

Encoder and Decoder

Here's the implementation of the encoder layer and decoder layer for a transformer model:

```
1 class EncoderLayer(BaseLayer):
2     def __init__(self, d_model, num_heads, d_ff):
3         """
4         Initialize the encoder layer.
5
6         Args:
7             d_model: Dimension of the model.
8             num_heads: Number of attention heads.
9             d_ff: Dimension of the feed-forward network.
10        """
11        self.self_attn = MultiHeadAttention(d_model, num_heads)
12        self.norm1 = LayerNorm()
13        self.ffn = FeedForward(d_model, d_ff)
14        self.norm2 = LayerNorm()
15        self.residual1 = ResidualConnection()
16        self.residual2 = ResidualConnection()
17
18    def forward(self, inputs, mask=None):
19        """
20        Forward pass of the encoder layer.
21
22        Args:
23            inputs: Input tensor of shape (batch_size, seq_length, d_model)
24            mask: Optional mask tensor of shape (batch_size, 1, seq_length, seq_length)
25
26        Returns:
27            Output tensor of shape (batch_size, seq_length, d_model)
28        """
29        # Self-attention
30        attn_output = self.self_attn.forward(inputs, inputs, inputs, mask)
31        # Residual connection and layer normalization
32        residual_output1 = self.residual1.forward(inputs, attn_output)
33        norm_output1 = self.norm1.forward(residual_output1)
34
35        # Feed-forward network
36        ffn_output = self.ffn.forward(norm_output1)
37        # Residual connection and layer normalization
38        residual_output2 = self.residual2.forward(norm_output1, ffn_output)
39        norm_output2 = self.norm2.forward(residual_output2)
40
41        return norm_output2
42
43    def backward(self, upstream_grad):
44        """
45        Backward pass of the encoder layer.
46
47        Args:
48            upstream_grad: Gradient tensor of shape (batch_size, seq_length, d_model)
49
50        Returns:
```

```

51         Gradient tensor of shape (batch_size, seq_length, d_model)
52     """
53     # Backward through layer normalization and residual connection
54     grad_norm2 = self.norm2.backward(upstream_grad)
55     grad_residual2, grad_ffn = self.residual2.backward(grad_norm2)
56
57     # Backward through feed-forward network
58     grad_ffn = self.ffn.backward(grad_ffn)
59
60     # Backward through layer normalization and residual connection
61     grad_norm1 = self.norm1.backward(grad_residual2 + grad_ffn)
62     grad_residual1, grad_attn = self.residual1.backward(grad_norm1)
63
64     # Backward through self-attention
65     grad_attn = self.self_attn.backward(grad_attn + grad_residual1)
66
67     return grad_attn
68
69     def update_parameters(self, learning_rate):
70         """
71         Update parameters of the encoder layer.
72
73         Args:
74             learning_rate: Learning rate for parameter updates.
75         """
76         self.self_attn.update_parameters(learning_rate)
77         self.ffn.update_parameters(learning_rate)
78
79
80     class DecoderLayer(BaseLayer):
81         def __init__(self, d_model, num_heads, d_ff):
82             """
83             Initialize the decoder layer.
84
85             Args:
86                 d_model: Dimension of the model.
87                 num_heads: Number of attention heads.
88                 d_ff: Dimension of the feed-forward network.
89             """
90             self.masked_attn = MultiHeadAttention(d_model, num_heads)
91             self.norm1 = LayerNorm()
92             self.cross_attn = MultiHeadAttention(d_model, num_heads)
93             self.norm2 = LayerNorm()
94             self.ffn = FeedForward(d_model, d_ff)
95             self.norm3 = LayerNorm()
96             self.residual1 = ResidualConnection()
97             self.residual2 = ResidualConnection()
98             self.residual3 = ResidualConnection()
99
100         def forward(self, inputs, encoder_output, src_mask=None, tgt_mask=None):
101             """
102             Forward pass of the decoder layer.
103
104             Args:
105                 inputs: Input tensor of shape (batch_size, seq_length, d_model)
106                 encoder_output: Output tensor from the encoder of shape (batch_size,
seq_length, d_model)
107                 src_mask: Optional source mask tensor of shape (batch_size, 1, 1, seq_length)

