#[cfg(feature = "onnx")]

pub mod onnx {

use crate::TensorReturnEnum;

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

use crate::cli\_args::{

Args, ARGS, INPUTS, MODEL\_SPECS, OUTPUTS,

};

use crate::metrics::{self, CONVERTER\_TIME\_COLLECTOR};

use crate::predict\_service::Model;

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

use anyhow::Result;

use arrayvec::ArrayVec;

use dr\_transform::converter::{BatchPredictionRequestToTorchTensorConverter, Converter};

use itertools::Itertools;

use log::{debug, info};

use dr\_transform::ort::environment::Environment;

use dr\_transform::ort::session::Session;

use dr\_transform::ort::tensor::InputTensor;

use dr\_transform::ort::{ExecutionProvider, GraphOptimizationLevel, SessionBuilder};

use dr\_transform::ort::LoggingLevel;

use serde\_json::Value;

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

use std::sync::Arc;

use std::{fmt, fs};

use tokio::time::Instant;

lazy\_static! {

pub static ref ENVIRONMENT: Arc<Environment> = Arc::new(

Environment::builder()

.with\_name("onnx home")

.with\_log\_level(LoggingLevel::Error)

.with\_global\_thread\_pool(ARGS.onnx\_global\_thread\_pool\_options.clone())

.build()

.unwrap()

);

}

#[derive(Debug)]

pub struct OnnxModel {

pub session: Session,

pub model\_idx: usize,

pub version: i64,

pub export\_dir: String,

pub output\_filters: ArrayVec<usize, MAX\_NUM\_OUTPUTS>,

pub input\_converter: Box<dyn Converter>,

}

impl Display for OnnxModel {

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

write!(

f,

"idx: {}, onnx model\_name:{}, version:{}, output\_filters:{:?}, converter:{:}",

self.model\_idx,

MODEL\_SPECS[self.model\_idx],

self.version,

self.output\_filters,

self.input\_converter

)

}

}

impl Drop for OnnxModel {

fn drop(&mut self) {

if ARGS.profiling != None {

self.session.end\_profiling().map\_or\_else(

|e| {

info!("end profiling with some error:{:?}", e);

},

|f| {

info!("profiling ended with file:{}", f);

},

);

}

}

}

impl OnnxModel {

fn get\_output\_filters(session: &Session, idx: usize) -> ArrayVec<usize, MAX\_NUM\_OUTPUTS> {

OUTPUTS[idx]

.iter()

.map(|output| session.outputs.iter().position(|o| o.name == \*output))

.flatten()

.collect::<ArrayVec<usize, MAX\_NUM\_OUTPUTS>>()

}

#[cfg(target\_os = "linux")]

fn ep\_choices() -> Vec<ExecutionProvider> {

match ARGS.onnx\_gpu\_ep.as\_ref().map(|e| e.as\_str()) {

Some("onednn") => vec![Self::ep\_with\_options(ExecutionProvider::onednn())],

Some("tensorrt") => vec![Self::ep\_with\_options(ExecutionProvider::tensorrt())],

Some("cuda") => vec![Self::ep\_with\_options(ExecutionProvider::cuda())],

\_ => vec![Self::ep\_with\_options(ExecutionProvider::cpu())],

}

}

fn ep\_with\_options(mut ep: ExecutionProvider) -> ExecutionProvider {

for (ref k, ref v) in ARGS.onnx\_ep\_options.clone() {

ep = ep.with(k, v);

info!("setting option:{} -> {} and now ep is:{:?}", k, v, ep);

}

ep

}

#[cfg(target\_os = "macos")]

fn ep\_choices() -> Vec<ExecutionProvider> {

vec![Self::ep\_with\_options(ExecutionProvider::cpu())]

}

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

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

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

let mut builder = SessionBuilder::new(&ENVIRONMENT)?

.with\_optimization\_level(GraphOptimizationLevel::Level3)?

.with\_parallel\_execution(ARGS.onnx\_use\_parallel\_mode == "true")?;

if ARGS.onnx\_global\_thread\_pool\_options.is\_empty() {

builder = builder

.with\_inter\_threads(

utils::get\_config\_or(

model\_config,

"inter\_op\_parallelism",

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

)

.parse()?,

)?

.with\_intra\_threads(

utils::get\_config\_or(

model\_config,

"intra\_op\_parallelism",

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

)

.parse()?,

)?;

}

else {

builder = builder.with\_disable\_per\_session\_threads()?;

}

builder = builder

.with\_memory\_pattern(ARGS.onnx\_use\_memory\_pattern == "true")?

