

DINOv2.cpp

Alex Lavaee, Zachary Gentile, Michael Krah

May 5th, 2025

Abstract

In recent years, interest in edge computing for AI has grown rapidly. AI algorithms can be deployed on edge devices, such as smartphones, personal computers, and IoT devices, enabling a wider range of use cases. To support this growing need, we introduce dinov2.cpp, an inference engine for the DINOv2 family of vision transformer models designed for efficient inference on edge hardware. This inference engine was built in C++ using the ggml library. It achieves improved inference speeds and lower memory use compared to existing PyTorch implementations.

Introduction

Edge computing for AI involves the deployment of AI algorithms on local edge devices and sensors, which enables a wider range of AI use cases and provides substantial benefit. Edge computing allows for AI algorithms to be run locally on devices with limited hardware, ensuring devices do not need to be connected to the cloud for real-time inference. Some benefits provided include reduced memory and energy use, decreased latency, and faster compute. These also provide privacy benefits as data can be stored locally. Use cases include real-time processing for autonomous vehicles, analytics for security devices, and improved sensor understanding for robotics.

The Vision Transformer (ViT) architecture has achieved state-of-the art results on a wide range of traditional computer vision problems, pushing the capabilities of image analysis [1]. Among recent innovations, the DINOv2 family of vision transformer models achieves competitive results in classification, depth estimation, and semantic segmentation tasks. Developed by Meta, pretrained weights and model architectures have been made publicly available on GitHub [3]. Despite their improved performance, ViT models are more computationally expensive than convolutional neural networks. The ViT self-attention mechanism scales quadratically with input size, leading to increased memory usage and slower inference times. As a result, deployment of ViTs on systems with low resources can prove challenging and can fail to provide competitive speeds.

We introduce DINOv2.cpp, an inference engine for the DINOv2 family of models for use on edge devices. Written in C++ with minimal dependencies, our lightweight implementation achieves lower memory use and improved inference speeds on all models.

Related Work

vit.cpp [4]

Our work is inspired by vit.cpp, a C++ implementation of the ViT inference engine using the ggml library.

DINOv2 [3]

DINOv2 is a self-supervised vision model that trains Vision Transformers on a massive 142 million-image dataset. It combines two key objectives: aligning global representations through class-token loss and fine-graining features via masked patch-token loss. The model also integrates techniques like Sinkhorn-Knopp for clustering and KoLeo regularization for uniform feature distribution. Additionally, DINOv2 includes a high-resolution adaptation stage and produces dense patch embeddings to capture local image semantics.

Approach

We build on the existing vit.cpp codebase, updating and refactoring the implementation as required to support DINOv2. Steps taken involved writing a script to convert the model to GGUF format, loading model weights into ggml, rewriting the forward pass to support new layers, and updating quantization support. In addition to these steps, we upgraded the GGML version for the base implementation of vit.cpp. Following the initial implementation of the inference pipeline, we added support for realtime visualization of extracted features, flash attention, and positional embedding interpolation.

GGUF Format [2]

GGUF is a binary format that is optimized for quick loading and saving of models, making it highly efficient for inference purposes. GGUF is designed for use with GGML and other executors. A Python script is provided in vit.cpp to download a ViT model from HuggingFace and then convert it to a deprecated GGML format. We rewrote this script to account additional layers present in DINOv2, such as the register tokens, and to use the updated GGUF format.

Correcting for the DINOv2 Architecture

Adapting the vit.cpp pipeline involved rewriting the model loading and forward pass to handle the modified DINOv2 architecture. Memory must be manually allocated for each tensor and account for the variable size of different models. The GGUF file format provides metadata on the loaded model architecture allowing the inference pipeline to account for different model sizes and architectures. The layer structure of the model is detailed in Figure 1.

```
Processing variable: dinov2.embeddings.cls_token with shape: torch.Size([1, 1, 384]) and type: torch.float32
Processing variable: dinov2.embeddings.mask_token with shape: torch.Size([1, 384]) and type: torch.float32
Processing variable: dinov2.embeddings.position_embeddings with shape: torch.Size([1, 1370, 384]) and type: torch.float32
Processing variable: dinov2.embeddings.patch_embeddings.projection.weight with shape: torch.Size([384, 3, 14, 14]) and type: torch.float32
Processing variable: dinov2.embeddings.patch_embeddings.projection.bias with shape: torch.Size([384]) and type: torch.float32

Processing variable: dinov2.encoder.layer.0.norm1.weight with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.norm1.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.attention.query.weight with shape: torch.Size([384, 384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.attention.query.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.attention.key.weight with shape: torch.Size([384, 384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.attention.key.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.attention.value.weight with shape: torch.Size([384, 384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.attention.value.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.output.dense.weight with shape: torch.Size([384, 384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.attention.output.dense.bias with shape: torch.Size([384]) and type: torch.float32

Processing variable: dinov2.encoder.layer.0.layer_scale1.lambda1 with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.norm2.weight with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.norm2.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.mlp.fc1.weight with shape: torch.Size([1536, 384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.mlp.fc1.bias with shape: torch.Size([1536]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.mlp.fc2.weight with shape: torch.Size([384, 1536]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.mlp.fc2.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.encoder.layer.0.layer_scale2.lambda1 with shape: torch.Size([384]) and type: torch.float32
.
.
.
Processing variable: dinov2.encoder.layer.11.layer_scale2.lambda1 with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.layernorm.weight with shape: torch.Size([384]) and type: torch.float32
Processing variable: dinov2.layernorm.bias with shape: torch.Size([384]) and type: torch.float32
Processing variable: classifier.weight with shape: torch.Size([1000, 768]) and type: torch.float32
Processing variable: classifier.bias with shape: torch.Size([1000]) and type: torch.float32
```

