# *Chapter 8*: Working with Efficient Transformers

So far, you have learned how to design a **Natural Language Processing** (**NLP**) architecture to achieve successful task performance with transformers. In this chapter, you will learn how to make efficient models out of trained models using distillation, pruning, and quantization. Second, you will also gain knowledge about efficient sparse transformers such as Linformer, BigBird, Performer, and so on. You will see how they perform on various benchmarks, such as memory versus sequence length and speed versus sequence length. You will also see the practical use of model size reduction.

The importance of this chapter came to light as it is getting difficult to run large neural models under limited computational capacity. It is important to have a lighter general-purpose language model such as DistilBERT. This model can then be fine-tuned with good performance, like its non-distilled counterparts. Transformers-based architectures face complexity bottlenecks due to the quadratic complexity of the attention dot product in the transformers, especially for long-context NLP tasks. Character-based language models, speech processing, and long documents are among the long-context problems. In recent years, we have seen much progress in making self-attention more efficient, such as Reformer, Performer, and BigBird, as a solution to complexity.

In short, in this chapter, you will learn about the following topics:

Introduction to efficient, light, and fast transformers

Implementation for model size reduction

Working with efficient self-attention

## Technical requirements

We will be using the Jupyter Notebook to run our coding exercises, which require Python 3.6+, and the following packages need to be installed:

- TensorFlow
- PyTorch
- Transformers >=4.00
- Datasets
- sentence-transformers
- py3nvml

All notebooks with coding exercises are available at the following GitHub link:

https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH08

Check out the following link to see Code in Action Video:

https://bit.ly/3y5j9oZ

---

# Introduction to efficient, light, and fast transformers

Transformer-based models have distinctly achieved state-of-the-art results in many NLP problems at the cost of quadratic memory and computational complexity. We can highlight the issues regarding complexity as follows:

- The models are not able to efficiently process long sequences due to their self-attention mechanism, which scales quadratically with the sequence length.
- An experimental setup using a typical GPU with 16 GB can handle the sentences of 512 tokens for training and inference. However, longer entries can cause problems.
- The NLP models keep growing from the 110 million parameters of BERT-base to the 17 billion parameters of Turing-NLG and to the 175 billion parameters of

GPT-3. This notion raises concerns about computational and memory complexity.

We also need to care about costs, production, reproducibility, and sustainability. Hence, we need faster and lighter transformers, especially on edge devices.

Several approaches have been proposed to reduce computational complexity and memory footprint. Some of these approaches focus on changing the architecture and some do not alter the original architecture but instead make improvements to the trained model or to the training phase. We will divide them into two groups, model size reduction and efficient self-attention.

Model size reduction can be accomplished using three different approaches:

Knowledge distillation

Pruning

Quantization

Each of these three has its own way of reducing the size model, which we will describe in short in the *Implementation for model size reduction* section.

In knowledge distillation, a smaller transformer (student) can transfer the knowledge of a big model (teacher). We train the student model so that it can mimic the teacher's behavior or produce the same output for the same input. The distilled model may underperform the teacher. There is a trade-off between compression, speed, and performance.

Pruning is a model compression technique in machine learning that is used to reduce the size of the model by removing a section of the model that contributes little to producing results. The most typical example is decision tree pruning, which helps to reduce the model complexity and increase the generalization capacity of the model. Quantization changes model weight types from higher resolutions to lower resolutions. For example, we use a typical floating-point number (`float64`) consuming 64 bits of memory for each weight. Instead, we can use `int8` in quantization, which consumes 8 bits for each weight, and naturally has less accuracy in presenting numbers.

Self-attention heads are not optimized for long sequences. In order to solve this issue, many different approaches have been proposed. The most efficient approach is **Self-Attention Sparsification**, which we will discuss soon. The other most widely used approach is **Memory Efficient Backpropagation**. This approach balances a trade-off between the caching of intermediate results and re-computing. Intermediate activations computed during forward propagation are needed to compute gradients during backward propagation. Gradient checkpoints can reduce a substantial amount of memory footprint and computation. Another approach is **Pipeline Parallelism Algorithms**. Mini-batches are split into micro-batches and the parallelism pipeline takes advantage of using the waiting time during the forward and backward operations while transferring the batches to deep learning accelerators such as **Graphics Processing Unit (GPU)** or **Tensor Processing Unit (TPU)**.

**Parameter Sharing** can be counted as one of the first approaches towards efficient deep learning. The most typical example is RNN, as depicted in *Chapter 1*, *From Bag-of-Words to the Transformers*, where the units of unfolded representation use the shared parameters. Hence, the number of trainable parameters is not affected by the input size. Some shared parameters which are also called weight tying or weight replication, spread the network so that the number of trainable parameters is reduced. For instance, Linformer shares projection matrices across heads and layers. Reformer shares the query and key at the cost of performance loss.

Now let's try to understand these issues with corresponding practical examples.

---

# Implementation for model size reduction

Even though the transformer-based models achieve state-of-the-art results in many aspects of NLP, they usually share the very same problem: they are big models and are not fast enough to be used. In business cases where it is necessary to embed them inside a mobile application or in a web interface, it seems to be impossible if you try to use the original models.

In order to improve the speed and size of these models, some techniques are proposed, which are listed here:

- Distillation (also known as knowledge distillation)
- Pruning

Quantization

For each of these techniques, we provide a separate subsection to address the technical and theoretical insights.

## Working with DistilBERT for knowledge distillation

The process of transferring knowledge from a bigger model to a smaller one is called **knowledge distillation**. In other words, there is a teacher model and a student model; the teacher is typically a bigger and stronger model while the student is smaller and weaker.

This technique is used in various problems, from vision to acoustic models and NLP. A typical implementation of this technique is shown in *Figure 8.1*:

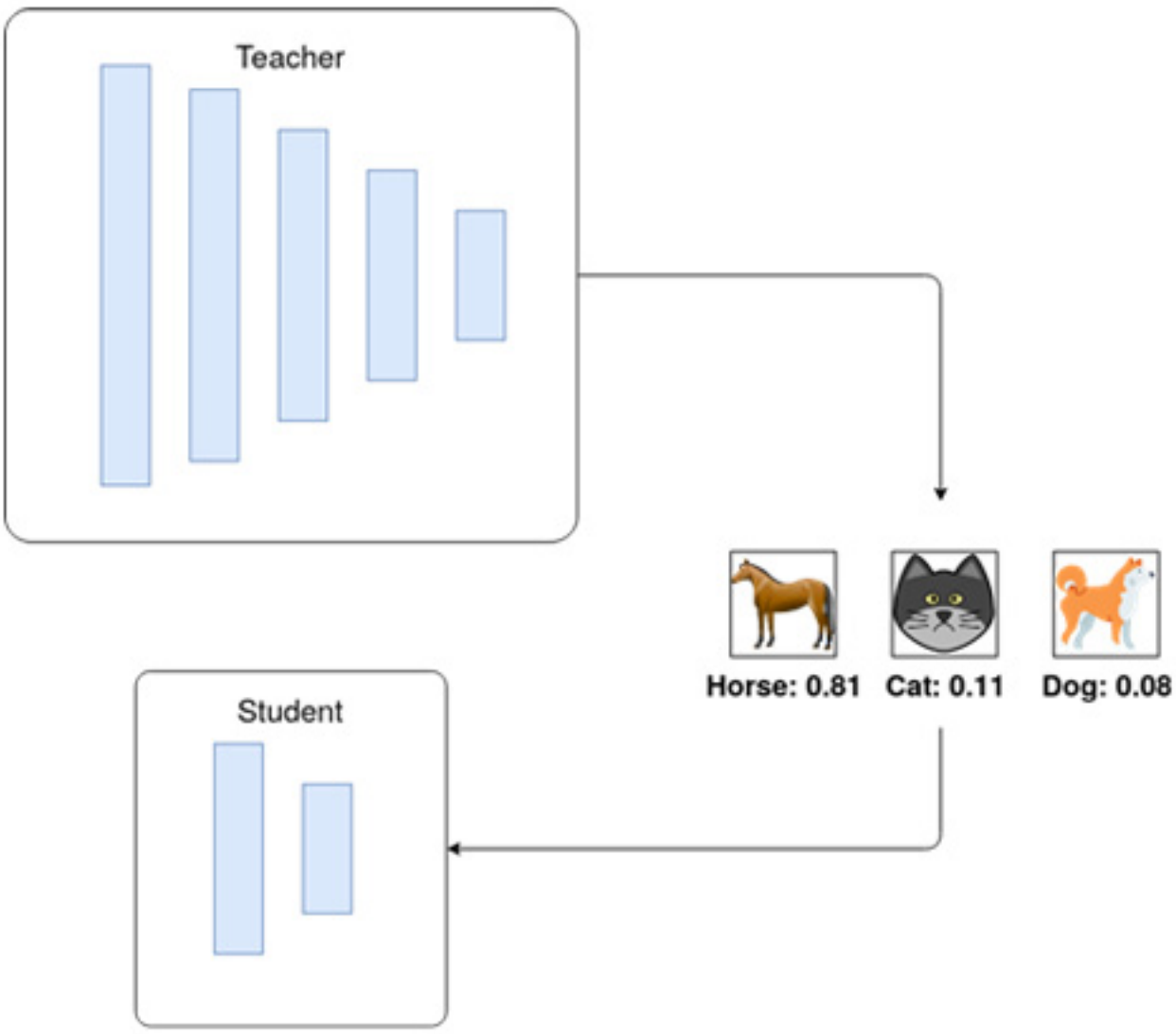


Figure 8.1 – Knowledge distillation for image classification

DistilBERT is one of the most important models in this field and has got the attention of researchers and even the businesses. This model, which tries to mimic the behavior of BERT-Base, has 50% fewer parameters and achieves 95% of the teacher model's performance.