```

```

108         tgt_mask: Optional target mask tensor of shape (batch_size, 1, seq_length,
seq_length)
109
110     Returns:
111         Output tensor of shape (batch_size, seq_length, d_model)
112     """
113     # Masked self-attention
114     attn_output = self.masked_attn.forward(inputs, inputs, inputs, tgt_mask)
115     # Residual connection and layer normalization
116     residual_output1 = self.residual1.forward(inputs, attn_output)
117     norm_output1 = self.norm1.forward(residual_output1)
118
119     # Cross-attention
120     cross_attn_output = self.cross_attn.forward(norm_output1, encoder_output,
encoder_output, src_mask)
121     # Residual connection and layer normalization
122     residual_output2 = self.residual2.forward(norm_output1, cross_attn_output)
123     norm_output2 = self.norm2.forward(residual_output2)
124
125     # Feed-forward network
126     ffn_output = self.ffn.forward(norm_output2)
127     # Residual connection and layer normalization
128     residual_output3 = self.residual3.forward(norm_output2, ffn_output)
129     norm_output3 = self.norm3.forward(residual_output3)
130
131     return norm_output3
132
133 def backward(self, upstream_grad):
134     """
135     Backward pass of the decoder layer.
136
137     Args:
138         upstream_grad: Gradient tensor of shape (batch_size, seq_length, d_model)
139
140     Returns:
141         Gradient tensor of shape (batch_size, seq_length, d_model)
142     """
143     # Backward through layer normalization and residual connection
144     grad_norm3 = self.norm3.backward(upstream_grad)
145     grad_residual3, grad_ffn = self.residual3.backward(grad_norm3)
146
147     # Backward through feed-forward network
148     grad_ffn = self.ffn.backward(grad_ffn)
149
150     # Backward through layer normalization and residual connection
151     grad_norm2 = self.norm2.backward(grad_residual3 + grad_ffn)
152     grad_residual2, grad_cross_attn = self.residual2.backward(grad_norm2)
153
154     # Backward through cross-attention
155     grad_cross_attn = self.cross_attn.backward(grad_cross_attn + grad_residual2)
156
157     # Backward through layer normalization and residual connection
158     grad_norm1 = self.norm1.backward(grad_cross_attn)
159     grad_residual1, grad_masked_attn = self.residual1.backward(grad_norm1)
160
161     # Backward through masked self-attention
162     grad_masked_attn = self.masked_attn.backward(grad_masked_attn + grad_residual1)
163

```

```

164         return grad_masked_attn
165
166     def update_parameters(self, learning_rate):
167         """
168         Update parameters of the decoder layer.
169
170         Args:
171             learning_rate: Learning rate for parameter updates.
172         """
173         self.masked_attn.update_parameters(learning_rate)
174         self.cross_attn.update_parameters(learning_rate)
175         self.ffn.update_parameters(learning_rate)

```

Explanation:

1. Encoder Layer:

- **Self-Attention:** Applies multi-head self-attention to the input.
- **Residual Connection and Layer Normalization:** Adds the input to the output of the self-attention and applies layer normalization.
- **Feed-Forward Network:** Passes the normalized output through a feed-forward network.
- **Residual Connection and Layer Normalization:** Adds the input to the output of the feed-forward network and applies layer normalization.

2. Decoder Layer:

- **Masked Self-Attention:** Applies multi-head self-attention with a mask to prevent attending to future positions.
- **Residual Connection and Layer Normalization:** Adds the input to the output of the masked self-attention and applies layer normalization.
- **Cross-Attention:** Applies multi-head attention using the encoder output as keys and values.
- **Residual Connection and Layer Normalization:** Adds the input to the output of the cross-attention and applies layer normalization.
- **Feed-Forward Network:** Passes the normalized output through a feed-forward network.
- **Residual Connection and Layer Normalization:** Adds the input to the output of the feed-forward network and applies layer normalization.

Usage Example:

```

1  # Hyperparameters
2  d_model = 512
3  num_heads = 8
4  d_ff = 2048
5  batch_size = 32
6  seq_len = 50
7
8  # Create encoder and decoder layers
9  encoder_layer = EncoderLayer(d_model, num_heads, d_ff)
10 decoder_layer = DecoderLayer(d_model, num_heads, d_ff)