Figure 1: An example of DINOv2’s model architecture information from our GGUF conversion script. It provides to us the exact name of each layer as well as its size and type, which is all the information that we need to exactly match while loading the model’s weights and building the forward pass.

Visualizing DINOv2’s Feature Extraction

dinov2.cpp supports parameters to enable classification or feature extraction and visualization. When visualizing features, the classification head is removed and the learned feature representation of the image will be output. To produce a visualization of features, principal component analysis (PCA) is run on the last layer of weights. OpenCV is used to convert and save the resulting image. Figure 2 shows an example of this visual feature representation.

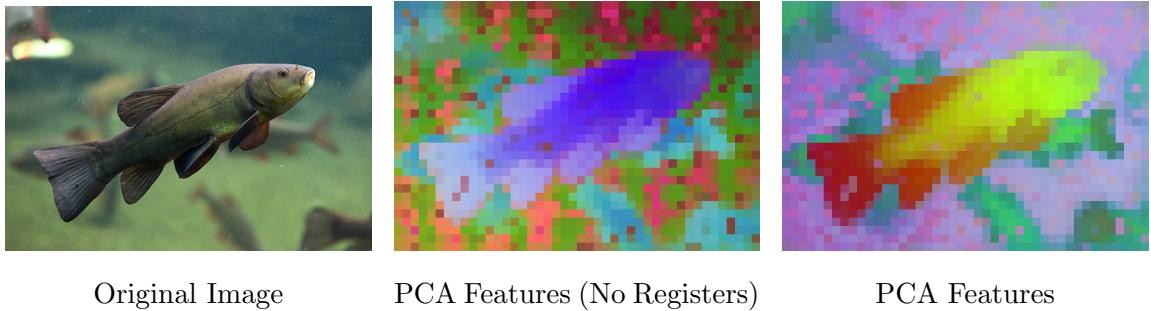


Figure 2: Left: Original. Middle: PCA Features without Register Tokens. Right: PCA features with Register Tokens.

Quantization

We adapted the ViT quantization script to support DINOv2. Table 1 details the different quantization formats supported with DINOv2.

Format	Description
4.0	4-bit round-to-nearest quantization (q). Each block has 32 weights.
4.1	4-bit round-to-nearest quantization (q). Each block has 32 weights.
5.0	5-bit round-to-nearest quantization (q). Each block has 32 weights.
5.1	5-bit round-to-nearest quantization (q). Each block has 32 weights.
8.0	8-bit round-to-nearest quantization (q). Each block has 32 weights.

Table 1: Supported quantization formats

Additional Features

Flash Attention:

To improve inference speed and reduce memory usage we implemented flash attention within the DINOv2 pipeline. Flash attention computes memory in a much more memory-

efficient manner using kernel fusion techniques. This can be optionally toggled to maximize performance on CPU.

Positional Embedding Interpolation:

When training, Vision Transformers rely on a fixed size positional embedding. However during inference, input image size may vary, especially as models are implemented on different devices. The interpolation process involves resizing the learned positional embeddings to match the number of patches generated by the new input resolution. This enables the model to generalize to arbitrary input size.

Cross-Platform Support:

We have tested dinov2.cpp on Windows, Linux, and Mac and have ensured that the inference pipeline works cross-platform.

Datasets

The original DINOv2 family of models was trained on the LVD-142M dataset, a curated set of 142 million images assembled from ImageNet-22k, Google Landmarks, and other fine-grained datasets. A self-supervised pipeline was developed for training, capturing features on the image and pixel level. The diverse dataset and self-supervised pipeline enables a wide degree of possible implementation in computer vision tasks without any finetuning. Existing tasks include classification, semantic segmentation, depth estimation, instance retrieval, and video understanding.

DINOv2.cpp retains these capabilities. Publicly available weights can be converted to the gguf format without any loss of accuracy. The inference pipeline supports images up to sizes of 518 by 518 pixels and automatically resizes input images, providing support for any image-based dataset. The diversity and size of the dataset used to train DINOv2 ensure that dinov2.cpp can support a wide range of real-time tasks in a variety of environments.

