```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import math
import copy
```

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        # Ensure that the model dimension (d_model) is divisible by the number of h
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        # Initialize dimensions
        self.d_model = d_model # Model's dimension
        self.num_heads = num_heads # Number of attention heads
        self.d_k = d_model // num_heads # Dimension of each head's key, query, and

        # Linear layers for transforming inputs
        self.W_q = nn.Linear(d_model, d_model) # Query transformation
        self.W_k = nn.Linear(d_model, d_model) # Key transformation
        self.W_v = nn.Linear(d_model, d_model) # Value transformation
        self.W_o = nn.Linear(d_model, d_model) # Output transformation

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # Calculate attention scores
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # Apply mask if provided (useful for preventing attention to certain parts
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)

        # Softmax is applied to obtain attention probabilities
        attn_probs = torch.softmax(attn_scores, dim=-1)

        # Multiply by values to obtain the final output
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        # Reshape the input to have num_heads for multi-head attention
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1

    def combine_heads(self, x):
        # Combine the multiple heads back to original shape
```

```
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_m

    def forward(self, Q, K, V, mask=None):
        # Apply linear transformations and split heads
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        # Perform scaled dot-product attention
        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)

        # Combine heads and apply output transformation
        output = self.W_o(self.combine_heads(attn_output))
        return output
```

Esta clase define la capa de atención multi-cabeza utilizada en el modelo Transformer.

init: En el constructor, se inicializan las dimensiones del modelo y se definen las capas lineales para transformar las entradas (Q, K, V). scaled_dot_product_attention: Calcula la atención ponderada utilizando el producto punto escalado. Aplica una máscara opcional para ignorar ciertas partes de la entrada. split_heads y combine_heads: Estas funciones ayudan a dividir y combinar las cabezas de atención múltiple. forward: Realiza la atención multi-cabeza, calcula la salida y la transforma.

```
class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
```

Esta clase define una capa de red neuronal de avance (feed-forward) utilizada en el modelo Transformer.

init: En el constructor, se definen dos capas lineales y una función de activación ReLU. forward: Realiza la transformación feed-forward de la entrada.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(math.log(10000

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

Esta clase define la codificación posicional que se agrega a las entradas del modelo.

init: En el constructor, se calcula la codificación posicional utilizando funciones sinusoidales y se registra como un tensor constante. forward: Agrega la codificación posicional a las entradas.

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

Esta clase define una capa del codificador en el modelo Transformer.

init: En el constructor, se definen las capas de atención propia y de avance, así como capas de normalización y dropout. forward: Realiza la operación de atención propia y avance en la entrada.

```python
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))
        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
        x = self.norm2(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x
```

Esta clase define una capa del decodificador en el modelo Transformer.

init: En el constructor, se definen las capas de atención propia, atención cruzada y de avance, junto con capas de normalización y dropout. forward: Realiza las operaciones de atención propia, atención cruzada y avance en la entrada.

```
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads, num_laye
        super(Transformer, self).__init__()
        self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)
        self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)

        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads, d_ff,
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff,

        self.fc = nn.Linear(d_model, tgt_vocab_size)
        self.dropout = nn.Dropout(dropout)

    def generate_mask(self, src, tgt):
        src_mask = (src != 0).unsqueeze(1).unsqueeze(2)
        tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)
        seq_length = tgt.size(1)
        nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length), diagon
        tgt_mask = tgt_mask & nopeak_mask
        return src_mask, tgt_mask

    def forward(self, src, tgt):
        src_mask, tgt_mask = self.generate_mask(src, tgt)
        src_embedded = self.dropout(self.positional_encoding(self.encoder_embedding
        tgt_embedded = self.dropout(self.positional_encoding(self.decoder_embedding

        enc_output = src_embedded
        for enc_layer in self.encoder_layers:
            enc_output = enc_layer(enc_output, src_mask)

        dec_output = tgt_embedded
        for dec_layer in self.decoder_layers:
            dec_output = dec_layer(dec_output, enc_output, src_mask, tgt_mask)

        output = self.fc(dec_output)
        return output
```

Esta clase define el modelo Transformer completo.

init: En el constructor, se definen las capas de embedding, codificación posicional, capas del codificador y decodificador, capa de salida y dropout. generate_mask: Genera máscaras para las secuencias de entrada y salida. forward: Realiza la propagación hacia adelante en el modelo, aplicando las capas del codificador y decodificador.

```
src_vocab_size = 5000
tgt_vocab_size = 5000
d_model = 512
num_heads = 8
num_layers = 6
d_ff = 2048
max_seq_length = 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_l

# Generate random sample data
src_data = torch.randint(1, src_vocab_size, (64, max_seq_length))  # (batch_size, s
tgt_data = torch.randint(1, tgt_vocab_size, (64, max_seq_length))  # (batch_size, s
```

En este bloque, se configuran los hiperparámetros del modelo Transformer, como el tamaño del vocabulario, la dimensión del modelo, el número de cabezas de atención, el número de capas, el tamaño máximo de secuencia y la probabilidad de dropout.

