Optimizer models

```
[ ]
1 model = torch.load("models/torch/resnet18.pth").to('cpu')
2 model.float()
3 model.half()
4 torch.save(model, 'models/torch/resnet18_float16.pth')
```

```
[50]
1 size = get_size_file('models/torch/resnet18.pth')
2 print(f"Size of model resnet18: {size:.1f} MB")
3 size = get_size_file('models/torch/resnet18_float16.pth')
4 print(f"Size of model resnet18_float16: {size:.1f} MB")
✓ 0.0s

Size of model resnet18: 42.7 MB
Size of model resnet18_float16: 21.4 MB
```

```
[51]
1 print(f"Time infer per image of resnet18_float32: {np.mean(time_infer_32bit): .4f} s")
2 print(f"Time infer per image of resnet18_float16: {np.mean(time_infer_16bit): .4f} s")
✓ 0.0s

Time infer per image of resnet18_float32:  0.0121 s
Time infer per image of resnet18_float16:  0.0045 s
```

Điểm Mạnh:

- Dễ chuyển đổi
- Dung lượng mô hình **giảm đi 50%**
- Tốc độ dự đoán **tăng 3 lần**

```python
import torch
import numpy as np

batch_size = [2, 4, 8, 16, 32, 64, 128]

for batch in batch_size:
    with torch.cuda.stream(torch.cuda.current_stream()):
        input = torch.randn(batch, 3, 224, 224)
        input = input.cuda()  # Chuyển input lên GPU bên trong stream
        with torch.no_grad():
            output_32bit = float_32(input)
            output_16bit = float_16(input.half())
    error = np.mean(output_32bit.cpu().numpy() - output_16bit.cpu().numpy())
    print(f"Batch {batch}: {error:.5f}")
```

[53] ✓ 2.5s

```
Batch 2: 0.00063
Batch 4: 0.00045
Batch 8: nan
Batch 16: 0.00043
Batch 32: 0.00065
Batch 64: nan
Batch 128: nan
```

Đây là đoạn code kiểm tra sự sai khác giữa mô hình float32 và float16.

Model float16: Bị NaN ở các batch_size 8, 64, 128
=> **Nó không stable**

Solution:

