Use Case:
Document Summarization

Exercises

1) Train a SOTA summarization model for long input documents
2) Try memory-optimization techniques
3) Explore how to design the attention pattern for a task
4) Convert an existing pretrained short model into a long one

Exercise 1 - Train a long-doc summarization model

- Task: Summarization
- Model: LongformerEncoderDecoder (LED)
- Dataset: arXiv
  - Given a paper text, generate its abstract
  - Input length: 16k tokens
  - Output length: 256 tokens
- Metric: rouge1
- Tools
  - Huggingface transformers: model implementation and pretrained weights
  - Pytorch Lightning: training loop
- Code: https://github.com/allenai/naacl2021-longdoc-tutorial

Code Skeleton

```python
class SummarizationDataset(Dataset):
    """
    HF arXiv Dataset Wrapper. It handles tokenization, max input/output seqlen,
    padding and batching
    """

    def __init__(self, hf_arxiv_dataset, tokenizer, args):

    def __len__(self):
        """Returns length of the dataset"""

    def __getitem__(self, idx):
        """
        Gets an example from the dataset. The input and output are tokenized
        and truncated to a certain seqlen
        """

    @staticmethod
    def collate_fn(batch):
        """
        Groups multiple examples into one batch with padding and tensorization.
        The collate function is called by PyTorch DataLoader
        """
```

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps model, data loading and training code"""

    def __init__(self, params):
        """Loads the model, the tokenizer and the metric."""

    def _set_global_attention_mask(self, input_ids):
        """Configure the global attention pattern based on the task"""

    def forward(self, input_ids, output_ids):
        """Call LEDForConditionalGeneration.forward"""

    def training_step(self, batch, batch_nb):
        """Call the forward pass then return loss"""

    def configure_optimizers(self):
        """Configure the optimizer and the learning rate scheduler"""

    def train_dataloader(self):
        """Get training dataloaders"""

    def val_dataloader(self):
        """Get validation dataloaders"""

    def validation_step(self, batch, batch_nb):
        """Validation - generate, compare with gold, compute rouge1, return result"""

    @staticmethod
    def add_model_specific_args(parser):
        """Command line arguments"""
```

```python
if __name__ == "__main__":
    """Read command line args, construct Pytorch Lightning module then train it"""
```

UWNLP AI2

Code Skeleton

```python
class SummarizationDataset(Dataset):
    """
    HF arXiv Dataset Wrapper. It handles tokenization, max input/output seqlen,
    padding and batching
    """

    def __init__(self, hf_arxiv_dataset, tokenizer, args):

    def __len__(self):
        """Returns length of the dataset"""

    def __getitem__(self, idx):
        """
        Gets an example from the dataset. The input and output are tokenized
        and truncated to a certain seqlen
        """

    @staticmethod
    def collate_fn(batch):
        """
        Groups multiple examples into one batch with padding and tensorization.
        The collate function is called by PyTorch DataLoader
        """
```

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps model, data loading and training code"""

    def __init__(self, params):
        """Loads the model, the tokenizer and the metric."""

    def _set_global_attention_mask(self, input_ids):
        """Configure the global attention pattern based on the task"""

    def forward(self, input_ids, output_ids):
        """Call LEDForConditionalGeneration.forward"""

    def training_step(self, batch, batch_nb):
        """Call the forward pass then return loss"""

    def configure_optimizers(self):
        """Configure the optimizer and the learning rate scheduler"""

    def train_dataloader(self):
        """Get training dataloaders"""

    def val_dataloader(self):
        """Get validation dataloaders"""

    def validation_step(self, batch, batch_nb):
        """Validation - generate, compare with gold, compute rouge1, return result"""

    @staticmethod
    def add_model_specific_args(parser):
        """Command line arguments"""
```

```python
if __name__ == "__main__":
    """Read command line args, construct Pytorch Lightning module then train it"""
```

```python
class SummarizationDataset(Dataset):
    """HF arXiv Dataset Wrapper. It handles tokenization, max input/output seqlen, padding and batching"""

    def __init__(self, hf_arxiv_dataset, tokenizer, args):
        self.hf_arxiv_dataset = hf_arxiv_dataset
        self.tokenizer = tokenizer
        self.args = args

    def __len__(self):
        """Returns length of the dataset"""
        return len(self.hf_arxiv_dataset)

    def __getitem__(self, idx):
        """Gets an example from the dataset. The input and output are tokenized and limited to a certain seqlen."""
        entry = self.hf_arxiv_dataset[idx]
        input_ids = self.tokenizer.encode(entry['article'], truncation=True, max_length=self.args.max_input_len)
        output_ids = self.tokenizer.encode(entry['abstract'], truncation=True, max_length=self.args.max_output_len)
        return torch.tensor(input_ids), torch.tensor(output_ids)

    @staticmethod
    def collate_fn(batch):
        """
        Groups multiple examples into one batch with padding and tensorization.
        The collate function is called by PyTorch DataLoader
        """
        pad_token_id = 1
        input_ids, output_ids = list(zip(*batch))
        input_ids = torch.nn.utils.rnn.pad_sequence(input_ids, batch_first=True, padding_value=pad_token_id)
        output_ids = torch.nn.utils.rnn.pad_sequence(output_ids, batch_first=True, padding_value=pad_token_id)
        return input_ids, output_ids
```

