CS 5223 - Grad Project

Tushaar Gangavarapu - TG352 and Lucas Molter - LM865

Professor: David Bindel

May 19th 2023

Cornell University

Introduction

The backbone of recent (and important) NLP developments has been the Attention matrix, first mentioned at the 2017 paper "Attention is All You Need". To explain it very briefly, this approach has allowed NLP models represent words as a weighted average of other words.

This idea might seem odd at first, but if we think about it we can understand the underlying motivation. In a sentence such as:

*1. The **ball** broke the window when the kid kicked it.*

*2. The Universities's winter **ball** was very well organized.*

It is only possible to understand the meaning of **ball** in each sentence given the context, which is equivalent to saying that the true meaning of **ball** is conditioned to the other words in the sentence. For us, humans, this process is naturally learned when we learn how to speak, but for NLP models the solution found was the computation of the Attention matrix:

$$Attention = softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$$

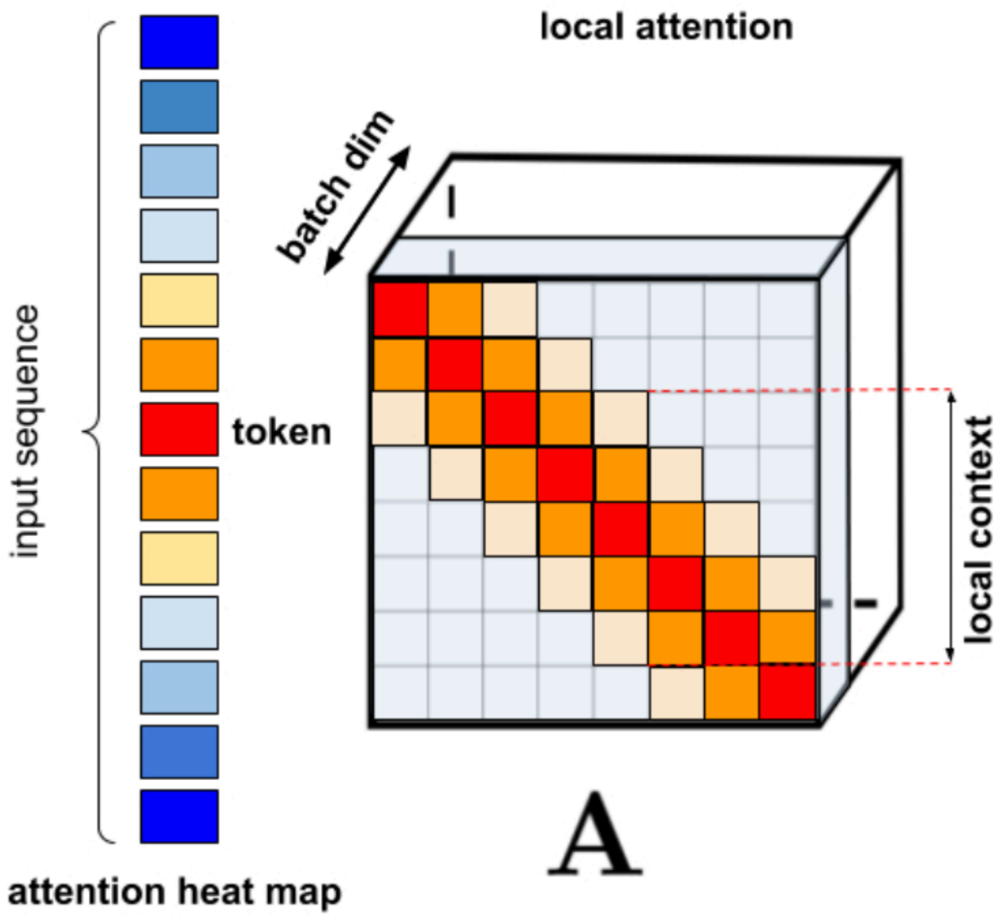
Where:

$$\begin{aligned} Q &= XW^Q \in \mathbb{R}^{L\times d} \\ K &= XW^K \in \mathbb{R}^{L\times d} \\ V &= XW^V \in \mathbb{R}^{L\times d} \end{aligned}$$

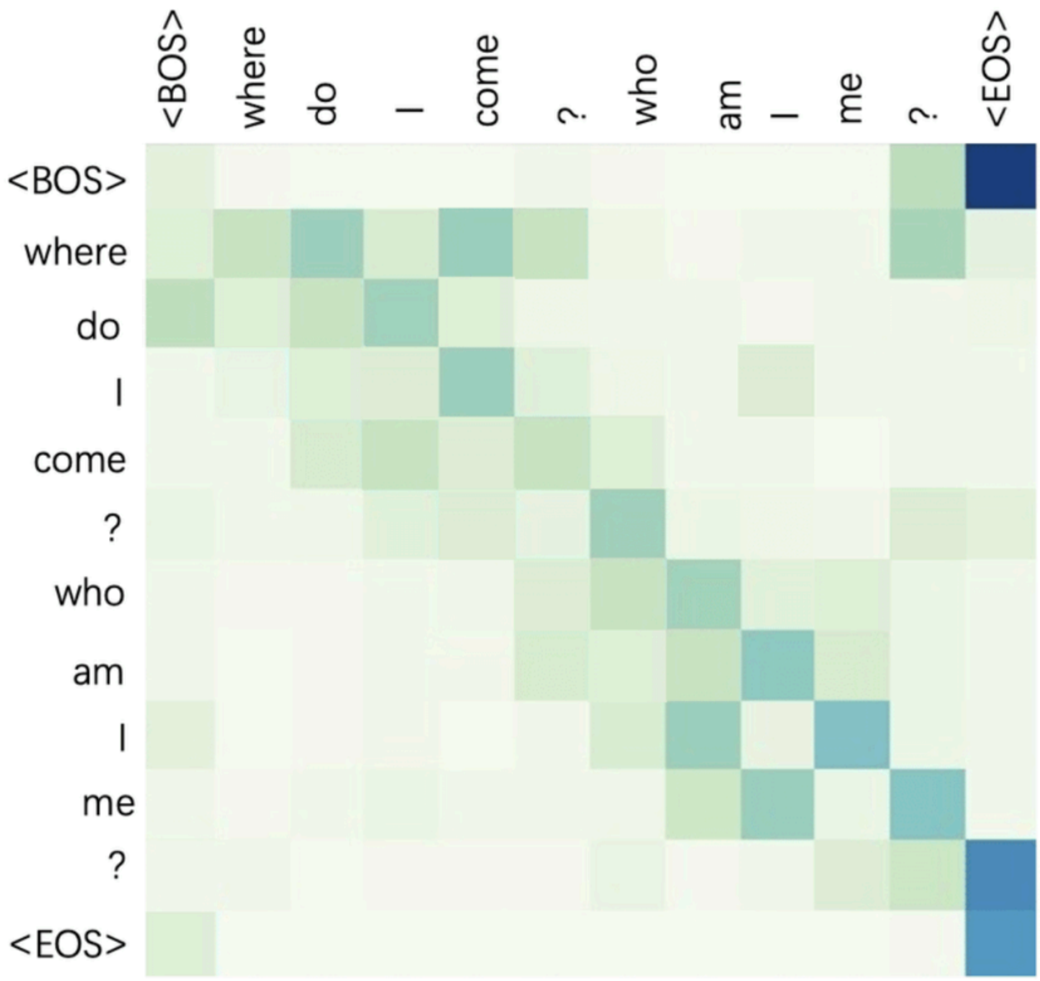
And more explicitly:

$$\begin{aligned} X \in \mathbb{R}^{L\times d} &= input\ embedding \equiv tokenized\ and\ projected\ words \\ L &= input\ sentence\ maximum\ length \\ d &= embedding\ dimension \\ W^Q \in \mathbb{R}^{d\times d} &= Model\ projection\ matrix\ Q \\ W^K \in \mathbb{R}^{d\times d} &= Model\ projection\ matrix\ K \\ W^V \in \mathbb{R}^{d\times d} &= Model\ projection\ matrix\ V \end{aligned}$$

There is a lot of NLP interpretations and nuances that we could explore here, but there would just make this report long (and possible boring). The main aspect here and theme of our CS5223 Grad project is the final structure that this Attention matrix assumes. As we can see in the images bellow, it is basically a diagonal matrix (convolution across neighbours or cells within a certain radius) added with a sparse matrix that takes care of the long range dependencies (just a pinch of NLP intuition here).



The image bellow is just a toy ilustration, but indeed helps to visualize the diagonal pattern:


<BOS>
where
do
I
come
?
who
am
I
me
?
<EOS>

Therefore, as discussed multiples times before, inspired by our project two, we decided to use FFT to compute this convolution faster and create a sparse light and easy to deal structure to accoun for the long range dependencies. Our expectation was a significant improvement in terms of speed given the simplificatin of the operations.

Before deriving our final approach, there is one final detail that is important to mention. Usually, the Attention operations happens in what is called "Multi-head Attention". Basically it means that the dimension d is going to be broken up into n (considering n heads) pieces of $\frac{d}{n}$. Each one of those pieces is individually processed by its head attention. In the end of the process all the outputs (from each head) are combined and reprojected to the same space by the matrix V. For more details and better intuition we recommend reading the paper "Attention is All You Need". However, to simplify and enhence comprehension, let's consider a single head, at least for now.

In more mathematical terms, the attention is as follows:

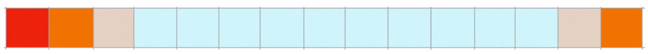
Let $X = (x^{(1)}, x^{(2)}, \dots, x^{(L)})$ be an input sequence of L tokens, where each token, $x^{(i)} \in \mathbb{R}^d$, is represented as a d-dimensional learnable embedding; $X \in \mathbb{R}^{L \times d}$. Let $W^K, W^Q, W^V \in \mathbb{R}^{d \times d}$ be learnable projection matrices (unique per layer, head).

Now, we have:

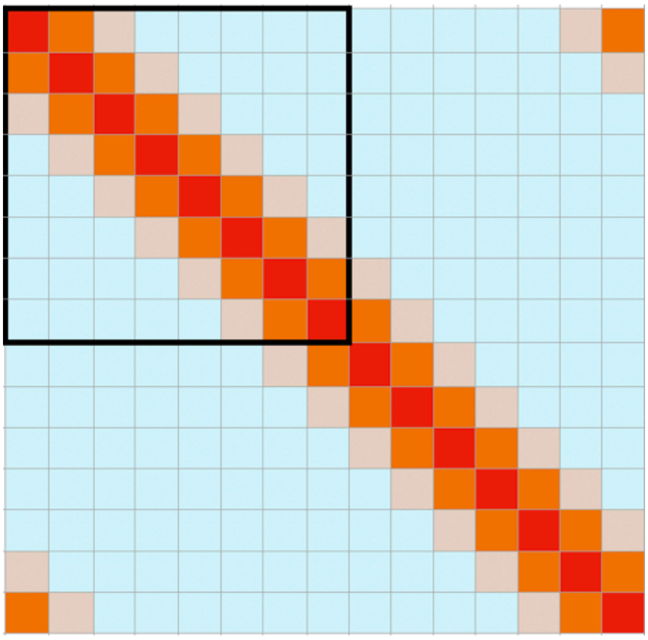
$$Q = XW^Q = \begin{bmatrix} \text{---}\ \left(x^{(1)}\right)^T W^Q\ \text{---} \\ \text{---}\ \left(x^{(2)}\right)^T W^Q\ \text{---} \\ \vdots \\ \text{---}\ \left(x^{(L)}\right)^T W^Q\ \text{---} \end{bmatrix} \in \mathbb{R}^{L \times d}$$
$$K = XW^K \in \mathbb{R}^{L \times d}$$
$$V = XW^V \in \mathbb{R}^{L \times d}$$

Still thinking one head at a time, our idea was basically create the following structure:

Let us apply a (trainable) filter kernel to mimic $QK^T \in \mathbb{R}^{L \times L}$ (viewed as $\mathbb{R}^{L^2}$). Unlike project 2, when we worked with circulant matrices, the expected structure of the Attention matrix (as shown in the images before) would be associated with a Toeplitz matrix, with a kernell ($\in \mathbb{R}^{2L-1}$) such as the image below:



Which generates now a circulant structure such as (we remark the dark boudries around the piece equivalent to the "original" attention):



This matrix we call the Toeplitz Augmentation Unit (*TAU*), where the dimensions are increased to accommodate the kernel and create the easy and fast to multiply circulant matrix. This happens because the convolution operation can be thought of as the linear mapping $H = Z^*\Sigma Z$ (where Z is the fast-Fourier matrix):

$$\tau = Z^*\Sigma Z$$

This τ "matrix" that accounts for the "weights" has then to multiply X to generate the new embeddings, and later on multiplied by another matrix that accounts for reprojection and across dimension (along d) interaction. Hence this part's entire operation is:

$$Z^*\Sigma Z X W^{\tau}$$

The other part, the sparse matrix doesn't have such refinement as our τ unit, but demands some explanation. From a mathematical perspective, to force a sparse matrix we used two techniques:

1. **Thresholding**: We forced our sparse matrix to have at most 30% of its entries non zero. If during backpropagation it happens to have more then 30% of the entries above zero we then keep only the 30% larger entries and zero out all the other ones.
2. **Regularization**: Aiming for some control over the sparse matrix, and avoid values that are too large, we also added to our loss function a penalization for the value in the entries of the sparse matrix, in other words, we added the norm of the matrix to the loss function.

Unfortunately, for NLP reasons (expressiveness and reprojections) we still have to multiply S by a parameter matrix $W^S, which reduces a little bit the speed up we were aiming for, but still resulted in a faster approach and with promissing accuracy results. In summary, what we did was:

$$softmax\left(\frac{QK^T}{\sqrt{d}}V\right) \approx (S(XW^X)W^S + Z^*\Sigma Z(XW^X)W^{\tau})$$

Given that the original paper that first computed the Attention Matrix does not use parallelism we also decided to not use it, and compare both structures with regular sequencial scripts, but we would like to remark that we could still deliver better results by making some further adaptations:

a. Creating faster sparse structures and operations wraping around Pytorch current classes b. Implementing parallelism in the multiple parts of our approach where it would be possible

To better illustrate the mentioned performance improvement we wrote testing scripts to evaluate execution time of the Attention Module and our Togepi Module to different sized random inputs. The results are in the end of the notebook.

PS: You are more than welcome to go over the code, but it is not necessary for the understanding of the results, hence you could jump directly to the end of the notebook.

togepi

toeplitz-based generative pretraining

In [1]:
```
!pip install torchinfo
```

```
Collecting torchinfo
  Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.8.0
```

In [2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torchinfo import summary

import math
from prettytable import PrettyTable

from dataclasses import dataclass
```

In [3]:
```
def print_params(module, print_vals=True):
    params_table = PrettyTable(['module', 'num_params', 'requires_g
    total_trainable_params = 0
    for name, param in module.named_parameters():
        params_table.add_row([name, param.numel(), param.requires_g
        if param.requires_grad:
            total_trainable_params = total_trainable_params + param
    print(params_table)
    if total_trainable_params > 1e6:
        print(f'total trainable params: {(total_trainable_params /
    else:
        print(f'total trainable params: {total_trainable_params}')
```

config

In [4]:
```
@dataclass
class TestTogepiConfig:
    # embedding
    vocab_size = 10  # includes special tokens ([PAD], [MASK], [CLS
    padding_idx = 0
    max_position_embeddings = 7  # includes proxy for padding token
    pad_position = 0
    num_token_types = 3  # includes padding token type
    pad_token_type = 0
    embedding_dim = 4
    embedding_dropout_proba = 0.1

    # attention
    causal_attn = True  # for generative pre-training
    num_attn_heads = 2
    attn_actn = 'gelu'
    sparse_dens = 0.3
    attn_dropout_proba = 0.1

test_config = TestTogepiConfig()
test_config.vocab_size
```

Out[4]: 10

embedding

In [5]:
```
class Embedding(nn.Module):
    def __init__(self, config):
        super().__init__()

        self._padding_idx = config.padding_idx
        self._pad_position = config.pad_position
        self._pad_token_type = config.pad_token_type

        self.tok_emb = nn.Embedding(num_embeddings=config.vocab_siz
        self.pos_emb = nn.Embedding(num_embeddings=config.max_posit
        self.type_emb = nn.Embedding(num_embeddings=config.num_toke

        nn.init.xavier_uniform_(self.tok_emb.weight.data)
        self.tok_emb.weight.data[self._padding_idx] = torch.zeros(c
        nn.init.xavier_uniform_(self.pos_emb.weight.data)
        self.tok_emb.weight.data[self._pad_position] = torch.zeros(
        nn.init.xavier_uniform_(self.type_emb.weight.data)
        self.tok_emb.weight.data[self._pad_token_type] = torch.zero

        self.layer_norm = nn.LayerNorm(normalized_shape=config.embe
        self.dropout = nn.Dropout(p=config.embedding_dropout_proba)

    def forward(self, input_ids, token_type_ids=None, padding_mask=
        # input_ids: (batch_size, max_length)
        # padding_mask: (batch_size, max_length)
        max_length = input_ids.shape[1]
```

```
        # assert(max_length == self.pos_emb.num_embeddings - 1)
        if padding_mask is None:
            # 1: no pad, 0: pad
            padding_mask = torch.where(input_ids == self._padding_i

        # position_ids: (batch_size, max_length)
        # assert(self._pad_position == 0)
        position_ids = torch.arange(max_length, dtype=torch.long, d
        position_ids = position_ids.unsqueeze(0).expand_as(input_id
        position_ids = position_ids.masked_fill(padding_mask == 0, 

