Memory Footprints of Selected DNN Optimization Algorithms

Eliah Windolph and Jack Ogaja

September 4, 2024

Introduction

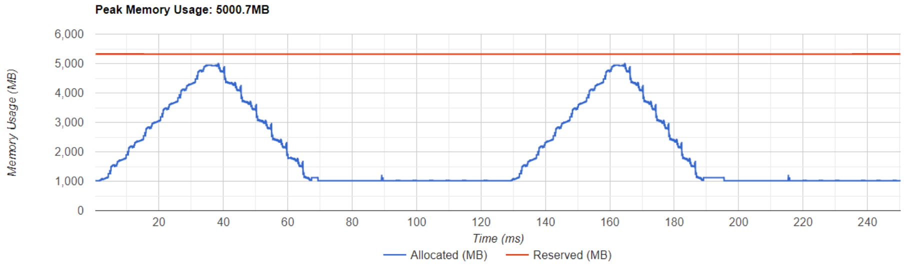
- Training costs increases with model sizes
- Memory signature of optimization algorithms can help in computational performance characterization of model training
- Allows employment of suitable performance optimization techniques to reduce training costs

Model

- All trials were run with the same model
- The model we used is a transformer for question answering called DistilBertForQuestionAnswering
- It has around 66.4 M trainable parameters
- Has a size of around 265 Mb

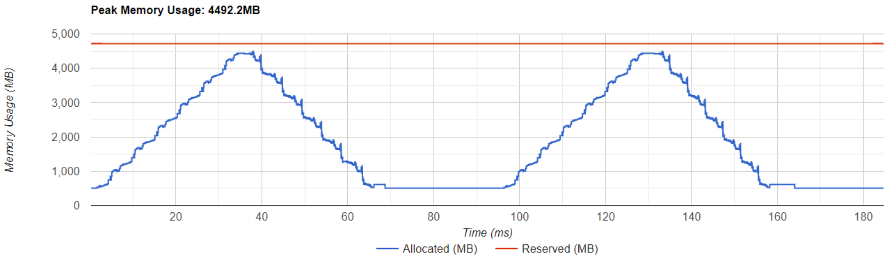
Adam

- Short for Adaptive Moment Estimation
- Combines the advantages of AdaGrad (adaptive gradient algorithm) and RMSProp (root mean squared propagation)
- Adapts the learning rate for each parameter regarding to its previous results



SGD

- Short for stochastic gradient descent
- Can be used with momentum (disabled in our runs)
- Can be used with weight decay (disabled in our runs)
- Updates the parameters based on the gradient and learning rate



Overview of Optimizers

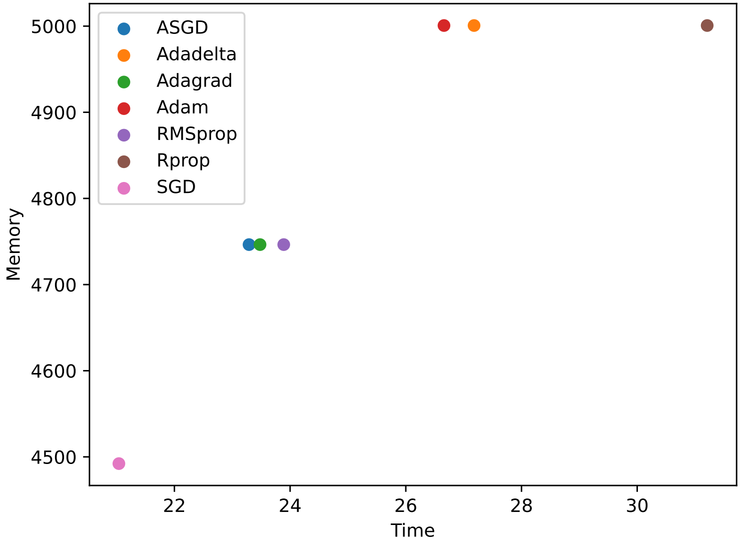
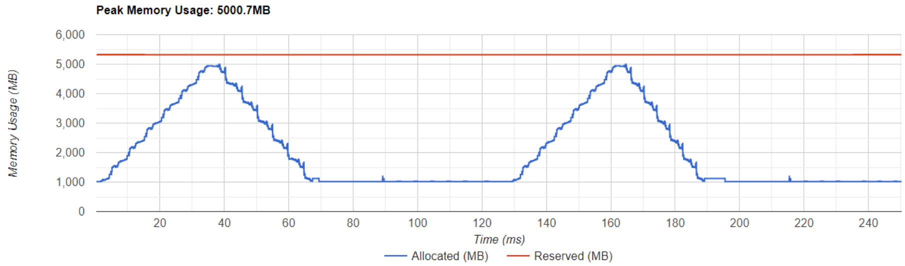