Some details are given as follows:

> DistilBert is 1.7x compressed and 1.6x faster with 97% relative performance (compared to original BERT).

Mini-BERT is 6x compressed, 3x faster, and has 98% relative performance.

TinyBERT is 7.5x compressed, has 9.4x speed, and 97% relative performance.

The distillation training step that is used to train the model is very simple using PyTorch (original description and code available at https://medium.com/huggingface/distilbert-8cf3380435b5):

Copy Explain

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Optimizer
KD_loss = nn.KLDivLoss(reduction='batchmean')
def kd_step(teacher: nn.Module,
            student: nn.Module,
            temperature: float,
            inputs: torch.tensor,
            optimizer: Optimizer):
    teacher.eval()
    student.train()
    with torch.no_grad():
        logits_t = teacher(inputs=inputs)
    logits_s = student(inputs=inputs)
    loss = KD_loss(input=F.log_softmax(
                           logits_s/temperature,
                           dim=-1),
                   target=F.softmax(
                           logits_t/temperature,
                           dim=-1))
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

This model-supervised training provides us with a smaller model that is very similar to the base model in behavior. However, the loss function used here is **Kullback-Leibler** loss to ensure that the student model mimics the good and bad aspects of the teacher model with no modification of the decision on the last softmax logits. This loss function shows how different two distributions are from each other; a greater difference means a higher loss value. The reason for using this loss function is to make the student model try to completely mimic the behavior of the teacher. The GLUE macro scores for BERT and DistilBERT are just 2.8% different.

# Pruning transformers

Pruning includes the process of setting weights at each layer to zero based on a pre-specified criterion. For example, a simple pruning algorithm could take the weights of each layer and set those that are below a threshold. This method eliminates weights that are very low in value and do not affect the results too much.

Likewise, we prune some redundant parts of the transformer network. The pruned networks are more likely to generalize better than the original one. We have seen a successful pruning operation because the pruning process probably keeps the true underlying explanatory factors and discards the redundant subnetwork. But we need to still train a large network. The reasonable strategy is that we train a neural network as large as possible. Then, the less salient weights or units whose removals have a small effect on the model performance are discarded.

There are two approaches:

**Unstructured pruning:** where individual weights with a small saliency (or the least weight magnitude) are removed no matter which part of the neural network they are located in.

**Structured pruning:** this approach prunes heads or layers.

However, the pruning process has to be compatible with modern GPUs.

Most libraries such as Torch or TensorFlow come with this capability. We will describe how it is possible to prune a model using Torch. There are many different methods to use in pruning (magnitude-based or mutual information-based). One of the simplest ones to understand and implement is the L1 pruning method. This method takes the weights of each layer and zeros out the ones with the lowest L1-norm. You can also specify what percentage of your weights must be converted to zero after pruning. In order to make this example more understandable and show its impact on the model, we'll use the text representation example from *Chapter 7*, *Text Representation*. We will prune the model and see how it performs after pruning:

1. We will use the Roberta model. You can load the model using the following code:

```
from sentence_transformers import SentenceTransformer
distilroberta = SentenceTransformer('stsb-distilroberta-base-v2')
```

2. You will also need to load metrics and datasets for evaluation:

```
from datasets import load_metric, load_dataset
stsb_metric = load_metric('glue', 'stsb')
stsb = load_dataset('glue', 'stsb')
mrpc_metric = load_metric('glue', 'mrpc')
mrpc = load_dataset('glue','mrpc')
```

3. In order to evaluate the model, just like in *Chapter 7*, *Text Representation*, you can use the following function:

```
import math
import tensorflow as tf
def roberta_sts_benchmark(batch):
    sts_encode1 = tf.nn.l2_normalize(
                distilroberta.encode(batch['sentence1']),
                axis=1)
    sts_encode2 = tf.nn.l2_normalize(
        distilroberta.encode(batch['sentence2']), axis=1)
    cosine_similarities = tf.reduce_sum(
        tf.multiply(sts_encode1, sts_encode2), axis=1)
    clip_cosine_similarities = tf.clip_by_value(cosine_similarities,-1.0,1.0)
    scores = 1.0 -\
              tf.acos(clip_cosine_similarities) / math.pi
return scores
```

4. And of course, it is required to set labels:

```
references = stsb['validation'][:]['label']
```

5. And to run the base model with no changes in it:

```
distilroberta_results = roberta_sts_benchmark(stsb['validation'])
```

6. After all these things are done, this is the step where we actually start to prune our model:

```
from torch.nn.utils import prune
pruner = prune.L1Unstructured(amount=0.2)
```

7. The previous code makes a pruning object using L1-norm pruning with 20% of the weights in each layer. To apply it to the model, you can use the following code:

```
state_dict = distilroberta.state_dict()
for key in state_dict.keys():
    if "weight" in key:
        state_dict[key] = pruner.prune(state_dict[key])
```

It will iteratively prune all layers that have weight in their name; in other words, we will prune all weight layers and not touch the layers that are biased. Of course, you can try that too for experimentation purposes.

8. And again, it is good to reload the state dictionary to the model:

```
distilroberta.load_state_dict(state_dict)
```

9. Now that we have done everything, we can test the new model:

```
distilroberta_results_p = roberta_sts_benchmark(stsb['validation'])
```

10. In order to have a good visual representation of the results, you can use the following code:

```
import pandas as pd
pd.DataFrame({
"DistillRoberta":stsb_metric.compute(predictions=distilroberta_results,
references=references),
"DistillRobertaPruned":stsb_metric.compute(predictions=distilroberta_results_p,
references=references)
})
```

The following screenshot shows the results:

| | DistillRoberta | DistillRobertaPruned |
|---|---|---|
| pearson | 0.888461 | 0.849915 |
| spearmanr | 0.889246 | 0.849125 |

Figure 8.2 – Comparison between original and pruned models

But what you did is you eliminated 20% of all weights of the model, reduced its size and computation cost, and lost 4% in performance. However, this step can be combined with other techniques such as quantization, which is explored in the next subsection.

This type of pruning is applied to some of the weights in a layer; however, it is also possible to completely drop some parts or layers of transformer architectures, for example, it is possible to drop some of the attention heads and track the changes too.

There are also other types of pruning algorithms available in PyTorch, such as iterative and global pruning, which are worth trying.

## Quantization

Quantization is a signal processing and communication term that is generally used to emphasize how much accuracy is presented by the data provided. More bits mean more accuracy and precision in terms of data resolution. For example, if you have a variable that is presented by 4 bits and you want to quantize it to 2 bits, it means you have to drop the accuracy of your resolution. With 4 bits, you can specify 16 different states, while with 2 bits you can distinguish 4 states. In other words, by reducing the resolution of your data from 4 to 2 bits, you are saving 50% more space and complexity.

Many popular libraries, such as TensorFlow, PyTorch, and MXNET, support mixed-precision operation. Recall the `fp16` parameter used in the `TrainingArguments` class in `chapter 05`. `fP16` increases computational efficiency since modern GPUs offer higher efficiency for reduced precision math, but the results are accumulated in `fP32`. Mixed-precision can reduce the memory usage required for training, which allows us to increase the batch size or model size.

Quantization can be applied to model weights to reduce their resolution and save computation time, memory, and storage. In this subsection, we will try to quantize the model that we pruned in the previous section:

1. In order to do so, you can use the following code to quantize your model in 8-bit integer representation instead of float:

```
import torch
distilroberta = torch.quantization.quantize_dynamic(
            model=distilroberta,
            qconfig_spec = {
            torch.nn.Linear :
            torch.quantization.default_dynamic_qconfig,
                        },
            dtype=torch.qint8)
```

2. Afterwards, you can get the evaluation results by using the following code:

```
distilroberta_results_pq = roberta_sts_benchmark(stsb['validation'])
```

3. And as before, you can view the results:

```
pd.DataFrame({
"DistillRoberta":stsb_metric.compute(predictions=distilroberta_results,
references=references),
"DistillRobertaPruned":stsb_metric.compute(predictions=distilroberta_results_p,
references=references),
"DistillRobertaPrunedQINT8":stsb_metric.compute(predictions=distilroberta_results_pq
references=references)
})
```

The results can be seen as follows:

| | DistillRoberta | DistillRobertaPruned | DistillRobertaPrunedQINT8 |
|---|---|---|---|
| pearson | 0.888461 | 0.849915 | 0.826784 |
| spearmanr | 0.889246 | 0.849125 | 0.824857 |

Figure 8.3 – Comparison between original, pruned, and quantized models

4. Until now, you just used a distilled model, pruned it, and then you quantized it to reduce its size and complexity. Let's see how much space you have saved by saving the model:

```
distilroberta.save("model_pq")
```

Use the following code in order to see the model size:

```
ls model_pq/0_Transformer/ -l --block-size=M | grep pytorch_model.bin
-rw-r--r-- 1 root 191M May 23 14:53 pytorch_model.bin
```

Copy Explain

As you can see, it is 191 MB. The initial size of the model was 313 MB, which means we managed to decrease the size of the model to 61% of its original size and just lost 6%-6.5% in terms of performance. Please note that the `block-size` parameter may fail on a Mac, and it is required to use `-lh` instead.