        # token_type_ids: (batch_size, max_length)
        if token_type_ids is None:
            # assert(self._pad_token_type == 0)
            token_type_ids = torch.ones_like(input_ids)  # assuming
        token_type_ids = token_type_ids.masked_fill(padding_mask ==

        token_embeddings = self.tok_emb(input_ids)
        position_embeddings = self.pos_emb(position_ids)
        token_type_embeddings = self.type_emb(token_type_ids)

        return self.dropout(self.layer_norm(token_embeddings + posi

test_input_ids = torch.tensor([[1, 2, 3, 4, 0, 0], [3, 4, 5, 6, 7, 
test_emb_obj = Embedding(test_config)
test_emb = test_emb_obj(test_input_ids)
print_params(test_emb_obj)
test_emb, test_emb.shape
```

```
+-------------------+------------+---------------+
|       module      | num_params | requires_grad |
+-------------------+------------+---------------+
|   tok_emb.weight  |     40     |      True     |
|   pos_emb.weight  |     28     |      True     |
|  type_emb.weight  |     12     |      True     |
| layer_norm.weight |     4      |      True     |
|  layer_norm.bias  |     4      |      True     |
+-------------------+------------+---------------+
total trainable params: 88
```

Out[5]:
```
(tensor([[[-1.7468,  1.0223, -0.1759,  0.9004],
          [ 0.7343,  1.2877, -0.0000, -1.6012],
          [-1.7097,  1.3248, -0.1124,  0.4973],
          [ 1.1672,  1.0530, -1.0768, -1.1434],
          [ 0.8496,  0.2151, -1.8746,  0.8099],
          [ 0.8496,  0.2151, -1.8746,  0.8099]],

         [[-1.5472,  1.3251, -0.5070,  0.7291],
          [ 0.5841,  1.3160, -0.2221, -1.6780],
          [-1.1318, -0.9334,  1.6043,  0.4609],
          [ 0.5997,  1.4710, -1.4034, -0.6673],
          [-0.0000,  1.4685, -0.9210,  0.6706],
          [-1.6175,  0.0000,  0.6469, -0.3610]]], grad_fn=<MulBack
ward0>),
```

```
torch.Size([2, 6, 4]))
```

attention

In [6]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()

        assert(config.embedding_dim % config.num_attn_heads == 0)

        self._num_heads = config.num_attn_heads
        self._per_head_dim = config.embedding_dim // config.num_attn
        max_length = config.max_position_embeddings - 1

        self.wq = nn.Linear(in_features=config.embedding_dim, out_fe
        self.wk = nn.Linear(in_features=config.embedding_dim, out_fe
        self.wv = nn.Linear(in_features=config.embedding_dim, out_fe
        nn.init.xavier_normal_(self.wq.weight.data)
        nn.init.xavier_normal_(self.wk.weight.data)
        nn.init.xavier_normal_(self.wv.weight.data)

        self._causal = config.causal_attn
        if config.causal_attn:
            self.register_buffer('causal_attn_mask', torch.tril(torc

        self.wo = nn.Linear(in_features=config.embedding_dim, out_fe
        nn.init.xavier_normal_(self.wo.weight.data)

        self.layer_norm = nn.LayerNorm(normalized_shape=config.embed
        self.dropout = nn.Dropout(p=config.attn_dropout_proba)
        self.softmax = nn.Softmax(dim=-1)

    def _extend_padding_mask(self, padding_mask, embeddings):
        # padding_mask: (batch_size, max_length)
        if padding_mask is None:
            padding_mask = torch.ones(embeddings.shape[0], embedding

        extended_padding_mask = padding_mask.unsqueeze(1).unsqueeze(
        extended_padding_mask = extended_padding_mask.to(dtype=embed
        extended_padding_mask = (1 - extended_padding_mask) * -1e4
        return extended_padding_mask

    def forward(self, embeddings, padding_mask=None):
        batch_size = embeddings.shape[0]
        max_length = embeddings.shape[1]
        embedding_dim = embeddings.shape[2]

        # embeddings: (batch_size, max_length, embedding_dim)
        # attn_mask: 1 = non-pad, 0 = pad
        # projected_*: (batch_size, max_length, num_heads * per_head
        projected_query = self.wq(embeddings)
        projected_key = self.wk(embeddings)
```

```
        projected_value = self.wv(embeddings)

        sliced_projected_query = projected_query.view(batch_size, ma
        sliced_projected_key_tr = projected_query.view(batch_size, m
        sliced_projected_value = projected_query.view(batch_size, ma

        # attn_mat: (batch_size, num_heads, max_length, max_length)
        # attn_mat: QK' / sqrt(d)
        # attn_mask: set [pad] tok attn values to -inf
        attn_mat = torch.matmul(sliced_projected_query, sliced_proje
        attn_mat = attn_mat + self._extend_padding_mask(padding_mask
        if self._causal:
            attn_mat.masked_fill_(self.causal_attn_mask[:, :, :max_l
        # attn_probs: (batch_size, num_heads, max_length, max_length
        attn_probs = self.softmax(attn_mat)
        attn_probs = self.dropout(attn_probs)

        # ctx_vectors: (batch_size, num_heads, max_length, per_head_
        #    .permute: (batch_size, max_length, num_heads, per_head_
        #    .view   : (batch_size, max_length, num_heads * per_head
        ctx_vectors = torch.matmul(attn_probs, sliced_projected_valu
        attn_output = self.wo(ctx_vectors)
        attn_output = self.dropout(attn_output)

        return self.layer_norm(attn_output + embeddings), attn_probs

test_mha_obj = MultiHeadAttention(test_config)
test_padding_mask = torch.tensor([[1, 1, 1, 1, 0, 0], [1, 1, 1, 1, 1
test_mha_emb, test_mha_filters = test_mha_obj(test_emb, padding_mask
print_params(test_mha_obj)
test_mha_emb, test_mha_emb.shape
```

```
+-------------------+------------+---------------+
|       module      | num_params | requires_grad |
+-------------------+------------+---------------+
|     wq.weight     |     16     |      True     |
|      wq.bias      |     4      |      True     |
|     wk.weight     |     16     |      True     |
|      wk.bias      |     4      |      True     |
|     wv.weight     |     16     |      True     |
|      wv.bias      |     4      |      True     |
|     wo.weight     |     16     |      True     |
|      wo.bias      |     4      |      True     |
| layer_norm.weight |     4      |      True     |
|  layer_norm.bias  |     4      |      True     |
+-------------------+------------+---------------+
total trainable params: 88
```