Code Skeleton

```python
class SummarizationDataset(Dataset):
    """
    HF arXiv Dataset Wrapper. It handles tokenization, max input/output seqlen,
    padding and batching
    """

    def __init__(self, hf_arxiv_dataset, tokenizer, args):

    def __len__(self):
        """Returns length of the dataset"""

    def __getitem__(self, idx):
        """
        Gets an example from the dataset. The input and output are tokenized
        and truncated to a certain seqlen
        """

    @staticmethod
    def collate_fn(batch):
        """
        Groups multiple examples into one batch with padding and tensorization.
        The collate function is called by PyTorch DataLoader
        """
```

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps model, data loading and training code"""

    def __init__(self, params):
        """Loads the model, the tokenizer and the metric."""

    def _set_global_attention_mask(self, input_ids):
        """Configure the global attention pattern based on the task"""

    def forward(self, input_ids, output_ids):
        """Call LEDForConditionalGeneration.forward"""

    def training_step(self, batch, batch_nb):
        """Call the forward pass then return loss"""

    def configure_optimizers(self):
        """Configure the optimizer and the learning rate scheduler"""

    def train_dataloader(self):
        """Get training dataloaders"""

    def val_dataloader(self):
        """Get validation dataloaders"""

    def validation_step(self, batch, batch_nb):
        """Validation - generate, compare with gold, compute rouge1, return result"""

    @staticmethod
    def add_model_specific_args(parser):
        """Command line arguments"""
```

```python
if __name__ == "__main__":
    """Read command line args, construct Pytorch Lightning module then train it"""
```

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps up the model, data loading and training code"""

    def __init__(self, params):
        """Loads the model, the tokenizer and the metric."""
        super().__init__()
        self.args = params

        # Load and update config then load a pretrained LEDForConditionalGeneration
        config = AutoConfig.from_pretrained('allenai/led-base-16384')
        config.gradient_checkpointing = self.args.grad_ckpt
        config.attention_window = [self.args.attention_window] * len(config.attention_window)
        self.model = AutoModelForSeq2SeqLM.from_pretrained('allenai/led-base-16384', config=config)

        # Load tokenizer and metric
        self.tokenizer = AutoTokenizer.from_pretrained('allenai/led-base-16384', use_fast=True)
        self.rouge = datasets.load_metric('rouge')

    def _set_global_attention_mask(self, input_ids):
        '''Configure the global attention pattern based on the task'''

        # Local attention everywhere - no global attention
        global_attention_mask = torch.zeros(input_ids.shape, dtype=torch.long, device=input_ids.device)
        return global_attention_mask

    def forward(self, input_ids, output_ids):
        """Call LEDForConditionalGeneration.forward"""
        return self.model(input_ids,
                          attention_mask=(input_ids != self.tokenizer.pad_token_id),  # mask padding tokens
                          global_attention_mask=self._set_global_attention_mask(input_ids),  # set global attention
                          labels=output_ids, use_cache=False)

    def training_step(self, batch, batch_nb):
        """Call the forward pass then return loss"""
        outputs = self.forward(*batch)
        return {'loss': outputs.loss}
```

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps up the model, data loading and training code"""

    ....

    def validation_step(self, batch, batch_nb):
        """Validation - predict output, compare it with gold, compute rouge1, and return result"""
        # Generate
        input_ids, output_ids = batch
        generated_ids = self.model.generate(input_ids=input_ids,
                                            attention_mask=(input_ids != self.tokenizer.pad_token_id),
                                            global_attention_mask=self._set_global_attention_mask(input_ids),
                                            use_cache=True, max_length=self.args.max_output_len, num_beams=1)

        # Convert predicted and gold token ids to strings
        predictions = self.tokenizer.batch_decode(generated_ids.tolist(), skip_special_tokens=True)
        references = self.tokenizer.batch_decode(output_ids.tolist(), skip_special_tokens=True)

        # Compute rouge
        results = self.rouge.compute(predictions=predictions, references=references)
        rouge1 = input_ids.new_zeros(1) + results["rouge1"].mid.fmeasure

        # Log metric
        self.log('val_rouge1', rouge1, on_step=False, on_epoch=True, sync_dist=True, prog_bar=True)
        return rouge1
```

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps up the model, data loading and training code"""

    ....

    def configure_optimizers(self):
        """Configure the optimizer and the learning rate scheduler"""
        optimizer = torch.optim.Adam(self.model.parameters(), lr=self.args.lr)
        dataset_size = len(self.hf_dataset['train'])
        gpu_count = torch.cuda.device_count()
        num_steps = dataset_size * self.args.epochs / gpu_count / self.args.grad_accum / self.args.batch_size
        scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=self.args.warmup,
                                                    num_training_steps=num_steps)
        return [optimizer], [{"scheduler": scheduler, "interval": "step"}]

    def _get_dataloader(self, split_name, is_train):
        """Get training and validation dataloaders"""
        dataset_split = self.hf_dataset[split_name]
        dataset = SummarizationDataset(hf_dataset=dataset_split, tokenizer=self.tokenizer, args=self.args)
        sampler = torch.utils.data.distributed.DistributedSampler(dataset, shuffle=is_train)
        return DataLoader(dataset, batch_size=self.args.batch_size, shuffle=(sampler is None),
                          num_workers=self.args.num_workers, sampler=sampler,
                          collate_fn=SummarizationDataset.collate_fn)

    def train_dataloader(self):
        return self._get_dataloader('train', is_train=True)

    def val_dataloader(self):
        return self._get_dataloader('validation', is_train=False)
```