Up to this point, you have learned about pruning and quantization in terms of practical model preparation for industrial usage. However, you also gained information about the distillation process and how it can be useful. There are many other ways to perform pruning and quantization, which can be a good step to go in after reading this section. For more information and guides, you can take a look at **movement pruning** at https://github.com/huggingface/block_movement_pruning. This kind of pruning is a simple and deterministic first-order weight pruning approach. It uses the weight changes in training to find out which weights are more likely to be unused to have less effect on the result.

---

# Working with efficient self-attention

Efficient approaches restrict the attention mechanism to get an effective transformer model because the computational and memory complexity of a transformer is mostly due to the self-attention mechanism. The attention mechanism scales quadratically with respect to the input sequence length. For short input, quadratic complexity may not be an issue. However, to process longer documents, we need to improve the attention mechanism that scales linearly with sequence length.

We can roughly group the efficient attention solutions into three types:

- Sparse attention with fixed patterns
- Learnable sparse patterns
- Low-rank factorization/kernel function

Let's begin with sparse attention based on a fixed pattern next.

## Sparse attention with fixed patterns

Recall that the attention mechanism is made up of a query, key, and values as roughly formulated here:

$$Attention\,(Q,K,V) = Score\,(Q,K)\,.V$$

Here, the `Score` function, which is mostly softmax, performs QKT multiplication that requires O(n2) memory and computational complexity since a token position attends to all other token positions in full self-attention mode to build its position embeddings. We repeat the same process for all token positions to get their embeddings, leading to a quadratic complexity problem. It is a very expensive way of learning, especially for long-context NLP problems. It is natural to ask the question do we need such a dense interaction or is there a cheaper way to do the calculations? Many researchers have addressed this problem and employed a variety of techniques to mitigate the complexity burden and to reduce the quadratic complexity of the self-attention mechanism. They have mostly made a trade-off between performance, computation, and memory, especially for long documents.

The simplest way of reducing complexity is to sparsify the full self-attention matrix or find another cheaper way to approximate full attention. Sparse attention patterns formulate how to connect/disconnect certain positions without disturbing the flow of information through layers, which helps the model to track long-term dependency and to build sentence-level encoding.

Full self-attention and sparse attention are depicted in *Figure 8.4* in that order, where the rows correspond to output positions and the columns are for the inputs. A full self-attention model would directly transfer the information between any two positions. On the other hand, in localized sliding window attention, which is sparse attention, as shown on the right of the figure, the empty cells mean that there is no interaction between the corresponding input-output position. The sparse model in the figure is based on fixed patterns that are certain manually designed rules. More specifically, it is localized sliding window attention that was one of the first proposed methods, also known as the local-based fixed pattern approach. The assumption behind it is that useful information is located in each position neighbor. Each query token attends to window/2 key tokens to the left and window/2 key tokens to the right of that position. In the following example, the window size is selected as 4. This rule applies to every layer in the transformer in the same way. In some studies, the window size is increased as it moves towards the layers further.

The following figure simply depicts the difference between full and sparse attention:

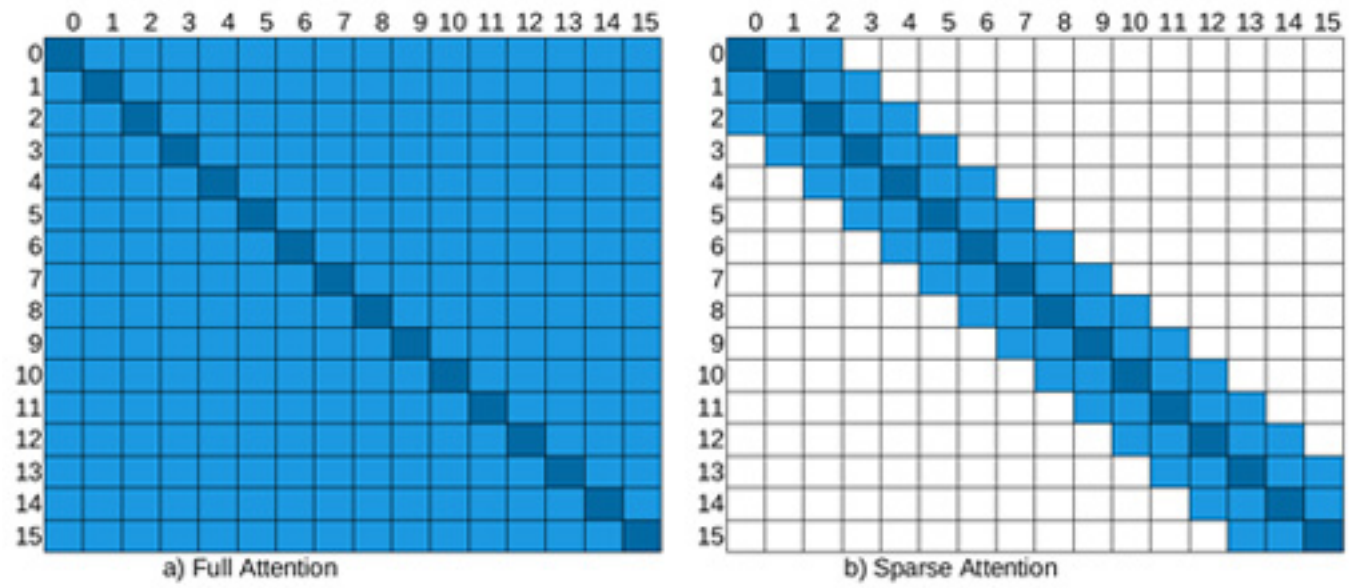


Figure 8.4 – Full attention versus sparse attention

In sparse mode, the information is transmitted through connected nodes (non-empty cells) in the model. For example, the output position 7 of the sparse attention matrix cannot directly attend to the input position 3 (please see the sparse matrix at the right of *Figure 8.4*) since the cell (7,3) is seen as empty. However, position 7 indirectly attends to position 3 via the token position 5, that is (7->5, 5->3 => 7->3). The figure also illustrates that while the full self-attention incurs n2 number of active cells (vertex), the sparse model does roughly *5×n*.

Another important type is global attention. A few selected tokens or a few injected tokens are used as global attention that can attend to all other positions and be attended by them. Hence, the maximum path distance between any two token positions is equal to 2. Suppose we have a sentence *[GLB, the, cat, is, very, sad]* where **Global (GLB)** is an injected global token and the window size is 2, which means a token can attend to only its immediate left-right tokens and to GLB as well. There is no direct interaction from *cat* to *sad*. But we can follow *cat-> GLB, GLB-> sad* interactions, which creates a hyperlink through the GLB token. The global tokens can be selected from existing tokens or added like *(CLS)*. As shown in the following screenshot, the first two token positions are selected as global tokens:

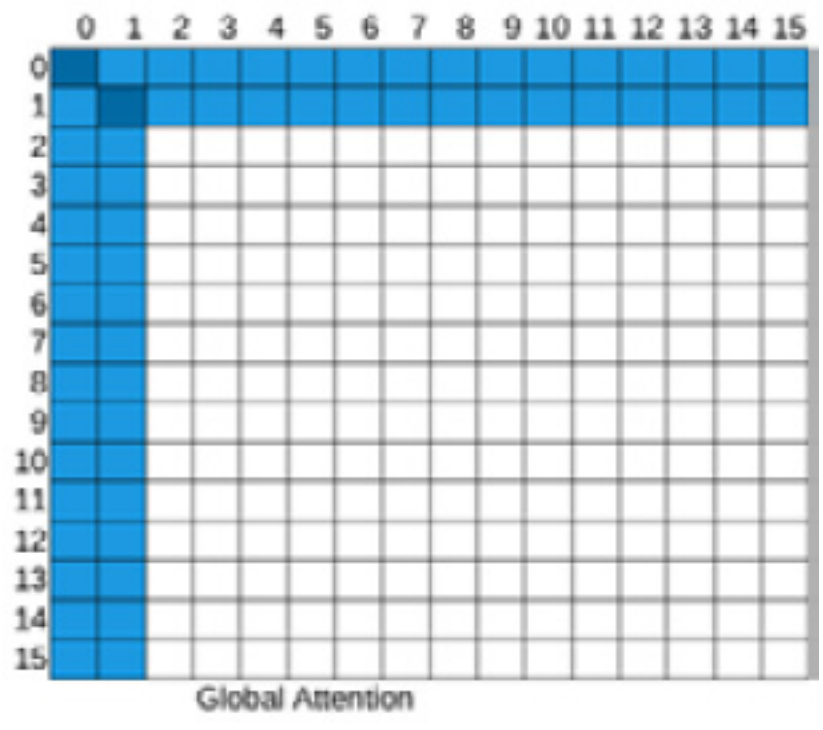

Figure 8.5 – Global attention

By the way, these global tokens don't have to be at the beginning of the sentence either. For example, the longformer model randomly selects global tokens in addition to the first two tokens..

