# 1a

Identify the differences between Google Patent's architecture and the code given in the module:

1. The design layout given out in Google's patent is for their translation services, so they have used both encoder, decoder networks while we have just used the decoder only architecture.
2. Google's patent has the provision to combine inputs from encoder and decoder into a multi-attention layer, our code avoids this step because of the decoder-only architecture.
3. A significant deviation is in the positional encoder where original architecture employs a sinusoidal version,but the code uses learned positional encoders.
4. The initial discrepancy I identified lies in the embedding size used within the code, where embed_size is set to 128, whereas the paper specifies d_model as 512. This is modifiable based on the application

# 1b

Modifications on the code:

1. Modified the code by adding code to perform positional encoding using sinusoidal waveform.
2. Also modified the code to run the code on gpu.

```
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
import math

''' Look at all previous tokens to generate next
    @Author: Uzair Ahmad
    2022
    +TransformerBlock
'''


class TransformerBlockLM(nn.Module):
    class TransformerBlock(nn.Module):
        def __init__(self, head_count, in_size, out_size):
            super().__init__()
            self.comm = TransformerBlockLM.MultiHeadAttention(head_count=head_count,
                                                              in_size=in_size,
                                                              out_size=out_size)
            self.think = TransformerBlockLM.MLP(embed_size=out_size)

        def forward(self, x):
            return x + self.think(x + self.comm(x))

    class MLP(nn.Module):
        # FFNN (embed_size, embed_size*4, embed_size)
        def __init__(self, embed_size):
            super().__init__()
            self.mlp = nn.Sequential(nn.Linear(embed_size, embed_size * 4),
                                     nn.ReLU(),
                                     nn.Linear(embed_size * 4, embed_size))
            self.layerNorm = nn.LayerNorm(embed_size)

        def forward(self, x):  # think
            return self.layerNorm(self.mlp(x))  # paper - after
            # return self.mlp(self.layerNorm(x)) # alternate - before

    class MultiHeadAttention(nn.Module):
        """
        multiple parallel SA heads (communication among words)
        """

        def __init__(self, head_count, in_size, out_size):
            super().__init__()
            self.heads = nn.ModuleList(
                TransformerBlockLM.SelfAttentionHead(in_size, out_size // head_count)
                for _ in range(head_count)
            )
            self.layerNorm = nn.LayerNorm(out_size)
```

```
            self.layerNorm = nn.LayerNorm(out_size)
            # self.proj = nn.Linear(out_size, out_size)

        def forward(self, x):
            # concat over channel/embeddings_size dimension
            return self.layerNorm(torch.cat([head(x) for head in self.heads], dim=-1))  # paper - after
            # return torch.cat([head(self.layerNorm(x)) for head in self.heads], dim=-1) # alternate - before
            # return self.proj(torch.cat([head(x) for head in self.heads], dim=-1))

    class SelfAttentionHead(nn.Module):
        def __init__(self, in_size, out_size):
            """
            in_size is embed_size
            out_size is head_size
            """
            super().__init__()
            self.head_size = out_size
            self.K = nn.Linear(in_size, self.head_size, bias=False)
            self.Q = nn.Linear(in_size, self.head_size, bias=False)
            self.V = nn.Linear(in_size, self.head_size, bias=False)

        def forward(self, x):
            keys = self.K(x)
            queries = self.Q(x)
            # affinities :
            # all the queries will dot-product with all the keys
            # transpose (swap) second dimension (input_length) with third (head_size)
            keys_t = keys.transpose(1, 2)
            autocorrs = (queries @ keys_t) * (self.head_size ** -0.5)  # (batch_size x input_length x input_length)
            '''
            (batch_size x input_length x embed_size) @ (batch_size x embed_size x input_length) ----> (batch_size x input_ler
            '''
            autocorrs = torch.tril(autocorrs)
            autocorrs = autocorrs.masked_fill(autocorrs == 0, float('-inf'))
            autocorrs = torch.softmax(autocorrs, dim=-1)
            values = self.V(x)  # (batch_size x input_length x head_size)
            out = autocorrs @ values
            return out

    def __init__(self, batch_size=4,
                 input_length=8,
                 embed_size=16,
                 sa_head_size=8,
                 sa_multihead_count=4,
                 pos_embed=False,
                 include_mlp=False):
        super().__init__()
        self.blocks = None
        self.ffn = None
        self.sa_heads = None
        # sa_head_size head_size of self-attention module
        self.sa_head_size = sa_head_size
        self.sa_multihead_count = sa_multihead_count

        self.val_data = None
        self.train_data = None
        self.val_text = None
        self.train_text = None
        self.K = None
        self.linear_sahead_to_vocab = None
        self.vocab = None
        self.token_embeddings_table = None
        self.vocab_size = None
        self.encoder = None
        self.decoder = None
