CVPR
SEATTLE, WA JUNE 17-21, 2024

intel.

OpenVINO™
CVPR 2024

# Edge-Optimized Deep Learning: Harnessing Generative AI and Computer Vision with Open-Source Libraries

# Module 3: Optimization with NNCF for Computer Vision and Gen AI


Alexander Kozlov

OpenVINO Low-precision Architect

Contributors: Paula Ramos, PhD and Raymond Lo, PhD

NNCF Team: Andrey Anufriev, Andrey Churkin, Alexander Dokuchaev, Aleksei Kashapov, Daniil Lyakhov, Nikolay Lyalyushkin, Nikita Malinin, Maksim Proshin, Nikita Savelyev, Vasily Shamporov, Alexander Suslov, Liubov Talamanova, Aidova Ekaterina

# Outline

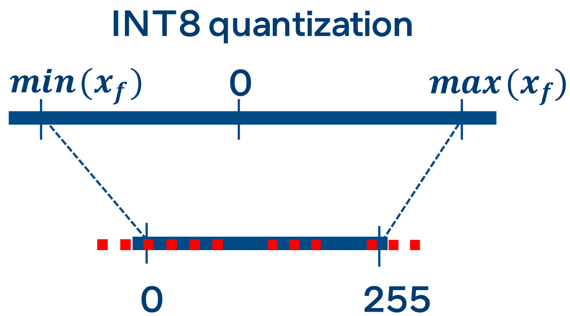
| | |
|---|---|
| • Computer Vision model optimization | 10 minutes |
| • General aspects of Gen.AI model optimization | 20 minutes |
| • Environment setup | 5 minutes |
| • LLM optimization | |
|     • 8-bit weight quantization | 10 minutes |
|     • 4-bit mixed-precision weight quantization | 10 minutes |
|     • Full quantization of CLIP | 10 minutes |
| • Diffusion pipeline optimization | |
|     • Full quantization of UNet in SD1.x-2.x | 10 minutes |
|     • Hybrid quantization | 10 minutes |
| • Conclusion and plans | 5 minutes |

# Computer Vision Model Optimization

- Why optimization is so important for deployment?
- DNN optimization methods
  - Pruning/Sparsity
  - Neural Architecture Search
  - Distillation
  - Quantization – easy to go!

# Quantization: concept

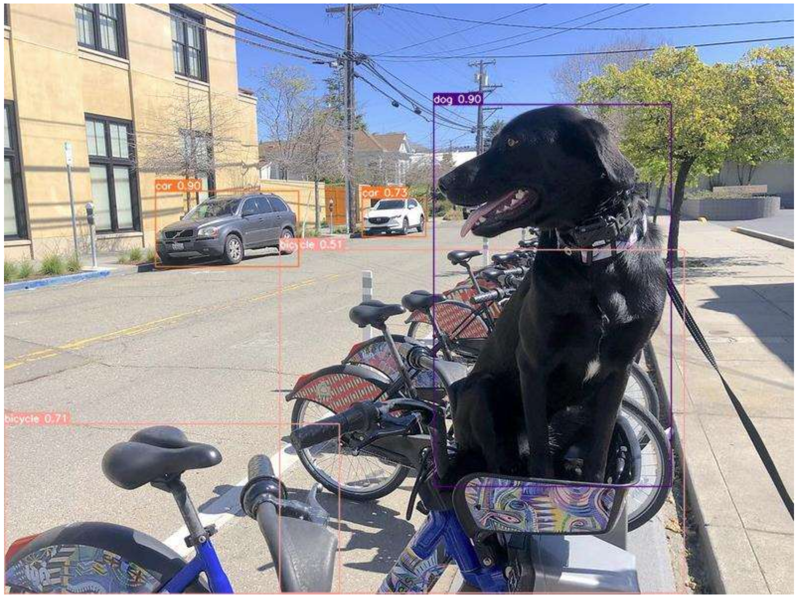
- Easy concept but lots of tricks
- Formula:
  - $y_{f32} = w_{f32} \times a_{f32}$, where $\times$ is dot product
  - Let's find scales so that $w_{f32} = s^{w}_{f32} \cdot w_{i8}$ and $a_{f32} = s^{a}_{f32} \cdot a_{i8}$
  - $s^{w}_{f32}$ and $s^{a}_{f32}$ are estimated for each layer
  - Now: $y_{f32} = w_{f32} \times a_{f32} = s^{w}_{f32} \cdot w_{i8} \times s^{a}_{f32} \cdot a_{i8} = s^{w}_{f32} \cdot s^{a}_{f32} \cdot w_{i8} \times a_{i8}$
  - Thus, we can compute $w_{i8} \times a_{i8}$ in a 8-bit precision more efficiently



