Implementing Nested Tensors for Transformer Models

Kushaan Gowda (kg3081), Harsh Benahalkar (hb2776), and Siddarth Ijju (si2462)

Problem Statement

Problem: PyTorch implementations of the transformer are computationally expensive and memory intensive

Solution: use PyTorch's new NestedTensor functionality to reduce the memory footprint of transformer implementations

Expected Impact:

- Potential memory improvements from reducing padding token usage
- Potential speed improvements from removing operations on padded tokens

Problem Motivation

- Transformers are critical to the performance of large language models but are incredibly computationally expensive to run in practice.
- One bottleneck in the current PyTorch transformer mechanism is that each sequence in a batch must be padded to the same sequence length.
- One way of defining sequence length is by the largest input batch dimension, meaning that on average over half of an input tensor could be padded.
- This also directly affects the attention masks since they must also adhere to the largest sequence length in a batch.
- The memory footprint from padded tokens represents a significant obstacle in speeding up inference for transformer-based architectures

Background Work

- Continuous Batching
  - Improves batching for LLM inference in production, but still requires padding
- FlashAttention
  - Improves the memory footprint of naive attention through fused kernels, tile blocking, and dynamic attention mask computation
- Sparse Attention
  - Since a significant number of tokens are padded (effectively 0) in many practical applications, sparse mechanisms can significantly improve performance overall
- Other Attention Variants
  - Linear, Sliding Window, Memory Efficient, Paged
- NestedTensor
  - Allows for variable lengths of tensors, which is very useful for batching and removing padding

Technical Challenges and Bottlenecks

NestedTensor

- Many features not fully implemented - work around these to implement in forward pass.
- shape() is not defined for a nested tensor - work around to use specific dimensions
- Cannot perform batched operations - work on each tensor in a batch individually

Attention

- Often uses a positional embedding that is closely related to the sequence length of the input
- Not immediately clear how to change attention mechanisms for variable length inputs
- Many attention mechanisms are abstracted out e.g. FlashAttention's fused kernel

Approach (1/2)

1. Created the forward inference pipeline with a custom Dataset and custom Datacollator to handle nested and padded tensors.
2. Modified the fms code to handle nested and padded tensors simultaneously (baseline).
3. Performed time profiling and memory profiling on multiple batches of data.
4. Re-modified the fms code (optimized).

```python
class NestedTensorCollator():

    def __init__(self, tokenizer, device, max_model_size, is_nest_required):
        self.tokenizer = tokenizer
        self.is_nest_required = is_nest_required
        self.max_model_size = max_model_size
        self.device = device

    def __call__(self, examples):
        """ tokenize string data and then nest it """
        features = list(map(lambda x : x["features"], examples))
        labels = torch.tensor(list(map(lambda x : x["labels"], examples)))

        if self.is_nest_required:
            features = self.tokenizer(
                features,
                return_tensors=None,
                padding=False,
                truncation=False,
            )
            input_ids, attention_mask = [], []
            for input_id, a_mask in zip(features["input_ids"], features["attention_mask"]):
                input_ids.append(torch.tensor(input_id).remainder(self.max_model_size - 1))
                attention_mask.append(torch.tensor(a_mask).remainder(self.max_model_size - 1))

            input_ids = torch.nested.nested_tensor(input_ids).to(self.device)
            attention_mask = torch.nested.nested_tensor(attention_mask).to(self.device)

        else:
            self.tokenizer.pad_token = self.tokenizer.eos_token

            features = self.tokenizer(
                features,
                return_tensors="pt",
                padding="max_length",
                max_length=100,
                truncation=True,
            )
            input_ids = features["input_ids"].remainder(self.max_model_size - 1).to(self.device)
            attention_mask = features["attention_mask"].remainder(self.max_model_size - 1).to(self.device)

        return {"input_ids": input_ids, "attention_mask": attention_mask, "labels": labels}
```