- Dùng **Mixed Precision Training**
- Dùng **ONNX 16bit**

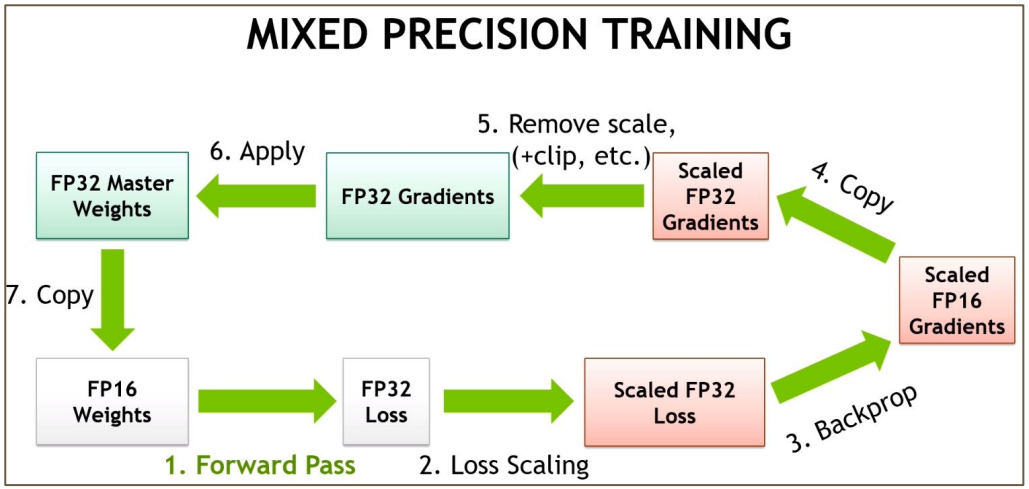
```python
output_16bit
```

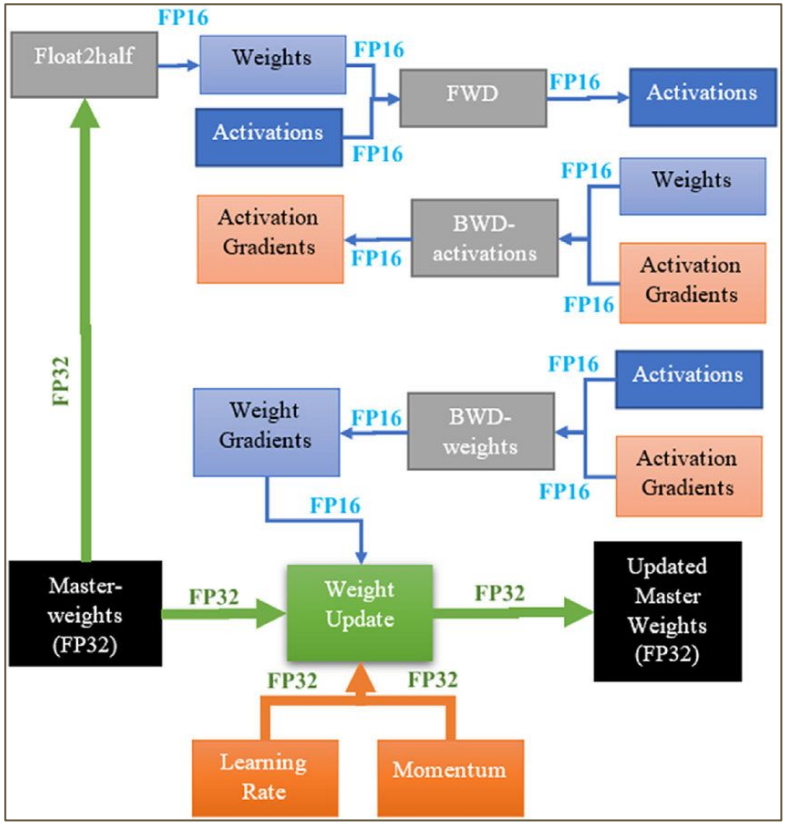
✓ 0.0s

```
tensor([[nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan, nan]], device='cuda:0', dtype=torch.float16
```

1. Convert_pytorch_32bit_to_16bit

Mixed Precision Training





Mixed Precision Training

https://pytorch.org/docs/stable/notes/amp_examples.html

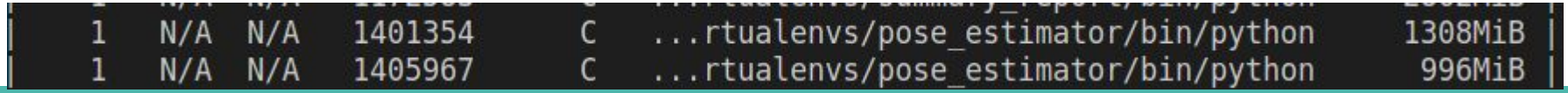
Ưu điểm:

- **Giảm dung lượng GPU yêu cầu cho các phép tính:** Do các phép tính diễn ra trên fp16 nên dung lượng tính toán giảm nhiều.
- **Hiệu quả trên một số phần cứng hiện đại :** Một số phần cứng GPU Nvidia, được tối ưu hóa cho các phép tính fp16, cho kết quả infer nhanh hơn.
- **Tăng tốc độ huấn luyện:** Do phần forward tính toán trên fp16 nên việc tính toán nhanh.

Nhược điểm:

- **Cần nhiều dung lượng GPU:** do việc nó cần tạo ra hai bản sao cho fp32 và fp16.
- **Không phải GPU Nvidia nào hỗ trợ tốt fp16:** một số phần cứng chạy trên fp32 nhanh hơn fp16.

Nhận xét: Nên dùng MPT khi muốn training nhanh mô hình, trong điều kiện tài nguyên thoải mái.



Mixed Precision Training

```python
import torch
from torch.cuda.amp import GradScaler, autocast

dataloader = ... # Tạo dataloader
epochs = ... # Số epoch cần huấn luyện
criterion = ... # Khởi tạo hàm loss
model = ... # Khởi tạo mô hình
optimizer = ... # Khởi tạo trình tối ưu hóa
scaler = GradScaler() # Khởi tạo scaler cho mixed precision

for epoch in range(epochs):
    for batch in dataloader:
        inputs, targets = batch
        inputs, targets = inputs.cuda(), targets.cuda()

        optimizer.zero_grad()

        with autocast():
            outputs = model(inputs)
            loss = criterion(outputs, targets)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
```

