**Nama : Aisa Selvira Q.A**

**NIM : 22/498516/TK/54690**

# TUGAS 2

## 1. Pendahuluan

Laporan ini membahas hasil implementasi program Transformer Decoder yang dibuat sepenuhnya menggunakan NumPy tanpa bantuan framework pembelajaran mesin seperti PyTorch atau TensorFlow. Tujuan utama dari implementasi ini adalah memahami secara mendalam bagaimana mekanisme internal Transformer bekerja, dimulai dari proses *embedding* hingga terbentuknya distribusi probabilitas token berikutnya.

Program ini terdiri dari beberapa komponen inti yang disusun menyerupai arsitektur model GPT (*decoder-only transformer*). Komponen tersebut mencakup *embedding layer*, *positional encoding*, *multi-head attention*, *feed-forward network*, dan *layer normalization*. Setiap bagian diimplementasikan secara manual untuk memberikan pemahaman konseptual terhadap alur komputasi di dalam Transformer.

## 2. Inisialisasi dan Fungsi Pendukung

Pada bagian awal, program melakukan inisialisasi beberapa fungsi penting yang menjadi dasar bagi seluruh perhitungan di dalam model. Fungsi softmax digunakan untuk menormalkan nilai *logits* menjadi distribusi probabilitas sehingga total seluruh nilainya berjumlah satu. Sedangkan fungsi GELU (Gaussian Error Linear Unit) berfungsi sebagai aktivasi non-linear yang digunakan dalam jaringan *feed-forward*. Fungsi aktivasi ini memberikan hasil yang lebih halus dan stabil dibandingkan aktivasi ReLU, sehingga umum digunakan pada model modern seperti Transformer.

Selain itu, terdapat kelas LayerNorm yang berfungsi menormalkan setiap masukan agar memiliki rata-rata nol dan varians satu. Proses normalisasi ini penting untuk menjaga kestabilan data selama melewati lapisan-lapisan model, serta mencegah terjadinya pergeseran nilai yang dapat mengganggu proses pembelajaran atau perhitungan.

## 3. Proses Token Embedding

```python
# --------------------------- Embedding --------------------------------
class TokenEmbedding:
    def __init__(self, vocab_size, d_model):
        self.vocab_size = vocab_size
        self.d_model = d_model
        # initialize embedding matrix: (vocab, d_model)
        self.W = np.random.randn(vocab_size, d_model) / np.sqrt(d_model)

    def __call__(self, token_ids):
        # token_ids: [batch, seq_len]
        return self.W[token_ids]  # returns [batch, seq_len, d_model]
```

Bagian ini bertanggung jawab untuk mengubah token yang direpresentasikan sebagai angka menjadi bentuk vektor berdimensi tetap yang dapat dipahami oleh model. Kelas TokenEmbedding membuat matriks embedding berukuran sesuai dengan jumlah kosakata dan dimensi model (*d_model*). Matriks tersebut diisi dengan nilai acak yang dibagi berdasarkan akar dimensi model untuk menjaga kestabilan skala.

Ketika input token diberikan dalam bentuk indeks numerik, setiap indeks akan digunakan untuk mengambil baris dari matriks embedding yang sesuai. Hasilnya adalah representasi vektor dari setiap token. Menariknya, bobot embedding ini juga digunakan kembali pada lapisan keluaran melalui mekanisme weight tying, yaitu dengan memanfaatkan transpose dari matriks embedding sebagai bobot keluaran. Pendekatan ini tidak hanya menghemat jumlah parameter, tetapi juga menjaga konsistensi antara representasi input dan output.