Custom DataCollator

Approach (2/2)

fms/models/llama.py
  class LLaMABlock → forward
  class LLAMA → _helper

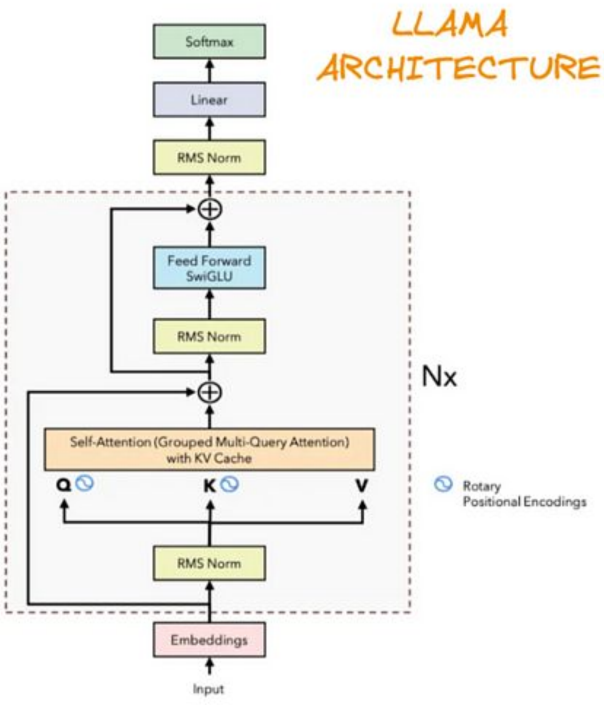
fms/models/hf/modeling_hf_adapter.py
  class HFDecoderModelArchitecture → _produce_decoder_attention_mask_from_hf

fms/models/hf/utils.py → mask_2d_to_3d

fms/models/attention.py
  class MultiHeadAttention → forward

fms/models/layernorm.py
  class LayerNormParameterized → forward

fms/models/positions.py
  class RotaryEmbedding → adjusted_qk



Llama architecture

Current Implementation

main.py

load data, model, run on sample batch, and save to file

eval.py

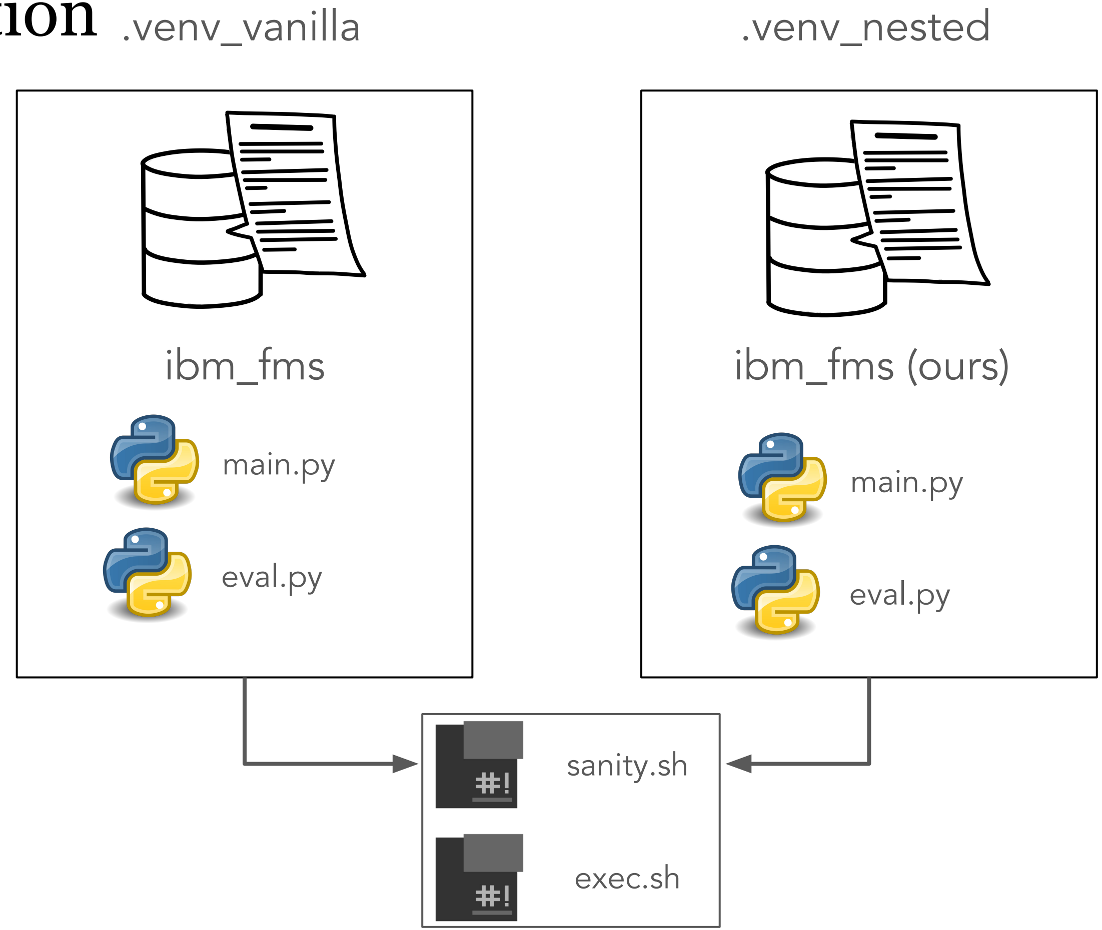
truncate and compare real and nested tensor outputs

