Serving BERT in production

PyTorch

HUGGING FACE

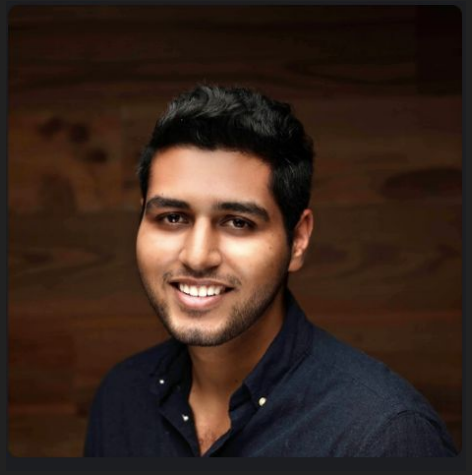
https://bit.ly/pytorch-workshop-2021

Image Ref

Agenda

1. Machine Learning at Walmart
2. Review some DL concepts + Hands-on
3. Optimizing the model for production + Hands-on
4. Break
5. Deploying the model + Hands-on
6. Lessons Learned
7. Q/A

About Us



Adway Dhillon

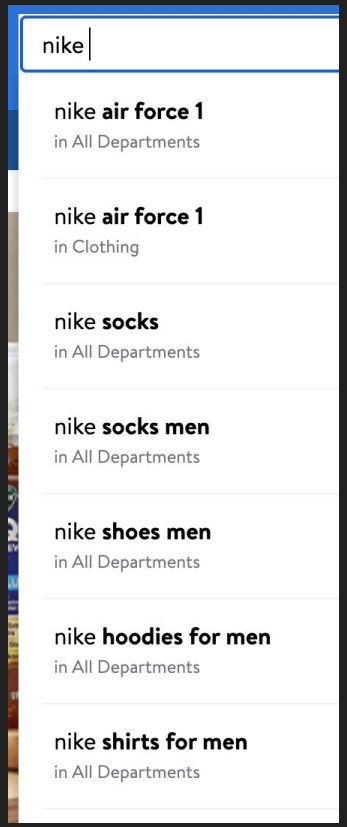
Nidhin Pattaiyil

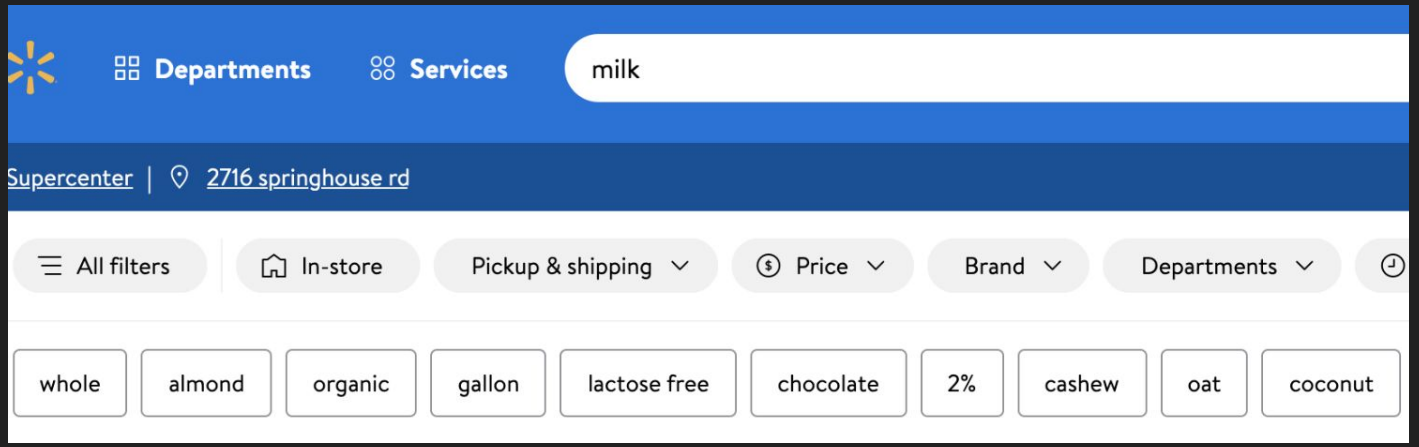
Walmart: ML Engineers on the Search team