Figure: Peak memory consumption vs average training time

Mixed Precision

- Not all variables need to be stored in 32bit (faster and less memory usage)
- Gradients are calculated in 16bits as well but will get converted back for the optimization step
- The model is in both 16-bit and 32-bit Precision on the GPU (around 1.5 times the original memory)
- Depending on model and batch size we can either increase or decrease the memory consumption

Mixed Precision - Adam



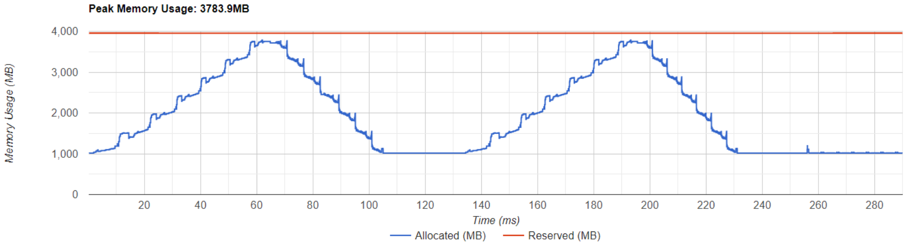
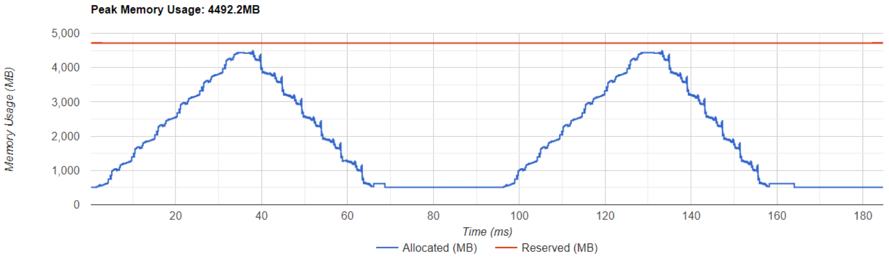


Figure: Training for 2 epochs (Top: Base; Bottom: Mixed precision)

Mixed Precision - SGD



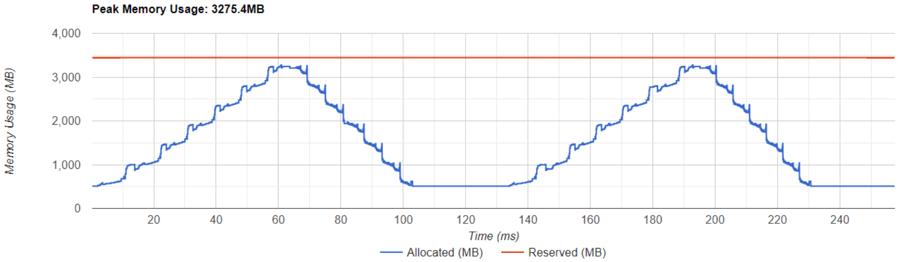
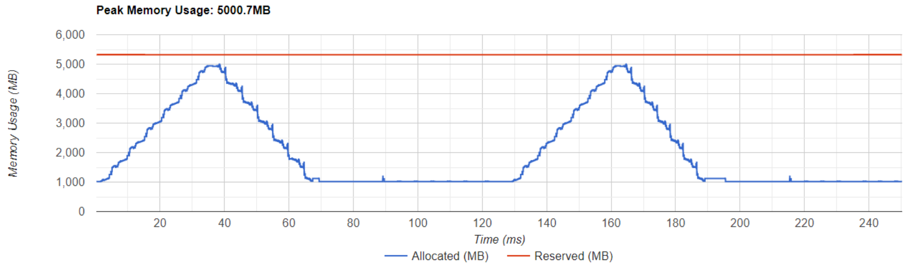


Figure: Training for 2 epochs (Top: Base; Bottom: Mixed precision)

Gradient Checkpointing

- For the backward pass, all activations from the forward pass are saved
- This creates a huge memory overhead
- You could recalculate them but this would create a huge calculation overhead
- Gradient Checkpointing takes the middle road by saving only parts of the activations
- The other needed activations will get recalculated from the last checkpoint

