Model Packaging and Serving

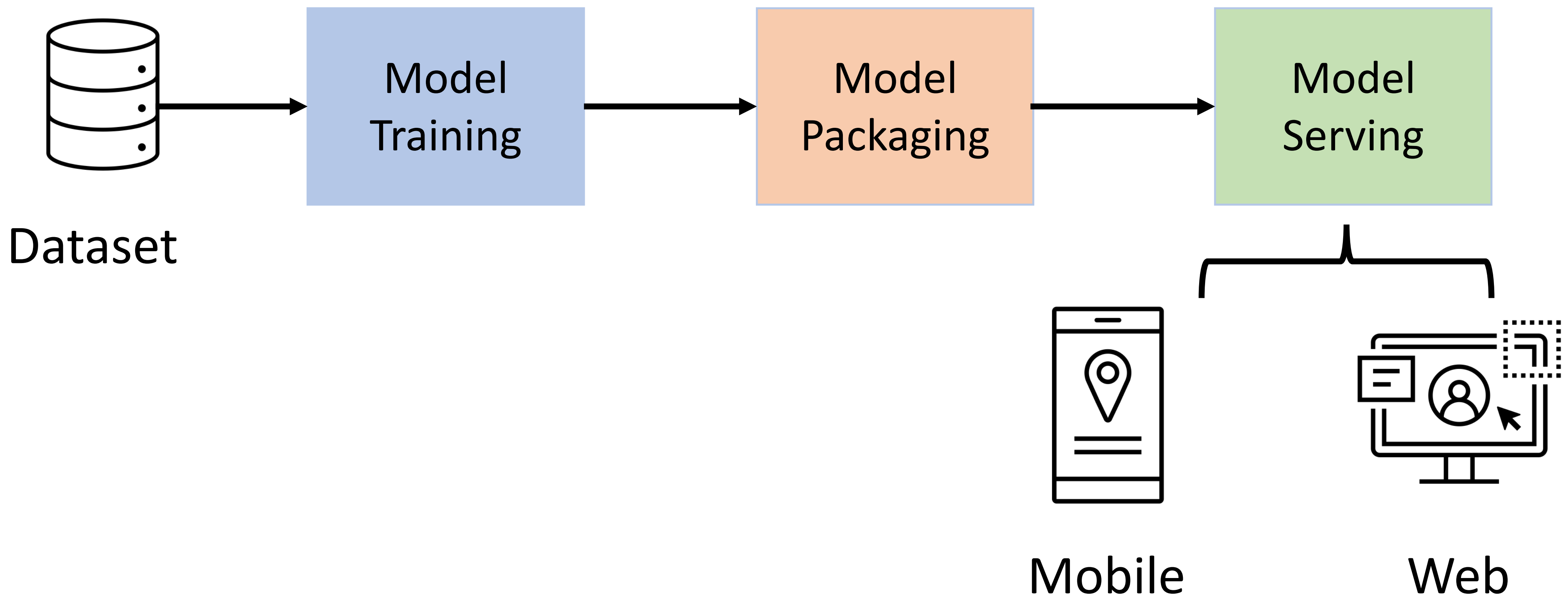
Rowel Atienza, PhD
University of the Philippines
github.com/roatienza

2023

Model Packaging and Serving

- Optimizes models for inference/deployment
- Enables trained models available for applications to use
- Maximizes the utility of the underlying AI accelerators

Model Deployment Pipeline



Gradio or Streamlit Front-end

Model Packaging

- Model packaging is a process that involves packaging model artifacts, dependencies, configuration files, and metadata into a single format for effortless distribution, installation, and reuse.
- The ultimate aim is to simplify the process of deploying a model, making the process of taking it to production seamless.
- Easy to install, reproducible, versioned, documented

https://neptune.ai/blog/ml-model-packaging

Model Formats

- ONNX
- TensorRT
- TorchScript
- Build your own
  - GGML https://github.com/ggerganov/ggml
  - vLLM https://github.com/vllm-project/vllm
  - S-LoRA https://github.com/S-LoRA/S-LoRA

ONNX

- ONNX is an open format built to represent machine learning models.
- ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers.
- Interoperability
- Hardware access

https://onnx.ai/

Frameworks & Converters

Use the frameworks you already know and love.