Code Skeleton

```
class SummarizationDataset(Dataset):
    """
    HF arXiv Dataset Wrapper. It handles tokenization, max input/output seqlen,
    padding and batching
    """

    def __init__(self, hf_arxiv_dataset, tokenizer, args):

    def __len__(self):
        """Returns length of the dataset"""

    def __getitem__(self, idx):
        """
        Gets an example from the dataset. The input and output are tokenized
        and truncated to a certain seqlen
        """

    @staticmethod
    def collate_fn(batch):
        """
        Groups multiple examples into one batch with padding and tensorization.
        The collate function is called by PyTorch DataLoader
        """
```

```
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps model, data loading and training code"""

    def __init__(self, params):
        """Loads the model, the tokenizer and the metric."""

    def _set_global_attention_mask(self, input_ids):
        """Configure the global attention pattern based on the task"""

    def forward(self, input_ids, output_ids):
        """Call LEDForConditionalGeneration.forward"""

    def training_step(self, batch, batch_nb):
        """Call the forward pass then return loss"""

    def configure_optimizers(self):
        """Configure the optimizer and the learning rate scheduler"""

    def train_dataloader(self):
        """Get training dataloaders"""

    def val_dataloader(self):
        """Get validation dataloaders"""

    def validation_step(self, batch, batch_nb):
        """Validation - generate, compare with gold, compute rouge1, return result"""

    @staticmethod
    def add_model_specific_args(parser):
        """Command line arguments"""
```

```
if __name__ == "__main__":
    """Read command line args, construct Pytorch Lightning module then train it"""
```

```python
if __name__ == "__main__":
    # Setup command line args
    main_arg_parser = argparse.ArgumentParser(description="summarization")
    parser = Summarizer.add_model_specific_args(main_arg_parser)
    args = parser.parse_args()

    # Init a PL module
    set_seed(args.seed)
    summarizer = Summarizer(args)

    # Load the arXiv dataset from HF datasets
    summarizer.hf_dataset = datasets.load_dataset('scientific_papers', 'arxiv')

    # Construct a PL trainer
    trainer = pl.Trainer(gpus=-1,
                         accelerator='ddp',
                         plugins=[pl.plugins.DDPPlugin(find_unused_parameters=False)],  # for grad. checkpointing
                         max_epochs=args.epochs,
                         replace_sampler_ddp=False,
                         num_sanity_val_steps=0,
                         limit_val_batches=args.limit_val_batches,
                         limit_train_batches=args.limit_train_batches,
                         precision=16 if args.fp16 else 32,
                         accumulate_grad_batches=args.grad_accum,
                         )
    # Start training
    trainer.fit(summarizer)
```

Exercise 1 - Train a summarization model

- Around 100 lines of code
- Simply load the model then train it on input/output pairs no matter how long the input is.

- Result
    - val_rouge1: 43.2 (sota 46.6 using LED-large)

```
CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7  python summarization.py  \
    --limit_val_batches 1.0   --limit_train_batches 1.0  \
    --max_input_len 16384  \
    --batch_size 1  --grad_accum 2  \
    --fp16  --grad_ckpt
```

Exercises 2 - Try memory optimization methods

Target configuration

- input seqlen = 16k tokens
- batch size = 4
- gpus: 1 x 16GB

GPU memory is not enough to run this configuration

Exercises 2 - Try memory optimization methods

In general, memory is the bottleneck for long-document models.

Efficient transformer models reduce memory requirements of self-attention …

but, the model still needs a lot of memory for the feed forward layers.

Memory optimizations:

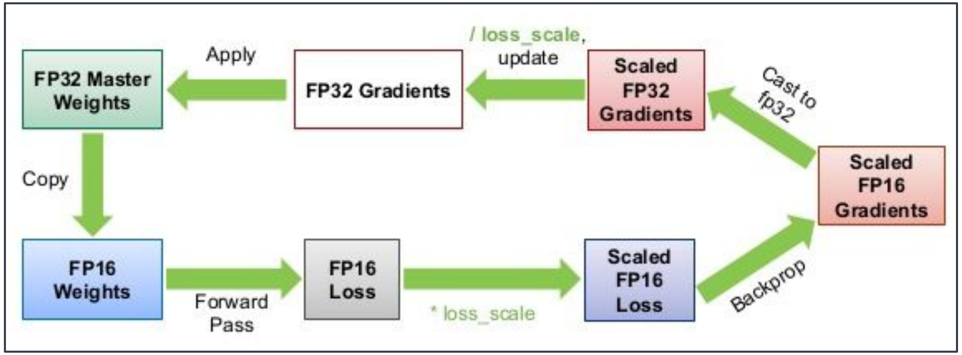
- Truncation: limit input sequence length
- Automatic Mixed precision (fp16)
- Gradient accumulation
- Gradient checkpointing