Machine Learning at Walmart Search

- SpellCheck
- Typeahead
- Related Search / Guided Navigation
- Item & SKU Understanding
- Query Understanding (product, gender, age, size and other attributes)
- Personalized Ranking

ML in Search: Typeahead / Guided Navigation





ML in Search: Query / Item Understanding

Query: Nike Men Shoes

Brand: NIKE
Gender: Men
Product Type:
- Athletic Shoes : 99%
- Casual and Dress Shoes: 55 %

Item

Brand: Nike
Color: Black / White
Shoe Size: 7.5M

What were the requirements

- Current production models written in TensorFlow and served with Java Native Interface
- serve large models like BERT
- Support Pytorch / Onnx
- SLA < 40 ms

Intro

Setup

- Repo:
- https://bit.ly/pytorch-workshop-2021

Dataset

- [Amazon Berkeley Objects (ABO) Dataset](#) (from UC Berkeley and Amazon)

Product metadata


```
{
  "item_id": "B075X4QMX3",
  "domain_name": "amazon.com",
  "item_name": [
    {
      "language_tag": "en_US",
      "value": "Stone & Beam W
    }
    {
      "language_tag": "zh_CN",
      "value": "亚马逊品牌 – Sto
    },
```


Metadata includes multilingual title, brand, model, year, product type, color, description, dimensions, weight, material, pattern, and style.

Catalog Images

For the 147,702 products, we provide 398,212 unique catalog images in high resolution.

360° Images

For more than 8,200 products, the dataset includes a sequence of 72 images, capturing the product every 5° in azimuth, for a total of 586,584 images.

3D Models

The dataset contains high-quality 3D models with 4K texture maps for physically based rendering for more than 7'900 products. The models are provided in the standard glTF 2.0 ↗ format.

Dataset

- Amazon Berkeley Objects (ABO) Dataset (from Amazon and UC Berkeley)

	brand	item_id	item_name	product_type
0	Amazon Essentials	B07HL25ZQM	Amazon Essentials Bib Set of 6	BABY_PRODUCT
1	AmazonBasics	B0825D4F6R	AmazonBasics 3-Tier Deluxe Tower Laundry Dryin...	HOME
2	Amazon Brand - Solimo	B07TF1FCFD	Amazon Brand - Solimo Designer Number Eight 3D...	CELLULAR_PHONE_CASE
3	Amazon Brand - Solimo	B08569SRJD	Amazon Brand - Solimo Designer Dark Night View...	CELLULAR_PHONE_CASE
4	Stone & Beam	B07B4G5RBN	Stone & Beam Varon 过渡日床, 灰石色	CHAIR

Demo

```
! curl -X POST http://localhost:9080/predictions/pt_classifier \
       -H 'Content-Type: application/json' \
       -d '{"text":"herbal tea","request_id":"test_id"}'
```


```
[
  {
    "TEA": 0.7232242226600647,
    "GROCERY": 0.2596974670886934,
    "COFFEE": 0.006111665163189173,
    "HERB": 0.006096727680414915,
    "EDIBLE_OIL_VEGETABLE": 0.0011588548077270389
  }
]
```