There are four more widely seen patterns. **Random attention** (the first matrix in *Figure 8.6*) is used to ease the flow of information by randomly selecting from existing tokens. But most of the time, we employ random attention as part of a **combined pattern** (the bottom-left matrix) that consists of a combination of other models. **Dilated attention** is similar to the sliding window, but some gaps are put in the window as shown at the top right of *Figure 8.6*:

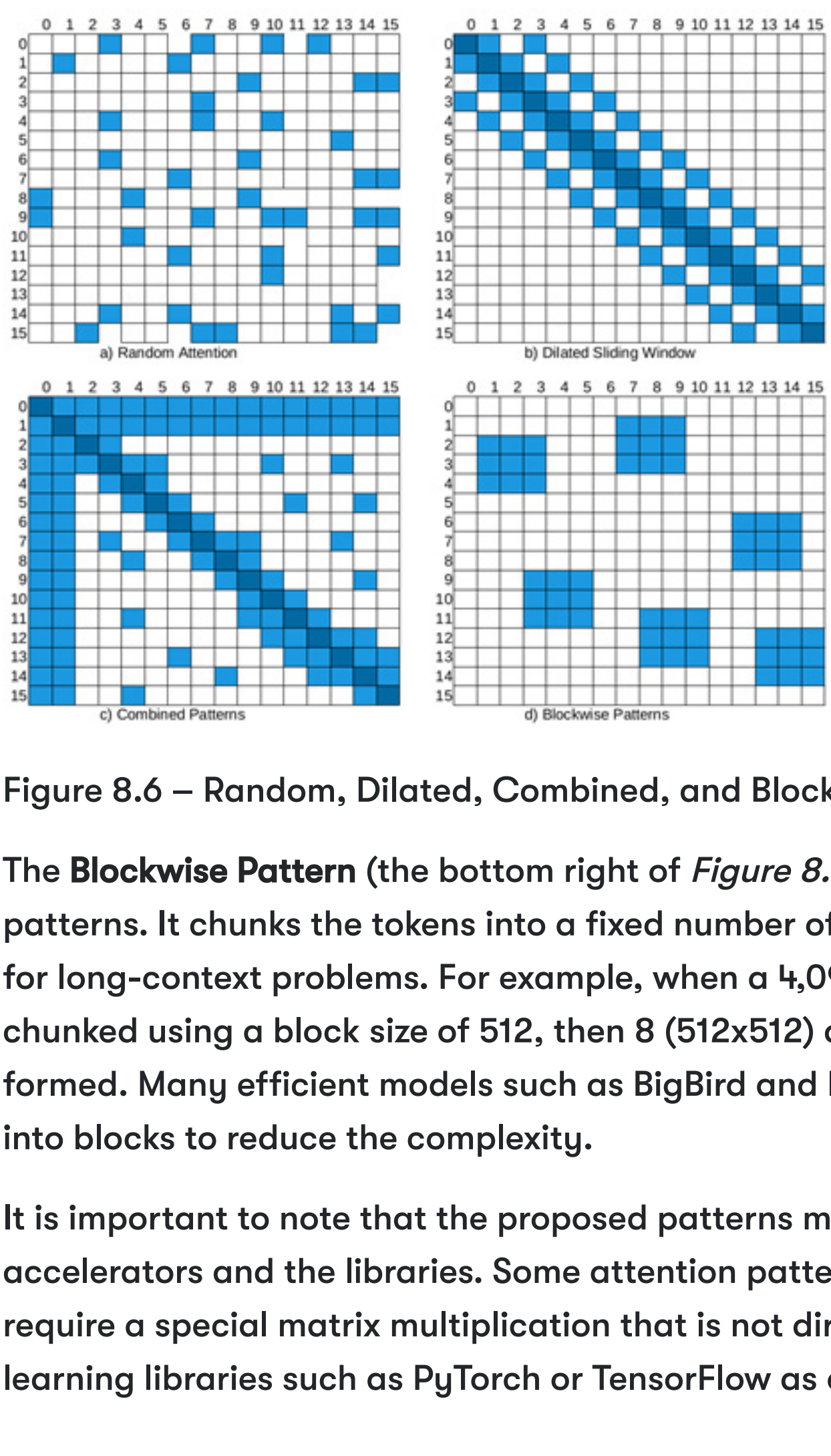


Figure 8.6 – Random, Dilated, Combined, and Blockwise

The **Blockwise Pattern** (the bottom right of *Figure 8.6*) provides a basis for other patterns. It chunks the tokens into a fixed number of blocks, which is especially useful for long-context problems. For example, when a 4,096x4,096 attention matrix is chunked using a block size of 512, then 8 (512x512) query blocks and key blocks are formed. Many efficient models such as BigBird and Reformer mostly chunk tokens into blocks to reduce the complexity.

It is important to note that the proposed patterns must be supported by the accelerators and the libraries. Some attention patterns such as dilated patterns require a special matrix multiplication that is not directly supported in current deep learning libraries such as PyTorch or TensorFlow as of writing this chapter.

We are ready to run some experiments for efficient transformers. We will proceed with models that are supported by the Transformers library and that have checkpoints on the HuggingFace platform. **Longformer** is one of the models that use sparse attention. It uses a combination of a sliding window and global attention. It supports dilated sliding window attention as well:

1. Before we start, we need to install the `py3nvml` package for benchmarking. Please recall that we already discussed how to apply benchmarking in *Chapter 2, A Hands-On Introduction to the Subject*:

```
!pip install py3nvml
```

2. We also need to check our devices to ensure that there is no running process:

```
!nvidia-smi
```

The output is as follows:

```
Sat May 22 13:43:18 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.19.01    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   36C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 8.7 – GPU usage

3. Currently, the Longformer author has shared a couple of checkpoints. The following code snippet loads the Longformer checkpoint `allenai/longformer-base-4096` and processes a long text:

```
from transformers import LongformerTokenizer, LongformerForSequenceClassification
import torch
tokenizer = LongformerTokenizer.from_pretrained(
    'allenai/longformer-base-4096')
model=LongformerForSequenceClassification.from_pretrained(
    'allenai/longformer-base-4096')
sequence= "hello "*4093
inputs = tokenizer(sequence, return_tensors="pt")
print("input shape: ",inputs.input_ids.shape)
outputs = model(**inputs)
```

4. The output is as follows:

```
input shape:  torch.Size([1, 4096])
```

> As seen, Longformer can process a sequence up to the length of `4096`. When we pass a sequence whose length is more than `4096`, which is the limit, you will get the error `IndexError: index out of range in self`.

Longformer's default `attention_window` is `512`, which is the size of the attention window around each token. With the following code, we instantiate two Longformer configuration objects, where the first one is the default Longformer, and the second is a lighter one where we set the window size to a smaller value such as `4` so that the model becomes lighter:

1. Please pay attention to the following examples. We will always call `XformerConfig.from_pretrained()`. This call does not download the actual weights of the model checkpoint, instead only downloading the configuration from the HuggingFace Hub. Throughout this section, since we will not fine-tune, we only need the configuration:

```
from transformers import LongformerConfig, \
PyTorchBenchmark, PyTorchBenchmarkArguments
config_longformer=LongformerConfig.from_pretrained(
    "allenai/longformer-base-4096")
config_longformer_window4=LongformerConfig.from_pretrained(
    "allenai/longformer-base-4096",
     attention_window=4)
```

2. With these configuration instances, you can train your Longformer language model with your own datasets passing the configuration object to the Longformer model as follows:

```
from transformers import LongformerModel
model = LongformerModel(config_longformer)
```

Other than training a Longformer model, you can also fine-tune a trained checkpoint to a downstream task. To do so, you can continue by applying the code as shown in `Chapter 03` for language model training and `Chapter 05–06` for fine-tuning.

3. We will now compare the time and memory performance of these two configurations with various lengths of input [`128, 256, 512, 1024, 2048, 4096`] by utilizing `PyTorchBenchmark` as follows:

```
sequence_lengths=[128,256,512,1024,2048,4096]
models=["config_longformer","config_longformer_window4"]
configs=[eval(m) for m in models]
benchmark_args = PyTorchBenchmarkArguments(
    sequence_lengths= sequence_lengths,
    batch_sizes=[1],
    models= models)
benchmark = PyTorchBenchmark(
    configs=configs,
    args=benchmark_args)
results = benchmark.run()
```

4. The output is the following:

```
1 / 2
2 / 2
====================       INFERENCE - SPEED - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length     Time in s
--------------------------------------------------------------------------------
      config_longformer              1              128            0.036
      config_longformer              1              256            0.036
      config_longformer              1              512            0.036
      config_longformer              1              1024           0.064
      config_longformer              1              2048           0.117
      config_longformer              1              4096           0.226
  config_longformer_window4          1              128            0.019
  config_longformer_window4          1              256            0.022
  config_longformer_window4          1              512            0.028
  config_longformer_window4          1              1024           0.044
  config_longformer_window4          1              2048           0.074
  config_longformer_window4          1              4096           0.136
--------------------------------------------------------------------------------

====================      INFERENCE - MEMORY - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length    Memory in MB
--------------------------------------------------------------------------------
      config_longformer              1              128             1595
      config_longformer              1              256             1595
      config_longformer              1              512             1595
      config_longformer              1              1024            1679
      config_longformer              1              2048            1793
      config_longformer              1              4096            2089
  config_longformer_window4          1              128             1525
  config_longformer_window4          1              256             1527
  config_longformer_window4          1              512             1541
  config_longformer_window4          1              1024            1561
  config_longformer_window4          1              2048            1643
  config_longformer_window4          1              4096            1763
--------------------------------------------------------------------------------
```

Figure 8.8 – Benchmark results

Some hints for `PyTorchBenchmarkArguments`: if you like to see the performance for training as well as inference, you should set the argument `training` to `True` (the default is `False`). You also may want to see your current environment information. You can do so by setting `no_env_print` to `False`; the default is `True`.

Let's visualize the performance to be more interpretable. To do so, we define a `plotMe()` function since we will need that function for further experiments as well. The function plots the inference performance in terms of both running time complexity by default or memory footprint properl:.

1. Here is the function definition:

```
import matplotlib.pyplot as plt
def plotMe(results,title="Time"):
    plt.figure(figsize=(8,8))
    fmts= ["rs--","go--","b+-","c-o"]
    q=results.memory_inference_result
    if title=="Time":
        q=results.time_inference_result
    models=list(q.keys())
    seq=list(q[models[0]]['result'][1].keys())
    models_perf=[list(q[m]['result'][1].values()) \
        for m in models]
    plt.xlabel('Sequence Length')
    plt.ylabel(title)
    plt.title('Inference Result')
    for perf,fmt in zip(models_perf,fmts):
        plt.plot(seq, perf,fmt)
    plt.legend(models)
    plt.show()
```

2. Let's see the computational performance of two Longformer configurations, as follows:

```
plotMe(results)
```

This plots the following chart:

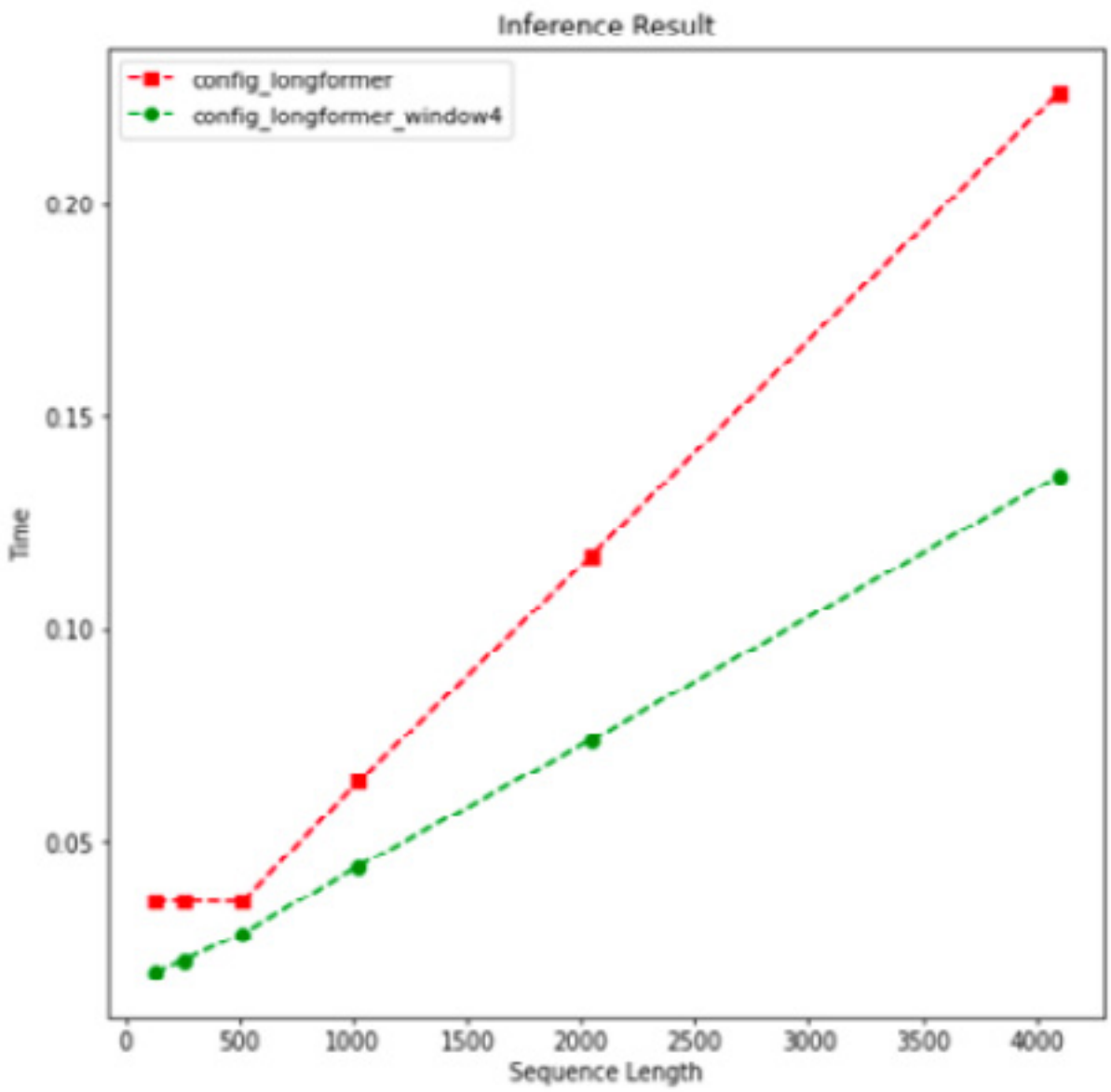


Figure 8.9 – Speed performance over sequence length (Longformer)

In this and the next examples, we see the main differentiation between a heavy model and a light model starting from length 512. The preceding figure shows the lighter Longformer model in green (the one with a window length of 4) performs better in terms of time complexity as expected. We also see that two Longformer models process the input with linear time complexity.

3. Let's evaluate these two models in terms of memory performance:

```
plotMe(results, "Memory")
```

This plots the following:

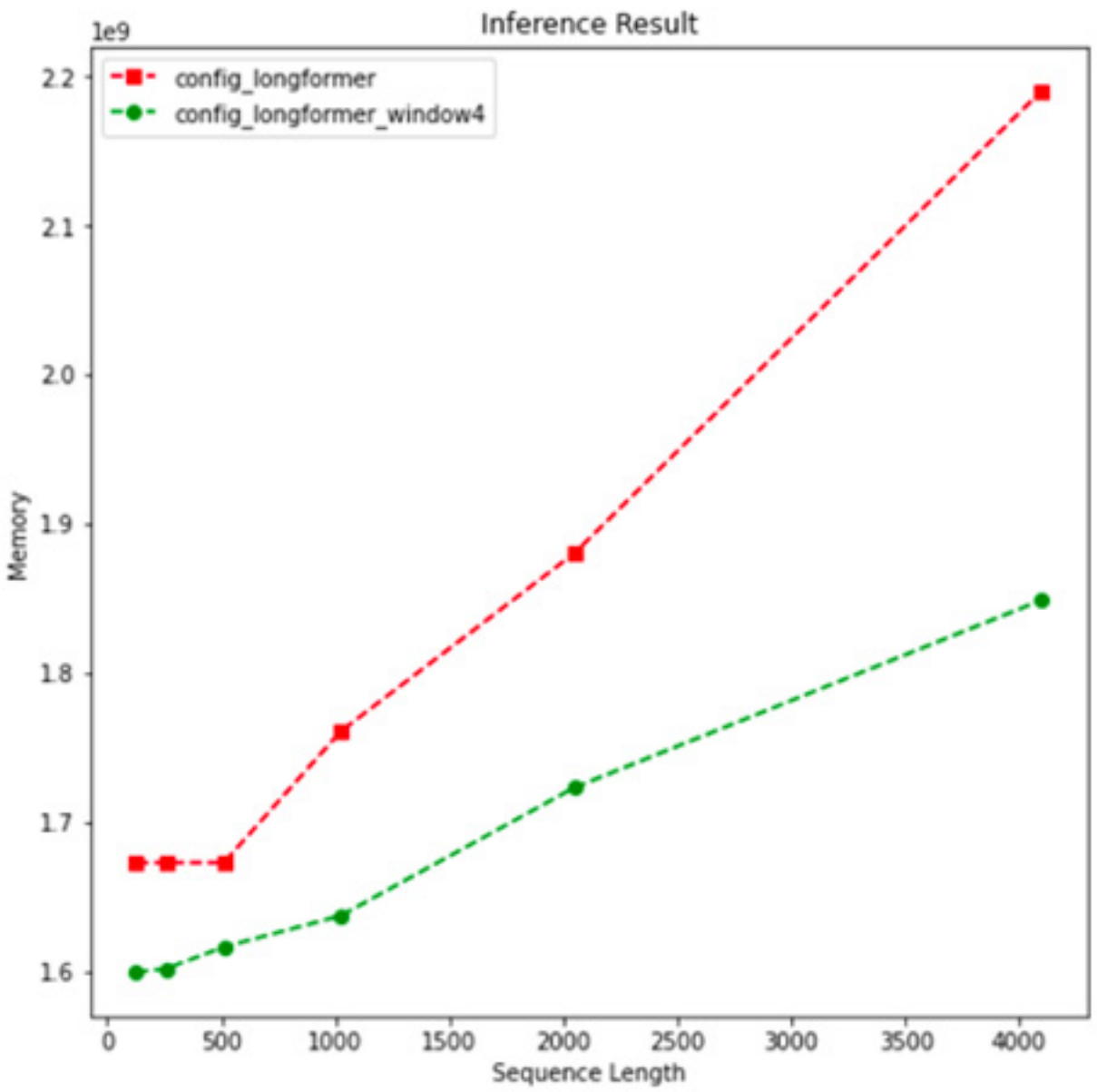


Figure 8.10 – Memory performance over sequence length (Longformer)

Again, up to length 512, there is no substantial differentiation. For the rest, we see a similar memory performance to the time performance. It is clear to say that the memory complexity of the Longformer self-attention is linear. On the other hand, let me bring to your attention that we are not saying anything about model task performance yet.

Thanks to the `PyTorchBenchmark` script, we have cross-checked these models. This script is very useful when we choose which configuration the language model should be trained with. It will be vital before starting the real language model training and fine-tuning.

Another best-performing model exploiting sparse attention is BigBird (Zohen et al. 2020). The authors claimed that their sparse attention mechanism (they called it a generalized attention mechanism) preserves all the functionality of the full self-attention mechanism of vanilla transformers in linear time. The authors treated the attention matrix as a directed graph so that they leveraged graph theory algorithms. They took inspiration from the graph sparsification algorithm, which approximates a given graph `G` by graph `G'` with fewer edges or vertices.

BigBird is a block-wise attention model and can handle sequences up to a length of `4096`. It first blockifies the attention pattern by packing queries and keys together and then defines attention on these blocks. They utilize random, sliding window, and global attention.

4. Let's load and use the BigBird model checkpoint configuration just like the Longformer transformer model. There are a couple of BigBird checkpoints shared by the developers in the HuggingFace Hub. We select the original BigBird model, `google/bigbird-roberta-base`, which is warm started from a RoBERTa checkpoint. Once again, we're not downloading the model checkpoint weights but the configuration instead. The `BigBirdConfig` implementation allows us to compare full self-attention and sparse attention. Thus, we can observe and check whether the sparsification will reduce the full-attention O(n^2) complexity to a lower level. Once again, up to a length of 512, we do not clearly observe quadratic complexity. We can see the complexity from this level on. Setting the attention type to original-full will give us a full self-attention model. For comparison, we created two types of configurations: the first one is BigBird's original sparse approach, the second is a model that uses the full self-attention model.
5. We call them `sparseBird` and `fullBird` in order as follows:

Copy Explain