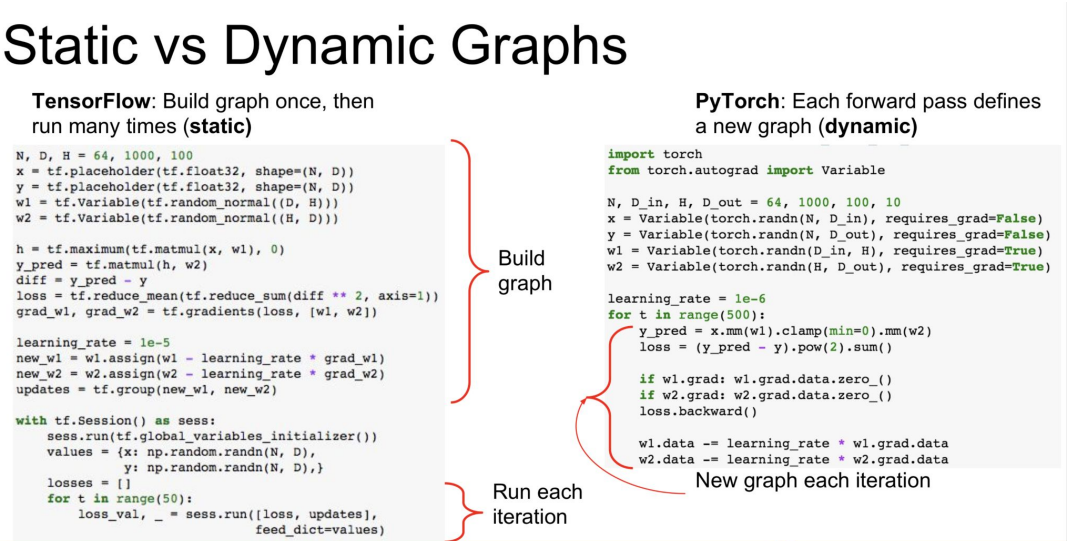
```python
import torch

dataloader = ... # Tạo dataloader
epochs = ... # Số epoch cần huấn luyện
criterion = ... # Khởi tạo hàm loss
model = ... # Khởi tạo mô hình
optimizer = ... # Khởi tạo trình tối ưu hóa
scheduler = ... # Khởi tạo scheduler

for epoch in range(epochs):
    for batch in dataloader:
        inputs, targets = batch
        inputs, targets = inputs.cuda(), targets.cuda()

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

    scheduler.step() # Cập nhật learning rate sau mỗi epoch
```

Do tính toán fp16 dễ bị tràn số (overflow computing), dùng một cái scaler để chỉnh gradient trước khi cập nhật trọng số , giúp quá trình cập nhật ổn định hơn.

JIT (Just in Time)

JIT trong **TorchScript** là quá trình biến đổi một mô hình Pytorch từ dạng **đồ thị tính toán động (Dynamic Computation Graph)** sang dạng **đồ thị tính toán tĩnh (Static Computation Graph)**.

- Dynamic Computation Graph “define-by-run": chỉ có pytorch dùng
- Static Computation Graph “define-and-run" : Tensorflow, TensorRT, TorchScript, ONNX




```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Torchscript

```python
import torch
# Load models pytorch
model_pytorch_cuda = torch.load("models/torch/resnet18.pth").to('cuda')
# convert model to static graph
sample_input_cuda = torch.randn(1,3,224,224).cuda()
traced_cuda = torch.jit.trace(model_pytorch_cuda, sample_input_cuda)
torch.jit.save(traced_cuda, "cuda.pt")


# Load jit model
model_jit = torch.jit.load("cuda.pt")

# Inference model
model_jit(sample_input_cuda)
```

[41] ✓ 0.4s

```
tensor([[ 0.3873,  2.0765, -0.4736, -6.3287,  2.5076, -3.1566]],
       device='cuda:0', grad_fn=<AddmmBackward0>)
```

Torchscript

```python
import torch
# Load models pytorch
model_pytorch_cuda = torch.load("models/torch/resnet18.pth").to('cuda')
# convert model to static graph
sample_input_cuda = torch.randn(1,3,224,224).cuda()
traced_cuda = torch.jit.trace(model_pytorch_cuda, sample_input_cuda)
torch.jit.save(traced_cuda, "cuda.pt")