Review Some Deep Learning Concepts

Models before BERT

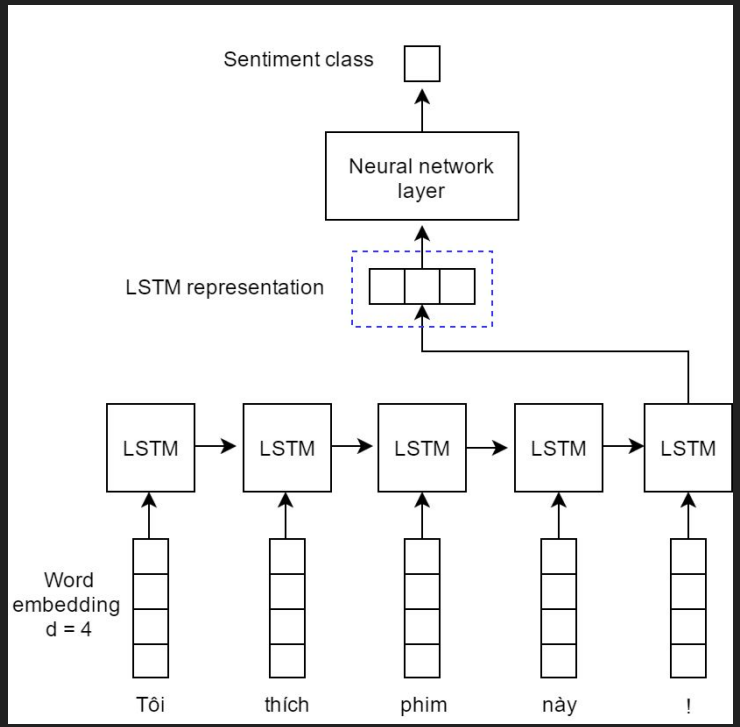


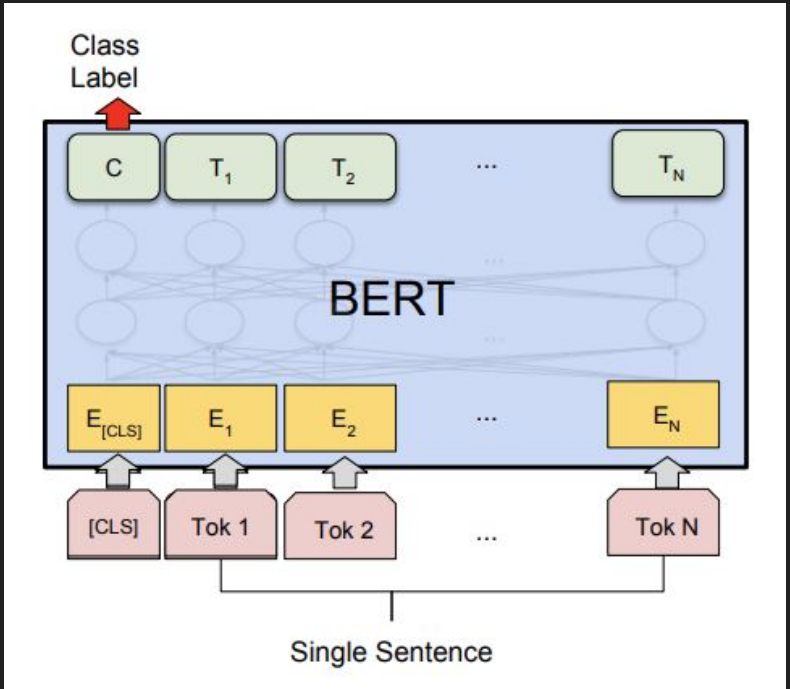
Image Ref

- Word Embedding Layer (Word2Vec, GLOVE, ELMO)
- Bidirectional LSTM

Cons:

- Slow to train RNN cells due to recurrence
- Not deeply bi-directional and hence context is lost, since left to right and right to left contexts are learnt separately and then concatenated
- Transformer models help alleviate these

BERT



Source : [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding]

Some Use Cases:

- Neural Machine Translation
- Question Answering
- Sentiment Analysis

Problems Solved:

- Contextualized Embeddings
- Can be trained on a larger corpus in a semi-supervised manner;
- General Language Model trained on Masked Word prediction
- Tokenizer solves for out of vocab words

Architecture:

- Attention is all you need
- stacked Transformer's Encoder model.

BERT: Different Architectures

For environments with limited compute resources, the BERT recipe has different model sizes with different attention heads and layers of complexity. The following table shows some of them with their corresponding GLUE scores.