Caffe2

Yandex CatBoost

Chainer

Cognitive Toolkit

CoreML

Optimum

Keras

LibSVM

MATLAB

[M]^{S}MindSpore

composer

mxnet

MyCaffe™

NCNN

NeoML

Neural Network Libraries

飞桨 PaddlePaddle

PaddlePaddle

PyTorch

sas

SIEMENS

Simio Forward Thinking

Apache SINGA

scikit learn

SciKit Learn

Tengine

TensorFlow

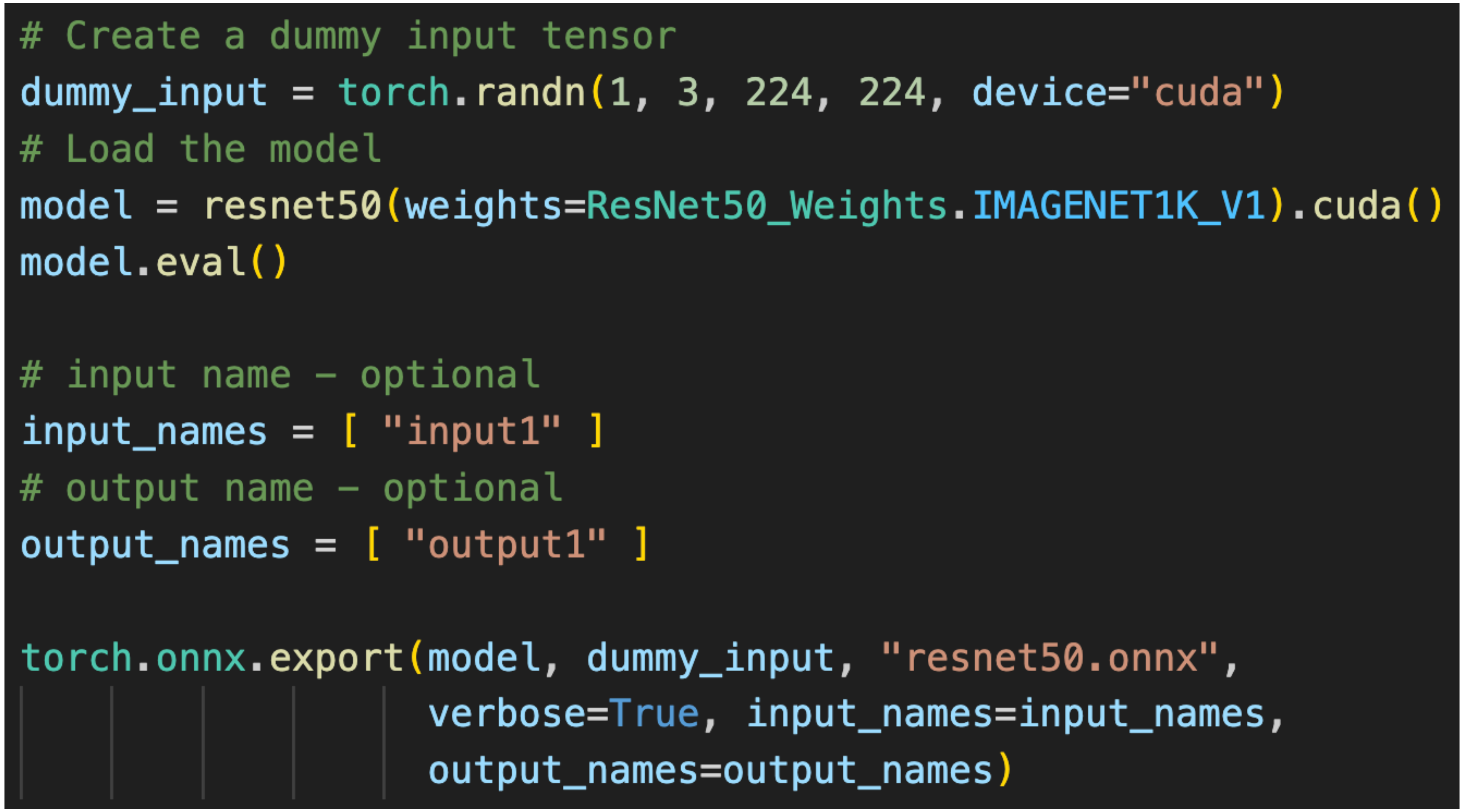
Tribuo

dmlc XGBoost

WOLFRAM

ZAMA

Torch to ONNX



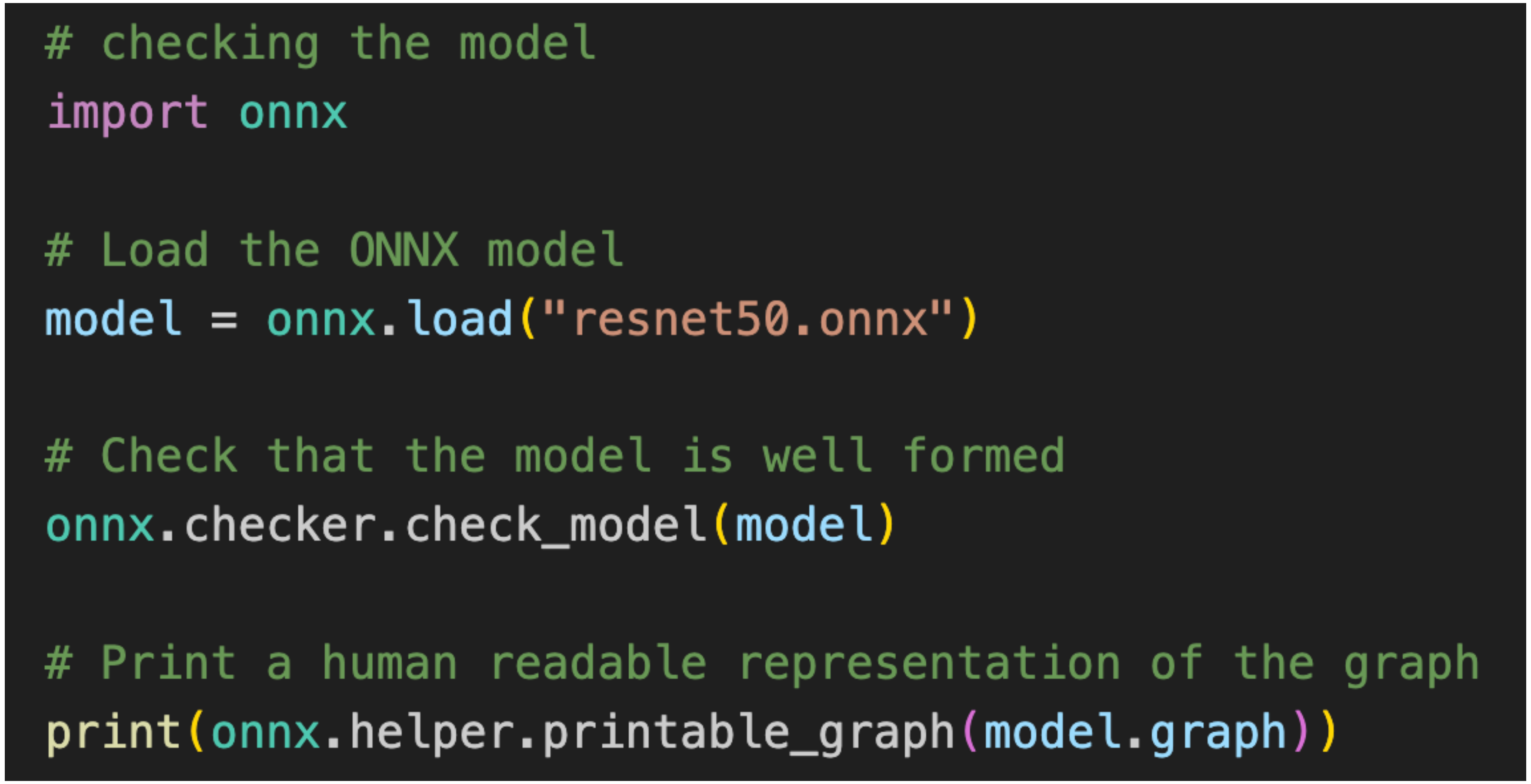
```python
# Create a dummy input tensor
dummy_input = torch.randn(1, 3, 224, 224, device="cuda")
# Load the model
model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V1).cuda()
model.eval()

# input name - optional
input_names = [ "input1" ]
# output name - optional
output_names = [ "output1" ]

torch.onnx.export(model, dummy_input, "resnet50.onnx",
                  verbose=True, input_names=input_names,
                  output_names=output_names)
```

Install ONNX support for all AI accelerators

```
pip install onnx
pip install onnxruntime-gpu
conda install cudatoolkit
pip install --upgrade setuptools pip
pip install nvidia-pyindex
pip install --upgrade nvidia-tensorrt
sudo apt-get install python3-libnvinfer-dev
```