Automatic Mixed Precision (AMP)

- Half-precision (FP16)
  - Uses 16bit to store floating points (in contrast with standard single-precision (FP32))
  - The low precision of FP16 is good enough for most NN computations
  - Smaller models and activations ⇒ less memory
- But lower precision ⇒ numerical instabilities

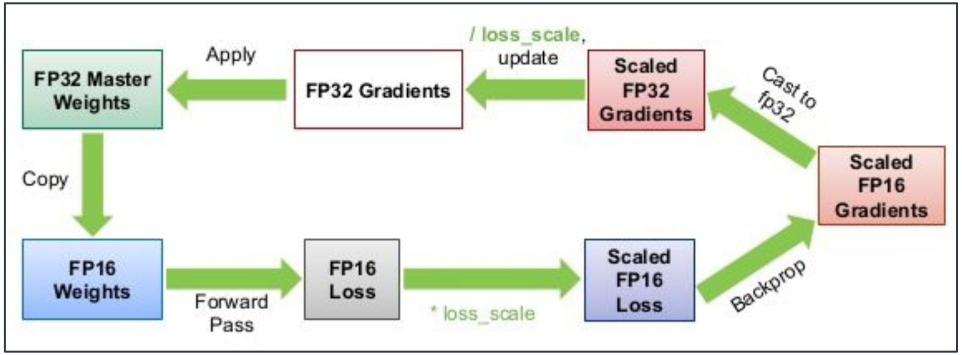
Automatic Mixed Precision (AMP)

- Automatic mixed precision
  - Uses a mix of FP16 and FP32
  - Automatically switching between them
  - The tool knows which operations to run in FP16 vs. Fp32
  - Numerical instabilities still happen



Automatic Mixed Precision (AMP)

- Automatic mixed precision
  - Uses a mix of FP16 and FP32
  - Automatically switching between them
  - The tool knows which operations to run in FP16 vs. Fp32
  - Numerical instabilities still happen
- Tools
  - Natively supported in Pytorch>=1.6.0
  - Pytorch lightning: flag



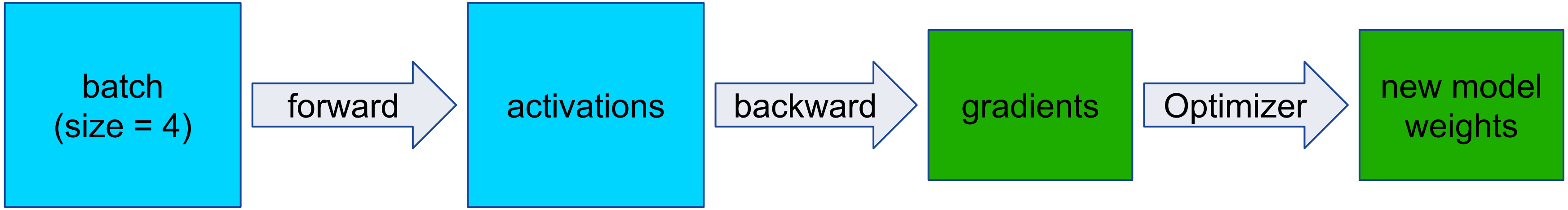
Gradient Accumulation

- Simulates a large **effective batch size** using a much smaller actual batch size

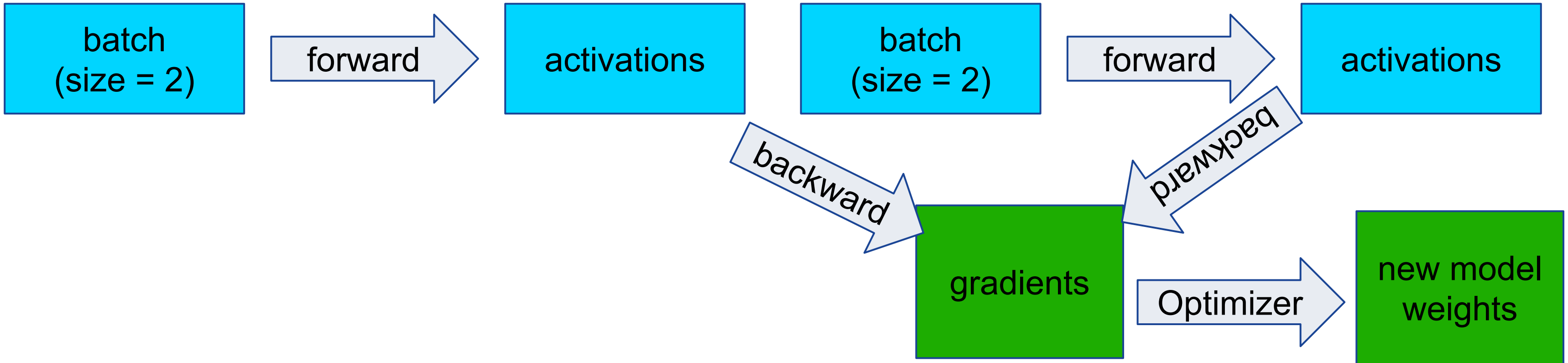
- Splits the large batch into smaller ones, then average their gradients

