# Conclusions

* Base time is between 1.5 (google colab cpu has worse MHz/Flops so 1.9s)
* The most significant improvement is by increasing the number of cores (for number of intraop threads)
  + Using 2 cores 0.8s
  + Using 4 cores yields almost 3X improvement to 0.55-0.6s
  + More cores might yield even better results
* If Padding is above 30% on average, than BetterTransformer yields the Best results results (0.4s or less for 4 cores, 0.9s or less for single core) - 2-4x improvemtn
* If padding is less than 30% on average, ONNXRuntime gets the best results (~0.45s for 4 cores, 1.2-1.3 for single core) - 20% improvement
  + Torch.compile has similar performance but a bit worse
* Another suggestion is to take the ONNX and run it in an optimized environment which is not python-based (like Triton) but this would require much more work for adjusting the 3rd party library for low gain

# Optimizations Done

1. Disable Grad reduces time by 0.15-+0.04 (2.1 -> 1.95) (99% confidence interval)
2. Warm up of each of models (so first latency is good also)
3. Parallelism (On docker with 4 cores on top of Intel i7 with 8 cores)

| Model | Num intraop threads | Num interop threads | Avg latency | Std latency |  |
| --- | --- | --- | --- | --- | --- |
| SecureBERT-NER | 1 | 1 | 1.5102315552012866 | 0.19554347445076628 | The difference between google colab stems from better CPU with better Mhz and GFLOPS (though I don’t have exact model of google colab to validate) and maybe improved pytorch version |
| CyNER | 1 | 1 | 1.4509599943493687 | 0.24199747813164316 |  |
| SecureBERT-NER | 2 | 1 | 0.8082285765991655 | 0.11678470829347322 |  |
| SecureBERT-NER | 4 | 1 | 0.5954501538775688 | 0.1338181948222099 |  |
| CyNER | 4 | 1 | 0.5606438285389612 | 0.10230861944329567 |  |
| SecureBERT-NER | 1 | 2 | 1.3364805042050605 | 0.13526702722227046 |  |
| SecureBERT-NER | 1 | 4 | 1.305621906768444 | 0.12424509846707077 |  |
| SecureBERT-NER | 2 | 2 | 0.7841833940772123 | 0.13218908589029263 |  |
| SecureBERT-NER | 4 | 2 | 0.5597864611204281 | 0.07165015451709449 |  |
| SecureBERT-NER | 2 | 4 | 0.8178367070680441 | 0.0901761492062893 |  |
| SecureBERT-NER | 4 | 4 | 0.5378288761820904 | 0.07472172220947865 |  |

1. BetterTransformer

| Model | Num intraop threads | Num interop threads | Avg latency | Std latency |  |
| --- | --- | --- | --- | --- | --- |
| SecureBERT-NER | 1 | 1 | 0.6895338764717412 | 0.19530775433446212 | 49% padding (10 sentences) |
| CyNER | 1 | 1 | 0.8154453098080879 | 0.22415405953672174 | 38% padding (10 sentences) |
| SecureBERT-NER | 4 | 1 | 0.2780284538518551 | 0.08812454086256907 | 49% padding (10 sentences) |
| SecureBERT-NER | 4 | 1 | 0.3976306877835259 | 0.15960970223382873 | 39% padding (12 sentences) |
| SecureBERT-NER | 4 | 1 | 0.4061382705470938 | 0.13310687426939113 | 31% padding (14 sentences) |
| SecureBERT-NER | 4 | 1 | 0.5211266861405484 | 0.17570878623945085 | 26% padding (16 sentences) |
| SecureBERT-NER | 4 | 1 | 0.6309557678811837 | 0.22014856100592425 | 25% padding (18 sentences) |
| CyNER | 4 | 1 | 0.33162659614585166 | 0.11026302331761305 | 38% padding (10 sentences) |

1. Torch.compile

| Model | Num intraop threads | Num interop threads | Avg latency | Std latency |  |
| --- | --- | --- | --- | --- | --- |
| SecureBERT-NER | 1 | 1 | 1.342572338705839 | 0.2757763648877847 |  |
| CyNER | 1 | 1 | 1.3607064551392267 | 0.2010354623044591 |  |
| SecureBERT-NER | 4 | 1 | 0.5562962699075078 | 0.11093142163532954 |  |
| CyNER | 4 | 1 | 0.4962373268465663 | 0.07803008132029937 |  |

