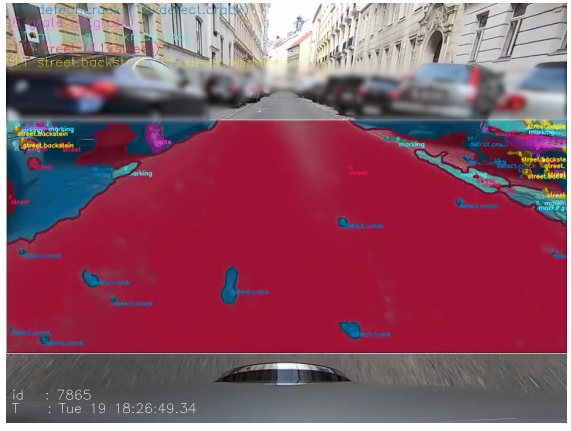
Computer Vision Models in Production

- Who am I?
- Development Environment
- Production Environment
- Deployment Approaches

Who am I?

Computer science student.

Data engineer at MoonVision.



Computer Vision for automotive and food service industry.

Development Environment

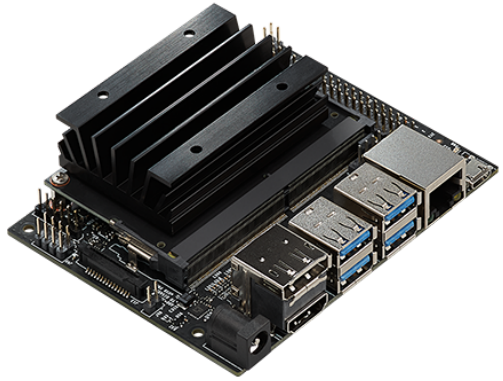
- Python
- Conda
- PyTorch
- Cuda

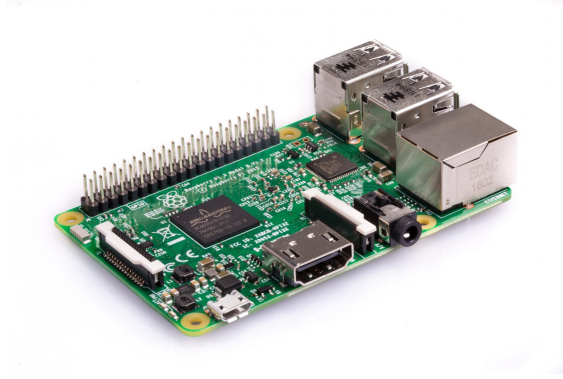
Production Environment

Infrastructure as a Service

- Free choice of deployment environment
- Existing systems
- Scalability

Embedded Systems

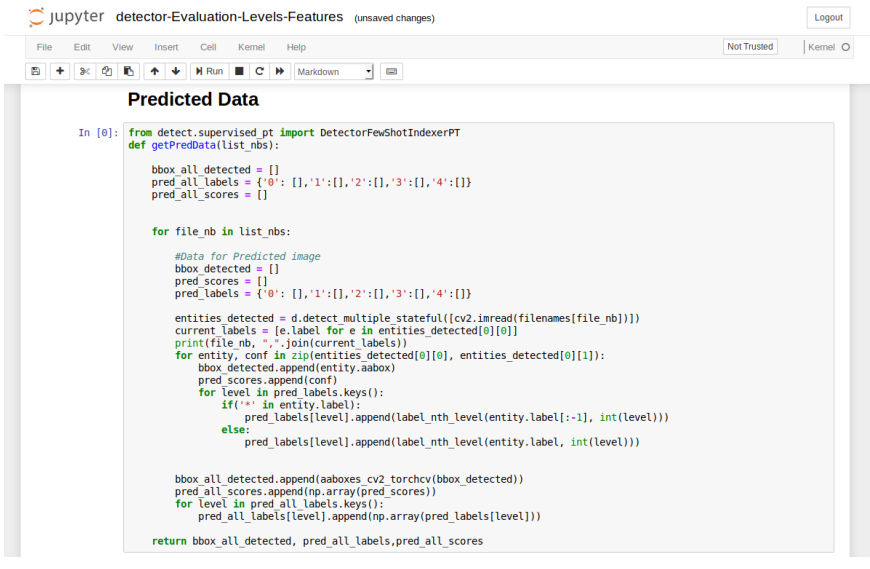


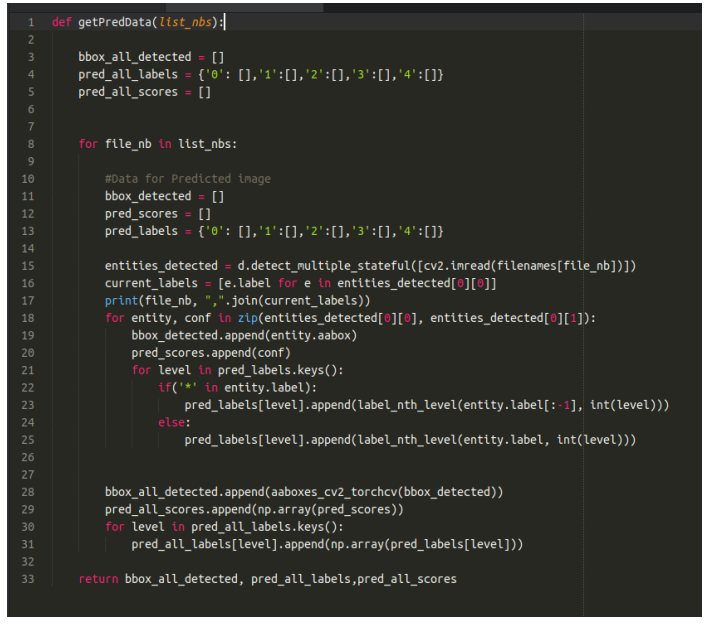


- Often less powerful
- Different processor architectures

Deployment Approaches

Replicate Development Environment in Production





+ Sometimes less complicated
- Slower
- Limited amount of target platforms

ONNX

- Open Neural Network Exchange format
- Defines a computation graph model as well as a set of built-in operators and data types.

Supported by a Wide Range of Vendors

Trace PyTorch model

```
# model = ...
torch.onnx.export(
    model,
    args=torch.randn(10, 3, 224, 224),
    f='ptmodel.onnx')
```

Trace Tensorflow model

```
onnx_graph = tf2onnx.tfonnx.process_tf_graph(
    sess.graph,
    input_names=["input:0"],
    output_names=["output:0"])

model_proto = onnx_graph.make_model("test")
with open("tfmodel.onnx", "wb") as f:
    f.write(model_proto.SerializeToString())
```

Inference with Caffe2

```
model = onnx.load("ptmodel.onnx")
prepared_backend = onnx_caffe2.backend.prepare(model)
prepared_backend(data)
```

Inference with ONNX Runtime

```