Gradient Accumulation

No gradient accumulation



With 2-step gradient accumulation



Gradient Checkpointing*

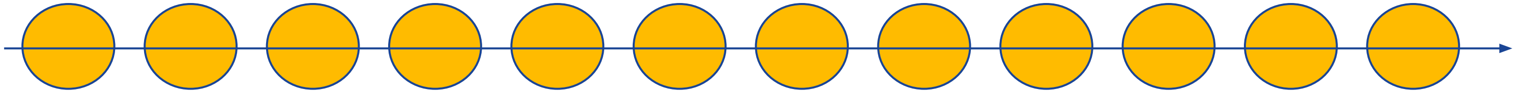
- Reduces memory usage at the expense of additional compute
- In the forward pass, saves only a selected set of activations, not all of them
- In the backward pass, recompute the missing activations to compute gradients

*Hermans et. al., ACML 2017

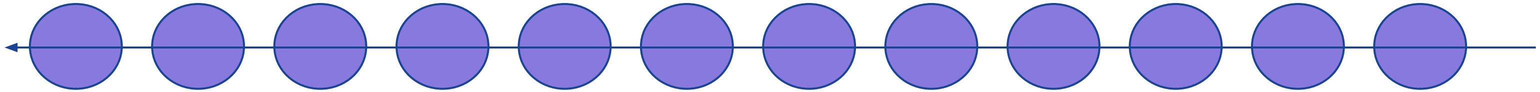
Gradient Checkpointing

- No gradient checkpointing

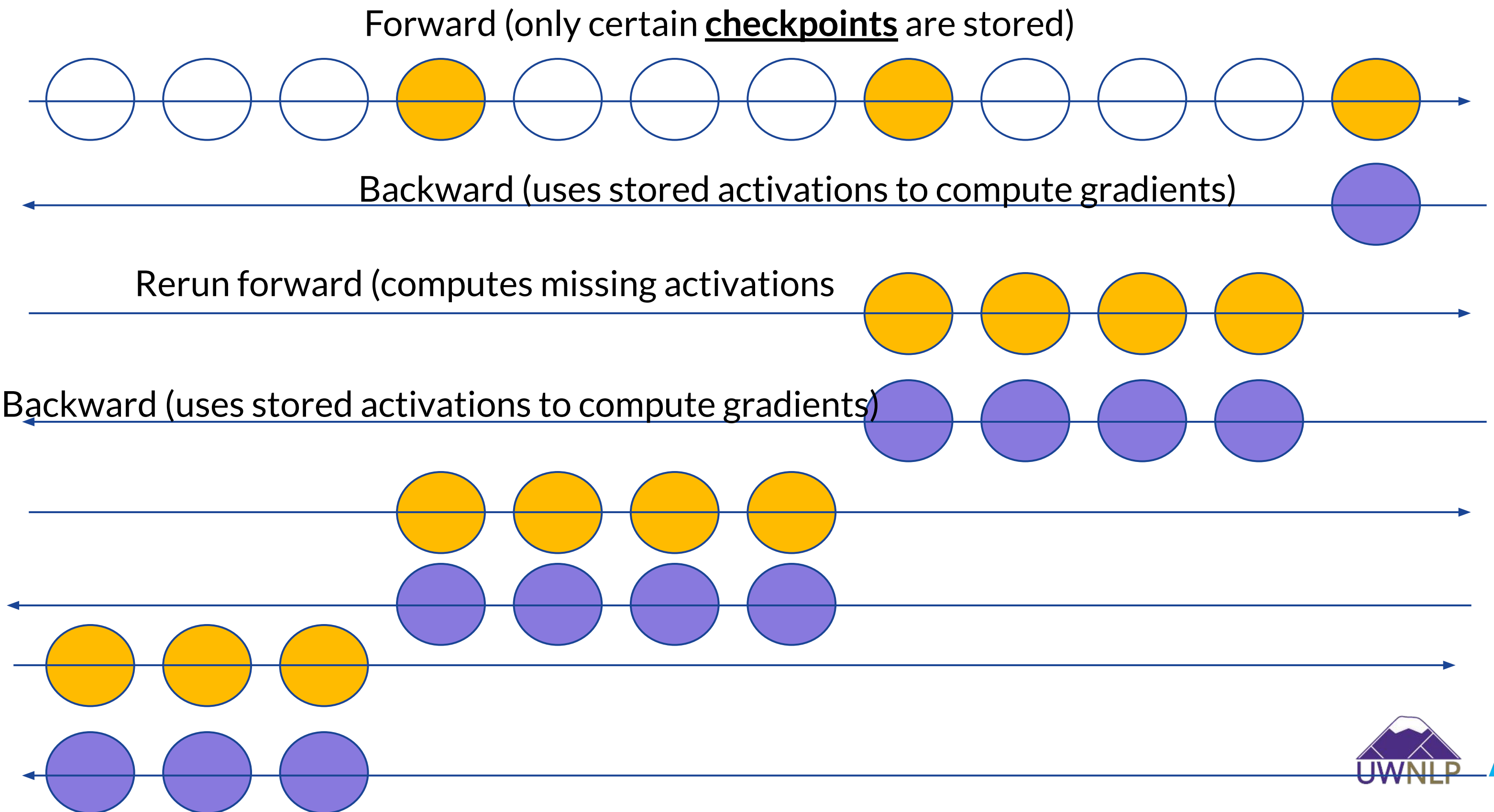
Forward (computes and **<u>stores</u>** activations)