1. ONNXRuntime

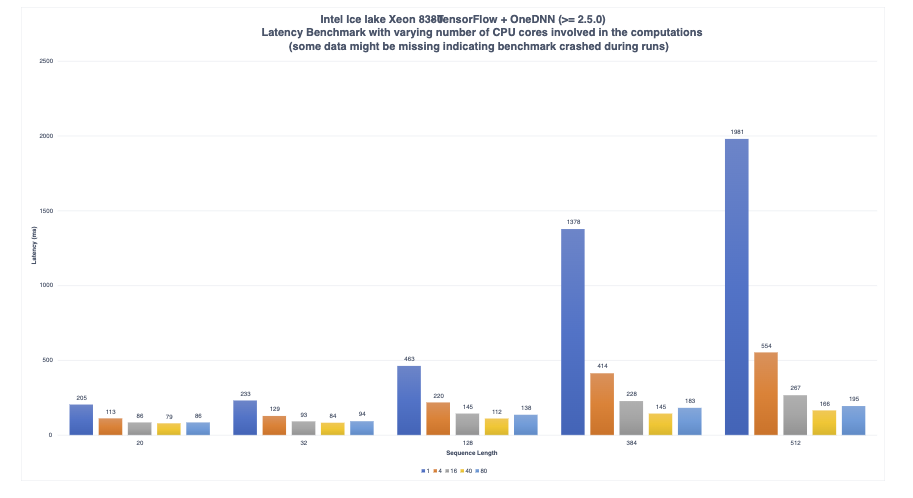
| Model | Num intraop threads | Num interop threads | Avg latency | Std latency |  |
| --- | --- | --- | --- | --- | --- |
| SecureBERT-NER | - | - | 0.4421567029731218 | 0.11711714822974974 |  |
| CyNER | - | - | 0.4675106288388718 | 0.10004503544709067 |  |
| SecureBERT-NER | 1 | 1 | 1.1829631449871285 | 0.11467656964293334 |  |
| CyNER | 1 | 1 | 1.3406732917525048 | 0.30026859157358887 |  |
| SecureBERT-NER | 4 | 1 | 0.46935342598793117 | 0.10525853363118189 |  |
| CyNER | 4 | 1 | 0.49505405128002167 | 0.12563548742195063 |  |

# Nir Suggestions

1. BetterTransformer/FastTransformer (<https://fast-transformers.github.io/>)
2. Docker that runs on ONNX (triton <https://github.com/triton-inference-server/server>)

# Type of Optimizations

1. Pytorch/Huggingface-level optimizations
   1. Disable Grad
   2. Decide the best “compilation/optimation” of the following (or if can be stacked):
      1. Torch.compile / torch.fx / torchscript (graph mode, maybe also run in C++)
      2. model.to\_bettertransformer() (Optimum)
         1. <https://medium.com/pytorch/bettertransformer-out-of-the-box-performance-for-huggingface-transformers-3fbe27d50ab2>
         2. Model using BetterTransformer is 1.03x faster than the original PyTorch model (throughput).
         3. Model using BetterTransformer is 2.17x faster than the original PyTorch model (throughput) - using sparsity.
      3. FastTransformer (<https://fast-transformers.github.io/>)
      4. ONNX Runtime (ORT)
         1. <https://github.com/huggingface/optimum?tab=readme-ov-file#onnx--onnx-runtime>
         2. <https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html>
      5. Docker that runs on ONNX (triton <https://github.com/triton-inference-server/server>)
      6. <https://huggingface.co/docs/optimum/intel/index> - OpenVino
      7. <https://blog.ml6.eu/openvino-vs-onnx-for-transformers-in-production-3e10c01520c8> - OpenVino vs ONNX (both are same)
      8. <https://blog.ml6.eu/bert-is-eating-your-cash-quantization-and-onnxruntime-to-save-ea6dc84dcd88>
   3. Torch.set\_num\_threads (according to graph in <https://huggingface.co/blog/bert-cpu-scaling-part-2> we used single core to get to 1.981s which doesn’t make sense on google colab that should have 2 cores)

 2. CPU-level optimizations

* 1. Intel Extension for Pytorch (IPEX)

1. Compression algorithms - Trade accuracy with speed
   1. Pruning
   2. Quantization (though it might increase latency)
   3. Knowledge Distillation

