Nibroos Aurore Majiid Haryanto
22/494882/TK/54329

Architecture

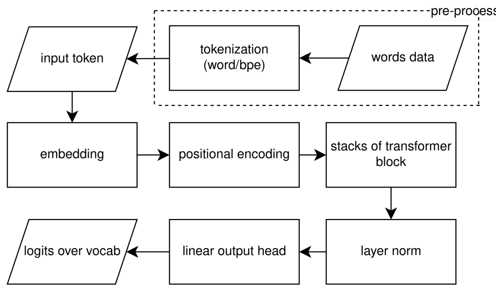

Figure 1. miniGPT Architecture

The miniGPT Architecture begins by converting raw text into numerical tokens using methods like Byte Pair Encoding (BPE), which are then mapped to continuous Token Embeddings. To account for word order, Positional Encoding is added to these embeddings. This prepared sequence then enters a stack of Transformer Blocks, the model's core, which uses multi-head self-attention to contextualize and refine the token representations across multiple layers. After the processing blocks, a final Layer Norm stabilizes the data before the Linear Output Head projects the features back into the size of the vocabulary. This final step generates Logits over vocab, representing the unnormalized scores for every possible next word. These logits are used to calculate loss during training or to sample the next token during text generation.

Require Components and Explanation

Token Embedding

```python
class Embedding(Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

        self.W = np.random.randn(vocab_size, embed_dim) * 0.02
        self.dW = None
        self.cache_input = None

    def forward(self, X):
        self.cache_input = X  # (B, T)
        out = self.W[X]  # (B, T, embed_dim)
        return out

    def backward(self, dZ):
        self.dW = np.zeros_like(self.W)
        X = self.cache_input
        np.add.at(self.dW, X.flatten(), dZ.reshape(-1, self.embed_dim))

    def params(self):
        return {"W": self.W}

    def grads(self):
        return {"W": self.dW}
```

Positional Encoding

```python
class PositionalEncoding(Module):
    def __init__(self, max_len, d_model):
        super().__init__()
        self.max_len = max_len
        self.d_model = d_model

        self.W = np.random.randn(max_len, d_model) * 0.02
        self.dW = None
        self.cache_input = None

    def forward(self, X):
        self.cache_input = X
        B, T, C = X.shape

        pos_emb = self.W[:T, :]  # (T, C)
        out = X + pos_emb  # (B, T, C)
        return out

    def backward(self, dZ):
        B, T, C = dZ.shape
        self.dW = np.zeros_like(self.W)

        # out = X + W[:T, :], so ∂out/∂W[t] = 1 for every batch element at position t
        # ∂L/∂W[t] = sum over all batch gradients at position t
        self.dW[:T, :] = np.sum(dZ, axis=0)
        return dZ

    def params(self):
        return {"W": self.W}

    def grads(self):
        return {"W": self.dW}
```

Scaled Dot-Product Attention dengan softmax

```python
# scaled dot product
scores = Q @ K.swapaxes(-2, -1) / np.sqrt(self.d_k)  # (B, nh, T, T)

if mask is not None:
    scores = scores + mask  # (B, nh, T, T)

original_shape = scores.shape
scores_reshaped = scores.reshape(-1, scores.shape[-1])  # (B*nh*T, T)
# scaled dot product with softmax
attn_weights_reshaped = self.softmax(scores_reshaped)  # (B*nh*T, T)
attn_weights = attn_weights_reshaped.reshape(original_shape)  # (B, nh, T, T)
```

Causal Masking

```python
def _create_causal_mask(self, T):
    mask = np.triu(np.ones((T, T)), k=1) * -1e9  #
    return mask[None, None, :, :]
```

Feed-Forward Network (FFN)