.with\_execution\_providers(&OnnxModel::ep\_choices())?;

match &ARGS.profiling {

Some(p) => {

debug!("Enable profiling, writing to {}", \*p);

builder = builder.with\_profiling(p)?

}

\_ => {}

}

let session = builder.with\_model\_from\_file(&export\_dir)?;

info!(

"inputs: {:?}, outputs: {:?}",

session.inputs.iter().format(","),

session.outputs.iter().format(",")

);

fs::read\_to\_string(&meta\_info)

.ok()

.map(|info| info!("meta info:{}", info));

let output\_filters = OnnxModel::get\_output\_filters(&session, idx);

let mut reporting\_feature\_ids: Vec<(i64, &str)> = vec![];

let input\_spec\_cell = &INPUTS[idx];

if input\_spec\_cell.get().is\_none() {

let input\_spec = session

.inputs

.iter()

.map(|input| input.name.clone())

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

input\_spec\_cell.set(input\_spec.clone()).map\_or\_else(

|\_| info!("unable to set the input\_spec for model {}", idx),

|\_| info!("auto detect and set the inputs: {:?}", input\_spec),

);

}

ARGS.onnx\_report\_discrete\_feature\_ids

.iter()

.for\_each(|ids| {

ids.split(",")

.filter(|s| !s.is\_empty())

.map(|s| s.parse::<i64>().unwrap())

.for\_each(|id| reporting\_feature\_ids.push((id, "discrete")))

});

ARGS.onnx\_report\_continuous\_feature\_ids

.iter()

.for\_each(|ids| {

ids.split(",")

.filter(|s| !s.is\_empty())

.map(|s| s.parse::<i64>().unwrap())

.for\_each(|id| reporting\_feature\_ids.push((id, "continuous")))

});

let onnx\_model = OnnxModel {

session,

model\_idx: idx,

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

export\_dir,

output\_filters,

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

&ARGS.model\_dir[idx],

&version,

reporting\_feature\_ids,

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

)?),

};

onnx\_model.warmup()?;

Ok(onnx\_model)

}

}

///Currently we only assume the input as just one string tensor.

///The string tensor will be be converted to the actual raw tensors.

/// The converter we are using is very specific to home.

/// It reads a BatchDataRecord thrift and decode it to a batch of raw input tensors.

/// Navi will then do server side batching and feed it to ONNX runtime

impl Model for OnnxModel {

//TODO: implement a generic online warmup for all runtimes

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

Ok(())

}

#[inline(always)]

fn do\_predict(

&self,

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

\_: u64,

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

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

let (inputs, batch\_ends): (Vec<Vec<InputTensor>>, Vec<Vec<usize>>) = batched\_tensors

.into\_iter()

.map(|batched\_tensor| {

match batched\_tensor.tensor\_data {

TensorInputEnum::String(t) if ARGS.onnx\_use\_converter.is\_some() => {

let start = Instant::now();

let (inputs, batch\_ends) = self.input\_converter.convert(t);

// info!("batch\_ends:{:?}", batch\_ends);

CONVERTER\_TIME\_COLLECTOR

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

.observe(

start.elapsed().as\_micros() as f64

/ (\*batch\_ends.last().unwrap() as f64),

);

(inputs, batch\_ends)

}

\_ => unimplemented!(),

}

})

.unzip();

//invariant we only support one input as string. will relax later

assert\_eq!(inputs.len(), 1);

let output\_tensors = self

.session

.run(inputs.into\_iter().flatten().collect::<Vec<\_>>())

.unwrap();

self.output\_filters

.iter()

.map(|&idx| {

let mut size = 1usize;

let output = output\_tensors[idx].try\_extract::<f32>().unwrap();

for &dim in self.session.outputs[idx].dimensions.iter().flatten() {

size \*= dim as usize;

}

let tensor\_ends = batch\_ends[0]

.iter()

.map(|&batch| batch \* size)

.collect::<Vec<\_>>();

(

//only works for batch major

//TODO: to\_vec() obviously wasteful, especially for large batches(GPU) . Will refactor to

//break up output and return Vec<Vec<TensorScore>> here

TensorReturnEnum::FloatTensorReturn(Box::new(output.view().as\_slice().unwrap().to\_vec(),

)),

tensor\_ends,

)

})

.unzip()

}

#[inline(always)]

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

self.model\_idx

}

#[inline(always)]

fn version(&self) -> i64 {

self.version

}

}

}