Out[6]:
```
(tensor([[[-1.2223,  0.5883, -0.6726,  1.3067],
          [-0.2960,  1.3438,  0.3586, -1.4064],
          [-1.4819, -0.0752,  0.2372,  1.3199],
          [-0.1796,  1.5546, -0.1372, -1.2378],
          [-1.0302,  0.7668, -0.9444,  1.2078],
          [-1.0423,  0.9359, -0.9548,  1.0612]],
```

```
          [[-1.1850, -0.0741, -0.3184,  1.5776],
           [-0.9913,  1.0017,  0.9983, -1.0087],
           [ 1.1699, -1.5675,  0.4169, -0.0194],
           [-1.1786,  1.5858, -0.1229, -0.2843],
           [-1.2998,  1.1983, -0.6086,  0.7100],
           [ 0.0582, -1.6466,  0.9034,  0.6850]]],
         grad_fn=<NativeLayerNormBackward0>),
 torch.Size([2, 6, 4]))
```

In [7]:

```
class TogepiMultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()

        assert(config.embedding_dim % config.num_attn_heads == 0)

        self._num_heads = config.num_attn_heads
        self._per_head_dim = config.embedding_dim // config.num_attn
        max_length = config.max_position_embeddings - 1  # one posit
        self._training_max_length = max_length

        # out_features: (num_heads * per_head_dim)
        self.pre_proj = nn.Linear(in_features=config.embedding_dim,
        self.pre_sparse_proj = nn.Linear(in_features=config.embeddin
        nn.init.xavier_normal_(self.pre_proj.weight.data)
        nn.init.xavier_normal_(self.pre_sparse_proj.weight.data)

        # randomly initialize point-spread functions, one per head
        # psf: [tok_weight, [tok_-1_weights, tok_-2_weight, ...], [.
        self.toeplitz_psfs = nn.Parameter(torch.randn(self._num_head
        self.attn_actn = F.gelu if config.attn_actn == 'gelu' else F
        self.post_conv_proj = nn.Linear(in_features=config.embedding
        nn.init.xavier_normal_(self.toeplitz_psfs.data)
        nn.init.xavier_normal_(self.post_conv_proj.weight.data)

        num_nonzero = int(max_length * max_length * config.sparse_de
        sparse_idxs = torch.randint(0, max_length, (num_nonzero, 2))
        sparse_vals = torch.randn(num_nonzero)
        self.sparse = nn.Parameter(torch.sparse_coo_tensor(sparse_id

        self._causal = config.causal_attn
        if config.causal_attn:
            # causal_psf_mask: ignore the tokens appearing ahead of
            self.register_buffer('causal_psf_mask', torch.tensor([1]
            self.register_buffer('causal_sparse_mask', torch.tril(to

        self.layer_norm = nn.LayerNorm(normalized_shape=config.embed
        self.dropout = nn.Dropout(p=config.attn_dropout_proba)

    def forward(self, embeddings, padding_mask=None, softmax_psf_wei
        # embeddings: (batch_size, max_length, embedding_dim)
        # padding_mask: (batch_size, max_length)
        batch_size = embeddings.shape[0]
```

```
        max_length = embeddings.shape[1]
        embedding_dim = embeddings.shape[2]

        # expanded_padding_mask: (batch_size, max_length, 1)
        # 1: no pad, 0: pad
        expanded_padding_mask = None
        if padding_mask is not None:
            expanded_padding_mask = padding_mask.unsqueeze(2)

        # pre_proj_emb: (batch_size, max_length, num_heads * per_hea
        pre_proj_emb = self.pre_proj(embeddings)
        if padding_mask is not None:
            pre_proj_emb.masked_fill_(expanded_padding_mask == 0, 0)
        # padded_embeddings: (batch_size, 2 * max_length - 1, embedd
        # F.pad: pad=(padding_left, padding_right, padding_top, padd
        pre_proj_padded_embeddings = F.pad(pre_proj_emb, pad=(0, 0,
        # pre_proj_padded_embeddings: (batch_size, num_heads, 2 * ma
        pre_proj_padded_embeddings = pre_proj_padded_embeddings.view

        psfs_weights = self.toeplitz_psfs.data
        if self._causal:
            if self._training_max_length == max_length:
                psfs_weights.masked_fill_(self.causal_psf_mask == 0,
            else:
                # at inference time, the max_length changes per prom
                causal_psf_mask = torch.tensor([1] + [1] * (max_leng
                psfs_weights.masked_fill_(causal_psf_mask == 0, 0)
        if softmax_psf_weights:
            psfs_weights = F.softmax(psfs_weights, dim=1)
        psfs_fft = torch.fft.fftn(psfs_weights, dim=(1, 2))
        emb_fft = torch.fft.fftn(pre_proj_padded_embeddings, dim=(2,
        # conv_output: (batch_size, num_heads, max_length, per_head_
        conv_output = torch.real(torch.fft.ifftn(psfs_fft * emb_fft,
        # conv_output: (batch_size, max_length, num_heads * per_head
        conv_output = self.attn_actn(conv_output).permute(0, 2, 1, 3
        conv_emb = self.post_conv_proj(conv_output)


        sparse_data = self.sparse.data
        if self._causal:
            sparse_data.masked_fill_(self.causal_sparse_mask[:max_le
        pre_sparse_emb = self.pre_sparse_proj(pre_proj_emb)
        if padding_mask is not None:
            pre_sparse_emb.masked_fill_(expanded_padding_mask == 0,
        sparse_emb = torch.matmul(sparse_data, pre_sparse_emb)

        togepi_emb = self.dropout(conv_emb + sparse_emb)
        return self.layer_norm(togepi_emb + embeddings)

test_togepi_mha_obj = TogepiMultiHeadAttention(test_config)
test_padding_mask = torch.tensor([[1, 1, 1, 1, 0, 0], [1, 1, 1, 1, 1
test_togepi_mha_emb = test_togepi_mha_obj(test_emb, padding_mask=tes
print_params(test_togepi_mha_obj)
test_togepi_mha_emb, test_togepi_mha_emb.shape
```

```
+------------------------+------------+---------------+
|         module         | num_params | requires_grad |
+------------------------+------------+---------------+
|     toeplitz_psfs      |     44     |      True     |
|         sparse         |     36     |      True     |
|    pre_proj.weight     |     16     |      True     |
|     pre_proj.bias      |     4      |      True     |
| pre_sparse_proj.weight |     16     |      True     |
|  pre_sparse_proj.bias  |     4      |      True     |
| post_conv_proj.weight  |     16     |      True     |
|  post_conv_proj.bias   |     4      |      True     |
|   layer_norm.weight    |     4      |      True     |
|    layer_norm.bias     |     4      |      True     |
+------------------------+------------+---------------+
total trainable params: 148
```