# Load jit model
model_jit = torch.jit.load("cuda.pt")

# Inference model
model_jit(sample_input_cuda)
```

[43] ✓ 0.4s

```
tensor([[-0.3191,  2.0507, -0.0977, -6.1900,  3.0482, -3.3211]],
       device='cuda:0', grad_fn=<AddmmBackward0>)
```

```python
# Một cách khác, nhưng cách này giúp tối ưu, nhưng không lưu được mô hình
frozen_mod = torch.jit.optimize_for_inference(torch.jit.script(model_pytorch_cuda.eval()))
frozen_mod(sample_input_cuda)
```

[44] ✓ 0.2s

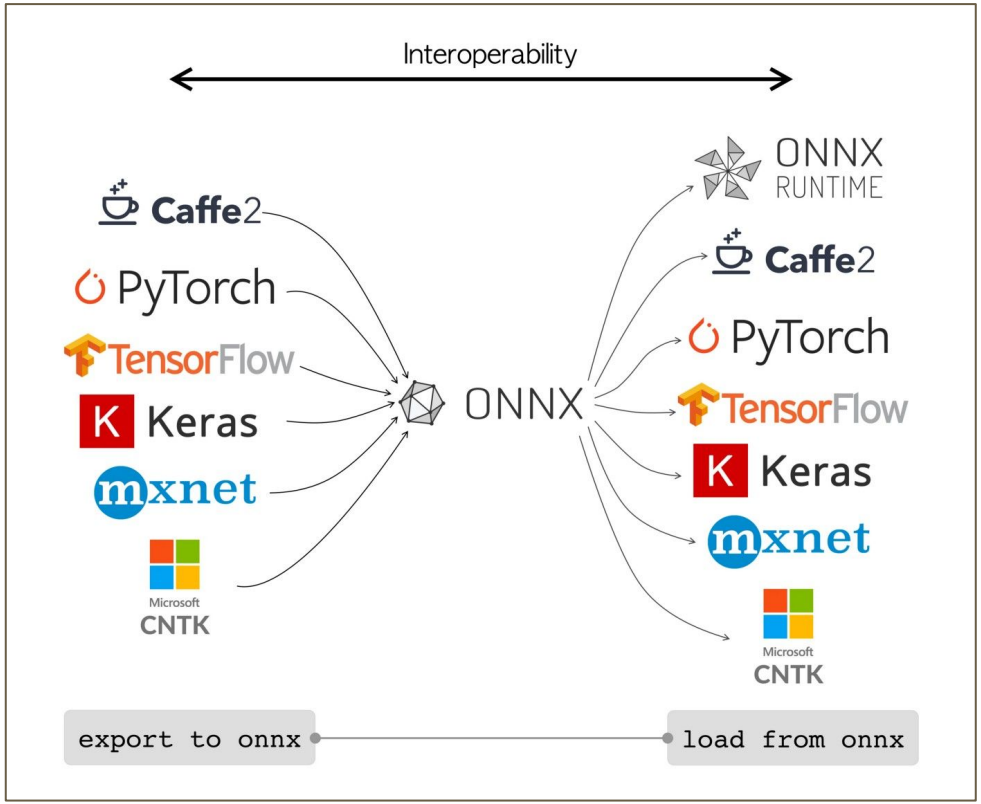
```
tensor([[-0.3191,  2.0507, -0.0977, -6.1900,  3.0482, -3.3211]],