**4. Penambahan Informasi Posisi (Positional Encoding)**

```python
# ------------------------ Positional Encoding --------------------------
class PositionalEncoding:
    def __init__(self, d_model, max_len=1024, method='sinusoidal'):
        """
        method: 'sinusoidal' or 'rope'
        For 'rope', this class will prepare sinusoidal frequencies used by RoPE.
        """
        self.d_model = d_model
        self.max_len = max_len
        self.method = method
        if method == 'sinusoidal':
            pe = np.zeros((max_len, d_model))
            position = np.arange(0, max_len)[:, np.newaxis]
            div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))
            pe[:, 0::2] = np.sin(position * div_term)
            pe[:, 1::2] = np.cos(position * div_term)
            self.pe = pe  # shape (max_len, d_model)
        elif method == 'rope':
            # Prepare RoPE frequency terms for rotary embeddings.
            # We'll compute angles for each even dimension as in many RoPE implementations.
            half = d_model // 2
            inv_freq = 1.0 / (10000 ** (np.arange(0, half) / float(half)))
            # store inv_freq to compute sin/cos for positions at runtime
            self.inv_freq = inv_freq  # shape (half,)
        else:
            raise ValueError("method must be 'sinusoidal' or 'rope'")

    def sinusoidal(self, seq_len):
        if seq_len > self.max_len:
            raise ValueError("seq_len exceeds max_len")
        return self.pe[:seq_len]  # (seq_len, d_model)

    def rope_sin_cos(self, seq_len, head_dim):
        """
        For RoPE, compute sin and cos arrays for positions.
        Return sin, cos shaped (seq_len, head_dim) where head_dim is half*2 maybe.
        We'll create sin/cos for the head_dim dimension.
        """
        # Ensure head_dim is even
        if head_dim % 2 != 0:
            raise ValueError("head_dim must be even for RoPE")
        half = head_dim // 2
        pos = np.arange(seq_len)
        angles = np.outer(pos, self.inv_freq[:half])  # (seq_len, half)
        sin = np.sin(angles)  # (seq_len, half)
        cos = np.cos(angles)
        # Interleave to match head_dim: [sin0, sin1, ...] and same shape for cos by repeating
        sin_big = np.concatenate([sin, sin], axis=-1)  # (seq_len, head_dim)
        cos_big = np.concatenate([cos, cos], axis=-1)
        return sin_big, cos_big
```

Karena Transformer tidak memiliki urutan waktu seperti pada model berbasis RNN, maka diperlukan cara untuk memberikan informasi posisi pada setiap token. Pada implementasi ini digunakan sinusoidal positional encoding, yang menambahkan pola sinus dan kosinus dengan frekuensi berbeda untuk setiap dimensi vektor berdasarkan urutan posisi token.

Pendekatan sinusoidal ini memungkinkan model mengenali posisi relatif antar token tanpa perlu melatih parameter tambahan. Selain metode sinusoidal, kode juga menyediakan alternatif Rotary Positional Encoding (RoPE), tetapi dalam pengujian ini digunakan metode sinusoidal sesuai konfigurasi bawaan.

**5. Mekanisme Scaled Dot-Product Attention dan Masking**

```python
# -------------------- Causal Mask (for attention) ----------------------
def causal_mask(seq_len):
    # mask shape [seq_len, seq_len], True where masked (future positions)
    return np.triu(np.ones((seq_len, seq_len), dtype=bool), k=1)

# ---------------- Scaled Dot-Product Attention ----------------------
def scaled_dot_product_attention(q, k, v, mask=None):
    """
    q,k,v: [..., seq_q, head_dim] and [..., seq_k, head_dim]
    mask: broadcastable boolean mask where True means masked (should not attend)
    """
    dk = q.shape[-1]
    scores = np.matmul(q, np.swapaxes(k, -1, -2)) / np.sqrt(dk)  # [..., seq_q, seq_k]
    if mask is not None:
        # mask: (seq_q, seq_k) -> expand to broadcasting shape of scores
        scores = np.where(mask, -1e9, scores)
    attn = softmax(scores, axis=-1)
    out = np.matmul(attn, v)
    return out, attn
```

Komponen inti dari Transformer adalah mekanisme perhatian atau attention mechanism. Pada bagian ini, hubungan antar token dihitung menggunakan operasi *dot product* antara vektor *query*, *key*, dan *value*. Nilai hasil perkalian dibagi dengan akar dari dimensi *key* untuk mencegah nilai skor menjadi terlalu besar, lalu hasilnya dinormalisasi dengan fungsi softmax sehingga menghasilkan bobot perhatian.

Program juga menggunakan causal mask yang berbentuk matriks segitiga atas untuk memastikan setiap token hanya dapat memperhatikan dirinya sendiri dan token-token sebelumnya. Mekanisme ini penting karena menjaga sifat *autoregressive*, di mana model hanya menggunakan informasi masa lalu untuk memprediksi token berikutnya.

