#[macro\_use]

extern crate lazy\_static;

extern crate core;

use serde\_json::Value;

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

use tokio::time::Instant;

use std::ops::Deref;

use itertools::Itertools;

use crate::bootstrap::TensorInput;

use crate::predict\_service::Model;

use crate::tf\_proto::{DataType, TensorProto};

pub mod batch;

pub mod bootstrap;

pub mod cli\_args;

pub mod metrics;

pub mod onnx\_model;

pub mod predict\_service;

pub mod tf\_model;

pub mod torch\_model;

pub mod cores {

pub mod validator;

}

pub mod tf\_proto {

tonic::include\_proto!("tensorflow");

pub mod tensorflow\_serving {

tonic::include\_proto!("tensorflow.serving");

}

}

pub mod kf\_serving {

tonic::include\_proto!("inference");

}

#[cfg(test)]

mod tests {

use crate::cli\_args::Args;

#[test]

fn test\_version\_string\_to\_epoch() {

assert\_eq!(

Args::version\_str\_to\_epoch("2022-12-20T10:18:53.000Z").unwrap\_or(-1),

1671531533000

);

assert\_eq!(Args::version\_str\_to\_epoch("1203444").unwrap\_or(-1), 1203444);

}

}

mod utils {

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

use anyhow::Result;

use log::info;

use serde\_json::Value;

pub fn read\_config(meta\_file: &String) -> Result<Value> {

let json = std::fs::read\_to\_string(meta\_file)?;

let v: Value = serde\_json::from\_str(&json)?;

Ok(v)

}

pub fn get\_config\_or\_else<F>(model\_config: &Value, key: &str, default: F) -> String

where

F: FnOnce() -> String,

{

match model\_config[key] {

Value::String(ref v) => {

info!("from model\_config: {}={}", key, v);

v.to\_string()

}

Value::Number(ref num) => {

info!(

"from model\_config: {}={} (turn number into a string)",

key, num

);

num.to\_string()

}

\_ => {

let d = default();

info!("from default: {}={}", key, d);

d

}

}

}

pub fn get\_config\_or(model\_config: &Value, key: &str, default: &str) -> String {

get\_config\_or\_else(model\_config, key, || default.to\_string())

}

pub fn get\_meta\_dir() -> &'static str {

ARGS.meta\_json\_dir

.as\_ref()

.map(|s| s.as\_str())

.unwrap\_or\_else(|| {

let model\_dir = &ARGS.model\_dir[0];

let meta\_dir = &model\_dir[0..model\_dir.rfind(&MODEL\_SPECS[0]).unwrap()];

info!(

"no meta\_json\_dir specified, hence derive from first model dir:{}->{}",

model\_dir, meta\_dir

);

meta\_dir

})

}

}

pub type SerializedInput = Vec<u8>;

pub const VERSION: &str = env!("CARGO\_PKG\_VERSION");

pub const NAME: &str = env!("CARGO\_PKG\_NAME");

pub type ModelFactory<T> = fn(usize, String, &Value) -> anyhow::Result<T>;

pub const MAX\_NUM\_MODELS: usize = 16;

pub const MAX\_NUM\_OUTPUTS: usize = 30;

pub const MAX\_NUM\_INPUTS: usize = 120;

pub const META\_INFO: &str = "META.json";

//use a heap allocated generic type here so that both

//Tensorflow & Pytorch implementation can return their Tensor wrapped in a Box

//without an extra memcopy to Vec

pub type TensorReturn<T> = Box<dyn Deref<Target = [T]>>;

//returned tensor may be int64 i.e., a list of relevant ad ids

pub enum TensorReturnEnum {

FloatTensorReturn(TensorReturn<f32>),

StringTensorReturn(TensorReturn<String>),

Int64TensorReturn(TensorReturn<i64>),

Int32TensorReturn(TensorReturn<i32>),

}

impl TensorReturnEnum {

#[inline(always)]

pub fn slice(&self, start: usize, end: usize) -> TensorScores {

match self {

TensorReturnEnum::FloatTensorReturn(f32\_return) => {

TensorScores::Float32TensorScores(f32\_return[start..end].to\_vec())

}

TensorReturnEnum::Int64TensorReturn(i64\_return) => {

TensorScores::Int64TensorScores(i64\_return[start..end].to\_vec())

}

TensorReturnEnum::Int32TensorReturn(i32\_return) => {

TensorScores::Int32TensorScores(i32\_return[start..end].to\_vec())

}

TensorReturnEnum::StringTensorReturn(str\_return) => {

TensorScores::StringTensorScores(str\_return[start..end].to\_vec())

}

}

}

}

#[derive(Debug)]

pub enum PredictResult {

Ok(Vec<TensorScores>, i64),

DropDueToOverload,

ModelNotFound(usize),

ModelNotReady(usize),

ModelVersionNotFound(usize, i64),

}

#[derive(Debug)]

pub enum TensorScores {

Float32TensorScores(Vec<f32>),

Int64TensorScores(Vec<i64>),

Int32TensorScores(Vec<i32>),

StringTensorScores(Vec<String>),

}

impl TensorScores {

pub fn create\_tensor\_proto(self) -> TensorProto {

match self {

TensorScores::Float32TensorScores(f32\_tensor) => TensorProto {

dtype: DataType::DtFloat as i32,

float\_val: f32\_tensor,

..Default::default()

},

TensorScores::Int64TensorScores(i64\_tensor) => TensorProto {

dtype: DataType::DtInt64 as i32,

int64\_val: i64\_tensor,

..Default::default()

},

TensorScores::Int32TensorScores(i32\_tensor) => TensorProto {

dtype: DataType::DtInt32 as i32,

int\_val: i32\_tensor,

..Default::default()

},

TensorScores::StringTensorScores(str\_tensor) => TensorProto {

dtype: DataType::DtString as i32,

string\_val: str\_tensor.into\_iter().map(|s| s.into\_bytes()).collect\_vec(),

..Default::default()

},

}

}

pub fn len(&self) -> usize {

match &self {

TensorScores::Float32TensorScores(t) => t.len(),

TensorScores::Int64TensorScores(t) => t.len(),

TensorScores::Int32TensorScores(t) => t.len(),

TensorScores::StringTensorScores(t) => t.len(),

}

}

}

#[derive(Debug)]

pub enum PredictMessage<T: Model> {

Predict(

usize,

Option<i64>,

Vec<TensorInput>,

Sender<PredictResult>,

Instant,

),

UpsertModel(T),

/\*

#[allow(dead\_code)]

DeleteModel(usize),

\*/

}

#[derive(Debug)]

pub struct Callback(Sender<PredictResult>, usize);

pub const MAX\_VERSIONS\_PER\_MODEL: usize = 2;