sess = rt.InferenceSession("rf_iris.onnx")
input_name = sess.get_inputs()[0].name
label_name = sess.get_outputs()[0].name
pred_onx = sess.run(
    [label_name],
    {input_name: data})[0]
```

ONNX Opsets

- Versions the specified operations
- Version number used for addition of new operations, breaking changes and deprecation alike
- Frameworks decide which Opset versions they support
- Can be difficult to find library versions that work together

TorchScript

Tracing

```
def a_function(a, b):
    return a + b

traced_function = torch.jit.trace(
    a_function,
    (torch.tensor(0), torch.tensor(0))
```

Can also be used to trace entire modules:

```
torch.jit.trace(
    torchvision.models.resnet50().eval(),
    example_inputs=torch.randn(10,3,224,224))
```

Tracing conditions

```
def conditional(x):
    if x > 10:
        x += 10
    return x

trace_a = traced_function = torch.jit.trace(
    conditional, torch.tensor(0))

# returns 5
trace_a(torch.tensor(5))

trace_b = traced_function = torch.jit.trace(
    conditional, torch.tensor(100))

# returns 15
trace_b(torch.tensor(5))
```

Scripting

Subset of Python:

- Statically typed functions
- All return values must be tensors
- Function with `@script` decorator
- `ScriptModule`

Script Function

```
@torch.jit.script
def a_script_function(a, b):
    return a + b
```

Script Module

```
class AScriptModule(torch.jit.ScriptModule):
    def __init__(self):
        pass

    @torch.jit.script_method
    def forward(self, input):
        return input
```

If you want to learn more

ONNX: the long and collaborative road to machine learning portability, https://medium.com/moonvision/a6416a96e870