       device='cuda:0')
```

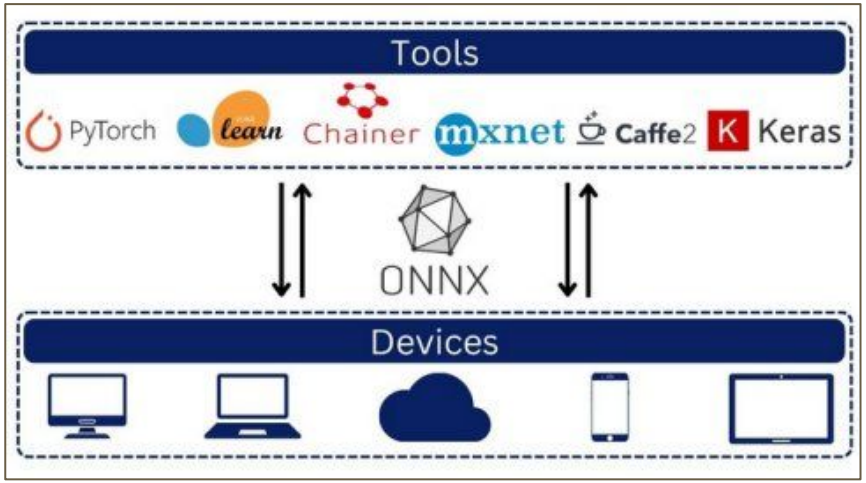
Model: Resnet18

Pytorch (GPU): **0.0240 ms**
Torchscript (GPU): 0.0313 ms
Torchscrip_Optimizer (GPU): 0.0287 ms

Pytorch (CPU): 0.4395 ms
Torchscript (CPU): 0.3081 ms
Torchscript_Optimizer (CPU): **0.1935 ms**

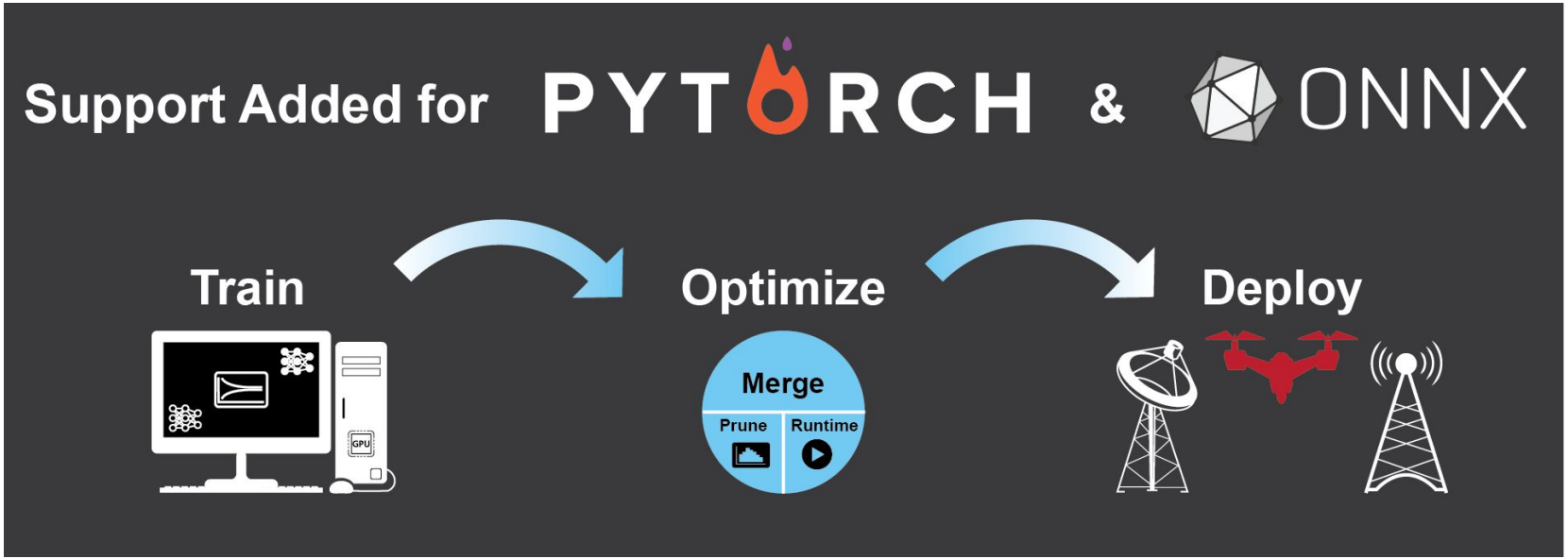
ONNX (Open Neural Network Exchange)





- Là một framework
- Giúp chuyển đổi giữa các framework với nhau
- Tối ưu hóa mô hình và triển khai dễ dàng trên máy tính, di động, thiết bị nhúng.

ONNX (Open Neural Network Exchange)



Pytorch to ONNX