https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html

Optional Checking



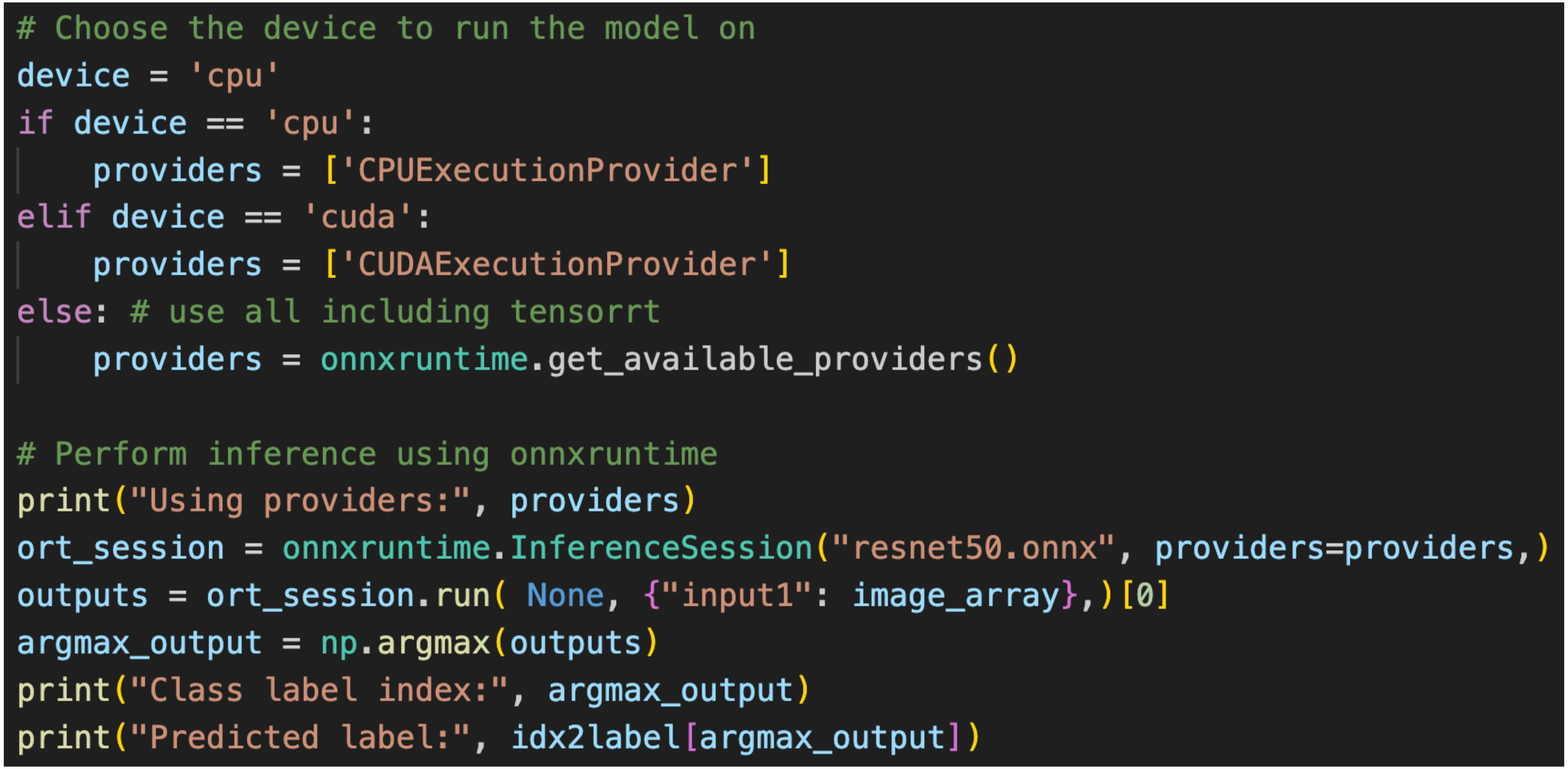
```python
# checking the model
import onnx

# Load the ONNX model
model = onnx.load("resnet50.onnx")

# Check that the model is well formed
onnx.checker.check_model(model)

# Print a human readable representation of the graph
print(onnx.helper.printable_graph(model.graph))
```

Check ONNX Execution Providers

```
import onnxruntime

onnxruntime.get_available_providers()
```

```
['TensorrtExecutionProvider',
'CUDAExecutionProvider',
'AzureExecutionProvider',
'CPUExecutionProvider']
```

ONNX Inference



```python
# Choose the device to run the model on
device = 'cpu'
if device == 'cpu':
    providers = ['CPUExecutionProvider']
elif device == 'cuda':
    providers = ['CUDAExecutionProvider']
else: # use all including tensorrt
    providers = onnxruntime.get_available_providers()

# Perform inference using onnxruntime
print("Using providers:", providers)
ort_session = onnxruntime.InferenceSession("resnet50.onnx", providers=providers,)
outputs = ort_session.run( None, {"input1": image_array},)[0]
argmax_output = np.argmax(outputs)
print("Class label index:", argmax_output)
print("Predicted label:", idx2label[argmax_output])
```

TensorRT

- TensorRT provides APIs via C++ and Python that help to express deep learning models via the Network Definition API or load a pre-defined model via the parsers that allow TensorRT to optimize and run them on an NVIDIA GPU.
- TensorRT applies graph optimizations, layer fusions, among other optimizations, while also finding the fastest implementation of that model leveraging a diverse collection of highly optimized kernels.
- TensorRT also supplies a runtime that you can use to execute this network on all of NVIDIA's GPU's
- TensorRT is ahead of time (AOT) compilation

https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html

Install

See ONNX install plus this one:

```
pip install tensorrt torch-tensorrt
```

TorchScript

- Serializable and optimizable models from PyTorch code
- A TorchScript program can be saved and loaded in a process where there is no Python dependency such as C++ programs
- Two approaches : Tracing and Scripting
- Just in time (JIT) compilation

https://pytorch.org/docs/stable/jit.html

Tracing

- Using *torch.jit.trace* and *torch.jit.trace_module*, convert an existing module or Python function into TorchScript ScriptFunction or ScriptModule that will be optimized using just-in-time compilation.
- Tracing is ideal for code that operates only on Tensors and lists, dictionaries, and tuples of Tensors.
- Do not use if there is a data dependent conditionals or loops within the model
- `torch.jit.trace()`

https://pytorch.org/docs/stable/generated/torch.jit.trace.html#torch.jit.trace

Data Dependent Control

```
def forward(self, x):
    b, c, h, w = x
    if c == 1:
        # process grayscale
    else:
        # process color
```

Scripting

- Scripting a function or nn.Module will inspect the source code, compile it as TorchScript code using the TorchScript compiler, and return a ScriptModule or ScriptFunction.
- A subset of the Python language, so not all features in Python work, but provides enough functionality to compute on tensors and do control-dependent operations
- `torch.jit.script()`

https://pytorch.org/docs/stable/generated/torch.jit.script.html#torch.jit.script

Tracing and Scripting

- Both turn a module or a function into a computational graph
- Not all modules/functions can be scripted or traced

https://ppwwyyxx.com/blog/2022/TorchScript-Tracing-vs-Scripting/

Tracing vs Scripting

- Traced module/function is simpler and faster
- Traced module/function does not corrupt the code
- Tracing is easier compared to scripting
- Only scripting supports data dependent conditionals and loops
- Can mix tracing and scripting