**6. Multi-Head Attention**

```python
# ---------------------- Multi-Head Attention -------------------------
class MultiHeadAttention:
    def __init__(self, d_model, num_heads, rope=False, max_len=512, posenc_obj=None):
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
        self.d_model = d_model
        self.num_heads = num_heads
        self.head_dim = d_model // num_heads
        self.rope = rope
        self.max_len = max_len
        # projection weights
        k = 1 / np.sqrt(d_model)
        self.W_q = np.random.randn(d_model, d_model) * k
        self.W_k = np.random.randn(d_model, d_model) * k
        self.W_v = np.random.randn(d_model, d_model) * k
        self.W_o = np.random.randn(d_model, d_model) * k
        # posenc_obj is PositionalEncoding instance when using RoPE
        self.posenc_obj = posenc_obj

    def split_heads(self, x):
        # x: [batch, seq_len, d_model] -> [batch, num_heads, seq_len, head_dim]
        b, s, _ = x.shape
        x = x.reshape(b, s, self.num_heads, self.head_dim)
        return np.transpose(x, (0, 2, 1, 3))

    def combine_heads(self, x):
        # x: [batch, num_heads, seq_len, head_dim] -> [batch, seq_len, d_model]
        b, h, s, hd = x.shape
        x = np.transpose(x, (0, 2, 1, 3)).reshape(b, s, h * hd)
        return x

    def apply_rope_to_head(self, x_head, sin, cos):
        """
        x_head: (batch, heads, seq_len, head_dim)
        sin, cos: (seq_len, head_dim) broadcast to (1,1,seq_len,head_dim)
        RoPE rotation: split head_dim into (x1, x2) halves and rotate:
        [x1, x2] -> [x1 * cos - x2 * sin, x2 * cos + x1 * sin]
        """
```

```
    # ensure head_dim even
    hd = x_head.shape[-1]
    half = hd // 2
    x1 = x_head[..., :half]
    x2 = x_head[..., half:]
    sin_b = sin[np.newaxis, np.newaxis, :, :half]
    cos_b = cos[np.newaxis, np.newaxis, :, :half]
    # rotate
    x1_rot = x1 * cos_b - x2 * sin_b
    x2_rot = x2 * cos_b + x1 * sin_b
    return np.concatenate([x1_rot, x2_rot], axis=-1)

ef __call__(self, x, mask=None):
    # x: [batch, seq_len, d_model]
    b, seq_len, _ = x.shape
    Q = x @ self.W_q  # [b, seq, d_model]
    K = x @ self.W_k
    V = x @ self.W_v
    Qh = self.split_heads(Q)  # [b, h, seq, head_dim]
    Kh = self.split_heads(K)
    Vh = self.split_heads(V)

    # Apply RoPE if requested (rotary positional embedding on Q and K)
    if self.rope and self.posenc_obj is not None:
        # get sin/cos for seq_len and head_dim
        sin_big, cos_big = self.posenc_obj.rope_sin_cos(seq_len, self.head_dim)
        # apply to Qh and Kh
        Qh = self.apply_rope_to_head(Qh, sin_big, cos_big)
        Kh = self.apply_rope_to_head(Kh, sin_big, cos_big)

    # prepare mask: (seq_len, seq_len) -> broadcastable to (b, h, seq, seq)
    attn_mask = None
    if mask is not None:
        attn_mask = mask[np.newaxis, np.newaxis, :, :]

    out_h, attn = scaled_dot_product_attention(Qh, Kh, Vh, mask=attn_mask)  # out_h: [b,h,seq,head_dim]
    out = self.combine_heads(out_h)  # [b, seq, d_model]
    out = out @ self.W_o  # final linear
    return out, attn
```

Bagian ini memperluas mekanisme perhatian menjadi beberapa *head* agar model dapat menangkap berbagai konteks secara paralel. Kelas MultiHeadAttention memecah representasi vektor menjadi beberapa bagian (*head*), di mana masing-masing bagian memiliki proyeksi sendiri untuk *query*, *key*, dan *value*. Setelah proses perhatian dilakukan, hasil dari seluruh *head* digabungkan kembali menjadi satu representasi utuh sebelum diproyeksikan ke dimensi awal.

Jumlah kepala perhatian yang digunakan pada konfigurasi ini adalah empat. Hal ini berarti model melakukan empat proses perhatian terpisah dalam satu waktu untuk mempelajari berbagai jenis hubungan antar token secara bersamaan. Hasil akhir dari proses ini kemudian dikirim ke lapisan berikutnya melalui jalur *residual connection*.