```
transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_l
```

En este bloque, se crea una instancia del modelo Transformer y se generan datos de muestra para entrenar y validar el modelo.

```
criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=

transformer.train()

for epoch in range(100):
    optimizer.zero_grad()
    output = transformer(src_data, tgt_data[:, :-1])
    loss = criterion(output.contiguous().view(-1, tgt_vocab_size), tgt_data[:, 1:].
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

```
Epoch: 42, Loss: 5.646734237670898
Epoch: 43, Loss: 5.588915824890137
Epoch: 44, Loss: 5.5259108543396
Epoch: 45, Loss: 5.471502304077148
Epoch: 46, Loss: 5.407161712646484
Epoch: 47, Loss: 5.342100143432617
Epoch: 48, Loss: 5.284428596496582
Epoch: 49, Loss: 5.231245994567871
```

```
Epoch: 50, Loss: 5.1717071533203125
Epoch: 51, Loss: 5.117407321929932
Epoch: 52, Loss: 5.058736324310303
Epoch: 53, Loss: 5.0040059089660645
Epoch: 54, Loss: 4.951104164123535
Epoch: 55, Loss: 4.89433479309082
Epoch: 56, Loss: 4.837600231170654
Epoch: 57, Loss: 4.790550231933594
Epoch: 58, Loss: 4.743412971496582
Epoch: 59, Loss: 4.685122013092041
Epoch: 60, Loss: 4.632752418518066
Epoch: 61, Loss: 4.582803249359131
Epoch: 62, Loss: 4.529180526733398
Epoch: 63, Loss: 4.472339153289795
Epoch: 64, Loss: 4.427853584289551
Epoch: 65, Loss: 4.371998310089111
Epoch: 66, Loss: 4.325108528137207
Epoch: 67, Loss: 4.27531623840332
Epoch: 68, Loss: 4.217070579528809
Epoch: 69, Loss: 4.170065402984619
Epoch: 70, Loss: 4.125426292419434
Epoch: 71, Loss: 4.071837902069092
Epoch: 72, Loss: 4.028285026550293
Epoch: 73, Loss: 3.9762043952941895
Epoch: 74, Loss: 3.9359443187713623
Epoch: 75, Loss: 3.871272563934326
Epoch: 76, Loss: 3.829887628555298
Epoch: 77, Loss: 3.7809715270996094
Epoch: 78, Loss: 3.7313687801361084
Epoch: 79, Loss: 3.6914589405059814
Epoch: 80, Loss: 3.643827199935913
Epoch: 81, Loss: 3.5913925170898438
Epoch: 82, Loss: 3.5483932495117188
Epoch: 83, Loss: 3.50618314743042
Epoch: 84, Loss: 3.451910972595215
Epoch: 85, Loss: 3.4077670574188232
Epoch: 86, Loss: 3.362776279449463
Epoch: 87, Loss: 3.3262393474578857
Epoch: 88, Loss: 3.281801700592041
Epoch: 89, Loss: 3.227851390838623
Epoch: 90, Loss: 3.187415838241577
Epoch: 91, Loss: 3.1422038078308105
Epoch: 92, Loss: 3.0958049297332764
Epoch: 93, Loss: 3.05653977394104
Epoch: 94, Loss: 3.01223087310791
Epoch: 95, Loss: 2.9734625816345215
Epoch: 96, Loss: 2.927741765975952
Epoch: 97, Loss: 2.887327194213867
Epoch: 98, Loss: 2.836761713027954
Epoch: 99, Loss: 2.796391010284424
Epoch: 100, Loss: 2.7539892196655273
```

Se define una función de pérdida (entropía cruzada categórica) y un optimizador (Adam) para entrenar el modelo. Se realiza un bucle de entrenamiento durante 100 épocas. En cada época, se calcula la pérdida y se actualizan los pesos del modelo.

```
transformer.eval()

# Generate random sample validation data
val_src_data = torch.randint(1, src_vocab_size, (64, max_seq_length))  # (batch_siz
val_tgt_data = torch.randint(1, tgt_vocab_size, (64, max_seq_length))  # (batch_siz

with torch.no_grad():

    val_output = transformer(val_src_data, val_tgt_data[:, :-1])
    val_loss = criterion(val_output.contiguous().view(-1, tgt_vocab_size), val_tgt_
    print(f"Validation Loss: {val_loss.item()}")
```

```
Validation Loss: 8.809242248535156
```

Se cambia el modelo al modo de evaluación y se calcula la pérdida en un conjunto de datos de validación.

En resumen, este código implementa un modelo Transformer y lo entrena en datos de muestra utilizando PyTorch. El modelo incluye capas de atención multi-cabeza, codificación posicional, capas de encoder y decoder, y se entrena para la tarea de predicción de secuencias.

COMBO: 1

[redacted]