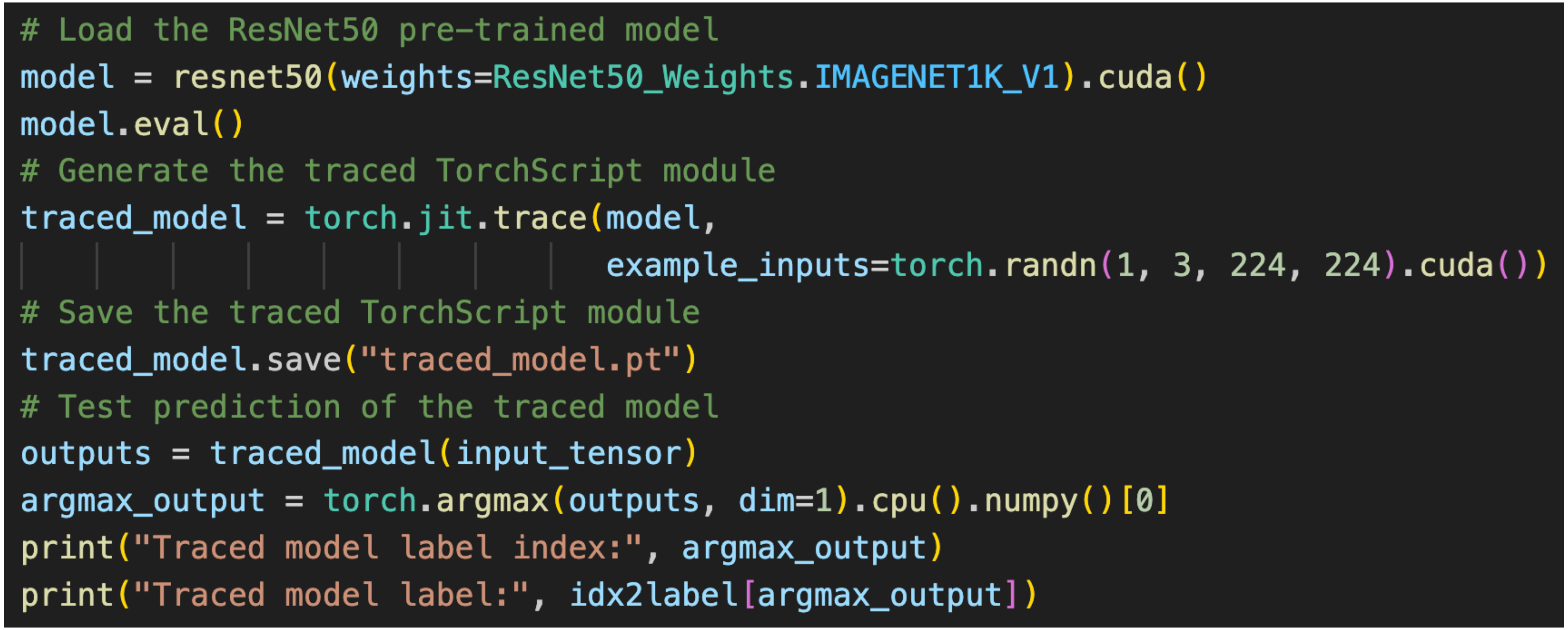
https://ppwwyyxx.com/blog/2022/TorchScript-Tracing-vs-Scripting/

Tracing and Scripting : Lessons

- Modularize your models – easier to debug and mix tracing and scripting
- Start with tracing first.
- Use scripting on functions/modules that could not be traced.

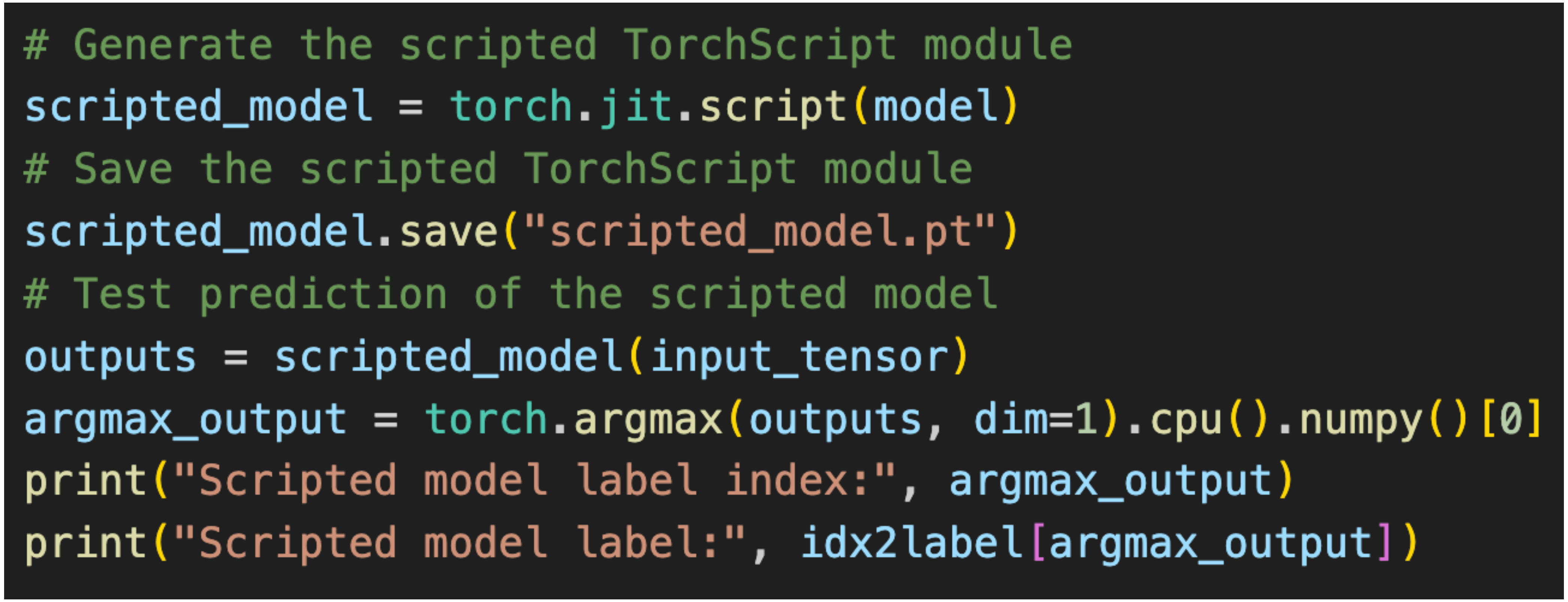
https://ppwwyyxx.com/blog/2022/TorchScript-Tracing-vs-Scripting/