```

```

11 |
12 | # Example inputs
13 | encoder_input = np.random.randn(batch_size, seq_len, d_model)
14 | decoder_input = np.random.randn(batch_size, seq_len, d_model)
15 | encoder_output = np.random.randn(batch_size, seq_len, d_model)
16 |
17 | # Create masks
18 | src_mask = np.random.randint(0, 2, (batch_size, 1, 1, seq_len))
19 | tgt_mask = np.random.randint(0, 2, (batch_size, 1, seq_len, seq_len))
20 |
21 | # Forward pass for encoder
22 | encoder_output = encoder_layer.forward(encoder_input, src_mask)
23 |
24 | # Forward pass for decoder
25 | decoder_output = decoder_layer.forward(decoder_input, encoder_output, src_mask, tgt_mask)
26 |
27 | # Simulate upstream gradients
28 | encoder_upstream_grad = np.random.randn(batch_size, seq_len, d_model)
29 | decoder_upstream_grad = np.random.randn(batch_size, seq_len, d_model)
30 |
31 | # Backward pass for encoder
32 | encoder_grad = encoder_layer.backward(encoder_upstream_grad)
33 |
34 | # Backward pass for decoder
35 | decoder_grad = decoder_layer.backward(decoder_upstream_grad)
36 |
37 | # Update parameters
38 | learning_rate = 0.001
39 | encoder_layer.update_parameters(learning_rate)
40 | decoder_layer.update_parameters(learning_rate)

```

Transformer Training

Training a transformer model typically involves the following parameters and practices:

Number of Epochs

- **Typical Range:** 10 to 100 epochs.
- **Considerations:** The number of epochs depends on the dataset size and complexity. Smaller datasets might require more epochs to achieve good performance, while larger datasets might converge in fewer epochs. It's common to use early stopping based on validation performance to prevent overfitting.

Batch Size

- **Typical Range:** 32 to 1024.
- **Considerations:** Larger batch sizes can lead to more stable training but require more memory. Common choices are 32, 64, 128, or 256. For very large datasets, effective batch sizes can be increased using gradient accumulation.

Optimizer

- **Adam Optimizer:** Yes, the Adam optimizer is commonly used for training transformer models due to its adaptive learning rate properties, which work well with the transformer architecture.
- **Typical Parameters:**
 - Learning rate: 1e-4 to 3e-5
 - Beta1: 0.9
 - Beta2: 0.999
 - Epsilon: 1e-8

Learning Rate Schedule

- **Warmup and Decay:** It's common to use a learning rate schedule with warmup steps followed by a decay. The learning rate is increased linearly during warmup and then decreased proportionally to the inverse square root of the number of steps.

Additional Training Considerations

- **Weight Initialization:** Use techniques like Xavier or He initialization.
- **Gradient Clipping:** Helps prevent exploding gradients, typically with a value of 1.0.
- **Mixed Precision Training:** Can speed up training and reduce memory usage.
- **Regularization:** Techniques like dropout and label smoothing are commonly used.

Example Training Setup

```
1  # Hyperparameters
2  num_epochs = 30
3  batch_size = 128
4  learning_rate = 1e-4
5  warmup_steps = 4000
6
7  # Optimizer
8  optimizer = AdamOptimizer(model.parameters(), lr=learning_rate, beta1=0.9, beta2=0.999,
                             epsilon=1e-8)
```

```

9
10 # Learning rate scheduler
11 scheduler = get_linear_schedule_with_warmup(
12     optimizer,
13     num_warmup_steps=warmup_steps,
14     num_training_steps=num_epochs * num_batches
15 )
16
17 # Training loop
18 for epoch in range(num_epochs):
19     model.train()
20     for batch in train_loader:
21         optimizer.zero_grad()
22         inputs, targets = batch
23         outputs = model(inputs)
24         loss = criterion(outputs, targets)
25         loss.backward()
26         optimizer.step()
27         scheduler.step()
28
29     # Validation
30     model.eval()
31     val_loss = 0.0
32     with torch.no_grad():
33         for batch in val_loader:
34             inputs, targets = batch
35             outputs = model(inputs)
36             loss = criterion(outputs, targets)
37             val_loss += loss.item()
38
39     print(f"Epoch {epoch+1}, Val Loss: {val_loss/len(val_loader)}")

```

These parameters and practices provide a good starting point for training transformer models, but they may need to be adjusted based on the specific dataset and task.