**7. Feed-Forward Network dan Decoder Block**

```python
# ------------------------- Feed-Forward --------------------------------
class FeedForward:
    def __init__(self, d_model, d_ff):
        k1 = 1 / np.sqrt(d_model)
        k2 = 1 / np.sqrt(d_ff)
        self.W1 = np.random.randn(d_model, d_ff) * k1
        self.b1 = np.zeros((d_ff,))
        self.W2 = np.random.randn(d_ff, d_model) * k2
        self.b2 = np.zeros((d_model,))

    def __call__(self, x):
        # x: [b, seq, d_model]
        x1 = x @ self.W1 + self.b1
        a1 = gelu(x1)
        x2 = a1 @ self.W2 + self.b2
        return x2
```

```python
# ----------------------- Decoder Block (pre-norm) --------------------
class DecoderBlock:
    def __init__(self, d_model, num_heads, d_ff, rope=False, posenc_obj=None):
        self.ln1 = LayerNorm(d_model)
        self.mha = MultiHeadAttention(d_model, num_heads, rope=rope, posenc_obj=posenc_obj)
        self.ln2 = LayerNorm(d_model)
        self.ff = FeedForward(d_model, d_ff)

    def __call__(self, x, mask=None):
        x_norm = self.ln1(x)
        mha_out, attn = self.mha(x_norm, mask=mask)
        x = x + mha_out
        x_norm2 = self.ln2(x)
        ff_out = self.ff(x_norm2)
        x = x + ff_out
        return x, attn
```

Setiap posisi token selanjutnya diproses oleh jaringan Feed-Forward Network (FFN) dua lapis yang dilengkapi dengan aktivasi GELU. Lapisan ini bertugas untuk memperluas dimensi representasi sebelum dikembalikan lagi ke dimensi awal.

Kelas DecoderBlock menyatukan beberapa komponen inti seperti *LayerNorm*, *MultiHeadAttention*, dan *Feed-Forward Network*. Setiap blok bekerja dengan prinsip *pre-normalization*, yaitu melakukan normalisasi sebelum perhitungan dimulai. Setelah mendapatkan hasil perhatian, nilai tersebut dijumlahkan dengan input semula menggunakan *residual connection*. Proses ini kemudian diulangi setelah melewati jaringan feed-forward, sehingga menjaga kestabilan dan konsistensi nilai sepanjang lapisan.

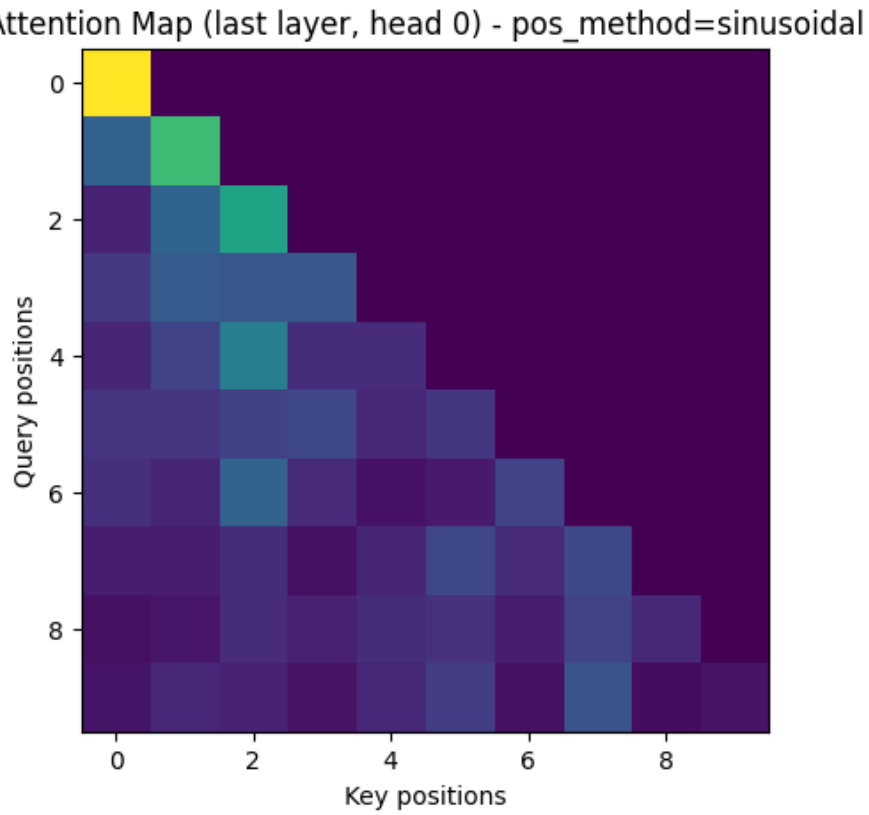
**8. Hasil Eksekusi Program**

```
Input token ids shape: (2, 10)
Logits shape: (2, 10, 1000)
Probabilities for last token shape: (2, 1000)
Sample logits (batch0, last pos, first 10): [-0.66862187 -0.01557043 -0.31070113  0.30578152  0.67496351 -1.51740577
 -2.00944891 -1.56804171  0.25936107  0.2532878 ]
Sum of prob distribution (batch0, last pos): 0.9999999999999999
Last-attention shape: (2, 4, 10, 10)
Top-5 next-token predictions (batch0, last pos): [819 877 974 194 391]
Attention row for position 2 (example): [0.094825 0.322204 0.582971 0.       0.       0.       0.       0.
 0.       0.       ]
```