MODEL	H	L	SST-2	QQP
BERT Tiny	128	2	83.2	62.2/83.4
BERT Mini	256	4	85.9	66.4/86.2
BERT Small	512	4	89.7	68.1/87.0
BERT Medium	512	8	89.6	69.6/87.9

Bi-LSTM vs BERT

Query: bedding for twin xl mattress

BI-LSTM Predictions:

- Mattress: 0.19
- Mattress Encasements: 0.10
- Bed Sheets: 0.05

BERT Predictions:

- Bed Sheets: 0.72
- Bedding Sets
- Bed-in-a-Bag: 0.26

F1 Score improved from 0.75 to 0.91

Hands-on

https://bit.ly/pytorch-workshop-2021

Notebook:
- 02_inference_review.ipynb
- 02_timing.ipynb

Tokenization (pre-processing)

```
from transformers import BertTokenizer
```

```
# Bert uses WordPiece Tokenizer
# splitting words either into the full forms
# (e.g., one word becomes one token) or into word piece
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```

```
tokenizer.tokenize("cheap nike men running shoes")
```

```
['cheap', 'nike', 'men', 'running', 'shoes']
```

```
# chep/runing is mispelled
tokenizer.tokenize("chep nike men shoes runing under 100$ ")
```

```
['che', '##p', 'nike', 'men', 'shoes', 'run', '##ing', 'under', '100', '$']
```

```
# size of vocabulary
tokenizer.vocab_size
```

```
30522
```

- BERT uses WordPiece Tokenizer
- A token is word or pieces of a word
- Vocab size is smaller than glove (30k vs 300k)

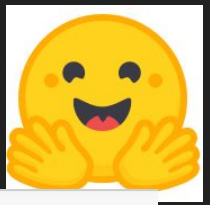
Sample Training Code (using HuggingFace)

```
# hugging face library to load existing/custom datasets
import datasets
# hugging face library contains tokenizers / models
import transformers
```

```
# dataset contains two columns "text/label"
raw_datasets = datasets.load_from_disk(dataset_path)
```

```
# use pretrained distilbert model
model = transformers.AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased"
                                                                         , num_labels=len(labels)
                                                                        ...
                                                                        )
```

Sample Training Code (using HuggingFace)



```
# use pretrained distilbert model
model = transformers.AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased"
                                                                         , num_labels=len(labels) ... )
```

```
training_args = transformers.TrainingArguments("trainer",num_train_epochs=5.
                                               )
trainer = transformers.Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets['train'],
    eval_dataset=tokenized_datasets['validation'],.... )
```

```
# train on datasets/argumets passed to trainer args
trainer.train()
```

Sample Inference Code

```
query = 'comfortable men sandals'
```

```
# compute input id / attention mask
tokenized_res = tokenizer.encode_plus(query, return_tensors="pt")
```

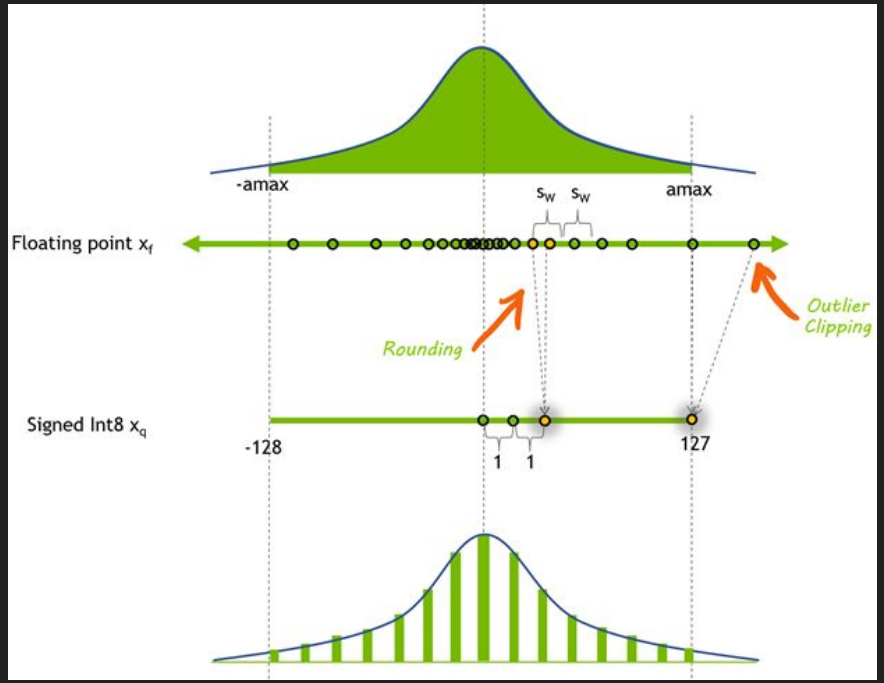
```
# pass input to model
model_res = model(**tokenized_res)
# get softmax of logits
logits = model_res.logits
softmax_res = torch.softmax(logits, dim=1).toList()[0]
```