Residual Connection + Layer Normalization

```python
class FeedForward(Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.linear1 = Linear(d_model, d_ff)
        self.relu = ReLU()
        self.linear2 = Linear(d_ff, d_model)

    def forward(self, X):
        X = self.linear1(X)  # (B, T, d_ff)
        X = self.relu(X)  # (B, T, d_ff)
        X = self.linear2(X)  # (B, T, d_model)
        return X

    def backward(self, dZ):
        dZ = self.linear2.backward(dZ)
        dZ = self.relu.backward(dZ)
        dZ = self.linear1.backward(dZ)
        return dZ

    def params(self):
        params = {}
        params.update({f"linear1.{k}": v for k, v in self.linear1.params().items()})
        params.update({f"linear2.{k}": v for k, v in self.linear2.params().items()})
        return params

    def grads(self):
        grads = {}
        grads.update({f"linear1.{k}": v for k, v in self.linear1.grads().items()})
        grads.update({f"linear2.{k}": v for k, v in self.linear2.grads().items()})
        return grads
```

```python
class TransformerBlock(Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, n_heads)
        self.ln1 = LayerNorm(d_model)
        self.ffn = FeedForward(d_model, d_ff)
        self.ln2 = LayerNorm(d_model)

    def forward(self, X, mask=None):
        ln1_out = self.ln1(X)  #Pre-Normalization
        attn_out = self.attn(ln1_out, mask)
        X = X + attn_out  #Residual Connection

        ln2_out = self.ln2(X)  #Pre-Normalization
        ffn_out = self.ffn(ln2_out)
        X = X + ffn_out  #Residual Connection
```

Multi-Head Attention

```python
class MultiHeadAttention(Module):
    """
    Multi-Head Attention (Attention is All You Need <3): https://arxiv.org/abs/1706.03762
    """

    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        self.W_q = Linear(d_model, d_model)
        self.W_k = Linear(d_model, d_model)
        self.W_v = Linear(d_model, d_model)
        self.W_o = Linear(d_model, d_model)
        self.softmax = Softmax()

        self.cache = {}

    def forward(self, X, mask=None):
        B, T, C = X.shape

        Q = self.W_q(X)  # (B, T, C)
        K = self.W_k(X)  # (B, T, C)
        V = self.W_v(X)  # (B, T, C)

        Q = Q.reshape(B, T, self.n_heads, self.d_k).swapaxes(1, 2)  # (B, nh, T, d_k)
        K = K.reshape(B, T, self.n_heads, self.d_k).swapaxes(1, 2)  # (B, nh, T, d_k)
        V = V.reshape(B, T, self.n_heads, self.d_k).swapaxes(1, 2)  # (B, nh, T, d_k)
```

Output Layer

```python
class miniGPT(Module):

    def __init__(self, vocab_size, max_len, d_model, n_heads, n_layers, d_ff):
        super().__init__()
        self.vocab_size = vocab_size
        self.max_len = max_len
        self.d_model = d_model

        self.tok_emb = Embedding(vocab_size, d_model)
        self.pos_emb = PositionalEncoding(max_len, d_model)
        self.blocks = [TransformerBlock(d_model, n_heads, d_ff) for _ in range(n_layers)]
        self.ln_f = LayerNorm(d_model)
        self.lm_head = Linear(d_model, vocab_size) #linear layer that as the language modeling 

        self.cache = {}

    def forward(self, X, targets=None):
        B, T = X.shape

        tok_emb = self.tok_emb(X)  # (B, T, C)
        X = self.pos_emb(tok_emb)  # (B, T, C)

        mask = self._create_causal_mask(T)  # (1, 1, T, T)

        for block in self.blocks:
            X = block(X, mask)  # (B, T, C)

        X = self.ln_f(X)  # (B, T, C)
        logits = self.lm_head(X)  # logits over vocab

        if targets is not None:
            loss = cross_entropy_loss(logits.reshape(-1, self.vocab_size), targets.reshape(-1))
            self.cache = {'logits': logits, 'targets': targets}
            return logits, loss

        return logits
```

Testing

```
Running simple verification tests...

--- Testing Model Output Dimensions ---
Test Passed: Output shape is correct (2, 8, 50).

--- Testing Softmax Properties ---
Test Passed: Probabilities sum to 1.
Test Passed: All probability values are within the [0, 1] range.

--- Testing Causal Masking ---
Test Passed: Causal mask correctly prevents attention to future tokens.

All simple tests completed.
```

Additional & Misc

Given the restrictions of this report to be at max 2 pages, all others explanation and elaboration is being further explained in the readme file of this repository: https://github.com/potreic/miniGPT.