Tracing ResNet50



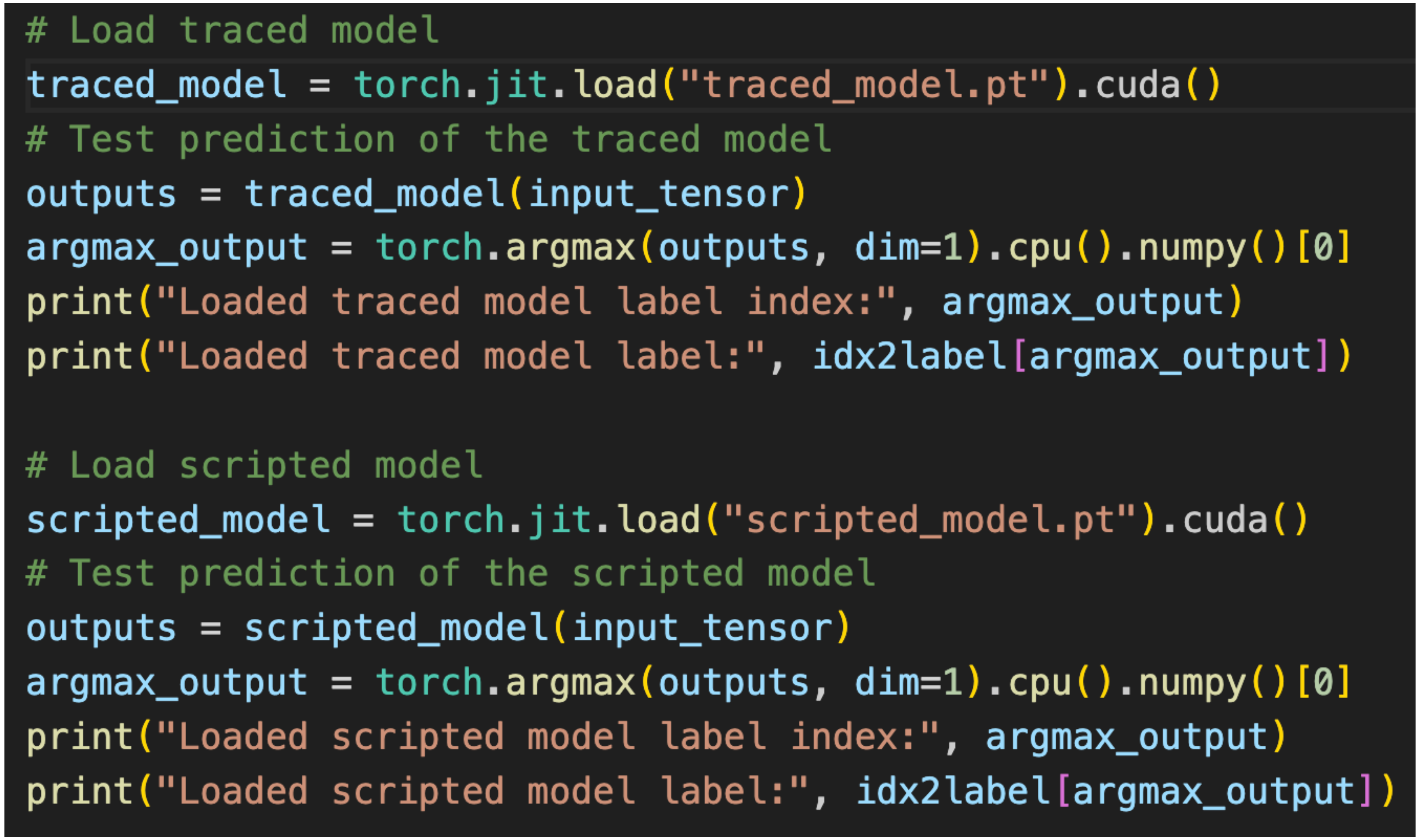
```python
# Load the ResNet50 pre-trained model
model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V1).cuda()
model.eval()
# Generate the traced TorchScript module
traced_model = torch.jit.trace(model,
                               example_inputs=torch.randn(1, 3, 224, 224).cuda())
# Save the traced TorchScript module
traced_model.save("traced_model.pt")
# Test prediction of the traced model
outputs = traced_model(input_tensor)
argmax_output = torch.argmax(outputs, dim=1).cpu().numpy()[0]
print("Traced model label index:", argmax_output)
print("Traced model label:", idx2label[argmax_output])
```

Scripting ResNet50



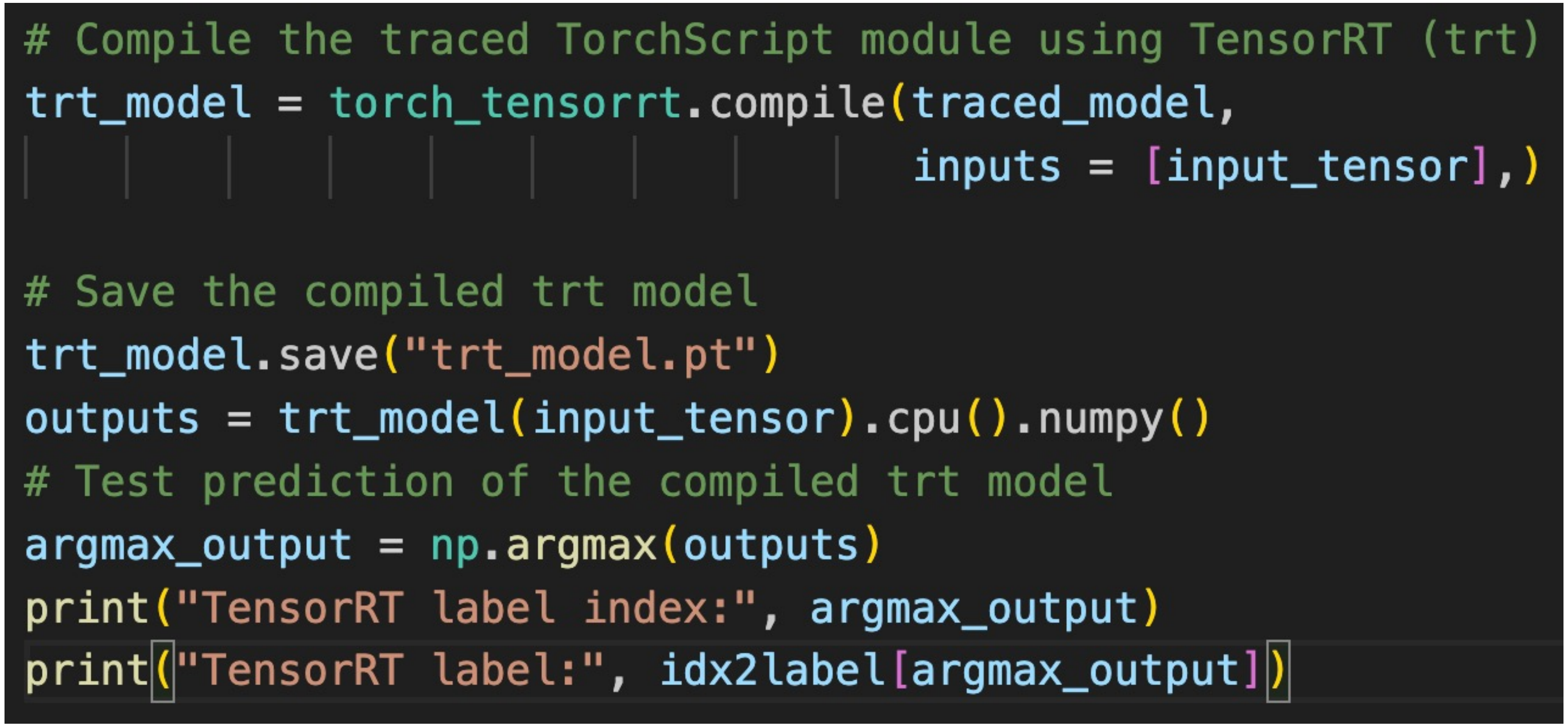
```python
# Generate the scripted TorchScript module
scripted_model = torch.jit.script(model)
# Save the scripted TorchScript module
scripted_model.save("scripted_model.pt")
# Test prediction of the scripted model
outputs = scripted_model(input_tensor)
argmax_output = torch.argmax(outputs, dim=1).cpu().numpy()[0]
print("Scripted model label index:", argmax_output)
print("Scripted model label:", idx2label[argmax_output])
```