# Quantization for CV Models

- Post-training quantization:
  - Good performance speedup (up to 4x) at small accuracy drop
  - Fast and easy-to-use
  - Can be even data-free
  - No fine-tuning and training HW is required

- Quantization-aware training (QAT)
  - Model fine-tuning w/ quantization simulation
  - Provides a better accuracy than PTQ



# OpenVINO™ & NNCF Quantization and Low-precision Inference

- **Neural Network Compression Framework (NNCF) is a part of the OpenVINO ecosystem to optimize models for deployment**
- **8-bit Post-training quantization (PTQ) API of NNCF**

```
from torchvision import datasets

dataset = datasets.ImageFolder(dataset_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False)

def transform_fn(data_item):
    images, _ = data_item
    return images

calibration_dataset = nncf.Dataset(dataloader, transform_fn)

quantized_model = nncf.quantize(model, calibration_dataset) # model is openvino.Model, ONNX, torch.nn.Module
```

- **Quantized model can be exported/saved and run with OpenVINO**
- **There is [Accuracy-aware](#) variant of PTQ that allows to control accuracy drop**

# Quantization-aware training API

- Workflow:
  - Quantize model the same way as in case of PTQ
  - Tune quantized model for a couple of epochs to restore accuracy
  - Available for PyTorch and TensorFlow

```
… # prepare datasets for PTQ and fine-tuning, optimizer and loss
quantized_model = nncf.quantize(model, calibration_dataset) # model is torch.nn.Module

# Fine-tune model to restore accuracy
for epoch in range(epochs):
    for images, target in train_loader:
        images = images.to(device)
        target = target.to(device)

        output = quantized_model(images)
        loss = criterion(output, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

  - [QAT example](#) for PyTorch

# Quantization of Gen.AI models

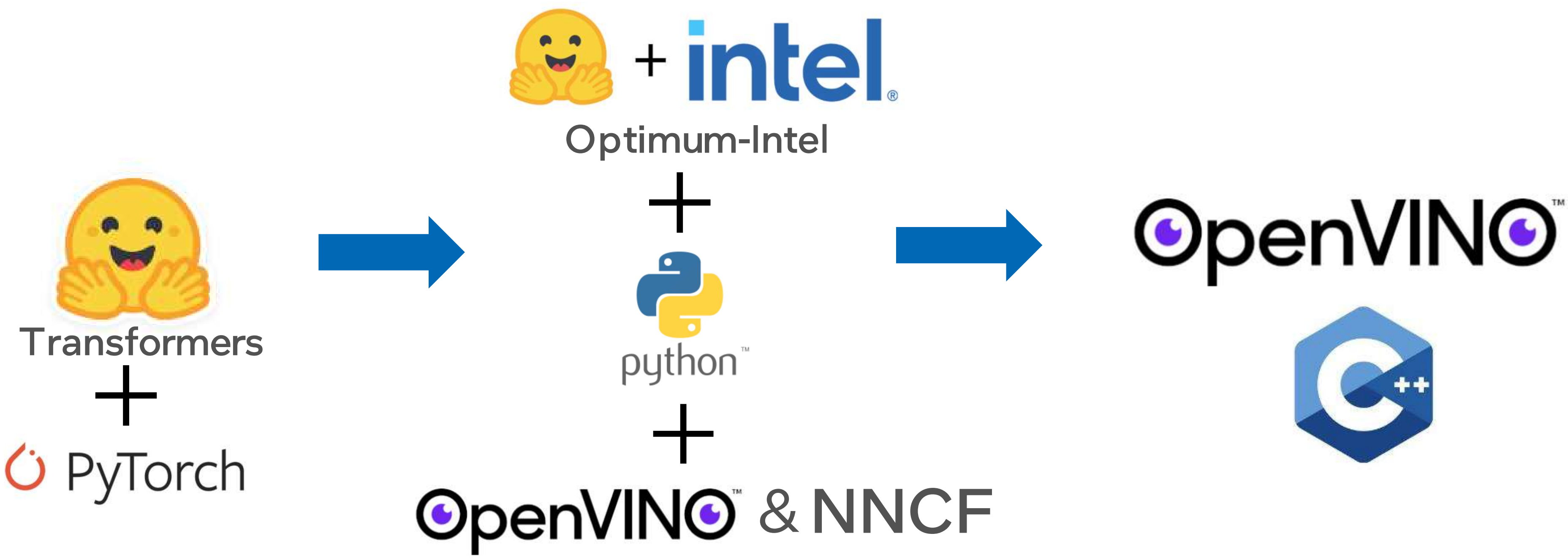
# General Aspects of Gen.AI Model Optimization

- Gen.AI models:
  - Large number of parameters (billions)
  - Trained on a large number of examples (trillions)
  - They often are memory-bound (LLMs) – data dominates on top of computations
- Optimization methods
  - Pruning/Sparsity
  - Neural Architecture Search
  - Distillation
  - Quantization – works!

# Quantization for Gen.AI

- Popular methods:
  - Quantization-aware training – hard to apply to Gen.AI due to the risk of overfitting to small datasets
  - Post-training quantization:
    - Full model quantization – activations are usually more error-prone when being quantized
    - Weight-only quantization
- Weight-only quantization
  - Reduces disk and memory footprint of the model
  - Shifts model from being memory-bound to compute-bound
  - Allows preserving accuracy after quantization
  - 8/4/3/2 bit and even 1.+ bit weight quantization are popular for Gen.AI models

# OpenVINO and NNCF Workflow for Gen.AI



# NNCF and OpenVINO features for Gen.AI

- Full 8-bit model quantization (ViT, CLIP, etc.)
  - Transformer scheme for quantization
  - SmoothQuant method
  - Bias correction
- 8/4 bit weights quantization (Diffusers, LLMs)
  - Data-aware precision selection
  - Activation-aware Weight Quantization (AWQ)
- Hybrid quantization (Diffusers)
  - Full quantization of Conv layers and Weight-only quantization of Transformer blocks
- Dynamic 8-bit quantization of activations (Diffusers, LLMs)
- KV-cache 8-bit quantization

# Native NNCF API for model optimization

- **INT8 compression (weights only)**