sanity.sh

check difference for non-nested tensors

exec.sh

check difference for nested tensors



Codebase for Implementation and Evaluation

Two repositories:

1. ibm-fms: https://github.com/benahalkar/ibm-fms
   a. Library changes
   b. Main changes: llama.py, modeling_hf_adapter.py, utils.py, attention.py, feedforward.py, layernorm.py, positions.py
2. nested-tensors: https://github.com/kushaangowda/nestedtensors-for-transformers
   a. Evaluation and comparison scripts
   b. Custom NestedTensor data generation, loading, and collation
   c. Time profiling and result matching
   d. Comparison library and utils for benchmarking and library debugging

Experiment Design Flow

Three avenues:

1. Sanity check - compare outputs from library after nested tensor changes to vanilla library on an input that is <u>not nested</u>
   a. Ensures changes do not break existing code
2. Comparison check - compare outputs from library after nested tensor changes to vanilla library on an input that is nested
   a. Checks that nested tensor changes function as intended and that output matches the original implementation
   b. Time profiling by saving tensors at various checkpoints in the library and recording time at that point
3. Custom LLaMa classes
   a. Demonstrate memory benefits of nested tensor in real encoder/decoder implementations

Experimental Evaluation (1/4)

Model: Llama

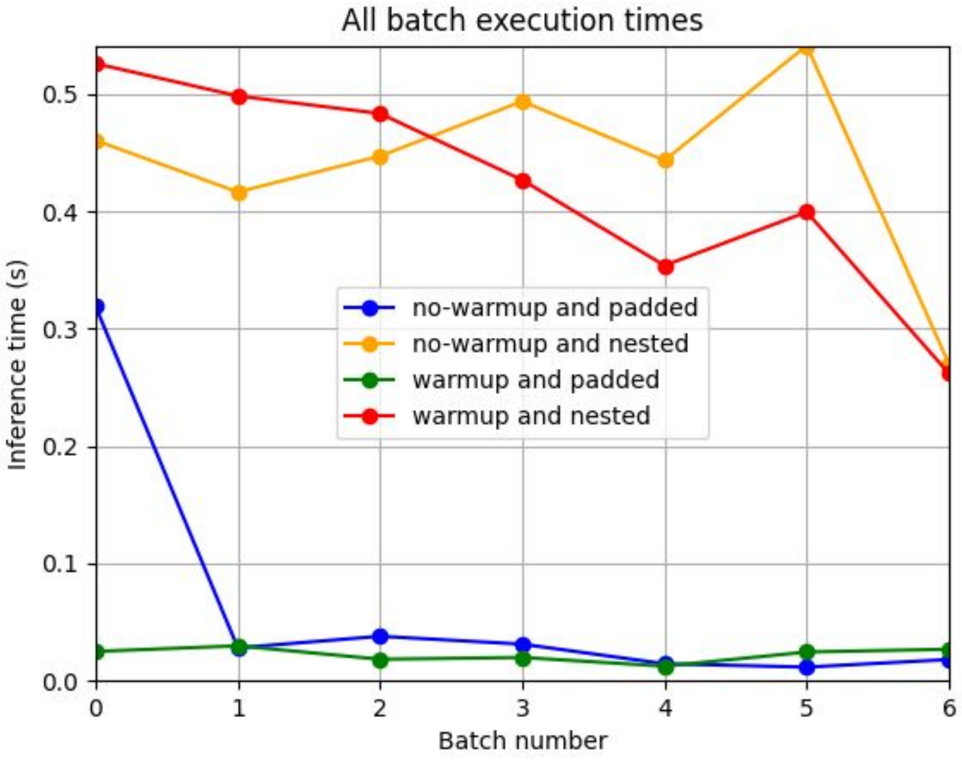
Seq len: 10 - 256

Samples: 200

Batch size: 32

Seed: 555

took average of 5 readings



Experimental Evaluation (2/4)

```
start = time.monotonic()
if not (k.is_nested ^ q.is_nested):
    if not k.is_nested:
        seq_len = max(k.size(1), q.size(1))
    else:
        seq_len = max(
            max([ele.size(0) for ele in k]),
            max([ele.size(0) for ele in q])
        )

else:
    raise ValueError("Either of the two, K or Q, is not nested... the other is")

torch.cuda.synchronize()
print(f"seqlen in position time: {time.monotonic()-start}")
```

fms/modules/positions.py

NESTED TENSORS	PADDED TENSORS
0.0099599	4.7594000e-05
0.0095281	4.1668999e-05
0.0035445	0.0059985
0.0009103	4.5513999e-05
0.0008670	3.9121000e-05

Execution Time (s)

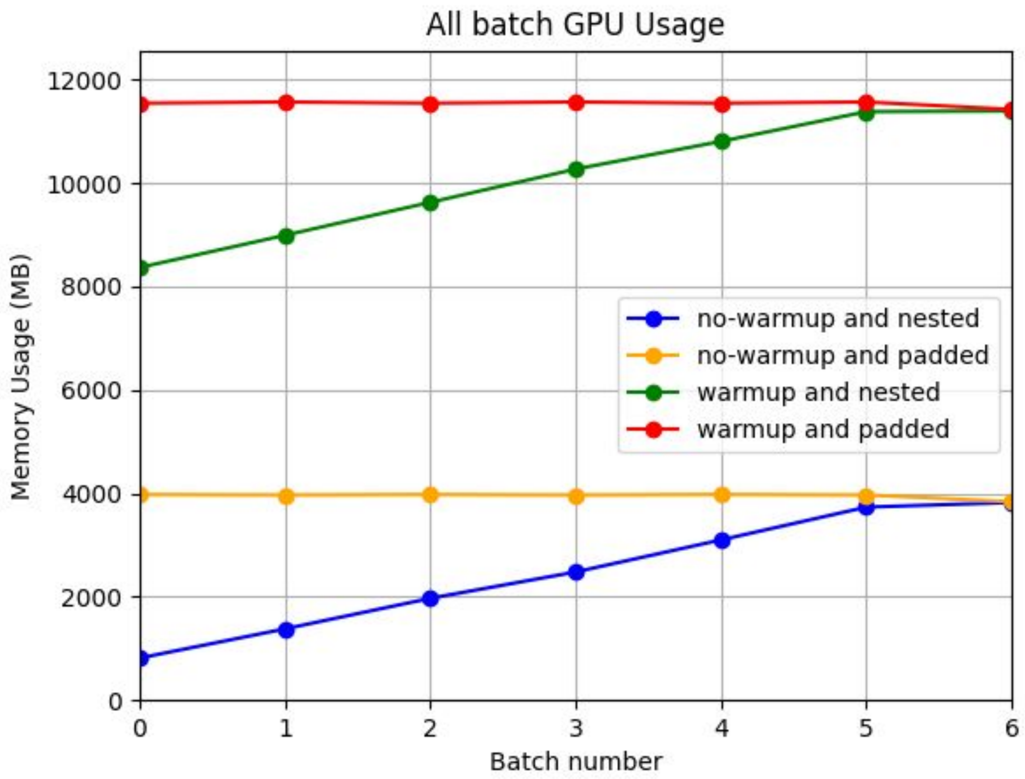
```
attn = F.scaled_dot_product_attention(
    queries,
    keys_e,
    values_e,
    attn_mask=attn_mask,
    dropout_p=self.p_dropout if self.training else 0.0,
    is_causal=is_causal_mask,
    scale=self.scale_factor,
)
```

fms/modules/attention.py

NESTED TENSORS	PADDED TENSORS
0.0041829	0.0002706
0.0040848	0.0001653
0.0039630	0.0001246
0.0051057	0.0001519
0.0038408	0.0001088