Reuse Traced or Scripted Models



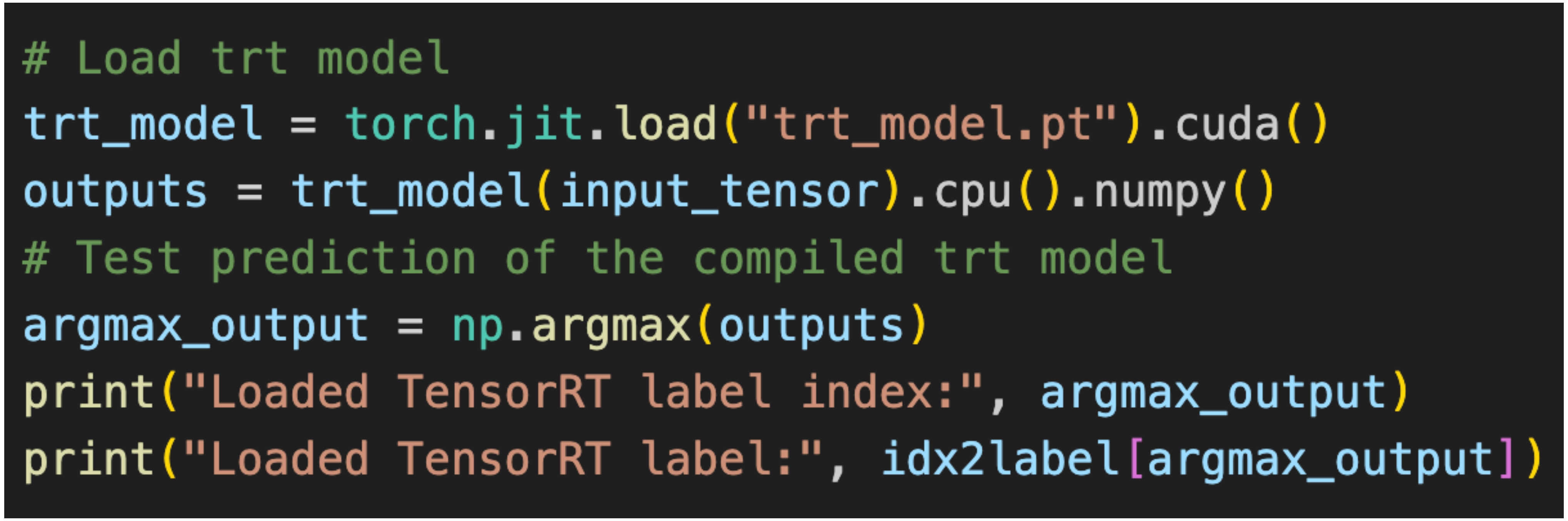
```python
# Load traced model
traced_model = torch.jit.load("traced_model.pt").cuda()
# Test prediction of the traced model
outputs = traced_model(input_tensor)
argmax_output = torch.argmax(outputs, dim=1).cpu().numpy()[0]
print("Loaded traced model label index:", argmax_output)
print("Loaded traced model label:", idx2label[argmax_output])

# Load scripted model
scripted_model = torch.jit.load("scripted_model.pt").cuda()
# Test prediction of the scripted model
outputs = scripted_model(input_tensor)
argmax_output = torch.argmax(outputs, dim=1).cpu().numpy()[0]
print("Loaded scripted model label index:", argmax_output)
print("Loaded scripted model label:", idx2label[argmax_output])
```

TensorRT ResNet50



```python
# Compile the traced TorchScript module using TensorRT (trt)
trt_model = torch_tensorrt.compile(traced_model,
                                   inputs = [input_tensor],)

# Save the compiled trt model
trt_model.save("trt_model.pt")
outputs = trt_model(input_tensor).cpu().numpy()
# Test prediction of the compiled trt model
argmax_output = np.argmax(outputs)
print("TensorRT label index:", argmax_output)
print("TensorRT label:", idx2label[argmax_output])
```

Reuse Compiled TensorRT Model



```python
# Load trt model
trt_model = torch.jit.load("trt_model.pt").cuda()
outputs = trt_model(input_tensor).cpu().numpy()
# Test prediction of the compiled trt model
argmax_output = np.argmax(outputs)
print("Loaded TensorRT label index:", argmax_output)
print("Loaded TensorRT label:", idx2label[argmax_output])
```

Model Serving

- Enables concurrent applications/processes to request for inference
- Enables inference of multiple and possible different models on the same machine AI accelerators (CPU, GPU)
- Can be deployed on edge, on-site or cloud
- Can be packaged in docker container and managed by orchestration

PyTriton Inference Server

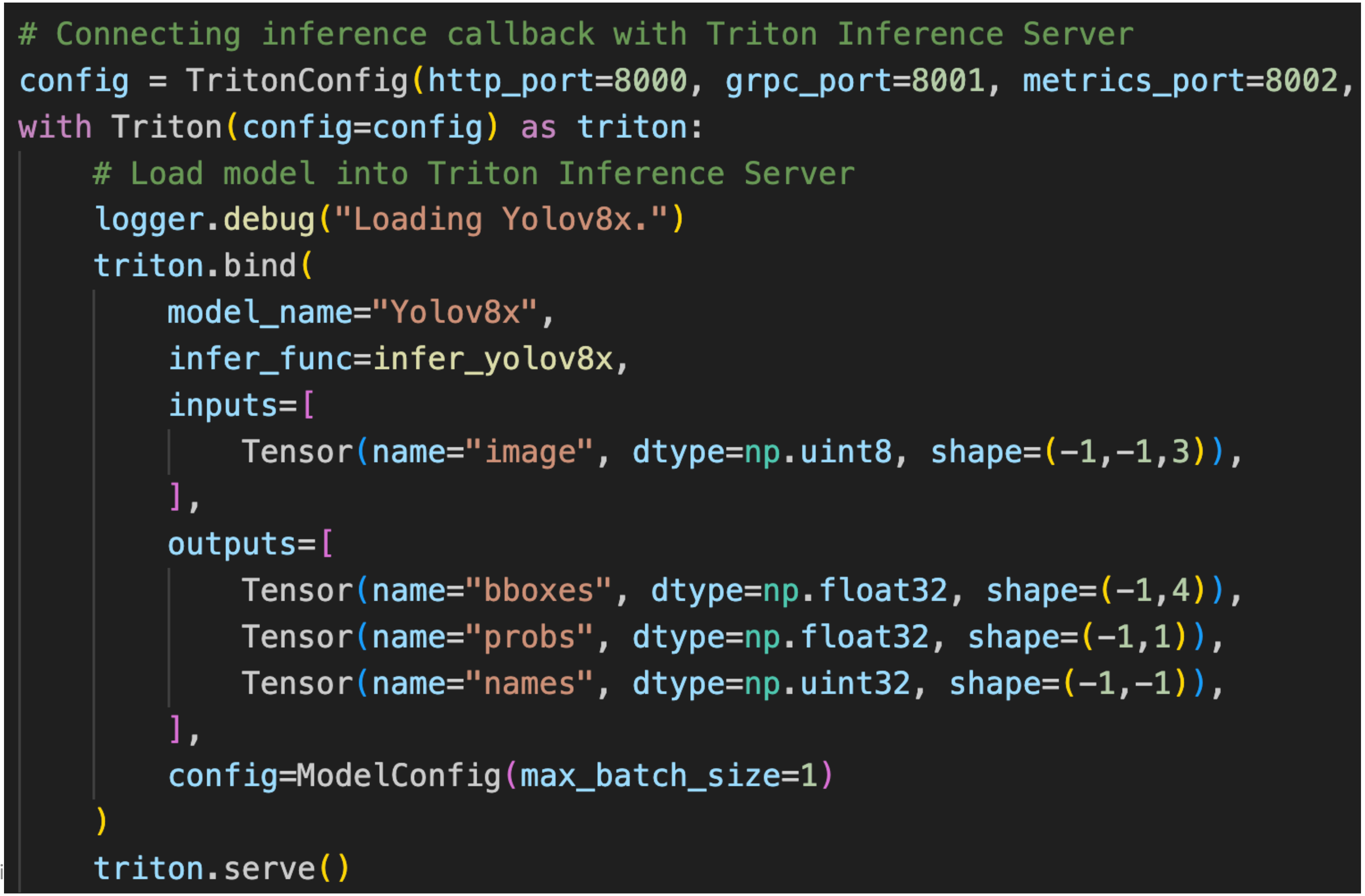
- PyTriton enables fast deployment of AI/ML models and python tools in a Flask API like manner.
- PyTriton can deploy model inference calls as commonly found in many github repos.
- Triton is already built-in the pytriton inference server so models in `torch, onnx, tf2` and `tensorrt` are also supported.
- PyTriton is like HuggingFace model hub except that we can deploy models on our local machines therefore enabling faster inference time and higher flexibility in terms of configuration.