Here's an implementation of the Transformer NMT model with the Adam optimizer and training loop:

```

1 class AdamOptimizer:
2     def __init__(self, params, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
3         self.params = params
4         self.lr = lr
5         self.beta1 = beta1
6         self.beta2 = beta2
7         self.epsilon = epsilon
8         self.m = None
9         self.v = None
10        self.t = 0
11
12        def update(self):
13            if self.m is None:
14                self.m = [np.zeros_like(param) for param in self.params]
15                self.v = [np.zeros_like(param) for param in self.params]
16
17            self.t += 1
18            for i, param in enumerate(self.params):

```

```

19         if param.dw is not None:
20             self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * param.dw
21             self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (param.dw ** 2)
22
23             m_hat = self.m[i] / (1 - self.beta1 ** self.t)
24             v_hat = self.v[i] / (1 - self.beta2 ** self.t)
25
26             param.W -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)
27             param.dw = None
28
29         if param.db is not None:
30             self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * param.db
31             self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (param.db ** 2)
32
33             m_hat = self.m[i] / (1 - self.beta1 ** self.t)
34             v_hat = self.v[i] / (1 - self.beta2 ** self.t)
35
36             param.b -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)
37             param.db = None
38
39
40 def train_model(model, train_loader, val_loader, num_epochs, optimizer, criterion):
41     for epoch in range(num_epochs):
42         model.train()
43         total_loss = 0.0
44         for batch in train_loader:
45             src_seq, tgt_seq = batch
46
47             optimizer.zero_grad()
48             outputs = model.forward(src_seq, tgt_seq[:, :-1])
49             loss = criterion(outputs, tgt_seq[:, 1:])
50             loss.backward()
51             optimizer.step()
52
53             total_loss += loss.item()
54
55         avg_train_loss = total_loss / len(train_loader)
56
57         model.eval()
58         val_loss = 0.0
59         with torch.no_grad():
60             for batch in val_loader:
61                 src_seq, tgt_seq = batch
62                 outputs = model.forward(src_seq, tgt_seq[:, :-1])
63                 loss = criterion(outputs, tgt_seq[:, 1:])
64                 val_loss += loss.item()
65
66         avg_val_loss = val_loss / len(val_loader)
67
68         print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss}, Val Loss: {avg_val_loss}")
69
70
71 # Example usage:
72 if __name__ == "__main__":
73     # Hyperparameters
74     src_vocab_size = 10000
75     tgt_vocab_size = 10000
76     d_model = 512

```



```

77     num_heads = 8
78     d_ff = 2048
79     num_layers = 6
80     max_seq_length = 100
81     batch_size = 32
82     num_epochs = 30
83     learning_rate = 0.0001
84
85     # Create model
86     model = TransformerNMT(
87         src_vocab_size,
88         tgt_vocab_size,
89         d_model,
90         num_heads,
91         d_ff,
92         num_layers,
93         max_seq_length
94     )
95
96     # Create optimizer
97     optimizer = AdamOptimizer(
98         [param for param in model.__dict__.values() if isinstance(param, BaseLayer)],
99         lr=learning_rate
100     )
101
102     # Create loss function
103     criterion = lambda outputs, targets: np.mean(
104         np.cross_entropy(outputs.reshape(-1, tgt_vocab_size), targets.reshape(-1))
105     )
106
107     # Create dummy data loaders
108     train_loader = [
109         (
110             np.random.randint(0, src_vocab_size, size=(batch_size, max_seq_length)),
111             np.random.randint(0, tgt_vocab_size, size=(batch_size, max_seq_length))
112         )
113         for _ in range(100)
114     ]
115     val_loader = [
116         (
117             np.random.randint(0, src_vocab_size, size=(batch_size, max_seq_length)),
118             np.random.randint(0, tgt_vocab_size, size=(batch_size, max_seq_length))
119         )
120         for _ in range(20)
121     ]
122
123     # Train the model
124     train_model(model, train_loader, val_loader, num_epochs, optimizer, criterion)

```

Explanation:

1. Adam Optimizer:

- Implements the Adam optimization algorithm with adaptive learning rates.
- Maintains first and second moment estimates of the gradients.
- Corrects bias in the moment estimates.

2. Training Loop:

- Iterates over epochs and batches.
- Computes the model's output and loss for each batch.
- Performs backpropagation and parameter updates using the Adam optimizer.
- Evaluates the model on the validation set after each epoch.

3. Loss Function:

- Uses cross-entropy loss to measure the difference between predicted and actual target sequences.

4. Data Loaders:

- Dummy data loaders are created for demonstration purposes.
- In practice, you would replace these with actual data loaders for your specific task.

This implementation provides a basic training setup for the Transformer NMT model using the Adam optimizer. You can adjust the hyperparameters and data loaders according to your specific requirements.