Out[7]:

```
(tensor([[[-1.6801,  0.7827,  0.1640,  0.7334],
          [ 0.3760,  1.4663, -0.8449, -0.9974],
          [-1.5970,  1.1562,  0.1317,  0.3091],
          [ 0.9828,  0.9007, -1.4156, -0.4679],
          [ 0.7972,  0.1035, -1.6645,  0.7637],
          [-0.1368,  0.4456, -1.5210,  1.2122]],

         [[-1.4156,  1.1995, -0.4087,  0.6248],
          [ 0.2022,  1.5489, -0.7595, -0.9916],
          [-0.6960, -1.0655,  1.5246,  0.2369],
          [ 0.3313,  1.1090, -1.6210,  0.1807],
          [ 0.1517,  1.2346, -1.5568,  0.1705],
          [-1.0827,  0.1097, -0.5957,  1.5687]]],
        grad_fn=<NativeLayerNormBackward0>),
 torch.Size([2, 6, 4]))
```

speed tests

sparse vs. dense

In [37]:
```
def create_sparse_mat(sparse_dens=0.3, max_length=512):
    num_nonzero = int(max_length * max_length * sparse_dens)
    sparse_idxs = torch.randint(0, max_length, (num_nonzero, 2))
    sparse_vals = torch.randn(num_nonzero)
    return torch.sparse_coo_tensor(sparse_idxs.t(), sparse_vals.abs

def create_emb(batch_size=32, max_length=512, embedding_dim=768):
    return torch.randn(batch_size, max_length, embedding_dim)

def sparse_matmul(sparse_mat, emb):
    # sparse_mat: (max_length, max_length)
    batch_size, max_length, embedding_dim = emb.shape
    return torch.sparse.mm(sparse_mat, emb.permute(1, 0, 2).reshape

def sparse_to_dense_matmul(sparse_mat, emb):
    # sparse_mat: (max_length, max_length)
    return torch.matmul(sparse_mat.to_dense(), emb)

def dense_matmul(dense_mat, emb):
    return torch.matmul(dense_mat, emb)
```

In [10]:
```
sparse_dens = 0.3
max_length = 512
batch_size = 32
embedding_dim = 768

sparse_mat = create_sparse_mat(sparse_dens=sparse_dens, max_length=
dense_mat = sparse_mat.to_dense()
emb = create_emb(batch_size=batch_size, max_length=max_length, embe
```

In [11]:
```
%%timeit
sparse_matmul(sparse_mat, emb)
```

```
317 ms ± 2.04 ms per loop (mean ± std. dev. of 7 runs, 1 loop each
)
```

In [12]:
```
%%timeit
sparse_to_dense_matmul(sparse_mat, emb)
```

```
25 ms ± 199 µs per loop (mean ± std. dev. of 7 runs, 10 loops each
)
```

In [13]:
```
%%timeit
dense_matmul(dense_mat, emb)
```

```
24.9 ms ± 215 µs per loop (mean ± std. dev. of 7 runs, 10 loops ea
ch)
```

bert-attention vs. togepi-attention

In [38]:
```
# https://aclanthology.org/2021.emnlp-main.831.pdf
@dataclass
class SpeedTestConfig:
    # embedding
    vocab_size = 30522
    padding_idx = 0
    max_position_embeddings = 1024 + 1 #L
    pad_position = 0
    num_token_types = 3
    pad_token_type = 0
    embedding_dim = 2048 #d
    embedding_dropout_proba = 0.1

    # attention
    causal_attn = True  # for generative pre-training
    num_attn_heads = 16
    attn_actn = 'gelu'
    sparse_dens = 0.3
    attn_dropout_proba = 0.1

    # training
    batch_size = 64

test_speed_config = SpeedTestConfig()
test_speed_config.vocab_size
```

Out[38]:
```
30522
```

In [39]:
```
test_input_ids = torch.randint(low=0, high=test_speed_config.max_po
                               size=(test_speed_config.batch_size,
test_input_ids.shape
```

Out[39]:
```
torch.Size([64, 1024])
```

In [40]:
```
test_emb_obj = Embedding(test_speed_config)
test_emb = test_emb_obj(test_input_ids)
print_params(test_emb_obj)
test_emb.shape
```

```
+-------------------+------------+---------------+
|       module      | num_params | requires_grad |
+-------------------+------------+---------------+
|   tok_emb.weight  |  62509056  |      True     |
|   pos_emb.weight  |  2099200   |      True     |
|  type_emb.weight  |    6144    |      True     |
| layer_norm.weight |    2048    |      True     |
|  layer_norm.bias  |    2048    |      True     |
+-------------------+------------+---------------+
total trainable params: 64.62M
```