https://developer.nvidia.com/blog/fast-and-scalable-ai-model-deployment-with-nvidia-triton-inference-server/

Install

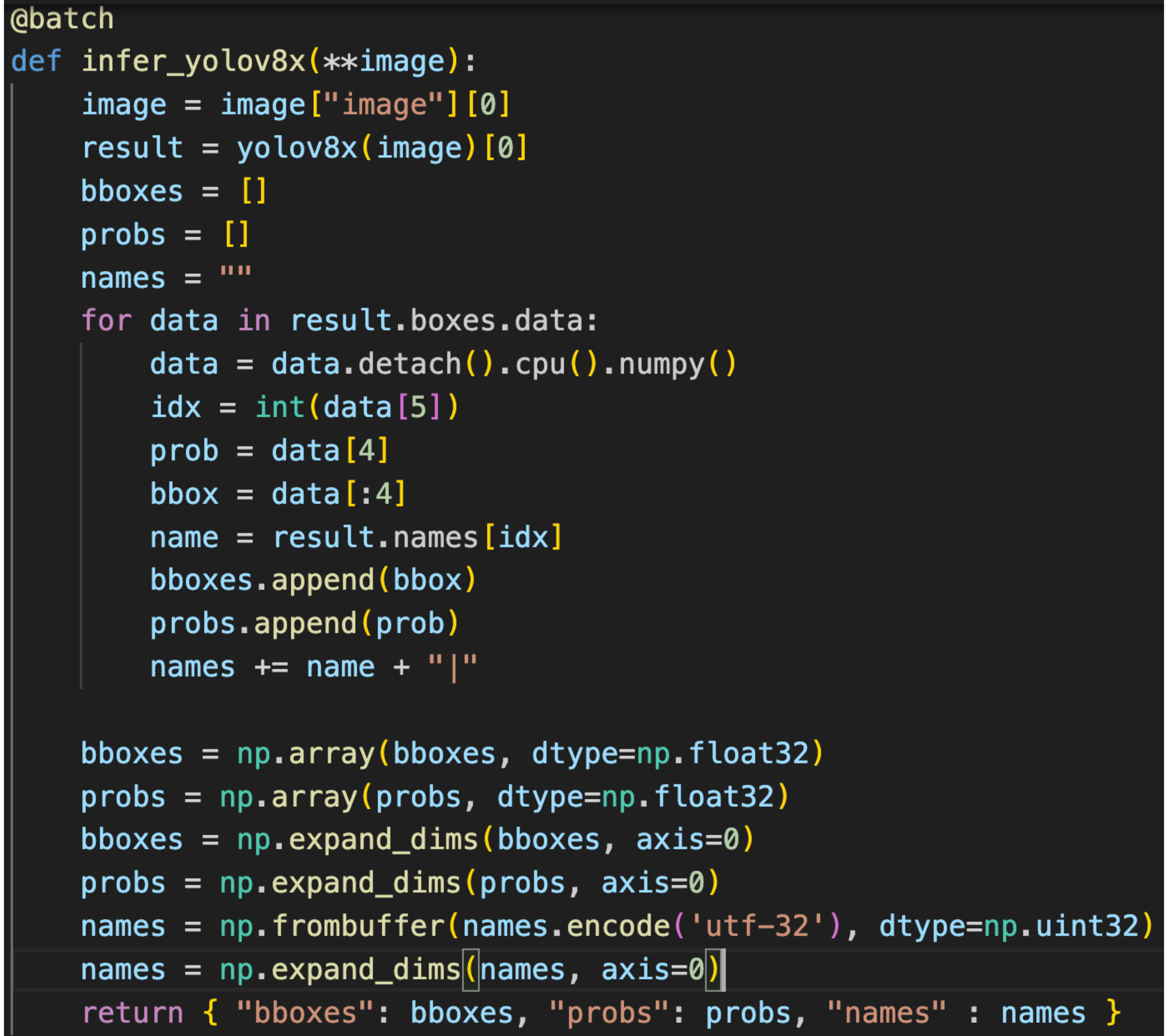
```
pip3 install -U nvidia-pytriton
pip3 install ultralytics
```

PyTriton Server



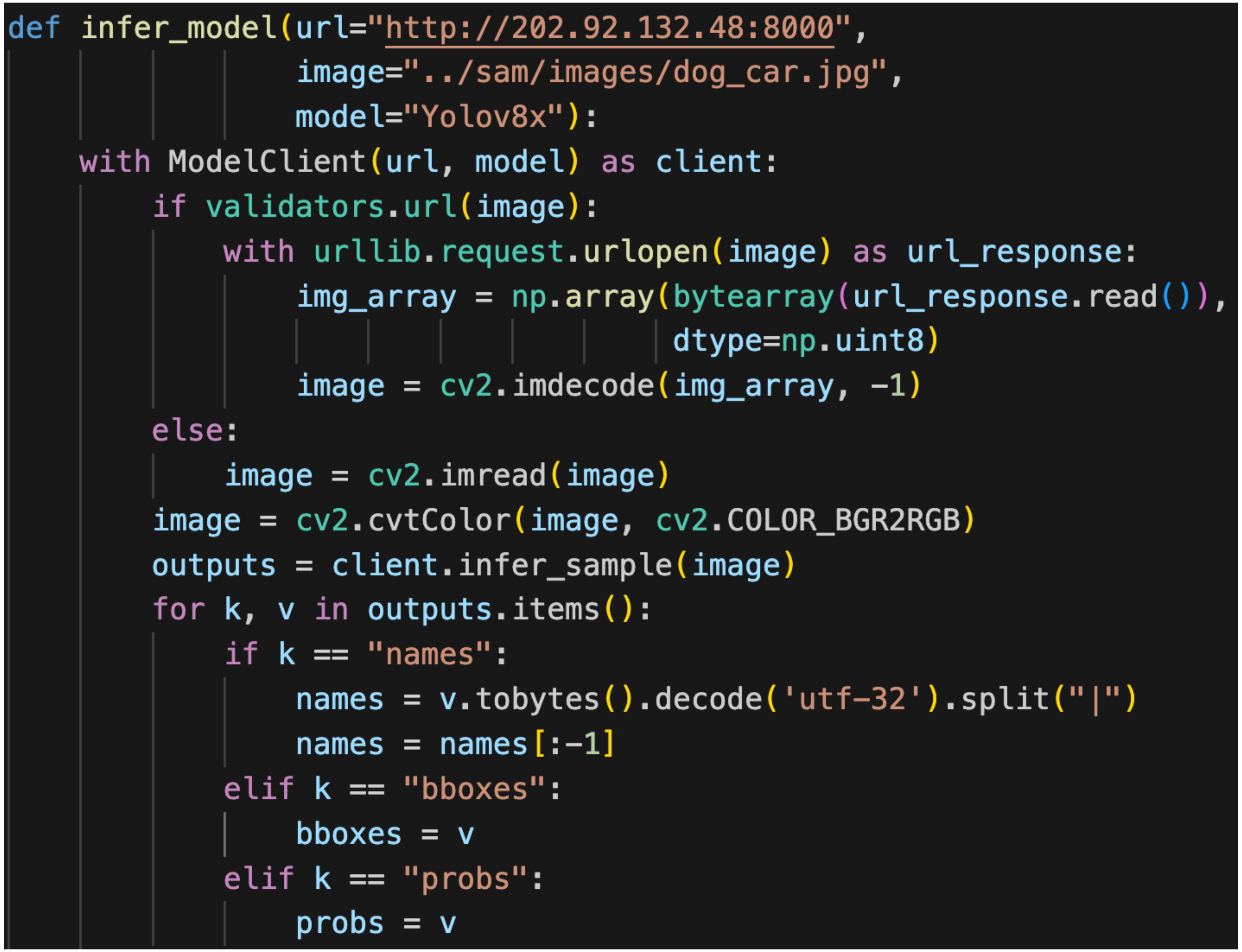
```
# Connecting inference callback with Triton Inference Server
config = TritonConfig(http_port=8000, grpc_port=8001, metrics_port=8002,
with Triton(config=config) as triton:
    # Load model into Triton Inference Server
    logger.debug("Loading Yolov8x.")
    triton.bind(
        model_name="Yolov8x",
        infer_func=infer_yolov8x,
        inputs=[
            Tensor(name="image", dtype=np.uint8, shape=(-1,-1,3)),
        ],
        outputs=[
            Tensor(name="bboxes", dtype=np.float32, shape=(-1,4)),
            Tensor(name="probs", dtype=np.float32, shape=(-1,1)),
            Tensor(name="names", dtype=np.uint32, shape=(-1,-1)),
        ],
        config=ModelConfig(max_batch_size=1)
    )
    triton.serve()
```

