"Profiling Memory and Performance of Large Language Models: A New Tool for Efficient Analysis".

To visualize the results of the ONNX model, you can use a variety of tools depending on the type of output you're interested in. Here are a few options:

1. ONNX Runtime: ONNX Runtime is a high-performance engine for deploying ONNX models. It includes a visualization tool that allows you to visualize the graph structure and analyze the performance of your model. You can install ONNX Runtime and use the onnxruntime-tools package to run the visualization tool.

2. Netron: Netron is a standalone tool that can visualize deep learning models in a variety of formats, including ONNX. You can simply load your ONNX model into Netron and it will display a graph of the model structure. Netron is available both as a web app and as a desktop app.

3. TensorBoard: TensorBoard is a web-based tool for visualizing and analyzing TensorFlow models, but it also has support for ONNX models. You can use the TensorFlow ONNX converter to convert your ONNX model to a TensorFlow graph, and then use TensorBoard to visualize it.

4. PyTorch: If you're using PyTorch, you can use the torch.onnx.export function to export your PyTorch model to ONNX format, and then use PyTorch's visualization tools to visualize the ONNX model. Specifically, you can use the torchviz package to generate a visualization of the computational graph of your PyTorch model.

These are just a few examples of the tools you can use to visualize ONNX models. The choice of tool depends on your specific needs and the format of the output you're interested in.

Abstract:

With the advent of deep learning, large language models have become increasingly popular and have shown remarkable performance in various natural language processing tasks. However, the efficient training and evaluation of these models require careful attention to the computational resources they consume. In this paper, we present a new tool for profiling the memory, time, and GPU usage of large language models. The tool uses the PyTorch profiler to generate detailed reports on the resource consumption of each layer in the model, allowing researchers and practitioners to identify potential bottlenecks and optimize their models accordingly.

Introduction:

Large language models have become a key component of many natural language processing applications. These models are typically based on deep neural networks and can require significant computational resources to train and evaluate. The efficient utilization of these resources is essential for achieving good performance and reducing training time. However, profiling the resource consumption of large language models can be challenging due to the complexity of the models and the large number of parameters they contain.

In this paper, we present a new tool for profiling the memory, time, and GPU usage of large language models. The tool is based on the PyTorch profiler, which is a built-in profiling tool in the PyTorch library. The tool generates detailed reports on the resource consumption of each layer in the model, allowing researchers and practitioners to identify potential bottlenecks and optimize their models accordingly.

Methodology:

To profile the resource consumption of large language models, we developed a Python script that uses the PyTorch profiler to generate detailed reports on the memory, time, and GPU usage of each layer in the model. The script takes as input a pre-trained language model and generates a report that shows the resource consumption of each layer in the model.

The script works by iterating over each layer in the model and profiling its resource consumption using the PyTorch profiler. The profiler collects detailed information on the CPU and GPU memory usage, the time taken for each operation, and the shape of the output tensor for each layer. The profiler outputs this information in a JSON format, which is then parsed by the script to generate a summary report.

Results:

To demonstrate the effectiveness of our tool, we evaluated it on two large language models: GPT-2 and BERT. We used the pre-trained models provided by the Hugging Face Transformers library and profiled the resource consumption of each layer in the model using our tool.

The profiling results showed that the resource consumption varied significantly across the layers in the models. For example, the embedding layer in both models consumed a significant amount of memory, while the attention and feedforward layers consumed a significant amount of time. The results also showed that the resource consumption varied across different model configurations and input sizes.

Conclusion:

In this paper, we presented a new tool for profiling the memory, time, and GPU usage of large language models. The tool uses the PyTorch profiler to generate detailed reports on the resource consumption of each layer in the model, allowing researchers and practitioners to identify potential bottlenecks and optimize their models accordingly.

The tool can be used to profile the resource consumption of different types of language models and configurations and can help researchers and practitioners optimize their models for efficient training and evaluation. Future work includes extending the tool to support other deep learning frameworks and adding support for distributed training and evaluation.

Reference: Toward Efficient Interactions between Python and Native Libraries

In the Python ecosystem, native libraries and downstream application codes evolve rapidly so they can interact in numerous and unexpected ways. Therefore, building a list to exhaust all interaction inefficiencies becomes infeasible. We seek a solution that will automatically identify the blocks of Python code that lead to inefficient interactions, through closing the knowledge gap between Python and native code. Existing profiling tools cannot address this issue. Python profiles [19, 22, 24, 49, 52, 55, 66, 67, 75] cannot step in native code so they do not know execution details. Native profiling tools [2, 9, 15, 44, 54, 62, 71, 72] can identify hotspots, which offer insights into problematic code blocks. However, because these tools do not have knowledge about Python code’s semantics, they cannot render detailed root cause and thus often make debugging remarkably challenging.