```
# get the label and probability sorted
predictions = list ( zip (labels , softmax_res ) )
predictions = sorted (predictions , key=lambda x:x[1] , reverse =True)
```

Optimizing the model for production

Post Training Optimization: Quantization

- Techniques for performing computations and storing tensors at lower bit than floating point precision; reduce memory and speed up compute
- Hardware support for INT8 computations is typically 2 to 4 times faster compared to FP32 compute.



[Achieving FP32 Accuracy for INT8 Inference Using Quantization Aware Training with NVIDIA TensorRT]

Post Training Optimization: Quantization

- Dynamic Quantization:
  - Weights are quantized
  - Activations quantized on the fly (don’t get memory benefits)
  - However, the activations are read and written to memory in floating point format

```
# create a model instance
model_fp32 = M()
# create a quantized model instance
model_int8 = torch.quantization.quantize_dynamic(
    model_fp32,  # the original model
    {torch.nn.Linear},  # a set of layers to dynamically quantize
    dtype=torch.qint8)  # the target dtype for quantized weights
```

Dynamic Quantization in PyTorch

Post Training Optimization: Quantization

- Static Quantization:
  - Weights / activations are quantized with a calibration data post training
  - Passing quantized values between operations instead of converting these values to floats - and then back to ints - between every operation, resulting in a significant speed-up.

- Quantization Aware Training:
  - quantization error is part of training error; training learns to reduce quantization error
  - Leads to the greatest accuracy

Quantization results

Model	Fp32 accuracy	Int 8 accuracy change	Strategy	Inference performance gain	Notes
BERT	0.902	0.895	Dynamic	1.8x 581 -> 313	Single thread; 1.6HZ;
Resnet-50	76.9	75.9	Static	2x 214->103	Single thread; 1.6GHZ
Mobilenet-v2	71.9	71.6	Quantization Aware Training	5.7x 97->17	Samsung S9

Reference: Introduction to Quantization on PyTorch ; pytorch.org

Post Training Optimization: Distillation

- Larger models have higher knowledge capacity than small models but not all of it may be used
- Transfer Knowledge from a **trained** large model to a smaller one

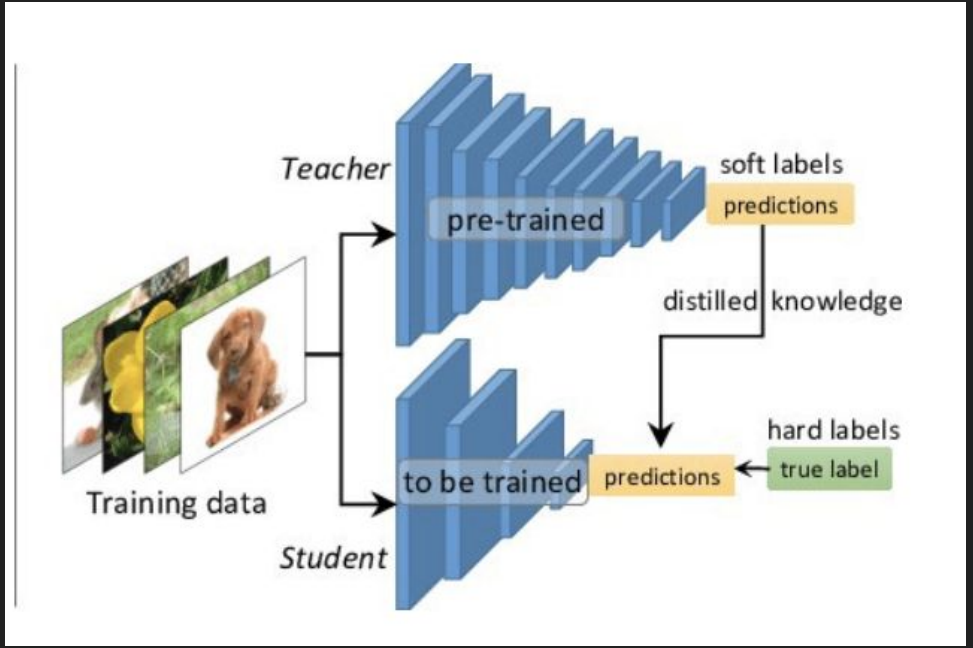


Image Reference:
https://towardsdatascience.com/knowledge-distillation-simplified-dd4973dbc764

Distillation Results

Model	SST-2	QQP
BERT (Large)	94.9	72.1
BERT Base	93.5	71.2
ELMO	90.4	63.1
BiLSTM	86.7	63.8
Distilled BiLSTM	90.7	68.2

Distilling Task-Specific Knowledge from BERT into Simple Neural Networks, 2018, Tang et all

Post Training Optimization: Pruning

- Sparse models are easier to compress.
- In a neural network, neurons and synapses can be ranked according to their contributions towards deep learning, based on their weights.
- Low contribution neurons can be pruned by setting individual weights in the weight matrix to zero.
- Pruning can decrease the size and complexity of the model, retaining the original model's architecture.