```python
def convert_pytorch2onnx(model, input_samples, path_onnx, mode = 'float32bit', device = 'cuda'):
    if mode == 'float16bit':
        print("float16bit")
        model.float()
        model.half()  # Chuyển mô hình sang float16
        input_samples = input_samples.half()
    model.to(device)
    model.eval()
    input_samples = input_samples.to(device)
    torch.onnx.export(
        model, # Model load bằng pytorch
        input_samples, # input với kích thước mong muốn
        path_onnx, # path_onnx bạn lưu
        verbose=False,  # Hiển thị thông báo trong quá trình chuyển đổi
        opset_version=12, # Đọc xem mỗi version sẽ có hỗ trợ onnx cho lớp nào
        do_constant_folding=True , # Dùng Constant-folding optimizer giúp cải thiện tốc độ và dung lượng
        input_names = ['images'],   # the model's input names
        output_names = ['output'], # the model's output names
        dynamic_axes={
                'images' : {0 : 'batch_size',},
                'output' : {0 : 'batch_size'}
            }
        )
```

✓ 0.0s

ONNX Runtime

ONNX Runtime: Là một trình thực thi mô hình Deep Learning

- Đa nền tảng: Chạy được trên Linux, Windows, MacOS
- Tối ưu hóa hiệu suất trên VPU (Thiết bị lượng tử), GPU, CPU
- API đa ngôn ngữ: C, Python, C++, Java
- Tích hợp với các công nghệ khác: TensorRT, openVINO

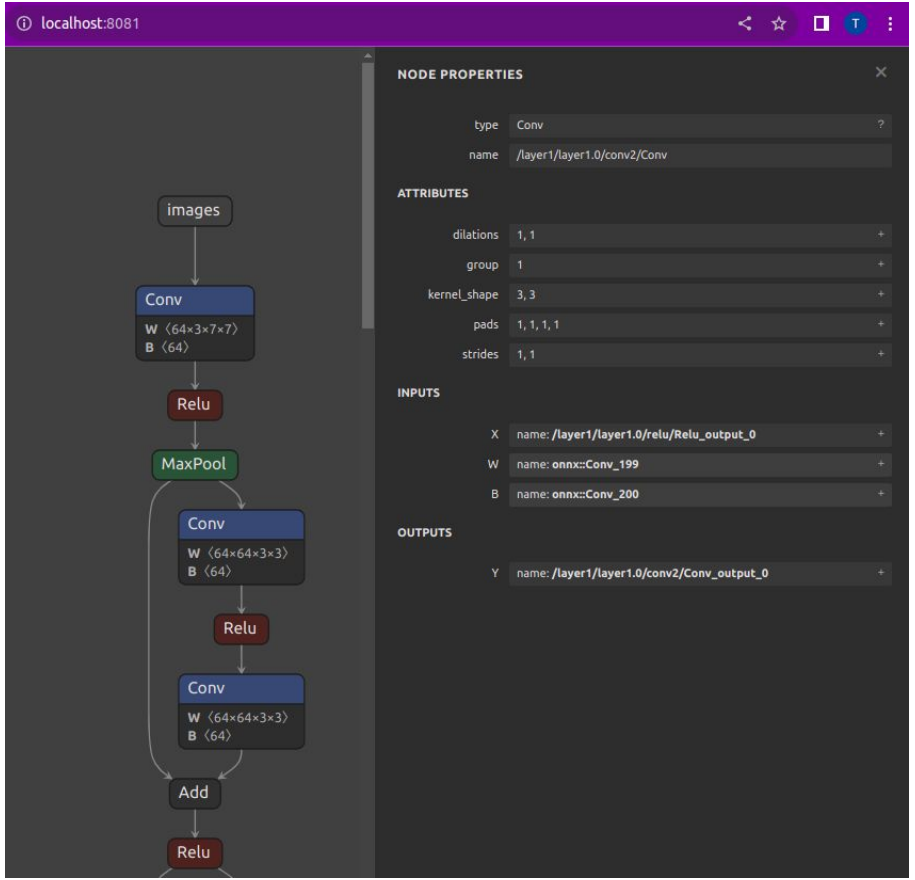
```
24
25  def load_onnx_model(path_onnx, providers=['CUDAExecutionProvider', 'CPUExecutionProvider']):
26      # Create an ONNX Runtime inference session for the ONNX model
27      ort_session = onnxruntime.InferenceSession(
28          path_onnx,
29          providers=providers
30          )
31      return ort_session
32
33  def onnx_infer(ort_session, input_data):
34      ort_inputs = {ort_session.get_inputs()[0].name: input_data}
35      ort_output = ort_session.run(None, ort_inputs)
36      return ort_output
```

ONNX Visualization