        self.vocab_size: int
        self.is_pos_emb = pos_embed
        self.include_mlp = include_mlp
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        # input_length = how many consecutive tokens/chars in one input
        self.input_length = input_length
        # batch_size = how many inputs are going to be processed in-parallel (on GPU)
        self.batch_size = batch_size
        # embed_size = embedding size
        self.embed_size = embed_size

        self.lm_head = None
        self.position_embeddings_table = None

    def forward(self, in_ids, target=None):
    # Embed the input ids using the token embeddings table
      in_ids_emb = self.token_embeddings_table(in_ids[:, -self.input_length:]).to(self.device)
      if self.is_pos_emb:
```

```
            # Compute the sinusoidal positional embeddings
            # Define a helper function inside the forward method
            def get_pos_emb(length, d_model, device):
                # Create a tensor of positions from 0 to length - 1
                pos = torch.arange(length, dtype=torch.float, device=device).unsqueeze(1)
                # Create a tensor of scaling factors for each dimension
                scale = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model)).to(self.device)
                # Create a zero tensor for the positional embeddings
                pe = torch.zeros(length, d_model, device=device)
                # Compute the sine and cosine values for the even and odd dimensions
                pe[:, 0::2] = torch.sin(pos * scale)
                pe[:, 1::2] = torch.cos(pos * scale)
                return pe
            # Get the positional embeddings for the input ids
            pos_emb = get_pos_emb(in_ids[:, -self.input_length:].shape[1], in_ids_emb.shape[-1], self.device)
            # Add the positional embeddings to the token embeddings
            in_ids_emb = in_ids_emb + pos_emb
        # Pass the embedded input through the transformer blocks
        block_outputs = self.blocks(in_ids_emb)
        # Project the output of the transformer blocks to the vocabulary size
        logits = self.linear_sahead_to_vocab(block_outputs)
        # Compute the cross-entropy loss if the target is given
        if target is not None:
            # Reshape the logits and the target to match the expected shape
            batch_size, input_length, vocab_size = logits.shape
            logits = logits.view(batch_size * input_length, vocab_size)
            target = target.view(batch_size * input_length)
            # Compute the cross-entropy loss
            ce_loss = F.cross_entropy(logits, target)
        else:
            # Set the loss to None
            ce_loss = None
        # Return the logits and the loss
        return logits, ce_loss

    def fit(self, train_iters=100, eval_iters=10, lr=0.0001):
        """
        train_iters = how many training iterations
        eval_iters = how many batches to evaluate to get average performance
        """
        optimizer = torch.optim.Adam(self.parameters(), lr=lr)
        for iteration in range(train_iters):
            if iteration % eval_iters == 0:
                avg_loss = self.eval_loss(eval_iters)
                print(f"iter {iteration}: train {avg_loss['train']} val {avg_loss['eval']}")
            inputs, targets = self.get_batch(split='train')
            _, ce_loss = self(inputs, targets)
            optimizer.zero_grad(set_to_none=True)  # clear gradients of previous step
            ce_loss.backward()  # propagate loss back to each unit in the network
            optimizer.step()  # update network parameters w.r.t the loss
        # torch.save(self, 'sa_pos_')

    def generate(self, context_token_ids, max_new_tokens):
        for _ in range(max_new_tokens):
            token_rep, _ = self(context_token_ids)
            last_token_rep = token_rep[:, -1, :]
            probs = F.softmax(last_token_rep, dim=1)
            next_token = torch.multinomial(probs, num_samples=1)
            context_token_ids = torch.cat((context_token_ids, next_token), dim=1)
        output_text = self.decoder(context_token_ids[0].tolist())
        return output_text

    @torch.no_grad()  # tell torch not to prepare for back-propagation (context manager)
    def eval_loss(self, eval_iters):
        perf = {}
        # set dropout and batch normalization layers to evaluation mode before running inference.
        self.eval()
        for split in ['train', 'eval']:
            losses = torch.zeros(eval_iters)
            for k in range(eval_iters):
                tokens, targets = self.get_batch(split)  # get random batch of inputs and targete
                _, ce_loss = self(tokens, targets)  # forward pass
                losses[k] = ce_loss.item()  # the value of loss tensor as a standard Python number
            perf[split] = losses.mean()
        self.train()  # turn-on training mode-
        return perf

    def prep(self, corpus):
        self.vocab = sorted(list(set(corpus)))
        self.vocab_size = len(self.vocab)
        c2i = {c: i for i, c in
               enumerate(self.vocab)}  # char c to integer i map. assign value i for every word in vocab
        i2c = {i: c for c, i in c2i.items()}  # integer i to char c map
```

```
        self.encoder = lambda doc: [c2i[c] for c in doc]
        self.decoder = lambda nums: ''.join([i2c[i] for i in nums])

        n = len(text)
        self.train_text = text[:int(n * 0.9)]
        self.val_text = text[int(n * 0.9):]