@article{Tan2021TowardEI, title={Toward efficient interactions between Python and native libraries}, author={Jialiang Tan and Yu Chen and Zhenming Liu and Bin Ren and Shuaiwen Song and Xipeng Shen and Xu Liu}, journal={Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering}, year={2021} }

**Existing Tools**

**Python performance analysis tools.**

**Native performance analysis tools.**

|  |  |  |  |
| --- | --- | --- | --- |
| Model Type | Layer Types | Memory Demand | Compute Demand |
| Image Classification | Convolutional, Pooling, Fully Connected | Medium to High | High |
| Object Detection | Convolutional, Pooling, Fully Connected | High | Very High |
| Semantic Segmentation | Convolutional, Upsampling, Fully Connected | Very High | Very High |
| Generative Adversarial Networks (GANs) | Convolutional, Upsampling, Fully Connected | High to Very High | High to Very High |
| Recurrent Neural Networks (RNNs) | Embedding, LSTM, GRU | Medium to High | High |
| Transformer-based Language Models (LLMs) | Embedding, Multi-Head Attention, Feedforward | Very High | Very High |

| **Model Type** | **Parameter Types** | **Memory Demand** | **Compute Demand** |
| --- | --- | --- | --- |
| Image Classification | Convolutional Layers, Fully Connected Layers | Depends on the size of input images, number of layers, and number of filters in each layer | Depends on the number of parameters and the size of input images |
| Recurrent Neural Networks (RNNs) | Embedding Layers, Recurrent Layers, Fully Connected Layers | Depends on the size of input sequences, number of layers, and number of hidden units in each layer | Depends on the number of parameters and the size of input sequences |
| Transformer-based Models | Embedding Layers, Self-Attention Layers, Feedforward Layers | Depends on the size of input sequences, number of layers, and number of attention heads in each layer | Depends on the number of parameters and the size of input sequences |
| Large Language Models (LLMs) | Embedding Layers, Self-Attention Layers, Feedforward Layers, Transformer Blocks | Depends on the size of input sequences, number of layers, and number of attention heads in each layer | Depends on the number of parameters and the size of input sequences |

Profiling a transformer model involves measuring and analyzing its performance characteristics, such as computational time, memory usage, and throughput. Here are some steps to follow when profiling a transformer model:

Define the metrics you want to measure: Before profiling the model, you need to decide which performance metrics you want to measure. These could include the model's inference time, the amount of memory used during inference, the number of operations performed by the model, and the model's throughput.

Use a profiling tool: There are many profiling tools available that can help you measure the performance of a transformer model. One popular tool is NVIDIA's TensorRT, which optimizes and accelerates the inference of deep learning models on NVIDIA GPUs. Another tool is PyTorch's autograd profiler, which can help you identify performance bottlenecks in your model.

Prepare your input data: To profile your transformer model, you need to provide it with input data that is representative of the data it will encounter during inference. This can help you get a better understanding of the model's performance under real-world conditions.

Run the profiling tool: Once you have prepared your input data and selected a profiling tool, you can run the tool to measure the performance of your transformer model. The tool will generate a report that summarizes the performance metrics you specified.

Analyze the results: After profiling your transformer model, you should analyze the results to identify any performance bottlenecks or areas for improvement. You may need to experiment with different hyperparameters or architectures to optimize the performance of your model.

Overall, profiling a transformer model is an important step in optimizing its performance and ensuring that it can meet the requirements of your application.

Profiling a transformer model involves measuring and analyzing its performance characteristics, such as computational time, memory usage, and throughput. Here are some steps to follow when profiling a transformer model:

1. Define the metrics you want to measure: Before profiling the model, you need to decide which performance metrics you want to measure. These could include the model's inference time, the amount of memory used during inference, the number of operations performed by the model, and the model's throughput.

2. Use a profiling tool: There are many profiling tools available that can help you measure the performance of a transformer model. One popular tool is NVIDIA's TensorRT, which optimizes and accelerates the inference of deep learning models on NVIDIA GPUs. Another tool is PyTorch's autograd profiler, which can help you identify performance bottlenecks in your model.