Gradient Checkpointing - Adam



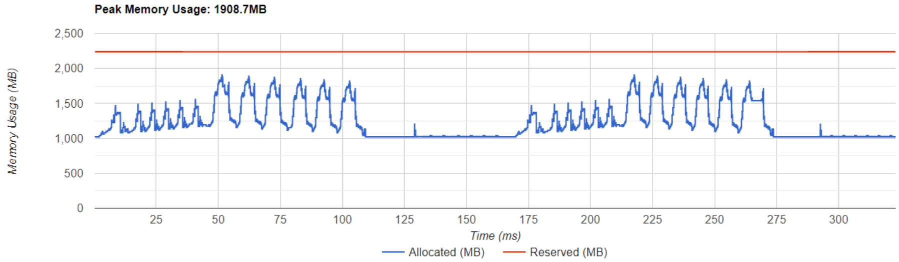
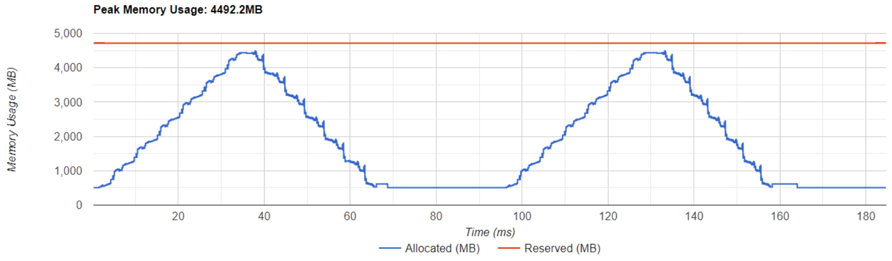


Figure: Training for 2 epochs (Top: Base; Bottom: Check-pointing)

Gradient Checkpointing - SGD



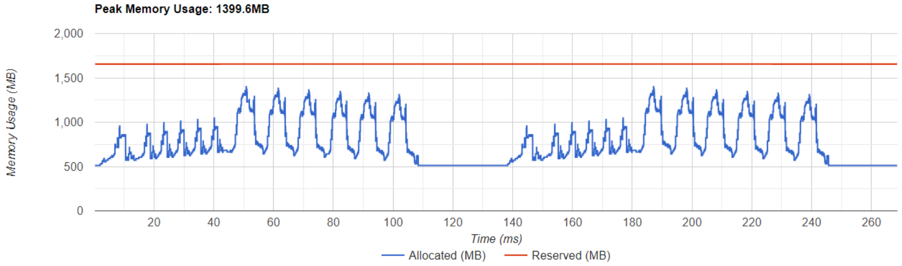


Figure: Training for 2 epochs (Top: Base; Bottom: Check-pointing)

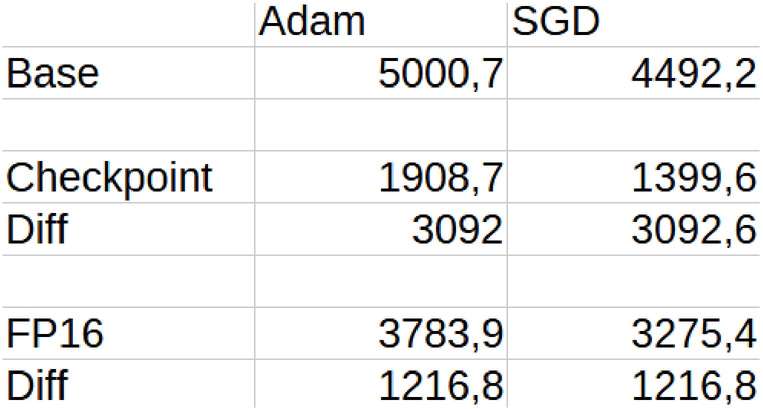
Batch Size

- Since we need to save all the activations for all inputs and calculate the gradients for all batch elements we can decrease the memory usage by decreasing the batch size
- Unless a optimizer saves additional parameters for each batch element the memory savings are optimizer independent and therefore not that important in our investigation

Gradient Accumulation

- If we want a higher batch size but we are limited by the memory one can use gradient accumulation
- Instead of running all the elements at once we have multiple forward and backward passes
- We accumulate the gradients before performing the optimization step
- Again should be optimizer independent

Memory Optimization Techniques



	Adam	SGD
Base	5000,7	4492,2
Checkpoint	1908,7	1399,6
Diff	3092	3092,6
FP16	3783,9	3275,4
Diff	1216,8	1216,8

Figure: Memory consumption in MB

Summary

- SGD has relatively lower average training time per epoch with lower peak memory consumption
- Adam and its derivatives (momentum-based) consume relatively higher memory