        self.train_data = torch.tensor(self.encoder(self.train_text), dtype=torch.long).to(self.device)
        self.val_data = torch.tensor(self.encoder(self.val_text), dtype=torch.long).to(self.device)

        # look-up table for embeddings (vocab_size x embed_size)
        # it will be mapping each token id to a vector of embed_size
        # a wrapper to store vector representations of each token
        self.token_embeddings_table = \
            nn.Embedding(self.vocab_size, self.embed_size).to(self.device)

        if self.is_pos_emb:
            self.position_embeddings_table = nn.Embedding(self.input_length, self.embed_size).to(self.device)

        self.blocks = nn.Sequential(
            TransformerBlockLM.TransformerBlock(head_count=self.sa_multihead_count,
                                                in_size=self.embed_size,
                                                out_size=self.sa_head_size).to(self.device),
            TransformerBlockLM.TransformerBlock(head_count=self.sa_multihead_count,
                                                in_size=self.embed_size,
                                                out_size=self.sa_head_size).to(self.device),
            TransformerBlockLM.TransformerBlock(head_count=self.sa_multihead_count,
                                                in_size=self.embed_size,
                                                out_size=self.sa_head_size).to(self.device),
            TransformerBlockLM.TransformerBlock(head_count=self.sa_multihead_count,
                                                in_size=self.embed_size,
                                                out_size=self.sa_head_size).to(self.device),
            TransformerBlockLM.TransformerBlock(head_count=self.sa_multihead_count,
                                                in_size=self.embed_size,
                                                out_size=self.sa_head_size).to(self.device),
            TransformerBlockLM.TransformerBlock(head_count=self.sa_multihead_count,
                                                in_size=self.embed_size,
                                                out_size=self.sa_head_size).to(self.device),
        )
        # linear projection of sa_head output to vocabulary
        self.linear_sahead_to_vocab = nn.Linear(self.sa_head_size, self.vocab_size).to(self.device)

    def get_batch(self, split='train'):
        data = self.train_data if split == 'train' else self.val_data
        # get random chunks of length batch_size from data
        ix = torch.randint(len(data) - self.input_length,
                           (self.batch_size,))
        inputs_batch = torch.stack([data[i:i + self.input_length] for i in ix])
        targets_batch = torch.stack([data[i + 1:i + self.input_length + 1] for i in ix])
        inputs_batch = inputs_batch.to(self.device)
        targets_batch = targets_batch.to(self.device)
        # inputs_batch is
        return inputs_batch, targets_batch


with open('./WarrenBuffet.txt', 'r') as f:
    text = f.read()

# text = 'a quick brown fox jumps over the lazy dog.\n ' \
#        'lazy dog and a quick brown fox.\n' \
#        'the dog is lazy and the fox jumps quickly.\n' \
#        'a fox jumps over the dog because he is lazy.\n' \
#        'dog is lazy and fox is brown. she quickly jumps over the lazy dog.'

model = TransformerBlockLM(batch_size=64,
                           input_length=32,
                           embed_size=128,
                           sa_multihead_count=8,
                           sa_head_size=128,
                           pos_embed=True,
                           include_mlp=True)
model = model.to(model.device)
model.prep(text)
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
print(f'params {sum([np.prod(p.size()) for p in model_parameters])}')
input_batch, output_batch = model.get_batch(split='train')
_, _ = model(input_batch, output_batch)
model.fit(train_iters=4000, eval_iters=1000, lr=1e-3)
```

```
params 1115739
iter 0: train 5.704237937927246 val 5.720011234283447
iter 1000: train 1.4822652339935303 val 1.6298166513442993
iter 2000: train 1.3000602722167969 val 1.5403438806533813
iter 3000: train 1.1915180683135986 val 1.528475284576416
```

```
outputs = model.generate(context_token_ids=torch.zeros((1, 1),
                                                        dtype=torch.long,
                                                        device=model.device),
                         max_new_tokens=1000)
print(outputs)
```

```
action In for meting $679 million. In 2006, specially public charging
the 80-year loan
of money. Charlie and I serie about $2 billion. I am eain it head 2005, him
commpanies. You can of you same we than told Sedream a CEO of A,

Nernigh Ne Marking, Fred, has never become on by its nothing: The Eleven "Every earnings paragre of this her purchase ac
of every investment for intrancial $100 lets year. At a
some probacks for the questions that
the most of our customers in the world
on Capmarks
and Picking it Jack bolt-or "them."

112


Desciones all of cters, weak, how many have
rolioys. All of their each substantial compilated returned by Justin books.


17


BERA period a posses to have a value of combined comities), no low
adding simple. Over time, the cost of the other industry. I fest implices and, esce- of since business and losses $650 r


26


The midame have none just moneting Seath at I decided, the induring of mysself
```