# Reading Material

* <https://huggingface.co/docs/transformers/main/perf_infer_cpu>
* <https://huggingface.co/docs/transformers/main/perf_torch_compile>
* <https://intel.github.io/intel-extension-for-pytorch/index.html#installation?platform=cpu&version=v2.1.100%2Bcpu>
* <https://pytorch.org/serve/performance_checklist.html>
  + Use model quantization (i.e. int8) for CPU inference. Explore different quantization options: dynamic quantization, static quantization, and quantization aware training, as well as tools such as Intel Neural Compressor that provide more sophisticated quantization methods. It is worth noting that quantization comes with some loss in accuracy and might not always offer significant speed up on some hardware thus this might not always be the right approach.
  + Balance throughput and latency with smart batching. While meeting the latency SLA try larger batch sizes to increase the throughput.
  + Try optimized inference engines such as **onnxruntime**, tensorRT, lightseq, ctranslate-2, etc. These engines often provide additional optimizations such as operator fusion, in addition to model quantization.
  + If working on CPU, you can try core pinning. You can find more information on how to work with this [in this blog post](https://pytorch.org/tutorials/intermediate/torchserve_with_ipex#grokking-pytorch-intel-cpu-performance-from-first-principles).
  + PyTorch: torch.set\_num\_threads(x)
  + <https://baiweiblog.wordpress.com/2017/11/02/how-to-set-processor-affinity-in-linux-using-taskset/#:~:text=To%20set%20the%20CPU%20affinity,can%20use%20pthread_setaffinity_np%20and%20pthread_attr_setaffinity_np>.
  + Improved cpu usage can improve by 2x
  + Docker preferences dictate how much CPU to use (I use 4 out of 8)
  + Launcher script - <https://github.com/huggingface/tune/blob/main/launcher.py>
* <https://pytorch.org/tutorials/intermediate/torchserve_with_ipex#grokking-pytorch-intel-cpu-performance-from-first-principles>
* <https://huggingface.co/blog/bert-cpu-scaling-part-1>
  + On the 2021 version, we kept the ability to run inference workloads through PyTorch and Tensorflow as in the previous blog [(1)](https://medium.com/huggingface/benchmarking-transformers-pytorch-and-tensorflow-e2917fb891c2) along with their traced counterpart [TorchScript (6)](https://pytorch.org/docs/stable/jit.html), [Google Accelerated Linear Algebra (XLA) (7)](https://www.tensorflow.org/xla).  
    Also, we decided to include support for [ONNX Runtime (8)](https://www.onnxruntime.ai/) as it provides many optimizations specifically targeting transformers based models which makes it a strong candidate to consider when discussing performance.
  + Last but not least, this new unified benchmarking environment will allow us to easily run inference for different scenarios such as [Quantized Models (Zafrir & al.) (9)](https://arxiv.org/abs/1910.06188) using less precise number representations (float16, int8, int4).
  + the results below were run on Amazon Web Services (AWS) c5.metal instance leveraging an Intel Xeon Platinum 8275 CPU (48 cores/96 threads). The choice of this instance provides all the useful CPU features to speedup Deep Learning workloads such as:
    - **AVX512** instructions set (which might not be leveraged out-of-the-box by the various frameworks)
    - **Intel Deep Learning Boost (also known as Vector Neural Network Instruction - VNNI)** which provides specialized CPU instructions for running quantized networks (using int8 data type)
  + The choice of using **metal instance** is to avoid any virtualization issue which can arise when using cloud providers. This gives us full control of the hardware, especially while targeting the NUMA (Non-Unified Memory Architecture) controller, which we will cover later in this post.
* <https://huggingface.co/blog/bert-cpu-scaling-part-2>
  + Back in April 2021, Intel launched its latest generation of Intel Xeon processors, codename Ice Lake, targeting more efficient and performant AI workloads. More precisely, Ice Lake Xeon CPUs can achieve up to 75% faster inference on a variety of NLP tasks when comparing against the previous generation of Cascade Lake Xeon processors
* <https://huggingface.co/blog/optimize-llm>
* <https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices>
* <https://rasa.com/blog/compressing-bert-for-faster-prediction-2/>
* <https://rasa.com/blog/pruning-bert-to-accelerate-inference/>
* <https://pytorch.org/tutorials/intermediate/torchserve_with_ipex#grokking-pytorch-intel-cpu-performance-from-first-principles>
* <https://pytorch.org/docs/stable/jit.html>

# Tasks

1. Finish the reading material and sort out methods
2. Solve the issue of entity mismatch in CyNER
   1. WARNING:root:entity mismatch: .NET 2.0 vs NET 2.0.
   2. WARNING:root:entity mismatch: .NET vs NET
   3. WARNING:root:entity mismatch: sysprep.exe vs ysprep.exe'
   4. WARNING:root:entity mismatch: .NET vs NET
   5. WARNING:root:entity mismatch: .NET vs NET
   6. WARNING:root:entity mismatch: TerafficAnalyzer (at) yahoo vs TerafficAnalyzer (at) yahoo.
   7. WARNING:root:entity mismatch: . vs N
3. On Laptop
   1. Create small framework to test and measure latency via MLFlow
   2. Try to understand how to parallize between 4 cores
   3. Use optimum/bettertransformer
   4. Use torch.compile
   5. Use ONNX
4. Compare performances of raw huggingface models
5. ONNXRuntime
   1. Check that ONNX output is ok
   2. Parallelization - <https://discuss.huggingface.co/t/pass-cpu-cores-to-speed-up-inference/19100>
6. Future tasks
   1. Create one library instead of the branches of changes for tner/cyner
      1. Create the library so it would call standalone service ?

# 

# שאלות למטה

1. איך אפשרי/צריך שה-process יהיה מורץ
   1. ישירות ?
   2. Docker ?
   3. Kubernetes (open shift) ?
2. מה הגרסאת linux של המכונה ?
3. מה ה-spec של ה-cpu (אפשר לשלוח פלט של lscpu)
4. מה ה-extension-ים של ה-processor-ים (אפשר לשלוח פלט של cat /proc/cpuinfo)