Introduction to Neural Network Intelligence (NNI) and Performance Analysis

Eliah Windolph and Jack Ogaja

September 4, 2024

Motivation

- Applying Machine Learning can be challenging especially for non-experts. Automation of model building and training (AutoML) can increase efficiency and productivity while applying machine learning
- AutoML can also be exploited for computational performance engineering and analysis

Introduction

We show how to use Neural Network Intelligence (NNI) to automate Hyperparameter Optimization (HPO) in Grete GPUs. NNI requires 3 files:

- The python code which includes your model and trainings loop
- A config file which will tell NNI how to run your code
- A search space, which includes the parameters which you try to optimize

Python File (PyTorch)

You only have to make some small adjustment to any existing PyTorch script.
NNI adds 3 important functions:

```
            nni.get_next_parameter()
            # This will be used to get the current parameters from the defined search space. It returns a dictionary.

```

```
            nni.report_intermediate_result()
            # This will be used to report the intermediate results which will show in the WebUI

```

```
            nni.report_final_result()
            # This will report the final result which will show in the WebUI as well

```

Python File (PyTorch)

For example, the modified lines could look like this:

```
def main():
    args = nni.get_next_parameter()
    optimizer = torch.optim.SGD(model.parameters, lr=args['lr'])
    ⋮
    train(...)
    nni.report_final_result(get_accuracy(model, data))

def test(...):
    ⋮
    nni.report_intermediate_result(get_accuracy(model, data))
```

The config file

This file contains the config which NNI will use to run your code. The most important configs are listed below. For more info look here.

- searchSpaceFile: Path to your search space
- trialCommand: The command which will run the python code
- trainingService: Switch between local, remote or more. In the cluster you will have to use local
- tuner: The tuner will choice which parameters will get tested next

The config file

An example might look like this:

```
searchSpaceFile: search_space.json
trialCommand: python3 mnist_tensorboard.py
trialConcurrency: 1
maxExperimentDuration: 20m
tuner:
  name: TPE
  classArgs:
    optimize_mode: maximize
trainingService:
  platform: local
  useActiveGpu: true
```

Search Space

The search space tells NNI which hyperparameters you are trying to optimizer over. The file is a json file with the following structure:

```
{
  "name": {_type": type, "_value": options}
  ⋮
},
```

The type tells NNI how to interpret the values. It could be a choice of a list or a random integer between two values. Depending on which tuner you use different types are allowed. For more info check here.

Search Space

An example from NNI for a mnist model looks like this:

```
{
  "batch_size": {"_type":"choice", "_value": [16, 32, 64, 128]},
  "hidden_size":{"_type":"choice","_value":[128, 256, 512, 1024]},
  "lr":{"_type":"choice","_value":[0.0001, 0.001, 0.01, 0.1]},
  "momentum":{"_type":"uniform","_value":[0, 1]}
}
```

PyTorch Profiler

Additionally you can run your PyTorch code with the PyTorch profiler and tensorboard. The tensorboard plugin will make it easier to examine the results since it shows them in a nice webui. In order to add the profiler you have just have to create the profiler object:

```
prof = torch.profiler.profile(
  on_trace_ready=torch.profiler.tensorboard_trace_handler(
    os.path.join(os.environ['NNI_OUTPUT_DIR'] ,'tensorboard')),
  profile_memory=True,
  with_stack=True)
```

For more information and parameters check here.

PyTorch Profiler

You then start the profiler with **prof.start()**. This will often be placed directly before the traninigs loop. After each epoch you will add the profiler step with **prof.step()**, before eventually closing the profiler with **prof.stop()**.
If we know want to take a look at the profiler we will need to install the profiler plugin by running **pip install torch_tb_profiler** in our python environment.

PyTorch Profiler

We then navigate to our nni-experiments folder and start tensorboard via the command:
tensorboard --logdir "<your experiment id>" --bind_all --port <your port>
Make sure to port forward this port to your local machine. You can then open your browser and open the webui of tensorboard and go to the pytorch profiler tab to see all the results.

Depending on your total amount of runs this make take a while!