```
model = nncf.compress_weights(model, mode=nncf.CompressWeightsMode.INT8_SYM) # model is ov.Model
```

- **INT4 data-free (weights only)**

```
model = nncf.compress_weights(model, mode=nncf.CompressWeightsMode.INT4_ASYM, ratio=0.8, group_size=64)
```

- **INT4 data-aware (weights only)**

```
calibration_dataset = nncf.Dataset(dataset, transform_fn)

model = nncf.compress_weights(model, mode=nncf.CompressWeightsMode.INT4_ASYM,
                              sensitivity_metric=nncf.SensitivityMetric.HESSIAN_INPUT_ACTIVATION,
                              awq=True, scale_estimation=True,
                              dataset=calibration_dataset)
```

**Note: this API also comes with AWQ and scale tuning methods that can be used stacked for better accuracy**

- **W8A8 quantization**

```
calibration_dataset = nncf.Dataset(dataset, transform_fn)

model = nncf.quantize(model, dataset=calibration_dataset, model_type=nncf.ModelType.TRANSFORMER)
```

# Optimum-Intel API on top of NNCF

- **INT8 weight quantization of LLMs**

```
model = OVModelForCausalLM.from_pretrained(MODEL_ID, quantization_config=dict(bits=8))
```

- **INT4-INT8 data-free weight quantization of LLMs**

```
model = OVModelForCausalLM.from_pretrained(MODEL_ID, quantization_config=dict(bits=4, ratio=0.8))
```

- **INT4-INT8 data-aware weight quantization of LLMs**

```
model = OVModelForCausalLM.from_pretrained(
    MODEL_ID,
    quantization_config=dict(bits=4, awq=True, ratio=0.8, dataset="ptb")
)
```

- **Hybrid quantization of Stable Diffusion pipeline**

```
model = OVStableDiffusionPipeline.from_pretrained(
    MODEL_ID,
    quantization_config=dict(bits=8, dataset="conceptual_captions")
)
```

# Practical part

# Setup Environment

- Create and activate a new environment:

```
$ python -m venv openvino
$ source openvino/bin/activate
```

- Install Anomalib + Optimum-Intel with OpenVINO and NNCF dependencies:

```
$ pip install openvino nncf
$ pip install anomalib[core]==1.0.0
$ pip install diffusers
$ pip install git+https://github.com/huggingface/optimum.git
$ pip install git+https://github.com/huggingface/optimum-intel.git
```

# QAT of STFPM PyTorch model from Anomalib

- Quantization of Student-Teacher Feature Pyramid Matching (STFPM) PyTorch model from Anomalib with NNCF QAT:
  - Load dataset
  - Load pre-trained baseline model check-point
  - Apply quantization
  - Fine-tune the model to restore accuracy
  - Compare accuracy/performance vs. baseline model

**GPU is required to run example*

# LLM: 8-bit Weight-only Quantization