3. Prepare your input data: To profile your transformer model, you need to provide it with input data that is representative of the data it will encounter during inference. This can help you get a better understanding of the model's performance under real-world conditions.

4. Run the profiling tool: Once you have prepared your input data and selected a profiling tool, you can run the tool to measure the performance of your transformer model. The tool will generate a report that summarizes the performance metrics you specified.

5. Analyze the results: After profiling your transformer model, you should analyze the results to identify any performance bottlenecks or areas for improvement. You may need to experiment with different hyperparameters or architectures to optimize the performance of your model.

Overall, profiling a transformer model is an important step in optimizing its performance and ensuring that it can meet the requirements of your application.

To estimate the memory and compute demands of a given model and determine if it will fit on a given GPU or machine, you can follow these steps:

1. Determine the size of the model: The size of the model is a key factor in determining its memory and compute demands. You can determine the size of the model by counting the number of parameters it has. This information is usually available in the documentation of the model or can be obtained using model inspection tools.

2. Determine the batch size: The batch size is the number of input examples that are processed at the same time. Larger batch sizes can lead to more efficient use of hardware resources, but also require more memory. You will need to choose a batch size that is appropriate for your hardware and use case.

3. Estimate the memory requirements: To estimate the memory requirements of the model, you can use the following formula:

memory requirements = model size \* batch size \* precision factor

The precision factor is the number of bytes required to store each number in the model. This is typically 2 bytes for half-precision (float16) and 4 bytes for single-precision (float32).

4. Estimate the compute requirements: The compute requirements of the model depend on the number of operations required to process each input example. You can estimate the number of operations by counting the number of operations required for each layer of the model and multiplying it by the number of layers. This information is also usually available in the documentation of the model or can be obtained using model inspection tools.

5. Estimate the queries per second: Once you have estimated the memory and compute requirements of the model, you can use this information to estimate the number of queries per second that can be supported by a given GPU or machine. This will depend on the hardware specifications of the machine, such as the number of GPU cores, memory bandwidth, and clock speed. You can use benchmarking tools to measure the performance of your model on a given machine and adjust your estimates accordingly.

By following these steps, you can estimate the memory and compute demands of a given model and determine if it will fit on a given GPU or machine, and how many queries per second can be supported.

FLOPS (Floating Point Operations per Second) is a measure of the computational complexity of a deep learning model. It is the number of floating point operations that can be executed per second by the hardware on which the model is running.

In the field of deep learning, FLOPS is often used as a proxy for the model's complexity and compute requirements. A model with a high number of FLOPS generally requires more computational resources and longer training times than a model with a lower number of FLOPS.

FLOPS is particularly important when working with large and complex models, such as those used in computer vision tasks like image classification or object detection. In these cases, the number of FLOPS required to process a single input can be in the billions or trillions, making it critical to use high-performance hardware such as GPUs or TPUs to achieve reasonable training times.

As deep learning models continue to grow in complexity and size, FLOPS is likely to remain an important metric for evaluating the computational requirements of these models and optimizing their performance.

Computing FLOPS, latency and fps of a model

It is important to have an idea of how to measure a video model’s speed, so that you can choose the model that suits best for your use case. In this tutorial, we provide two simple scripts to help you compute (1) FLOPS, (2) number of parameters, (3) fps and (4) latency. These four numbers will help you evaluate the speed of this model. To be specific, FLOPS means floating point operations per second, and fps means frame per second. In terms of comparison, (1) FLOPS, the lower the better, (2) number of parameters, the lower the better, (3) fps, the higher the better, (4) latency, the lower the better.

In terms of input, we use the setting in each model’s training config. For example, I3D models will use 32 frames with stride 2 in crop size 224, but R2+1D models will use 16 frames with stride 2 in crop size 112. This will make sure that the speed performance here correlates well with the reported accuracy number.

Introduction: Large Language Models (LLMs) have rapidly evolved in recent years, with the likes of GPT-3, T5, and BERT gaining widespread adoption in various domains, including natural language processing, computer vision, and speech recognition. LLMs have proved to be a game-changer in many tasks, but their increasing complexity and size have raised concerns about their ethical, legal, and social implications. In this paper, we aim to profile LLMs, their architecture, training process, and applications, and discuss the challenges and opportunities they present.

Architecture: LLMs are neural networks that process large amounts of text data to learn the underlying patterns and structure. They typically consist of multiple layers of neurons, with each layer processing a different aspect of the input data. The most common architecture used in LLMs is the Transformer architecture, which was introduced by Vaswani et al. (2017). The Transformer architecture consists of an encoder and a decoder, with the encoder being responsible for processing the input data and the decoder generating the output.