Out[40]:
```
torch.Size([64, 1024, 2048])
```

In [41]:
```
test_mha_obj = MultiHeadAttention(test_speed_config)
print_params(test_mha_obj)

test_togepi_mha_obj = TogepiMultiHeadAttention(test_speed_config)
print_params(test_togepi_mha_obj)
```

```
+-------------------+------------+---------------+
|       module      | num_params | requires_grad |
+-------------------+------------+---------------+
|     wq.weight     |  4194304   |      True     |
|      wq.bias      |    2048    |      True     |
|     wk.weight     |  4194304   |      True     |
|      wk.bias      |    2048    |      True     |
|     wv.weight     |  4194304   |      True     |
|      wv.bias      |    2048    |      True     |
|     wo.weight     |  4194304   |      True     |
|      wo.bias      |    2048    |      True     |
| layer_norm.weight |    2048    |      True     |
|  layer_norm.bias  |    2048    |      True     |
+-------------------+------------+---------------+
total trainable params: 16.79M
+------------------------+------------+---------------+
|         module         | num_params | requires_grad |
+------------------------+------------+---------------+
|     toeplitz_psfs      |  4192256   |      True     |
|         sparse         |  1048576   |      True     |
|    pre_proj.weight     |  4194304   |      True     |
|     pre_proj.bias      |    2048    |      True     |
| pre_sparse_proj.weight |  4194304   |      True     |
|  pre_sparse_proj.bias  |    2048    |      True     |
| post_conv_proj.weight  |  4194304   |      True     |
|  post_conv_proj.bias   |    2048    |      True     |
|   layer_norm.weight    |    2048    |      True     |
|    layer_norm.bias     |    2048    |      True     |
+------------------------+------------+---------------+
total trainable params: 17.83M
```

In [42]:
```
%%timeit
test_mha_emb, test_mha_filters = test_mha_obj(test_emb)
```

```
1min 14s ± 6.16 s per loop (mean ± std. dev. of 7 runs, 1 loop eac
h)
```