```
#!pip install netron
import netron
netron.start('models/onnx/vgg19_float32bit.onnx', 8081)
```

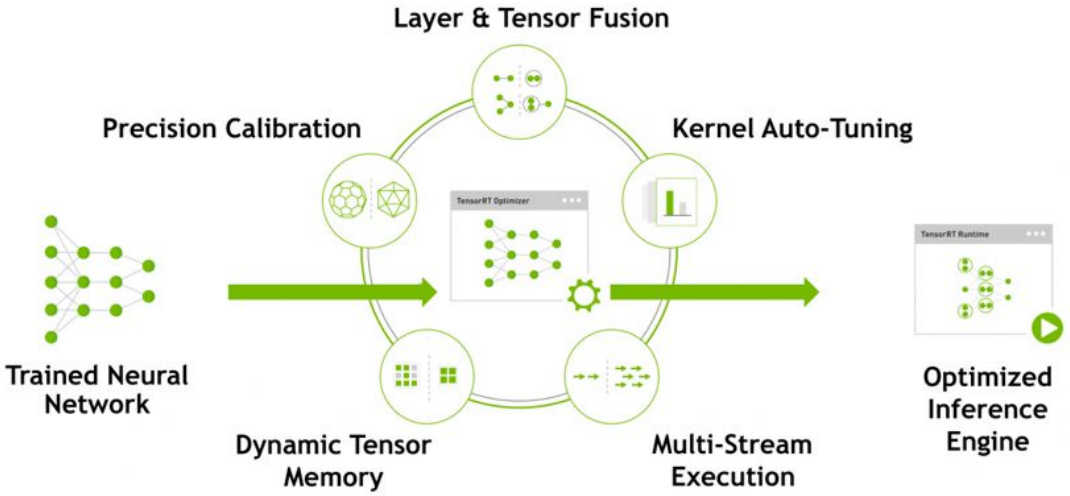
[38] ✓ 0.5s

