boolean(

Tensor::new(&TFModel::get\_tftensor\_dimensions(

t.as\_slice(),

input\_size,

batch\_size,

input.dims,

))

.with\_values(t.as\_slice())

.unwrap(),

),

}

}

fn fetch\_output<T: TensorType>(

args: &mut SessionRunArgs,

token\_output: &FetchToken,

batch\_size: u64,

output\_size: u64,

) -> (Tensor<T>, u64) {

let tensor\_output = args.fetch::<T>(\*token\_output).expect("fetch output failed");

let mut tensor\_width = tensor\_output.dims()[1];

if batch\_size == 1 && output\_size > 1 {

tensor\_width = tensor\_output.dims().iter().fold(1, |mut total, &val| {

total \*= val;

total

});

}

(tensor\_output, tensor\_width)

}

}

impl Model for TFModel {

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

// warm up

let warmup\_file = format!(

"{}/assets.extra/tf\_serving\_warmup\_requests",

self.export\_dir

);

if std::path::Path::new(&warmup\_file).exists() {

use std::io::Cursor;

info!(

"found warmup assets in {}, now perform warming up",

warmup\_file

);

let f = std::fs::File::open(warmup\_file).context("cannot open warmup file")?;

// let mut buf = Vec::new();

let read = std::io::BufReader::new(f);

let mut reader = RecordReader::new(read);

let mut warmup\_cnt = 0;

loop {

let next = reader.read\_next\_owned();

match next {

Ok(res) => match res {

Some(vec) => {

// info!("read one tfRecord");

match PredictionLog::decode(&mut Cursor::new(vec))

.context("can't parse PredictonLog")?

{

PredictionLog {

log\_metadata: \_,

log\_type:

Some(LogType::PredictLog(PredictLog {

request: Some(mut req),

response: \_,

})),

} => {

if warmup\_cnt == ARGS.max\_warmup\_records {

//warm up to max\_warmup\_records records

warn!(

"reached max warmup {} records, exit warmup for {}",

ARGS.max\_warmup\_records,

MODEL\_SPECS[self.model\_idx]

);

break;

}

self.do\_predict(

vec![req.take\_input\_vals(&self.input\_names)],

1,

);

sleep(Duration::from\_millis(100));

warmup\_cnt += 1;

}

\_ => error!("some wrong record in warming up file"),

}

}

None => {

info!("end of warmup file, warmed up with records: {}", warmup\_cnt);

break;

}

},

Err(RecordReadError::CorruptFile)

| Err(RecordReadError::IoError { .. }) => {

error!("read tfrecord error for warmup files, skip");

}

\_ => {}

}

}

}

Ok(())

}

#[inline(always)]

fn do\_predict(

&self,

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

batch\_size: u64,

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

// let mut batch\_ends = input\_tensors.iter().map(|t| t.len()).collect::<Vec<usize>>();

let output\_size = self.output\_names.len() as u64;

let input\_size = self.input\_names.len() as u64;

debug!(

"Request for Tensorflow with batch size: {} and input\_size: {}",

batch\_size, input\_size

);

// build a set of input TF tensors

let batch\_end = (1usize..=input\_tensors.len() as usize)

.into\_iter()

.collect\_vec();

let mut batch\_ends = vec![batch\_end; output\_size as usize];

let batched\_tensors = TensorInputEnum::merge\_batch(input\_tensors)

.into\_iter()

.enumerate()

.map(|(\_, i)| TFModel::convert\_to\_tftensor\_enum(i, input\_size, batch\_size))

.collect\_vec();

let mut args = SessionRunArgs::new();

for (index, tf\_tensor) in batched\_tensors.iter().enumerate() {

match tf\_tensor {

TFTensorEnum::String(inner) => args.add\_feed(&self.input\_ops[index], 0, inner),

TFTensorEnum::Int(inner) => args.add\_feed(&self.input\_ops[index], 0, inner),

TFTensorEnum::Int64(inner) => args.add\_feed(&self.input\_ops[index], 0, inner),

TFTensorEnum::Float(inner) => args.add\_feed(&self.input\_ops[index], 0, inner),

TFTensorEnum::Double(inner) => args.add\_feed(&self.input\_ops[index], 0, inner),

TFTensorEnum::Boolean(inner) => args.add\_feed(&self.input\_ops[index], 0, inner),

}

}

// For output ops, we receive the same op object by name. Actual tensor tokens are available at different offsets.

// Since indices are ordered, its important to specify output flag to Navi in the same order.

let token\_outputs = self

.output\_ops

.iter()

.enumerate()

.map(|(idx, op)| args.request\_fetch(op, idx as i32))

.collect\_vec();

match self.bundle.session.run(&mut args) {

Ok(\_) => (),

Err(e) => {

NUM\_REQUESTS\_FAILED.inc\_by(batch\_size);

NUM\_REQUESTS\_FAILED\_BY\_MODEL

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

.inc\_by(batch\_size);

INFERENCE\_FAILED\_REQUESTS\_BY\_MODEL

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

.inc\_by(batch\_size);

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

}

}

let mut predict\_return = vec![];

// Check the output.

for (index, token\_output) in token\_outputs.iter().enumerate() {

// same ops, with type info at different offsets.

let (res, width) = match self.output\_ops[index].output\_type(index) {

DataType::Float => {

let (tensor\_output, tensor\_width) =

TFModel::fetch\_output(&mut args, token\_output, batch\_size, output\_size);

(

TensorReturnEnum::FloatTensorReturn(Box::new(tensor\_output)),

tensor\_width,

)

}

DataType::Int64 => {

let (tensor\_output, tensor\_width) =

TFModel::fetch\_output(&mut args, token\_output, batch\_size, output\_size);

(

TensorReturnEnum::Int64TensorReturn(Box::new(tensor\_output)),

tensor\_width,

)

}

DataType::Int32 => {

let (tensor\_output, tensor\_width) =

TFModel::fetch\_output(&mut args, token\_output, batch\_size, output\_size);

(

TensorReturnEnum::Int32TensorReturn(Box::new(tensor\_output)),

tensor\_width,

)

}

DataType::String => {

let (tensor\_output, tensor\_width) =

TFModel::fetch\_output(&mut args, token\_output, batch\_size, output\_size);

(

TensorReturnEnum::StringTensorReturn(Box::new(tensor\_output)),

tensor\_width,

)

}

\_ => panic!("Unsupported return type!"),

};

let width = width as usize;

for b in batch\_ends[index].iter\_mut() {

\*b \*= width;

}

predict\_return.push(res)

}

//TODO: remove in the future

//TODO: support actual mtl model outputs

(predict\_return, batch\_ends)

}

#[inline(always)]

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

self.model\_idx

}

#[inline(always)]

fn version(&self) -> i64 {

self.version

}

}

}