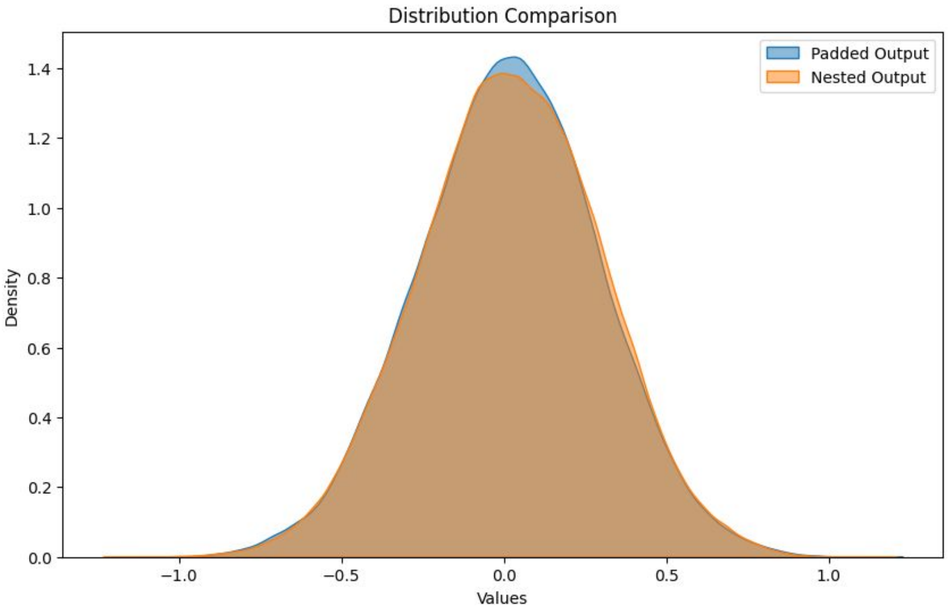
Execution Time (s)

Experimental Evaluation (3/4)



Output Distribution

Tolerance Issue with output logits:
2e-6



KDE Plot comparing outputs

```python
nested_in = torch.nested.nested_tensor([
    torch.randn(5, 4),
    torch.randn(1, 4),
    torch.randn(2, 4),
])

padded_in = torch.nested.to_padded_tensor(nested_in,0)

linear = Linear(4,3, bias=False)

padded_out = linear(padded_in)
nested_out = linear(nested_in)

nested_out-truncate_to_nested(nested_out,padded_out)
```

```
nested_tensor([
  tensor([[-8.731149137020111083980e-10,  0.000000000000000000000e+00,
            7.450580596923828125000e-09],
          [-2.235174179077148437500e-08,  1.490116119384765625000e-08,
           -3.725290298461914062500e-08],
          [ 1.490116119384765625000e-08,  2.980232238769531250000e-08,
            0.000000000000000000000e+00],
          [-7.450580596923828125000e-09,  2.980232238769531250000e-08,
           -5.960464477539062500000e-08],
```

Conclusion

- Nested Tensor implementation shows promise?
  - Memory footprint is much lower in custom-built framework
  - Certain native pytorch functions are slower for nested tensors as compared to padded tensors
  - Certain features like torch.ndim need to be used for faster computation
- Future work in fms still needed for full compatibility
  - Fundamental difference in how positional encoding works with nested tensors versus a padded tensor
  - Current design forces causal_mask to be true - account for other case as well
- Future work for nested tensors for ease of use
  - Compatibility function for conversions between nested and real (i.e. truncate_to_nested)
  - Change torch functional scaled_dot_product_attention to use nested tensors

Contributions

Harsh Benahalkar

- Dataset class and Datacollator class for HF and IBM inference pipeline.
- Baseline NestedTensor forward inference implementation.
- Tensor eval and time profiling for tolerance verification.

Kushaan Gowda

- Preliminary experiments on decoder models.
- Optimized implementation for NestedTensor on forward inference.
- Performed memory profiling on nested and padded tensors.

Sid Ijju

- Benchmarking inference setup and runs.
- Debug key issues in fms.
- Develop truncation methods and comparisons.
- Slides and report.