Training Process: LLMs are trained using a process called unsupervised learning, where the model learns from a large corpus of text data without any labeled examples. The training process involves optimizing a loss function that measures the difference between the predicted output and the ground truth. The training data is typically preprocessed to remove noise and irrelevant information, and the model is trained using stochastic gradient descent (SGD) or a variant of it, such as Adam.

Applications: LLMs have been applied to a wide range of natural language processing tasks, including text classification, sentiment analysis, language translation, and question answering. They have also been used in computer vision and speech recognition tasks. One of the most significant applications of LLMs is in language generation, where they can generate text that is indistinguishable from human-written text. This capability has significant implications in the field of natural language processing, where it can be used to generate more human-like chatbots, improve machine translation, and create more accurate text-to-speech models.

Challenges: LLMs come with a range of challenges, including ethical, legal, and social challenges. One of the most significant challenges is the potential for bias in the training data, which can result in biased models that perpetuate discrimination and inequality. Another challenge is the potential for misuse, where LLMs can be used to generate fake news or create deepfakes, leading to significant social and political implications.

Opportunities: Despite the challenges, LLMs present several opportunities, including their potential to improve the accuracy of natural language processing tasks, improve machine translation, and create more human-like chatbots. LLMs also have the potential to transform industries such as healthcare, where they can be used to generate more accurate medical diagnoses and predictions.

Conclusion: LLMs have revolutionized the field of natural language processing, and their capabilities have significant implications in various domains. While they present several challenges, including potential biases and misuse, LLMs also present several opportunities that can improve the accuracy of natural language processing tasks and transform industries such as healthcare. As LLMs continue to evolve, it is critical to address their challenges while leveraging their opportunities to create a more equitable and just society.

Title: Performance Optimization of Deep Learning Models using DeepSpeed, ONNXRuntime, PyTorch Profiler, and TensorBoard in the Hugging Face Ecosystem

Abstract: Deep learning models have become increasingly complex, requiring more computational power and longer training times. In response, researchers and practitioners have developed various tools and techniques to optimize the performance of these models. This paper presents a novel approach for performance optimization of deep learning models using the DeepSpeed library, the ONNXRuntime engine, the PyTorch Profiler, and TensorBoard in the Hugging Face ecosystem. Our approach allows for efficient distributed training, inference, and profiling of deep learning models, as well as visualization and analysis of performance metrics. We demonstrate the effectiveness of our approach through an example code for fine-tuning a BERT model on a text classification task.

Deep learning models have revolutionized the field of artificial intelligence, but their increasing complexity has led to significant computational challenges. To address these challenges, researchers have developed various tools and techniques to optimize the performance of these models. In this paper, we present a novel approach for performance optimization using the DeepSpeed library, the ONNXRuntime engine, the PyTorch Profiler, and TensorBoard in the Hugging Face ecosystem. Our approach enables efficient distributed training, inference, and profiling of deep learning models, as well as visualization and analysis of performance metrics. We demonstrate the effectiveness of our approach through an example code for fine-tuning a BERT model on a text classification task. Our results show that our approach significantly reduces training time and memory usage, while improving model performance. Overall, our approach provides a comprehensive solution for optimizing the performance of deep learning models in the Hugging Face ecosystem.

Here's an example introduction section:

Deep learning models have made significant advancements in the fields of computer vision, natural language processing, and other areas of artificial intelligence. However, as these models become increasingly complex, they require more computational power and longer training times. This has led to a growing need for efficient tools and techniques to optimize the performance of deep learning models. In recent years, researchers have developed several libraries and frameworks to address these challenges.

One such library is DeepSpeed, which enables efficient distributed training of deep learning models. DeepSpeed uses a novel approach to distributed training, which reduces the communication overhead between model replicas and allows for larger batch sizes. Another key tool for performance optimization is the ONNXRuntime engine, which enables faster inference by converting trained models to the optimized ONNX format. Additionally, the PyTorch Profiler provides a comprehensive suite of tools for profiling deep learning models, including detailed information on training time, memory usage, and other performance metrics.