Backward (uses stored activations to compute gradients)



Gradient Checkpointing



Gradient Checkpointing

- Compute: 2 x forward + 1 x backward
  - Usually fast
- Design consideration: where to place checkpoints?
  - Too many checkpoints ⇒ large memory to store activations of the checkpoints
  - Too few checkpoints => large memory to store activations between two checkpoints
- Pytorch support `torch.utils.checkpoint.checkpoint`
  - Check the code for two caveats to get to work with `DistributedDataParallel`
- LongformerEncoderDecoder (LED)
  - Already supported in the HF implementation (one checkpoint before each layer)

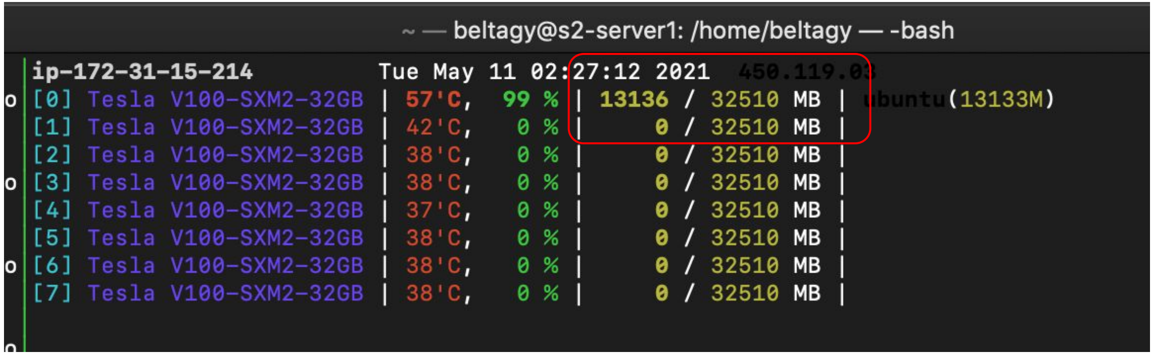
Demo

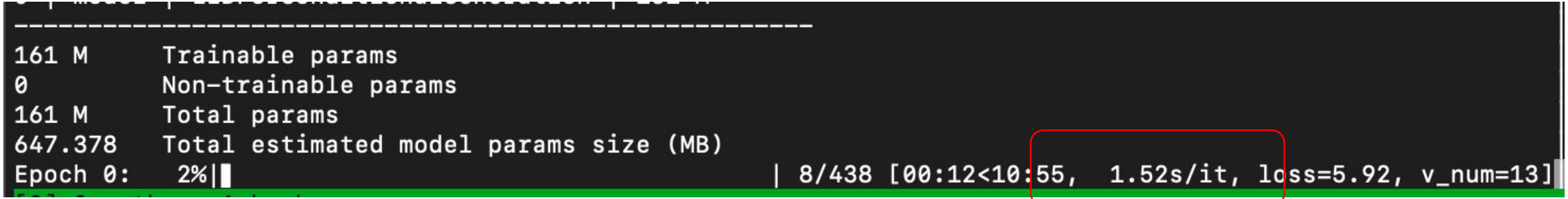
Target configuration

- input seqlen = 16k tokens
- output seqlen = 256 tokens
- model size: base
- batch size = 4
- gpus: 1 x 16GB

Demo

- Try the various memory optimizations and monitor memory and speed
- GPU: 32GB V100





Demo

```python
class SummarizationDataset(Dataset):
    """HF arXiv Dataset Wrapper. It handles tokenization, max input/output seqlen, padding and batching"""

    ...

    def __getitem__(self, idx):
        """Gets an example from the dataset. The input and output are tokenized and limited to a certain seqlen."""
        entry = self.hf_arxiv_dataset[idx]
        input_ids = self.tokenizer.encode(entry['article'], truncation=True, max_length=self.args.max_input_len,
                                          padding='max_length')  # padding to max seqlen for const memory/example
        output_ids = self.tokenizer.encode(entry['abstract'], truncation=True, max_length=self.args.max_output_len,
                                           padding='max_length')  # padding to max seqlen for const memory/example
        return torch.tensor(input_ids), torch.tensor(output_ids)
```

Demo

Demo

Demo

batch size	gradient accumulation	fp16	gradient checkpointing	max input length	time/step	memory
4	1	no	no	3k	1.5s	28GB
1	4	yes	yes	16k	5.96s	13GB
2	2	yes	yes	16k	5.92s	18GB