## 2a

Model performance: Perplexity

```
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader

def calculate_perplexity(model, eval_data):
    model.eval()  # Set the model to evaluation mode
    data_loader = DataLoader(eval_data, batch_size=model.batch_size, shuffle=False)

    total_loss = 0.0
```

Average Loss: 1.55 The average loss is a measure of how well the model is predicting the target sequence. Lower values indicate better performance. In this case, an average loss of 1.55 is relatively low, suggesting that, on average, the model is making accurate predictions.

Perplexity: 4.69 Perplexity is another way to measure the quality of a language model. It is a measure of how well the model predicts the next token in a sequence. Lower perplexity values indicate better performance. A perplexity of 4.69 is quite good and suggests that the model is generally effective at predicting the next token in the sequence.

## 2b

Most impressive text

```
seed_phrase = """The highlight of the year, however, was our July 5 th acquisition of most of ISCAR, an Israeli
company, and our new association with its chairman, Eitan Wertheimer, and CEO, Jacob Harpaz"""
context_token_ids = torch.tensor([model.encoder(seed_phrase)], dtype=torch.long).to(model.device)

outputs = model.generate(context_token_ids=context_token_ids, max_new_tokens=1000)
print(outputs)
```

```
The highlight of the year, however, was our July 5 th acquisition of most of ISCAR, an Israeli
company, and our new association with its chairman, Eitan Wertheimer, and CEO, Jacob Harpaz. And ISncor',

Abounts claims produce of Ederior in 1798, we well need Leceivable Berkshire. This losses
being slow wish perform, ployit deevelopied into the great authout 50% over at the bligared. They did of the old becaus
challege addvertisting mansude wondership or many self-salent in the future recommed of a policy recoverine to come fro
cash equivalent of completing to us that helple: And achieved, in a fline funding its this told you wisdom exposed us w
doil services. Because off financie
housing
commitments to ustriste
we agned to on depread pay in a
medical from Blumkin  share wither sume operation. No in the higgh-yieldings, that are exords. Overage the borne
Marmon partnelled Pocific, however, there is happer in bought North Clayton Home of Fruic operations,
the most holding trouble earned 9/30; can we unipos say, the housing
```

Even though the model is not up to the mark and there are a few grammatical, and spelling mistakes it can provide relevant text. Especially in this case when I fed the text about acquisition, it was able to look into the data for the data about acquisitions and provided us with information about other acquisitions. The most impactful design choices responsible for the text according to me are multi-head attention layers that we have developed, using which the text is able to relate and produce meaningful results by understanding context.