```
prune.random_unstructured(module, name="weight", amount=0.3)
```

Model Pruning in PyTorch

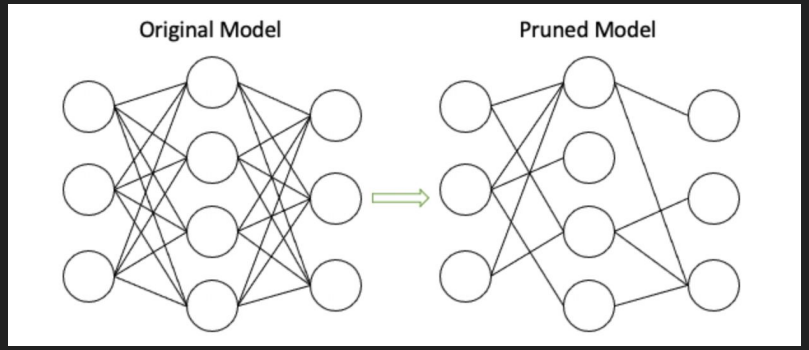


Image Reference:
https://towardsdatascience.com/pruning-deep-neural-network-56cae1ec5505

Eager Execution vs Script Mode

Eager Execution

- Good for **training** /prototyping / debugging
- requires a python process
- Slow for production use cases
- Flexible ; can apply any model techniques; easy to change
- Can use any of the libraries in python ecosystem

Script Mode

- Model is running in graph execution mode; better optimized for speed
- Only needs a C++ runtime; so can be loaded in other languages
- TorchScript is an optimized intermediate representation of a pytorch model
- Doesn’t support everything that python /eager model supported

TorchScript JIT: Tracing vs Scripting

Tracing

- Does not support python data structures / control flows in the model
- Requires a sample input

Scripting

- Supports custom control flow and python data structures
- Only requires the model

TorchScript: Script

```
import torch
class MyModule(torch.nn.Module):

  def __init__(self, N, M):
    super(MyModule, self).__init__()
    self.weight = torch.nn.Parameter(torch.rand(N, M))

  def forward(self, input):
    if input.sum() > 0:
      output = self.weight.mv(input)
    else:
      output = self.weight + input
    return output