FP16 is usually a good idea (as long as it doesn't cause NaN). It saves memory and compute

¼ batch size doesn't ¼ memory because it only affects activation while gpu memory is used by model and optimizer state as well

same applies to sequence length

Gradient checkpointing reduced memory 2.3x and increase compute to 1.24x. The memory saving is even larger for deeper models

Gradient accumulation saves memory at the expense of slight slow down. The slow down can be larger if the smaller batch size can't push the gpu to high utilization

Exercise 3 - Explore attention pattern designs

- Local attention
  - Larger attention window ⇒ more representative model
  - Smaller attention window ⇒ faster model
  - Mix of small and large attention window ⇒ balances representation and speed

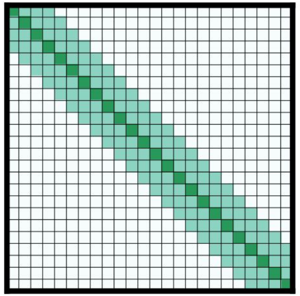
```
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps up the model, data loading and training code"""

    def __init__(self, params):
        """Loads the model, the tokenizer and the metric."""
        super().__init__()
        self.args = params

        # Load and update config then load a pretrained LEDForConditionalGeneration
        config = AutoConfig.from_pretrained('allenai/led-base-16384')
        config.gradient_checkpointing = self.args.grad_ckpt
        config.attention_window = [self.args.attention_window] * len(config.attention_window)
        self.model = AutoModelForSeq2SeqLM.from_pretrained('allenai/led-base-16384', config=config)
```

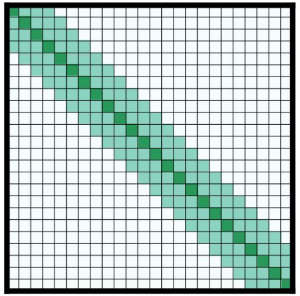
Exercise 3 - Explore attention pattern designs

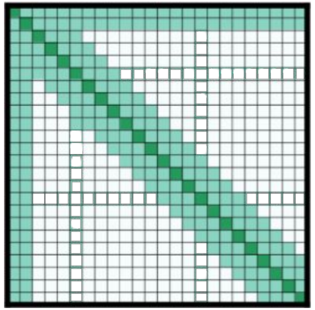
- Global attention
  - No global attention: works well for summarization because the decoder can see all input tokens



Exercise 3 - Explore attention pattern designs

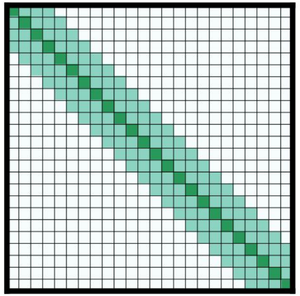
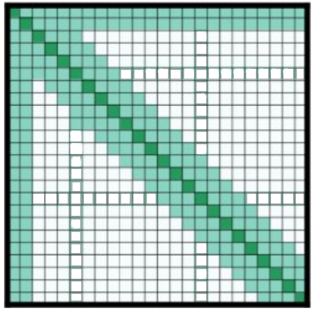
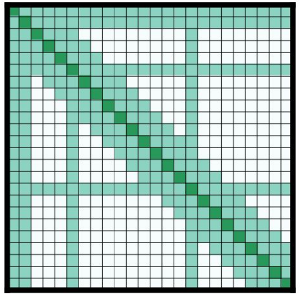
- Global attention
  - No global attention: works well for summarization because the decoder can see all input tokens
  - Global attention on a fixed-sized block at the beginning of the sequence
    - Intuition: model uses it as memory to store global information





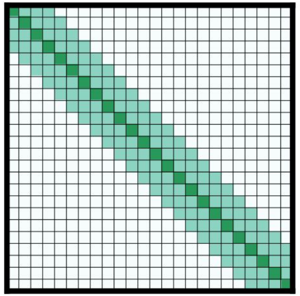
Exercise 3 - Explore attention pattern designs

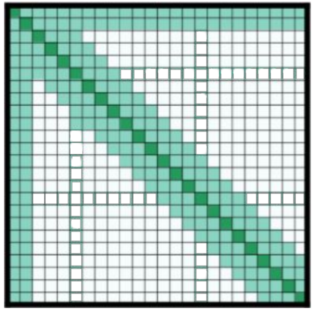
- Global attention
  - No global attention: works well for summarization because the decoder can see all input tokens
  - Global attention on a fixed-sized block at the beginning of the sequence
    - Intuition: model uses it as memory to store global information
  - Global attention **on periods** (one for each sentence)
    - Intuition: allow easy transfer of information between sentences
    - Works well for sentence-level extractive summarization

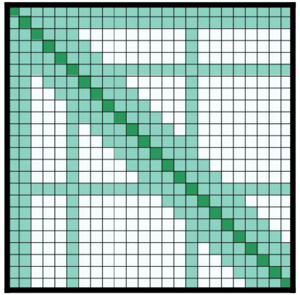




Exercise 3 - Explore attention pattern designs

- Global attention
  - No global attention: works well for summarization because the decoder can see all input tokens
  - Global attention on a fixed-sized block at the beginning of the sequence
    - Intuition: model uses it as memory to store global information
  - Global attention **on periods** (one for each sentence)
    - Intuition: allow easy transfer of information between sentences
    - Works well for sentence-level extractive summarization
- Design consideration: more global attention means more memory