```
from transformers import AutoTokenizer
from optimum.intel import OVModelForCausalLM
import openvino as ov

MODEL_ID = "microsoft/Phi-3-mini-4k-instruct"

model = OVModelForCausalLM.from_pretrained(MODEL_ID, export=True, quantization_config=dict(bits=8),
trust_remote_code=True)
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID, trust_remote_code=True)

template = "<|user|>\n{}<|end|>\n<|assistant|>"
question = "Hey, model! How are you today?"
prompt = template.format(question)
inputs = tokenizer(prompt, return_tensors="pt")
output = model.generate(**inputs, max_new_tokens=256)
answer = tokenizer.batch_decode(output, skip_special_tokens=True)[0]

print(answer)
```

**Try [this code](#) in your script/notebook*

# LLM: 4-bit Data-aware Weight-only Quantization

```
from transformers import AutoTokenizer
from optimum.intel import OVModelForCausalLM
import openvino as ov

MODEL_ID = "microsoft/Phi-3-mini-4k-instruct"

model = OVModelForCausalLM.from_pretrained(
    MODEL_ID,
    export=True,
    compile=False,
    quantization_config=dict(bits=4, sym=True, ratio=0.8, dataset="ptb"),
    trust_remote_code=True
)
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID, trust_remote_code=True)

template = "<|user|>\n{}<|end|>\n<|assistant|>"
question = "Hey, model! How are you today?"
prompt = template.format(question)

inputs = tokenizer(prompt, return_tensors="pt")
output = model.generate(**inputs, max_new_tokens=256)
answer = tokenizer.batch_decode(output, skip_special_tokens=True)[0]

print(answer)
```

**Try [this code](#) in your script/notebook*

# LLM: Dynamic Quantization and KV-cache Quantization

```python
from transformers import AutoTokenizer
from optimum.intel import OVModelForCausalLM
import openvino as ov

MODEL_ID = "microsoft/Phi-3-mini-4k-instruct"

model = OVModelForCausalLM.from_pretrained(
    MODEL_ID,
    export=True,
    compile=False,
    quantization_config=dict(bits=4, sym=True, ratio=0.8, dataset="ptb"),
    ov_config={"KV_CACHE_PRECISION": "u8", "DYNAMIC_QUANTIZATION_GROUP_SIZE": "32", "PERFORMANCE_HINT": "LATENCY"},
    trust_remote_code=True
)
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID, trust_remote_code=True)

template = "<|user|>\n{}<|end|>\n<|assistant|>"
question = "Hey, model! How are you today?"
prompt = template.format(question)

inputs = tokenizer(prompt, return_tensors="pt")
output = model.generate(**inputs, max_new_tokens=256)
answer = tokenizer.batch_decode(output, skip_special_tokens=True)[0]

print(answer)
```

**Try [this code](#) in your script/notebook*

# Full Quantization of CLIP Model

- CLIP is an important part of various pipelines (including Stable Diffusion)
- The image encoder of the CLIP pipeline can be fully quantized (weights and activations) to 8 bits to speed up the inference
- SmoothQuant method (w/ tuned parameter) is used to reduce quantization error
- Example in OpenVINO Notebooks

# Stable Diffusion: UNet Model Quantization

- UNet is the essential part of the Stable Diffusion pipeline which consumes most of the inference time
- Some Stable Diffusion models allow quantizing UNet to speed up the inference, e.g. SD Latent Consistency Model (LCM)
- LCM is trained to be resistant to perturbations. Thus, quantizing weights and activations does not drop the accuracy
- LCM Example in OpenVINO Notebooks

# Hybrid Quantization of Stable Diffusion

- Some parts of SD Pipeline are sensitive to quantization, e.g. decoder
- Hybrid approach allows keeping some sensitive parts unquantized:
  - UNet model:
    - Conv layers – are fully quantized
    - Transformer blocks – only weights are quantized
  - Other models (text/image encoders, decoder)
    - Weight-only quantization
- SD Example in the Hugging Face Optimum-Intel

# Useful Links

- Weight compression examples
- Hugging Face Optimum-Intel project
- OpenVINO GenAI C++ samples
- LLM benchmarking
- Quick LLM evaluation project

# Thank You