  # Compile the model code to a static representation
  my_script_module = torch.jit.script(MyModule(3, 4))

  # Save the compiled code and model data so it can be l
  my_script_module.save("my_script_module.pt")
```

TorchScript Timing

- Inference time for torch script model: cpu/gpu

	CPU (ms)	GPU (ms)
Pytorch	1339	460
TorchScript	768 (42%)	360 (21%)

Benchmarking Transformers: PyTorch and TensorFlow from HuggingFace
Full Table

Hands-on: Optimizing the model

- Quantizing model
- Converting the BERT model with torch script

https://bit.ly/pytorch-workshop-2021

Notebook: 03_optimizing_model

Deploying the model

Options for deploying PyTorch model

- Pure Flask / Fastapi
- Model Servers: TorchServe / Nvidia Triton / TensorFlow Serving

Benefits of TorchServe

- Optimized for serving models
- Configurable support for CUDA and GPU environments
- multi model serving
- model version for A/B testing
- server side batching
- support for pre and post processing

Packaging a model / MAR

- Torch Model Archiver: utility from pytorch team
- package model code and artifacts into one file
- Python model specific dependencies

```
# name and version of the model
MODEL_NAME="pt_classifier"
MODEL_VERSION="1.0"

# folder where model is saved
MODEL_STORE="model_store"
# path of saved pytorch models
MODEL_SERIALIZED_FILE="traced_model.pt"
# path of extra files to include
MODEL_EXTRA_FILES="index_to_name.json,setup_config.json"
# model code
MODEL_CODE="handler.py"
```

```
torch-model-archiver --model-name ${MODEL_NAME} \
--version ${MODEL_VERSION} \
--serialized-file ${MODEL_SERIALIZED_FILE} \
--export-path ${MODEL_STORE} \
--extra-files ${MODEL_EXTRA_FILES} \
--handler ${MODEL_CODE} \
```

PyTorch Base Handler

```python
from abc import ABC
import os
import torch
from ts.torch_handler.base_handler import BaseHandler

class CustomHandler(BaseHandler, ABC):

    # method is called when initializing each worker
    def initialize(self, ctx):
        model_dir = ctx.system_properties.get("model_dir")
        serialized_file =  ctx.manifest["model"]["serializedFile"]
        model_pt_path = os.path.join(model_dir, serialized_file)
        self.model = torch.jit.load(model_pt_path, map_location=self.device)
        ...

    # called before inference
    def preprocess(self, requests):
        ...
    # called for inference
    def inference(self, input_batch):
        ...
    # called after inference
    def postprocess(self, inference_output):
        ...
    def handle(self, data, context):
        ...
        data_preprocess = self.preprocess(data)
        data_inference = self.inference(data_preprocess)
        data_postprocess = self.postprocess(data_inference)
        return data_postprocess
```

Model Serving Handler at Walmart

- Acts as a layer on top of the BaseHandler provided by TorchServe
- All common initialization, pre-processing, error handling and metric collection steps can be abstracted into this custom handler class
- Any model specific logic can be added in the child class that inherits from the custom handler class
- This provides a resilient wrapper that can be extended and used by multiple teams to serve models through TorchServe, with minimal config and code changes

Built in handlers

- Pytorch has support for these handlers / use case
  - image_classifier
  - image_segmenter
  - object_detector
  - text_classifier
- Just need to provide model_file and index_to_name.json

Serving

torchserve --ts-config config.properties --start --model-store model_store