In this paper, we present a novel approach for performance optimization using the DeepSpeed library, the ONNXRuntime engine, the PyTorch Profiler, and TensorBoard in the Hugging Face ecosystem. Our approach enables efficient distributed training, inference, and profiling of deep learning models, as well as visualization and analysis of performance metrics. We demonstrate the effectiveness of our approach through an example code for fine-tuning a BERT model on a text classification task. Our results show that our approach significantly reduces training time and memory usage, while improving model performance. Overall, our approach provides a comprehensive solution for optimizing the performance of deep learning models in the Hugging Face ecosystem.

Here's an example related work section:

Several tools and frameworks have been developed to optimize the performance of deep learning models. In this section, we provide an overview of some of the related work.

TensorFlow is a popular deep learning framework that provides a suite of tools for optimizing model performance. One such tool is the TensorFlow Profiler, which enables detailed profiling of TensorFlow models. The TensorFlow Profiler can provide insights into training time, memory usage, and other performance metrics, allowing researchers to optimize their models for faster and more efficient training.

PyTorch is another popular deep learning framework that provides several tools for optimizing model performance. The PyTorch Profiler provides detailed profiling of PyTorch models, similar to the TensorFlow Profiler. Additionally, PyTorch supports distributed training using the DataParallel and DistributedDataParallel modules, which enable efficient training across multiple GPUs.

Another tool for optimizing model performance is NVIDIA's TensorRT. TensorRT is an inference engine that optimizes models for deployment on NVIDIA GPUs. TensorRT can significantly speed up inference times by optimizing the computation graph and using low-precision arithmetic.

Recently, the Hugging Face ecosystem has emerged as a popular framework for natural language processing tasks. The Hugging Face ecosystem provides several pre-trained models, including BERT, GPT-2, and T5, among others. Additionally, the ecosystem provides several tools for fine-tuning and deploying these models, including Transformers, Tokenizers, and Accelerate.

In this paper, we focus on the combination of DeepSpeed, ONNXRuntime, PyTorch Profiler, and TensorBoard in the Hugging Face ecosystem for performance optimization of deep learning models. Our approach provides a comprehensive solution for distributed training, inference, profiling, and visualization of performance metrics in the Hugging Face ecosystem.

When working with transformer models, the parameter types can be identified based on the names of the layers in the model. Here are some common naming patterns for transformer layers that can help you identify the parameter types:

• Embeddings: Parameters related to the input and output embeddings are usually named with the word "embeddings". For example, in the BERT model, the input and output embeddings are named "bert.embeddings.word\_embeddings" and "bert.embeddings.LayerNorm", respectively.

• FFN: Parameters related to the feedforward layer are usually named with the word "ffn". For example, in the BERT model, the feedforward layer is named "bert.encoder.layer.X.intermediate.dense" and "bert.encoder.layer.X.output.dense", where "X" is the layer number.

• Convolution: Parameters related to the convolutional layer are usually named with the word "conv". For example, in the GPT-2 model, the convolutional layer is named "transformer.h.X.attn.c\_attn.conv1d" and "transformer.h.X.attn.c\_proj.conv1d", where "X" is the layer number.

Note that these naming patterns may vary depending on the specific model you are working with, so it's always a good idea to check the documentation or the source code of the model to confirm the naming conventions for the different layers. Additionally, you can use the code snippet I provided earlier to print out the names and shapes of the parameters in the model and manually identify the layer types based on their names.

OpenLlama 13b and MPT 30B. At this point you might have lost track as to which model is the best to use understandably. The short answer is Llama 🦙 is still the best open source large language model but it does not come with a commercial license while Falcon 🦅 is the best commercial alternative.  
  
One thing to keep an eye on when reading about a new open source large language model is the number of tokens it was trained on. The most performant models at this point are trained with around 1T tokens. This applies to Llama 🦙, Falcon 🦅, MPT Ⓜ️ and StableLM 🦜 but not models like Pythia, Cerebras or Dolly.  
  
Assuming the model was trained on 1T tokens or more, then parameter size is a good indicator of performance. The most common sizes that LLMs come at are around 7b, 13b, 30B and 60B. At the moment Llama comes at all sizes while Falcon and MPT provide a model at the 30B range and StableLM stops at 7B. All have announced plans to train models at all these sizes so we should see more models released very soon.

There is an ongoing debate as to whether it is possible to replicate the performance of a larger model using a smaller one. Alpaca and friends first claimed this was possible with subsequent work (false promise of imitating proprietary llms) turning this around and then orca showing that its possible again by using chain of thoughts explanations to learn from. It is worth keeping in mind that most probably some form of compressing or distilling the knowledge from a large model is possible with the exact mechanism to be defined but this still means you need a teacher model and that its legal to do so.