Evaluation Results

We evaluate our results on inference speed and memory usage. When compared to DINOv2’s PyTorch implementation, we aim to achieve faster inference speeds while decreasing memory usage. To compare our results, we ran four benchmarks. First, we have two benchmarks without quantization. For these benchmarks, we run the small, base, large, and giant models 100 times and measure the average inference speed and memory usage. This is done for the model variants with register tokens and without register tokens. We also ran two benchmarks for quantized models. For each model we benchmark the 4_0, 4_1, 5_0, 5_1, and 8_0 formats across 100 inference runs each. We run this for the model variants

small, base, large, and giant, each with and without register tokens. Each experiment was run using Intel Core i9-14900HX. While it has 32 threads, we used 24 for these experiments as we found improved results.

Our benchmark results, detailed in Figure 3, show universal improvements in inference speed and memory use when compared to PyTorch. Our most notable improvement was with the small variant with register tokens. When compared to PyTorch we see an increase in speed from 297ms to 64ms, and a memory reduction from 455MB to 110MB.

Our quantization benchmark results, detailed in Figure 4, show universal improvements in inference speed and memory use when compared to the non-quantized models run in our pipeline. Our most notable improvement can be seen with the giant model with register tokens, as we see an increase in speed from 1995ms to 1281ms, and a memory reduction from 4400MB to 1281MB with quantization type 4.0.

Benchmark Results

Model	Max Mem (PyTorch)	Max Mem	Speed (PyTorch)	Speed
small	~ 455 MB	~ 110 MB	181 ms	62 ms
base	~ 720 MB	~ 367 MB	462 ms	197 ms
large	~ 1.55 GB	~ 1.2 GB	1288 ms	600 ms
giant	~ 4.8 GB	~ 4.4 GB	4384 ms	1969 ms

(a) Without Register Tokens

Model	Max Mem (PyTorch)	Max Mem	Speed (PyTorch)	Speed
small	~ 455 MB	~ 110 MB	297 ms	64 ms
base	~ 720 MB	~ 366 MB	436 ms	200 ms
large	~ 1.55 GB	~ 1.2 GB	1331 ms	597 ms
giant	~ 4.8 GB	~ 4.4 GB	4472 ms	1995 ms

(b) With Register Tokens

Figure 3: Peak memory and inference-speed comparison between PyTorch baseline and our custom build, without and with register tokens.

Quantization Benchmark Results

Model	Quant	Speed (ms)	Mem (MB)	Model	Quant	Speed (ms)	Mem (MB)
small	q4_0	46	49	small	q4_0	52	49
small	q4_1	48	51	small	q4_1	50	52
small	q5_0	63	54	small	q5_0	59	54
small	q5_1	58	57	small	q5_1	57	57
small	q8_0	50	70	small	q8_0	51	70
base	q4_0	141	129	base	q4_0	136	129
base	q4_1	135	140	base	q4_1	133	139
base	q5_0	162	150	base	q5_0	164	150
base	q5_1	161	160	base	q5_1	158	160
base	q8_0	125	212	base	q8_0	124	211
large	q4_0	389	371	large	q4_0	395	371
large	q4_1	382	407	large	q4_1	395	407
large	q5_0	497	444	large	q5_0	493	443
large	q5_1	478	480	large	q5_1	490	480
large	q8_0	348	661	large	q8_0	353	661
giant	q4_0	1268	1281	giant	q4_0	1275	1281
giant	q4_1	1248	1417	giant	q4_1	1261	1417
giant	q5_0	1625	1553	giant	q5_0	1615	1552
giant	q5_1	1576	1688	giant	q5_1	1583	1687
giant	q8_0	1059	2364	giant	q8_0	1065	2364

(a) Without Register Tokens

(b) With Register Tokens

Figure 4: Inference speed and memory footprint across quantization formats, without and with register tokens.

Conclusion

For our final project we developed dinov2.cpp, a lightweight C++ inference engine for the DINOV2 family of models. Our implementation enables efficient inference on edge devices, using the GGML tensor library with minimal dependencies.

Our benchmarks demonstrate significant reductions in memory usage and inference time across all model sizes when compared to the original PyTorch implementation. Quantization improves these results further, allowing for efficient model loading and inference. We provide support for classification and real-time feature extraction, giving insight into how the model understands visual information. We implemented flash attention, and positional embedding interpolation features, giving users the ability to choose between classification accuracy and inference speed.

Future work may include adding support for different visual analysis tasks, such as depth estimation and semantic segmentation.

Code Availability

DINOv2.cpp

References

- [1] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV]. URL: <https://arxiv.org/abs/2010.11929>.
- [2] Georgi Gerganov et al. *ggml-org/ggml*. May 2, 2025. URL: <https://github.com/ggml-org/ggml>.
- [3] Maxime Oquab et al. *DINOv2: Learning Robust Visual Features without Supervision*. 2024. arXiv: 2304.07193 [cs.CV]. URL: <https://arxiv.org/abs/2304.07193>.
- [4] Said Taghadouini, Georgi Gerganov, and Mehdi Elion. *staghado/vit.cpp*. Apr. 11, 2024. URL: <https://github.com/staghado/vit.cpp>.