```
from transformers import BigBirdConfig
# Default Bird with num_random_blocks=3, block_size=64
sparseBird = BigBirdConfig.from_pretrained(
    "google/bigbird-roberta-base")
fullBird = BigBirdConfig.from_pretrained(
    "google/bigbird-roberta-base",
    attention_type="original_full")
```

6. Please notice that for smaller sequence lengths up to 512, the BigBird model works as full self-attention mode due to block-size and sequence-length inconsistency:

```
sequence_lengths=[256,512,1024,2048, 3072, 4096]
models=["sparseBird","fullBird"]
configs=[eval(m) for m in models]
benchmark_args = PyTorchBenchmarkArguments(
    sequence_lengths=sequence_lengths,
    batch_sizes=[1],
    models=models)
benchmark = PyTorchBenchmark(
    configs=configs,
    args=benchmark_args)
results = benchmark.run()
```

The output is as follows:

```
====================       INFERENCE - SPEED - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length     Time in s
--------------------------------------------------------------------------------
          sparseBird                 1              256            0.014
          sparseBird                 1              512            0.029
          sparseBird                 1              1024           0.112
          sparseBird                 1              2048           0.288
          sparseBird                 1              3072           0.217
          sparseBird                 1              4096           0.279
           fullBird                  1              256            0.014
           fullBird                  1              512            0.024
           fullBird                  1              1024           0.049
           fullBird                  1              2048           0.123
           fullBird                  1              3072           0.232
           fullBird                  1              4096           0.348
--------------------------------------------------------------------------------

====================       INFERENCE - MEMORY - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length    Memory in MB
--------------------------------------------------------------------------------
          sparseBird                 1              256             1495
          sparseBird                 1              512             1571
          sparseBird                 1              1024            1777
          sparseBird                 1              2048            2195
          sparseBird                 1              3072            2591
          sparseBird                 1              4096            2969
           fullBird                  1              256             1495
           fullBird                  1              512             1571
           fullBird                  1              1024            1801
           fullBird                  1              2048            2257
           fullBird                  1              3072            2955
           fullBird                  1              4096            3835
--------------------------------------------------------------------------------
```

Figure 8.11 – Benchmark results (BigBird)

7. Again, we plot the time performance as follows:

```
plotMe(results)
```

This plots the following:

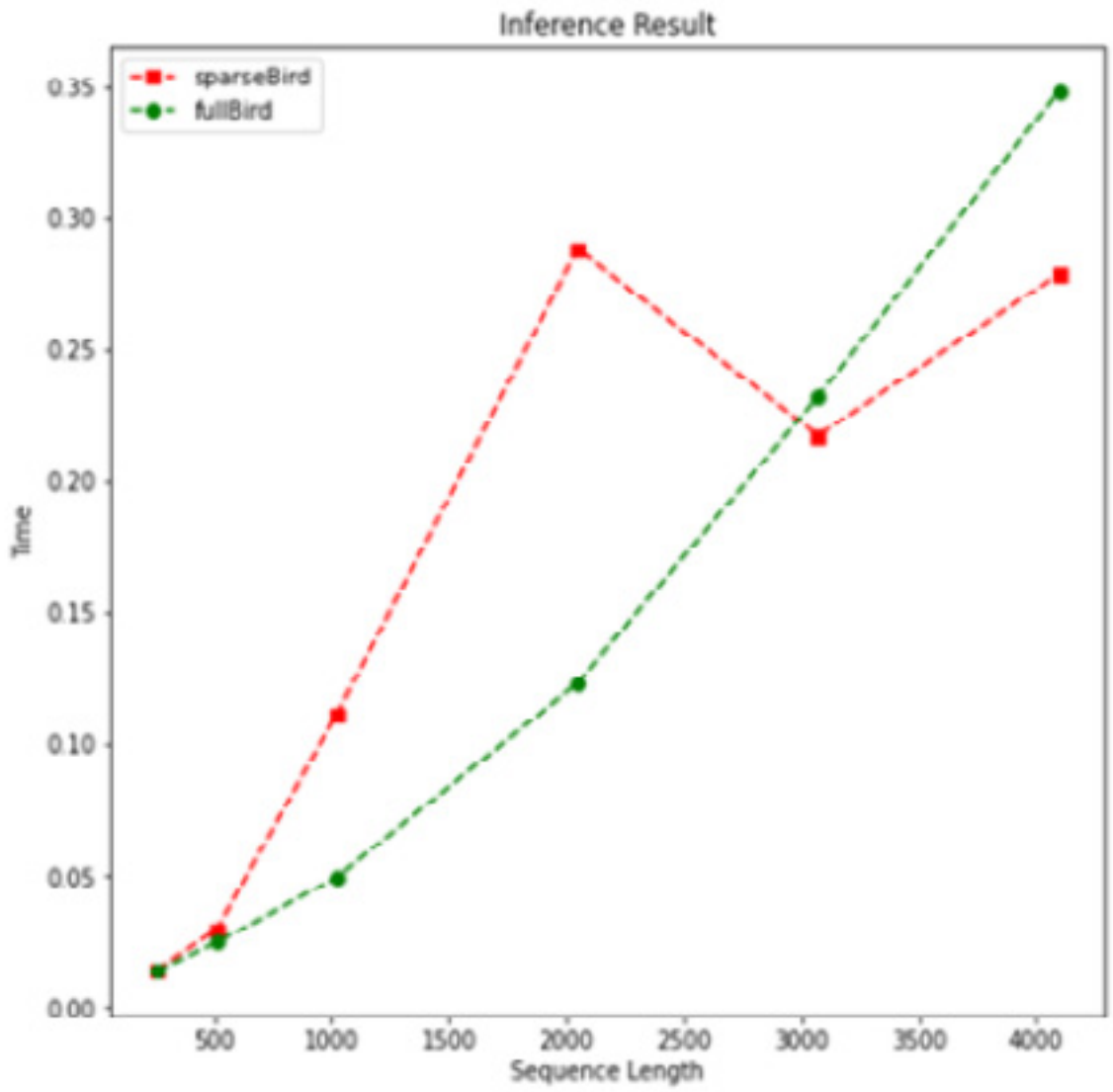


Figure 8.12 – Speed performance (BigBird)

To a certain extent, the full self-attention model performs better than a sparse model. However, we can observe the quadratic time complexity for `fullBird`. Hence, after a certain point, we also see that the sparse attention model abruptly outperforms it, when coming to an end.

8. Let's check the memory complexity as follows:

```
plotMe(results,"Memory")
```

Here is the output:

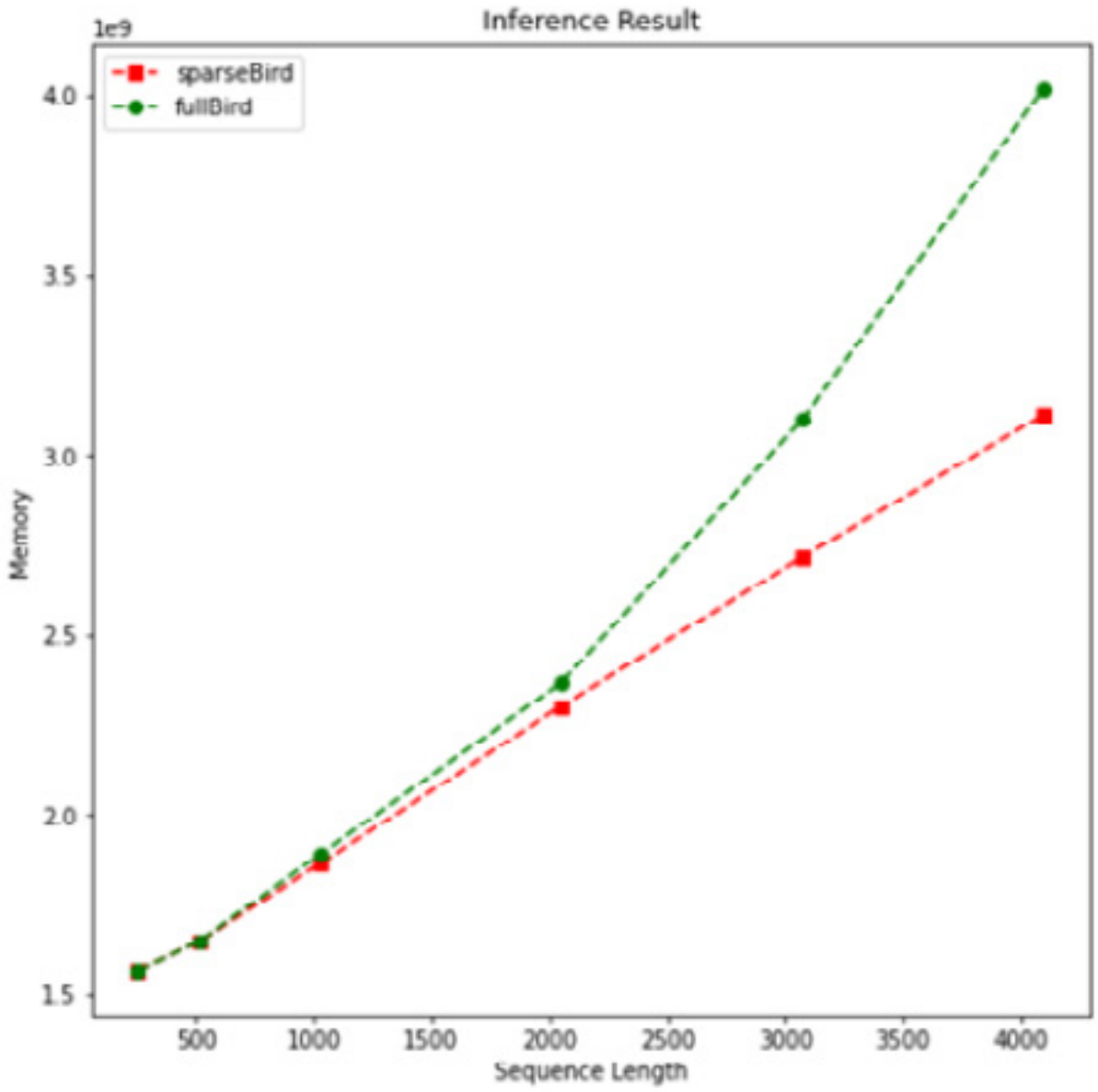


Figure 8.13 – Memory performance (BigBird)

In the preceding figure, we can clearly see linear and quadratic memory complexity. Once again, up to a certain point (a length of 2,000 in this example), we cannot speak of a clear distinction.

Next, let's discuss learnable patterns and work with models that can process longer input.

# Learnable patterns

Learning-based patterns are the alternatives to fixed (predefined) patterns. These approaches extract the patterns in an unsupervised data-driven fashion. They leverage some techniques to measure the similarity between the queries and the keys to properly cluster them. This transformer family learns first how to cluster the tokens and then restrict the interaction to get an optimum view of the attention matrix.

Now, we will do some experiments with Reformer as one of the important efficient models based on learnable patterns. Before that, let's address what the Reformer model contributes to the NLP field, as follows:

1. It employs **Local Self Attention** (**LSA**) that cuts the input into **n** chunks to reduce the complexity bottleneck. But this cutting process makes the boundary token unable to attend to its immediate neighbors. For example, in the chunks `[a,b,c]` and `[d,e,f]`, the token `d` cannot attend to its immediate context `c`. As a remedy, Reformer augments each chunk with the parameters that control the number of previous neighboring chunks.

2. The most important contribution of Reformer is to leverage the **Locality Sensitive Hashing** (**LSH**) function, which assigns the same value to similar query vectors. Attention could be approximated by only comparing the most similar vectors, which helps us reduce the dimensionality and then sparsify the matrix. It is a safe operation since the softmax function is highly dominated by large values and can ignore dissimilar vectors. Additionally, instead of finding the relevant keys to a given query, only similar queries are found and bucked. That is, the position of a query can only attend to the positions of other queries to which it has a high cosine similarity.
3. To reduce the memory footprint, Reformer uses reversible residual layers, which avoids the need to store the activations of all the layers to be reused for backpropagation, following the **Reversible Residual Network** (**RevNet**), because the activations of any layer can be recovered from the activation of the following layer.

   It is important to note that the Reformer model and many other efficient transformers are criticized as, in practice, they are only more efficient than the vanilla transformer when the input length is very long (*REF: Efficient Transformers*: A Survey, Yi Tay, Mostafa Dehghani, Dara Bahri, Donald Metzler). We made similar observations in our earlier experiments (please see the BigBird and Longformer experiment) .

4. Now we will conduct some experiments with Reformer. Thanks to the HuggingFace community again, the Transformers library provides us with Reformer implementation and its pre-trained checkpoints. We will load the configuration of the original checkpoint `google/reformer-enwik8` and also tweak some settings to work in full self-attention mode. When we set `lsh_attn_chunk_length` and `local_attn_chunk_length` to `16384`, which is the maximum length that Reformer can process, the Reformer instance will have no chance of local optimization and will automatically work like a vanilla transformer with full attention. We call it `fullReformer`. As for the original Reformer, we instantiate it with default parameters from the original checkpoint and call it `sparseReformer` as follows:

```
from transformers import ReformerConfig
fullReformer = ReformerConfig\
    .from_pretrained("google/reformer-enwik8",
        lsh_attn_chunk_length=16384,
        local_attn_chunk_length=16384)
sparseReformer = ReformerConfig\
    .from_pretrained("google/reformer-enwik8")
sequence_lengths=[256, 512, 1024, 2048, 4096, 8192, 12000]
models=["fullReformer","sparseReformer"]
configs=[eval(e) for e in models]
```

Please notice that the Reformer model can process sequences up to a length of 16384. But for the full self-attention mode, due to the accelerator capacity of our environment, the attention matrix does not fit on GPU, and we get a CUDA out of memory warning. Hence, we set the max length as 12000. If your environment is suitable, you can increase it.

5. Let's run the benchmark experiments as follows:

```
benchmark_args = PyTorchBenchmarkArguments(
    sequence_lengths=sequence_lengths,
    batch_sizes=[1],
    models=models)
benchmark = PyTorchBenchmark(
    configs=configs,
    args=benchmark_args)
result = benchmark.run()
```

The output is as follows:

```
====================       INFERENCE - SPEED - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length     Time in s
--------------------------------------------------------------------------------
         fullReformer                1              256            0.024
         fullReformer                1              512            0.036
         fullReformer                1             1024            0.075
         fullReformer                1             2048            0.196
         fullReformer                1             4096            0.529
         fullReformer                1             8192            1.722
         fullReformer                1            12000            3.443
       sparseReformer                1              256            0.026
       sparseReformer                1              512            0.049
       sparseReformer                1             1024            0.084
       sparseReformer                1             2048            0.165
       sparseReformer                1             4096            0.296
       sparseReformer                1             8192            0.576
       sparseReformer                1            12000            0.869
--------------------------------------------------------------------------------

====================       INFERENCE - MEMORY - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length    Memory in MB
--------------------------------------------------------------------------------
         fullReformer                1              256            1511
         fullReformer                1              512            1551
         fullReformer                1             1024            1671
         fullReformer                1             2048            2115
         fullReformer                1             4096            3791
         fullReformer                1             8192            8415
         fullReformer                1            12000           16191
       sparseReformer                1              256            1509
       sparseReformer                1              512            1705
       sparseReformer                1             1024            1885
       sparseReformer                1             2048            2339
       sparseReformer                1             4096            3143
       sparseReformer                1             8192            4803
       sparseReformer                1            12000            6367
--------------------------------------------------------------------------------
```

Figure 8.14 – Benchmark results

6. Let's visualize the time performance result as follows:

```
plotMe(result)
```

The output is the following:

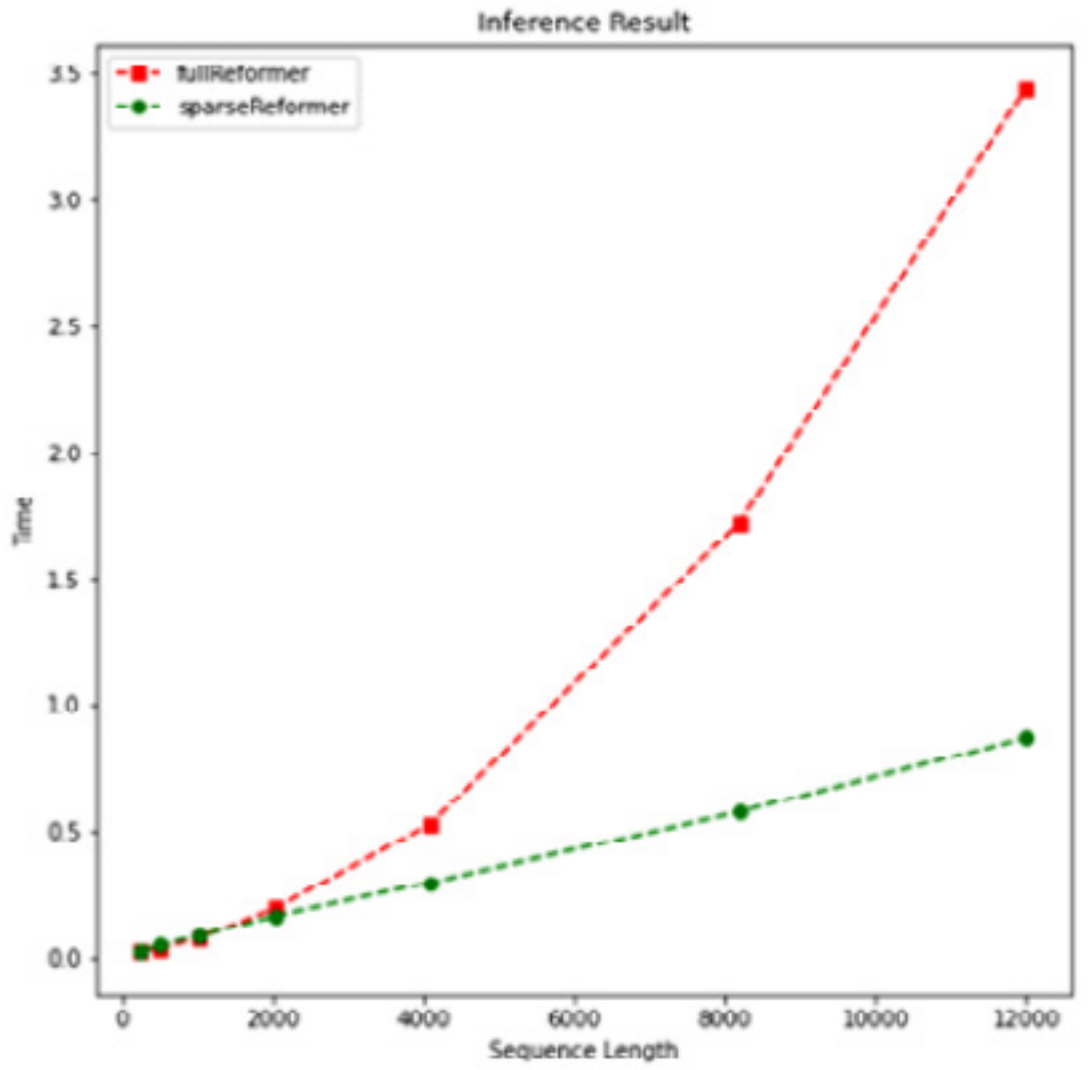


Figure 8.15 – Speed performance (Reformer)

7. We can see the linear and quadratic complexity of the models. We observe similar characteristics for the memory footprint by running the following line:

```
plotMe(result,"Memory Footprint")
```

Copy Explain

It plots the following:

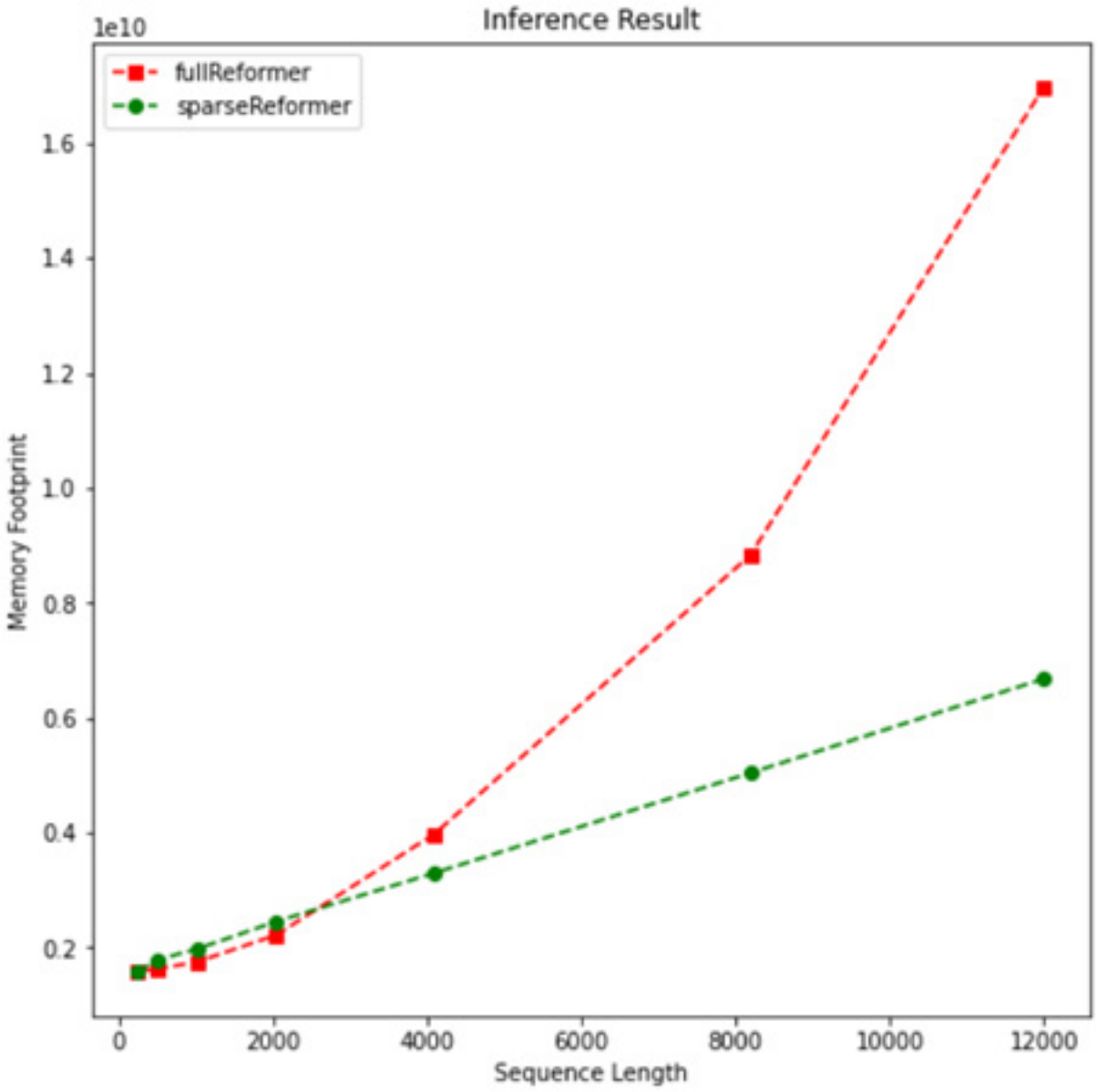


Figure 8.16 – Memory usage (Reformer)

Just as expected, Reformer with sparse attention produces a lightweight model. However, as was said before, we have difficulty observing the quadratic/linear complexity up to a certain length. As all these experiments indicate, efficient transformers can mitigate the time and memory complexity for longer text. What about task performance? How accurate would they be for classification or summarization tasks? To answer this, we will either start an experiment or have a look at the performance reports in the relevant articles of the models. For the experiment, you can repeat the code in `chapter 04` and `chapter 05` by instantiating an efficient model instead of a vanilla transformer. And you can track model performance and optimize it by using model tracking tools that we will discuss in detail in *Chapter 11*, *Attention Visualization and Experiment Tracking*.

## Low-rank factorization, kernel methods, and other approaches

The latest trend of the efficient model is to leverage low-rank approximations of the full self-attention matrix. These models are considered to be the lightest since they can reduce the self-attention complexity from *O(n2) to O(n)* in both computational time and memory footprint. Choosing a very small projection dimension *k*, such that *k << n*, then the memory and space complexity is highly reduced. Linformer and

Synthesizer are the models that efficiently approximate the full attention with a low-rank factorization. They decompose the dot-product $N \times N$ attention of the original transformer through linear projections.

Kernel attention is another method family that we have seen lately to improve efficiency by viewing the attention mechanism through kernelization. A kernel is a function that takes two vectors as arguments and returns the product of their projection with a feature map. It enables us to operate in high-dimensional feature space without even computing the coordinate of the data in that high-dimensional space, because computations within that space become more expensive. This is when the kernel trick comes into play. The efficient models based on kernelization enable us to re-write the self-attention mechanism to avoid explicitly computing the N×N matrix. In machine learning, the algorithm we hear the most about kernel methods is Support Vector Machines, where the radial basis function kernel or polynomial kernel are widely used, especially for nonlinearity. For transformers, the most notable examples are **Performer** and **Linear Transformers**.

---

# Summary

The importance of this chapter is that we have learned how to mitigate the burden of running large models under limited computational capacity. We first discussed and implemented how to make efficient models out of trained models using distillation, pruning, and quantization. It is important to pre-train a smaller general-purpose language model such as DistilBERT. Such light models can then be fine-tuned with good performance on a wide variety of problems compared to their non-distilled counterparts.

Second, we have gained knowledge about efficient sparse transformers that replace the full self-attention matrix with a sparse one using approximation techniques such as Linformer, BigBird, Performer, and so on. We have seen how they perform on various benchmarks such as computational complexity and memory complexity. The examples showed us these approaches are able to reduce the quadratic complexity to linear complexity without sacrificing the performance.

In the next chapter, we will discuss other important topics: cross-lingual/multi-lingual models.

---

# References


Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. arXiv preprint arXiv:1910.01108.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., & Weller, A. (2020). *Rethinking attention with performers*. arXiv preprint arXiv:2009.14794.

Wang, S., Li, B., Khabsa, M., Fang, H., & Ma, H. (2020). *Linformer: Self-attention with linear complexity*. arXiv preprint arXiv:2006.04768.

Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., ... & Ahmed, A. (2020). *Big bird: Transformers for longer sequences*. arXiv preprint arXiv:2007.14062.

Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). *Efficient transformers: A survey*. arXiv preprint arXiv:2009.06732.

Tay, Y., Bahri, D., Metzler, D., Juan, D. C., Zhao, Z., & Zheng, C. (2020). *Synthesizer: Rethinking self-attention in transformer models.* arXiv preprint arXiv:2005.00743.

Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020). *Reformer: The efficient transformer.* arXiv preprint arXiv:2001.04451.

Fournier, Q., Caron, G. M., & Aloise, D. (2021). *A Practical Survey on Faster and Lighter Transformers.* arXiv preprint arXiv:2103.14636.



Previous Chapter

Next Chapter