```
Serving 'models/onnx/vgg19_float32bit.onnx' at http://localhost:8081

('localhost', 8081)
```

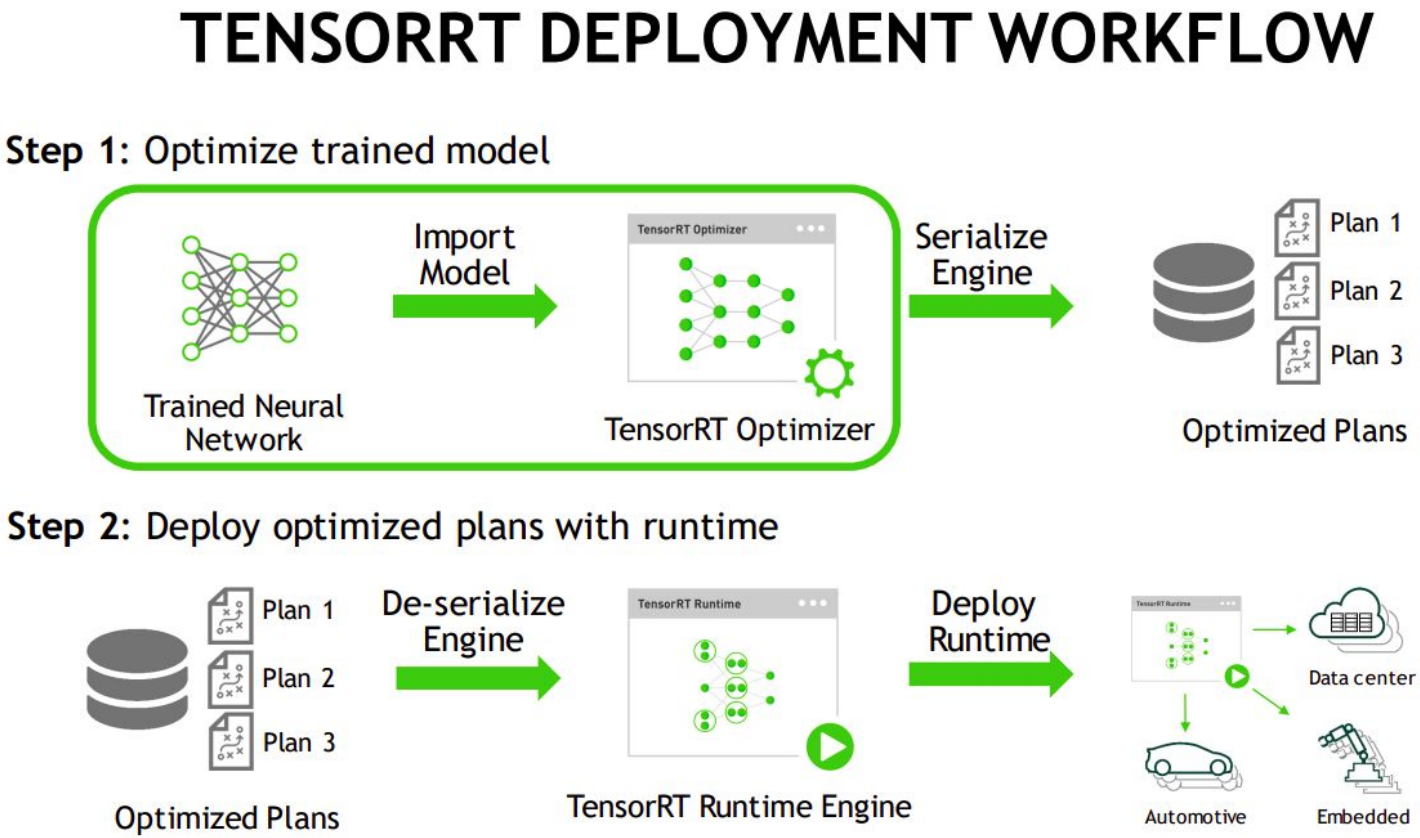


TensorRT (Tensor Run Time)

TensorRT là high-performance deep learning cho inference model. Đặc biệt chuyên tối ưu hóa hiệu suất của mô hình trên GPU của NVIDIA.
TensorRT giúp **Pruning** (Loại bỏ những trọng số không cần thiết), **Quantization** (Giảm thiểu số lượng bit biểu diễn trọng số) và **Fusion** (Ghép các phép tính lại với nhau).



TensorRT in Deployments



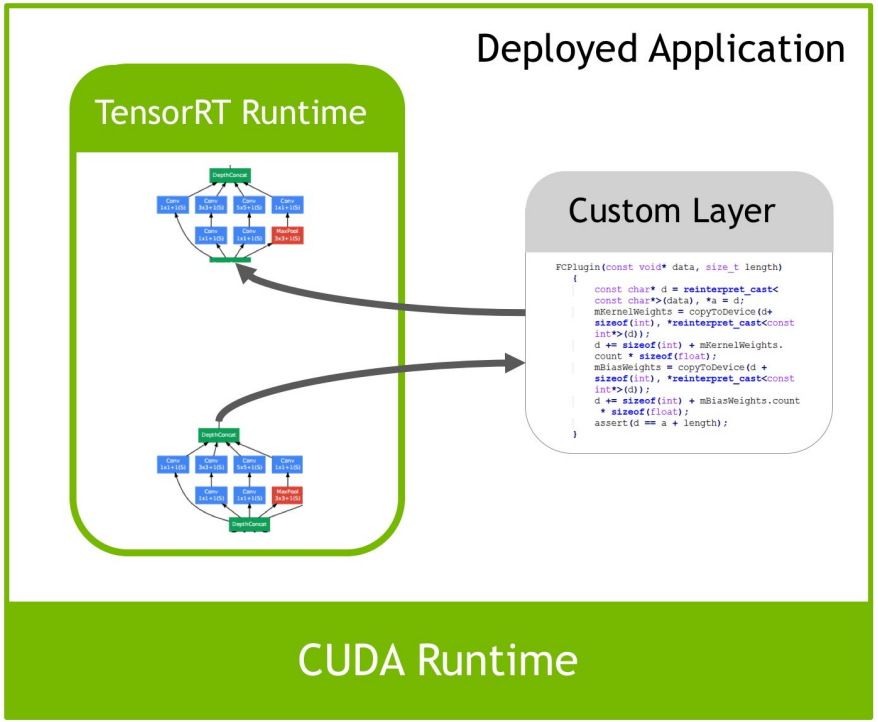
TensorRT in Deployments

TensorRT supported layers:

- Convolution
- LSTM and GRU
- Activation: ReLU, Tanh, Sigmoid
- Pooling: max and average
- Scaling
- Element wise operations
- LRN
- Fully-connected
- Softmax
- Deconvolution

TensorRT cung cấp Custom Layer API.

- Dùng C++ để code cuda để định nghĩa laye



https://github.com/ndtands/Speed_up_Model