In [43]:
```
%%timeit
test_togepi_mha_emb = test_togepi_mha_obj(test_emb)
```

```
16 s ± 419 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Conclusion and Results

As mentioned before, altough the improvement have effects in the speed of the NLP application, for the project regarding CS5223 we decided to make something very specific to compare those models. This jupyter notebook contains the functions and classes definitions and also some very basic tests. (Despite being basic in terms of dimensionality, you are more than welcome to run then and have fun). Seeking for more robust and concrete results we ran both methods, Attention and Togepi, in a server (same type of GPU), to compare their performance in multiple different scenarios.

We see an improvement across multiple cases. For very small settings (small L and small d) Togepi doesn't seems to be faster because of some overhead that is necessary and not related to the size of the inputs. However the NLP challange is dealing with larger inputs, settings in which Togepi has consistently shown do be faster than Attention.

Analyzing the complexity of both methods, when we take dimensions to infinity, they have indeed the same theoretical lower bound $\mathcal{O}(L^2 d)$, however, if we considerer each part of the code, the Attention method is $\mathcal{O}(Ld^2) + \mathcal{O}(L^2 d) + \mathcal{O}(L^2) + \mathcal{O}(L^2 d)$ while in our case, given the speed up from FFT we are only bounded by the three matrix multiplication at the sparse side which, disconsidering sparsity effect would be $\mathcal{O}(L^2 d)$.

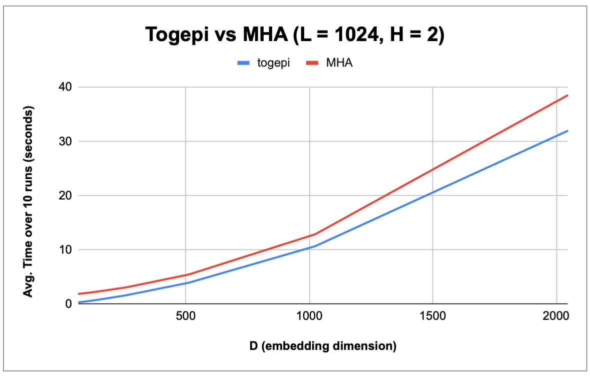
We show below the multiple tests we ran on the server:

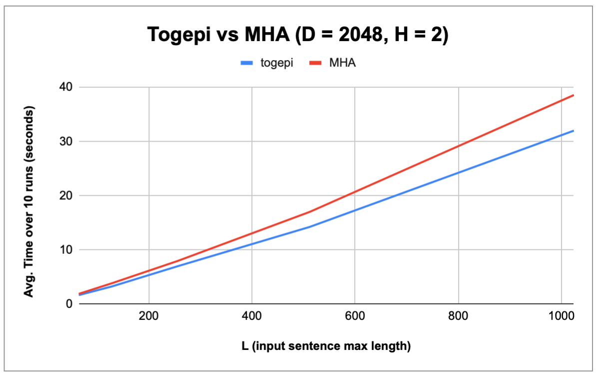
1. **Avg. Run Time (10 iterations) for 2 heads**

		L				
	Togepi H=2	**64**	**128**	**256**	**512**	**1024**
D	**64**	0,02	0,02	0,05	0,12	0,33
	128	0,03	0,05	0,12	0,25	0,69
	256	0,06	0,11	0,28	0,59	1,62
	512	0,17	0,32	0,72	1,52	3,95
	1024	0,5	1,01	2,14	4,45	10,68
	2048	1,67	3,26	7,00	14,24	31,97

		L				
	MHA H=2	**64**	**128**	**256**	**512**	**1024**
D	**64**	0,02	0,04	0,12	0,47	1,89
	128	0,02	0,05	0,16	0,57	2,22
	256	0,05	0,11	0,29	0,88	3,07
	512	0,14	0,31	0,72	1,86	5,46
	1024	0,5	1,05	2,23	5,13	12,88
	2048	1,9	3,85	7,97	17,01	38,53

		L				
	Togepi/MHA	**64**	**128**	**256**	**512**	**1024**
D	**64**	100,0%	50,0%	41,7%	25,5%	17,5%
	128	150,0%	100,0%	75,0%	43,9%	31,1%
	256	120,0%	100,0%	96,6%	67,0%	52,8%
	512	121,4%	103,2%	100,0%	81,7%	72,3%
	1024	100,0%	96,2%	96,0%	86,7%	82,9%
	2048	87,9%	84,7%	87,8%	83,7%	83,0%



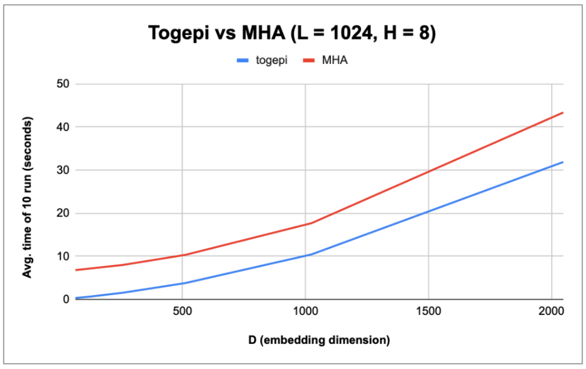


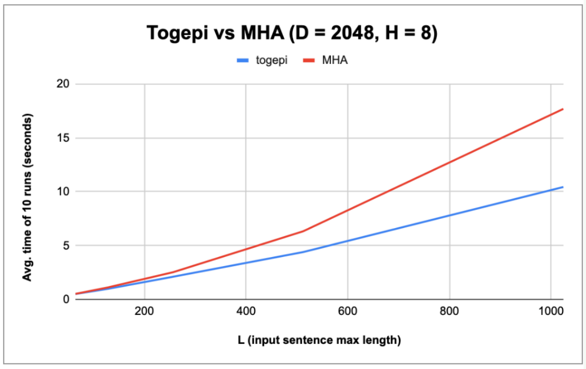
2. **Avg. Run Time (10 iterations) for 8 heads**

		L				
	Togepi H=8	64	128	256	512	1024
D	64	0,01	0,02	0,05	0,12	0,32
	128	0,03	0,04	0,11	0,25	0,67
	256	0,06	0,11	0,27	0,58	1,53
	512	0,16	0,32	0,72	1,54	3,79
	1024	0,5	0,98	2,1	4,39	10,43
	2048	1,69	3,31	7,02	14,51	31,88

		L				
	MHA H=8	64	128	256	512	1024
D	64	0,03	0,11	0,42	1,68	6,79
	128	0,04	0,12	0,45	1,79	7,19
	256	0,06	0,18	0,57	02.08	7,98
	512	0,16	0,37	1	3,07	10,34
	1024	0,51	1,11	2,52	6,32	17,68
	2048	1,9	3,9	8,27	18,22	43,37

		L				
	Togepi/MHA	64	128	256	512	1024
D	64	33,3%	18,2%	11,9%	7,1%	4,7%
	128	75,0%	33,3%	24,4%	14,0%	9,3%
	256	100,0%	61,1%	47,4%	0,0%	19,2%
	512	100,0%	86,5%	72,0%	50,2%	36,7%
	1024	98,0%	88,3%	83,3%	69,5%	59,0%
	2048	88,9%	84,9%	84,9%	79,6%	73,5%



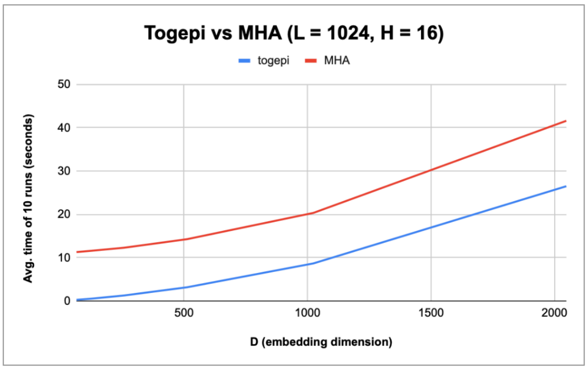


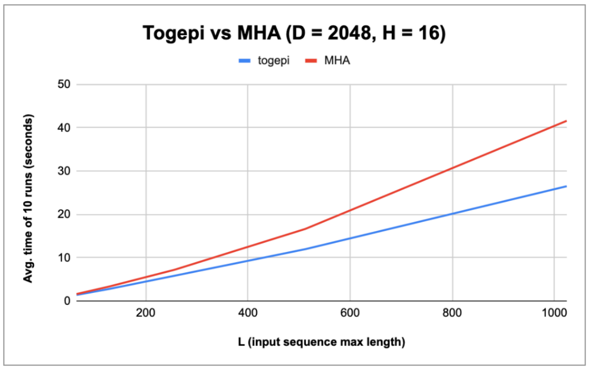
3. **Avg. Run Time (10 iterations) for 16 heads**

		L				
	Togepi H=16	64	128	256	512	1024
D	64	0,02	0,02	0,05	0,1	0,27
	128	0,02	0,04	0,1	0,2	0,56
	256	0,05	0,09	0,22	0,49	1,27
	512	0,14	0,26	0,59	1,25	3,17
	1024	0,42	0,81	1,8	3,77	8,68
	2048	1,4	2,78	5,83	11,96	26,49

		L				
	MHA H=16	64	128	256	512	1024
D	64	0,05	0,17	0,69	2,78	11,29
	128	0,05	0,19	0,71	2,89	11,61
	256	0,07	0,23	0,81	3,15	12,29
	512	0,15	0,4	1,17	3,96	14,27
	1024	0,45	1,01	2,46	6,65	20,34
	2048	1,61	3,34	7,21	16,63	41,59

		L				
	Togepi/MHA	64	128	256	512	1024
D	64	40,0%	11,8%	7,2%	3,6%	2,4%
	128	40,0%	21,1%	14,1%	6,9%	4,8%
	256	71,4%	39,1%	27,2%	15,6%	10,3%
	512	93,3%	65,0%	50,4%	31,6%	22,2%
	1024	93,3%	80,2%	73,2%	56,7%	42,7%
	2048	87,0%	83,2%	80,9%	71,9%	63,7%





Thanks for the great semester and have a great summer