PyTriton Server



```
@batch
def infer_yolov8x(**image):
    image = image["image"][0]
    result = yolov8x(image)[0]
    bboxes = []
    probs = []
    names = ""
    for data in result.boxes.data:
        data = data.detach().cpu().numpy()
        idx = int(data[5])
        prob = data[4]
        bbox = data[:4]
        name = result.names[idx]
        bboxes.append(bbox)
        probs.append(prob)
        names += name + "|"

    bboxes = np.array(bboxes, dtype=np.float32)
    probs = np.array(probs, dtype=np.float32)
    bboxes = np.expand_dims(bboxes, axis=0)
    probs = np.expand_dims(probs, axis=0)
    names = np.frombuffer(names.encode('utf-32'), dtype=np.uint32)
    names = np.expand_dims(names, axis=0)
    return { "bboxes": bboxes, "probs": probs, "names" : names }
```

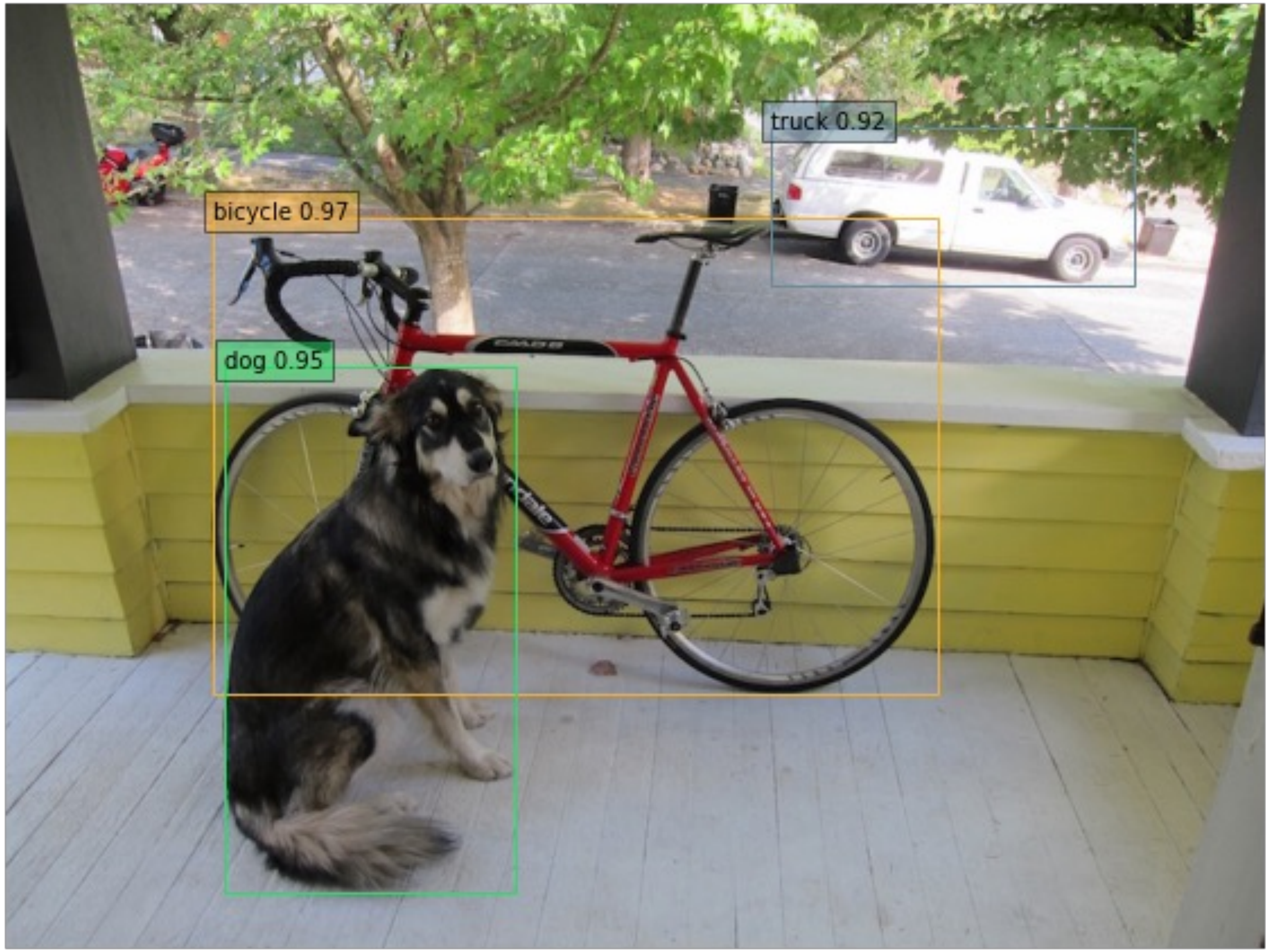
PyTriton Client



```python
def infer_model(url="http://202.92.132.48:8000",
                image="../sam/images/dog_car.jpg",
                model="Yolov8x"):
    with ModelClient(url, model) as client:
        if validators.url(image):
            with urllib.request.urlopen(image) as url_response:
                img_array = np.array(bytearray(url_response.read()),
                                     dtype=np.uint8)
                image = cv2.imdecode(img_array, -1)
        else:
            image = cv2.imread(image)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        outputs = client.infer_sample(image)
        for k, v in outputs.items():
            if k == "names":
                names = v.tobytes().decode('utf-32').split("|")
                names = names[:-1]
            elif k == "bboxes":
                bboxes = v
            elif k == "probs":
                probs = v
```

PyTriton Client Input

PyTriton Client Output



End