```
load_models=all
inference_address=http://0.0.0.0:8080
management_address=http://0.0.0.0:8081
metrics_address=http://0.0.0.0:8082
number_of_netty_threads=32
job_queue_size=1000
model_store=/home/model-server/model-store
batch_size=20
max_batch_delay=2
async_logging=true
models=\
```

APIS

A. Management Api: http://localhost:8081
   List of Models:
   - curl "http://localhost:8081/models"

   Get detail on model
   - curl http://localhost:8081/models/model_name

B. Inference Api: http://localhost:8080

   Inference:

   curl -X POST http://localhost:8080/predictions/model_name

Handson: Deploying the Model

- Package the given model using Torch Model Archive
- Write a custom handler to support preprocessing and post processing

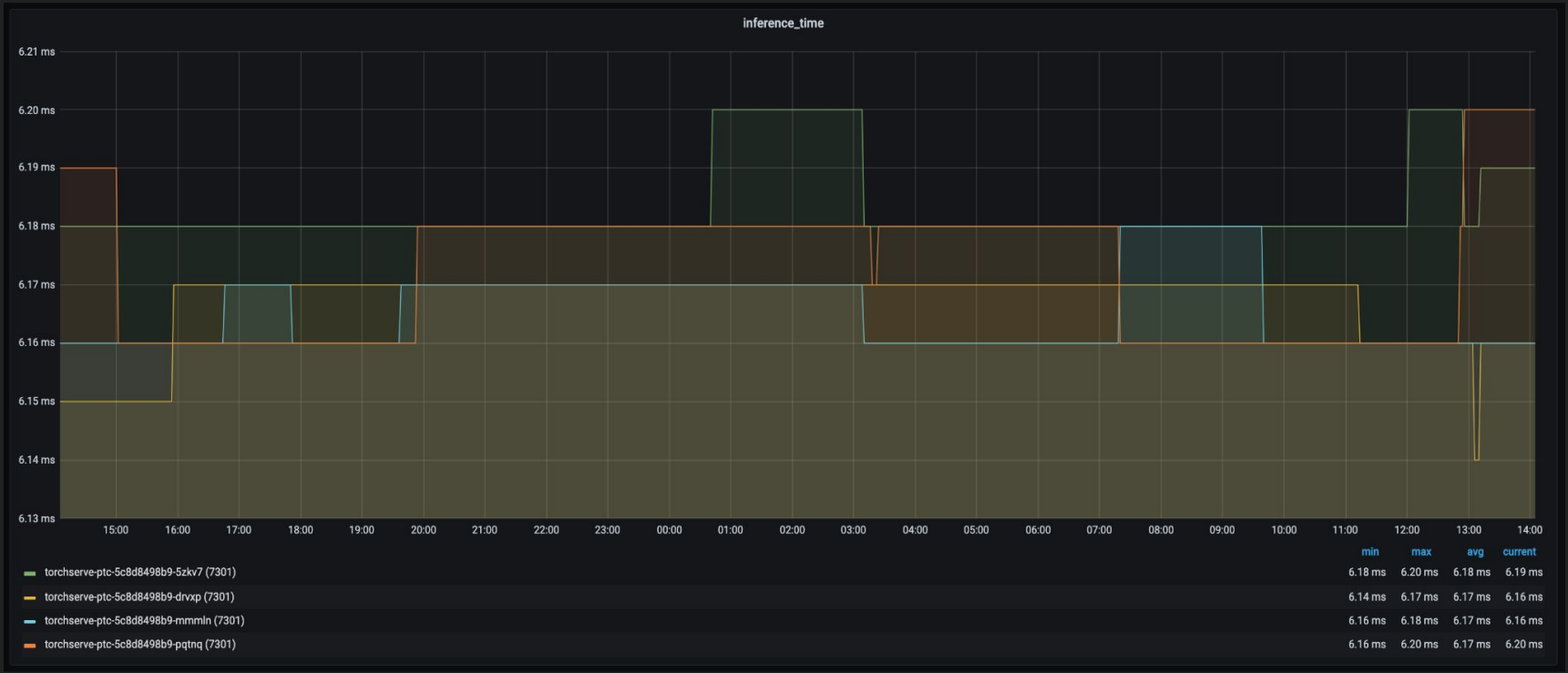
https://bit.ly/pytorch-workshop-2021

Notebook: 04_packaging

Lessons learned

Lesson Learned

- DistilledBertBase with quantization and other optimizations can be served on a CPU and meet < 40 ms SLA
- DistilledBertBase on nvidia-t4 can hit < 10 ms under peak traffic
- Throughput of 1500 RPS at under SLA



Q/A

Check out Walmart Global Tech for openings for Software Engineers, Machine Learning Engineers and Data Scientists: https://careers.walmart.com/technology