Tensorboard WebUI

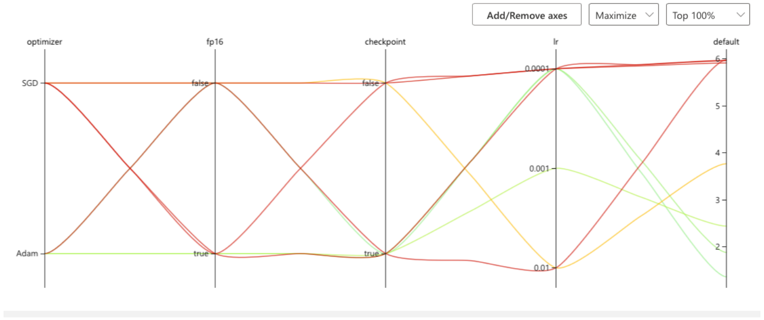


Figure: Hyperparameter values: Top 100 %

Tensorboard WebUI

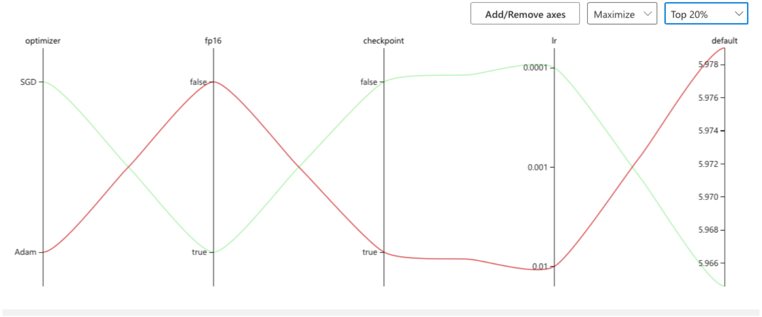


Figure: Hyperparameter values: Top 20 %

Tensorboard WebUI

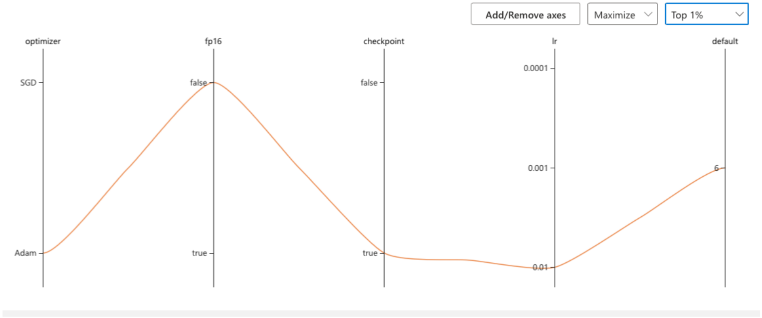
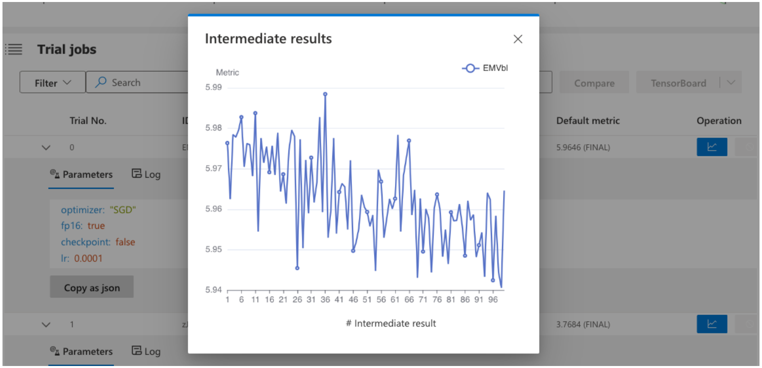


Figure: Hyperparameter values: Top 1 %

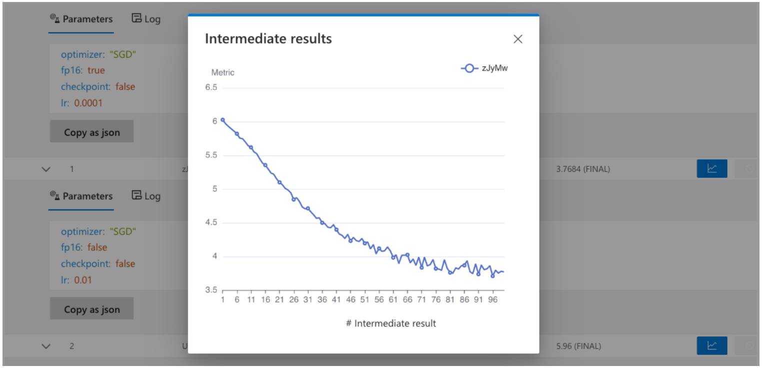
Tensorboard WebUI



optimizer: "SGD"
fp16: true
checkpoint: false
lr: 0.0001

Figure: Intermediate Results: Trial 1

Tensorboard WebUI



```
optimizer: "SGD"
fp16: true
checkpoint: false
lr: 0.0001
```

1 3.7684 (FINAL)

```
optimizer: "SGD"
fp16: false
checkpoint: false
lr: 0.01
```

2 5.96 (FINAL)

Figure: Intermediate Results: Trial 2