Exercise 3 - Explore attention pattern designs

```python
class Summarizer(pl.LightningModule):
    """Pytorch Lightning module. It wraps up the model, data loading and training code"""

    ...

    def _set_global_attention_mask(self, input_ids):
        """Configure the global attention pattern based on the task"""

        # Local attention everywhere - no global attention
        global_attention_mask = torch.zeros(input_ids.shape, dtype=torch.long, device=input_ids.device)

        # # Global attention on the first 100 tokens
        global_attention_mask[:, :100] = 1

        # # Global attention on periods
        global_attention_mask[(input_ids == self.tokenizer.convert_tokens_to_ids('.'))] = 1

        return global_attention_mask

    def forward(self, input_ids, output_ids):
        """Call LEDForConditionalGeneration.forward"""
        return self.model(input_ids,
                          attention_mask=(input_ids != self.tokenizer.pad_token_id),  # mask padding tokens
                          global_attention_mask=self._set_global_attention_mask(input_ids),  # set global attention
                          labels=output_ids, use_cache=False)
```

Exercise 4 - from short to long model

Convert a pretrained BART checkpoint into one that works for long sequences

Assume long input (max: 16k tokens) but relatively short output

Model components of interest (position embeddings and attention):

- Encoder position embeddings: extend to 16k
- Encoder self-attention: replace with LEDEncoderSelfAttention
- Decoder position embedding: no changes
- Decoder self-attention: no changes
- Decoder cross-attention: no changes

```python
import copy
from transformers import BartForConditionalGeneration
from transformers.models.led.modeling_led import LEDEncoderAttention

# desired configuration
max_input_len = 16384 + 2
attention_window = 1024

# load pretrained BART
model = BartForConditionalGeneration.from_pretrained('facebook/bart-base')

# *********** 1) position embeddings ***********

# read current size of position embedding
current_max_input_len, embed_size = model.model.encoder.embed_positions.weight.shape

# allocate a larger position embedding matrix for the encoder
new_encoder_pos_embed = model.model.encoder.embed_positions.weight.new_empty(max_input_len, embed_size)

# copy position embeddings over and over to initialize the new position embeddings
k = 2
step = current_max_input_len - 2
while k < max_input_len - 1:
    new_encoder_pos_embed[k:(k + step)] = model.model.encoder.embed_positions.weight[2:]
    k += step
model.model.encoder.embed_positions.weight.data = new_encoder_pos_embed

# inspecting position embedding size
print(model.model.encoder.embed_positions.weight.shape)
# > torch.Size([16386, 768])
```

```python
# *********** 2) self-attention ***********

# Prepare the config
model.config.attention_window = [attention_window] * model.config.num_hidden_layers
model.config.attention_probs_dropout_prob = 0

for i, layer in enumerate(model.model.encoder.layers):

    # initialize LEDEncoderAttention
    longformer_self_attn_for_bart = LEDEncoderAttention(model.config, layer_id=i)

    # copy pretrained weights
    longformer_self_attn_for_bart.longformer_self_attn.query = layer.self_attn.q_proj
    longformer_self_attn_for_bart.longformer_self_attn.key = layer.self_attn.k_proj
    longformer_self_attn_for_bart.longformer_self_attn.value = layer.self_attn.v_proj

    # initialize the global kqv. Make sure to use `copy.deepcopy`
    longformer_self_attn_for_bart.longformer_self_attn.query_global = copy.deepcopy(layer.self_attn.q_proj)
    longformer_self_attn_for_bart.longformer_self_attn.key_global = copy.deepcopy(layer.self_attn.k_proj)
    longformer_self_attn_for_bart.longformer_self_attn.value_global = copy.deepcopy(layer.self_attn.v_proj)

    # copy the output projection
    longformer_self_attn_for_bart.output = layer.self_attn.out_proj

    # replace the `modeling_bart.BartAttention` object with `LEDEncoderAttention`
    layer.self_attn = longformer_self_attn_for_bart

# inspecting one layer
print(model.model.encoder.layers[0])
# > BartEncoderLayer(
# >   (self_attn): LEDEncoderAttention(
# >     (longformer_self_attn): LEDEncoderSelfAttention(
# >       (query): Linear(in_features=768, out_features=768, bias=True)
# >       (key): Linear(in_features=768, out_features=768, bias=True)
# >       (value): Linear(in_features=768, out_features=768, bias=True)
# >       (query_global): Linear(in_features=768, out_features=768, bias=True)
# >       (key_global): Linear(in_features=768, out_features=768, bias=True)
# >       (value_global): Linear(in_features=768, out_features=768, bias=True)
# >     )
# >     (output): Linear(in_features=768, out_features=768, bias=True)
# >   )
# >   (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
# >   (fc1): Linear(in_features=768, out_features=3072, bias=True)
# >   (fc2): Linear(in_features=3072, out_features=768, bias=True)
# >   (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
# > )
```