Hasil eksekusi menunjukkan bahwa model Transformer Decoder bekerja dengan baik dan seluruh komponennya berfungsi sesuai dengan rancangan. Data masukan memiliki dimensi dua kali sepuluh, yang berarti terdapat dua batch data dengan sepuluh token pada setiap urutan. Setelah melewati proses embedding, penambahan posisi, serta beberapa lapisan decoder, model menghasilkan keluaran berupa logits dengan ukuran dua kali sepuluh kali seribu. Hasil ini menunjukkan bahwa untuk setiap token dalam urutan, model menghitung kemungkinan kemunculan dari seribu token yang ada di dalam kosakata.

Nilai hasil perhitungan softmax memiliki jumlah mendekati satu, yaitu 0.9999999999. Hal ini menandakan bahwa proses normalisasi berjalan dengan stabil dan akurat. Model juga menampilkan lima prediksi token berikutnya dengan kemungkinan tertinggi, yaitu token bernomor 819, 877, 974, 194, dan 391. Hasil ini menunjukkan bahwa model telah mampu memperkirakan token selanjutnya berdasarkan konteks urutan sebelumnya. Selain itu, bentuk peta perhatian terakhir memiliki ukuran dua kali empat kali sepuluh kali sepuluh, yang berarti terdapat dua batch data, empat kepala perhatian, dan sepuluh posisi token. Nilai perhatian terbesar terlihat pada token-token sebelumnya, yang menandakan bahwa mekanisme causal masking berjalan dengan benar.

**9. Visualisasi Attention Map**




Hasil visualisasi peta perhatian menunjukkan pola diagonal dari kiri atas ke kanan bawah. Pola ini menggambarkan bahwa setiap token memberikan perhatian pada dirinya sendiri dan pada token-token sebelumnya dalam urutan. Warna yang lebih terang pada gambar menandakan tingkat perhatian yang lebih tinggi, sementara warna yang lebih gelap menunjukkan perhatian yang lebih rendah.

Pola diagonal yang terbentuk membuktikan bahwa mekanisme masked self-attention berfungsi dengan baik, karena model tidak memberikan perhatian pada token di masa depan. Distribusi warna yang menurun secara bertahap juga menunjukkan bahwa model mampu menangkap konteks antar token dengan proporsional. Dengan demikian, hasil visualisasi ini membuktikan bahwa mekanisme perhatian berjalan optimal dan sesuai dengan karakteristik Transformer berbasis decoder seperti GPT.

**10. Kesimpulan**

Berdasarkan hasil eksekusi dan analisis keluaran, dapat disimpulkan bahwa seluruh komponen Transformer Decoder telah berfungsi dengan benar. Program berhasil membangun model Transformer secara menyeluruh menggunakan NumPy, menghasilkan distribusi probabilitas yang stabil, dan menampilkan peta perhatian yang sesuai dengan perilaku teoretis.

Proses embedding, penambahan posisi, multi-head attention, feed-forward, dan normalisasi berjalan sesuai urutan yang tepat. Visualisasi attention map juga membuktikan bahwa mekanisme

*causal masking* bekerja dengan baik. Implementasi ini tidak hanya menunjukkan keakuratan perhitungan matematis, tetapi juga menggambarkan secara jelas bagaimana model Transformer memproses informasi untuk memahami hubungan antar token dalam suatu urutan teks.