Here's the implementation of the TransformerNMT class with train() and eval() methods:

```
1 class TransformerNMT(BaseLayer):
2     def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads, d_ff,
3         num_layers, max_seq_length):
4         """
5         Initialize the Transformer NMT model.
6
7         Args:
8             src_vocab_size: Size of the source vocabulary.
9             tgt_vocab_size: Size of the target vocabulary.
10            d_model: Dimension of the model.
11            num_heads: Number of attention heads.
12            d_ff: Dimension of the feed-forward network.
13            num_layers: Number of encoder and decoder layers.
14            max_seq_length: Maximum sequence length.
15        """
16        self.src_embedding = EmbeddingLayer(src_vocab_size, d_model)
17        self.tgt_embedding = EmbeddingLayer(tgt_vocab_size, d_model)
18        self.positional_encoding = PositionalEmbeddingLayer(max_seq_length, d_model)
19
20        self.encoder_layers = [EncoderLayer(d_model, num_heads, d_ff) for _ in
21            range(num_layers)]
22        self.decoder_layers = [DecoderLayer(d_model, num_heads, d_ff) for _ in
23            range(num_layers)]
24
25        self.fc = LinearLayer(d_model, tgt_vocab_size)
26        self.training = True # Flag to indicate training mode
```

```

25     def forward(self, src_seq, tgt_seq, src_mask=None, tgt_mask=None):
26         """
27         Forward pass of the Transformer NMT model.
28
29         Args:
30             src_seq: Source sequence tensor of shape (batch_size, src_seq_length)
31             tgt_seq: Target sequence tensor of shape (batch_size, tgt_seq_length)
32             src_mask: Optional source mask tensor of shape (batch_size, 1, 1,
src_seq_length)
33             tgt_mask: Optional target mask tensor of shape (batch_size, 1, tgt_seq_length,
tgt_seq_length)
34
35         Returns:
36             Output tensor of shape (batch_size, tgt_seq_length, tgt_vocab_size)
37         """
38         # Embedding and positional encoding for source sequence
39         src_emb = self.src_embedding.forward(src_seq)
40         src_pos_emb = self.positional_encoding.forward(src_emb)
41
42         # Encoder
43         encoder_output = src_pos_emb
44         for layer in self.encoder_layers:
45             encoder_output = layer.forward(encoder_output, src_mask)
46
47         # Embedding and positional encoding for target sequence
48         tgt_emb = self.tgt_embedding.forward(tgt_seq)
49         tgt_pos_emb = self.positional_encoding.forward(tgt_emb)
50
51         # Decoder
52         decoder_output = tgt_pos_emb
53         for layer in self.decoder_layers:
54             decoder_output = layer.forward(decoder_output, encoder_output, src_mask,
tgt_mask)
55
56         # Final linear layer
57         output = self.fc.forward(decoder_output)
58
59         return output
60
61     def backward(self, upstream_grad):
62         """
63         Backward pass of the Transformer NMT model.
64
65         Args:
66             upstream_grad: Gradient tensor of shape (batch_size, tgt_seq_length,
tgt_vocab_size)
67
68         Returns:
69             Gradient tensor of shape (batch_size, tgt_seq_length, d_model)
70         """
71         grad_fc = self.fc.backward(upstream_grad)
72
73         grad_decoder = grad_fc
74         for layer in reversed(self.decoder_layers):
75             grad_decoder = layer.backward(grad_decoder)
76
77         grad_encoder = None
78         for layer in reversed(self.encoder_layers):

```

```

79         if grad_encoder is None:
80             grad_encoder = layer.backward(grad_decoder)
81         else:
82             grad_encoder = layer.backward(grad_encoder)
83
84     return grad_encoder
85
86 def update_parameters(self, learning_rate):
87     """
88     Update parameters of the Transformer NMT model.
89
90     Args:
91         learning_rate: Learning rate for parameter updates.
92     """
93     self.src_embedding.update_parameters(learning_rate)
94     self.tgt_embedding.update_parameters(learning_rate)
95     self.positional_encoding.update_parameters(learning_rate)
96
97     for layer in self.encoder_layers:
98         layer.update_parameters(learning_rate)
99
100    for layer in self.decoder_layers:
101        layer.update_parameters(learning_rate)
102
103    self.fc.update_parameters(learning_rate)
104
105 def inference(self, src_seq, src_mask, max_length=100, SOS_token=1, EOS_token=2):
106     """
107     Inference function for the Transformer NMT model.
108
109     Args:
110         src_seq: Source sequence tensor of shape (1, src_seq_length)
111         src_mask: Source mask tensor of shape (1, 1, 1, src_seq_length)
112         max_length: Maximum length of the generated sequence.
113         SOS_token: Start-of-sequence token.
114         EOS_token: End-of-sequence token.
115
116     Returns:
117         Generated sequence tensor of shape (1, tgt_seq_length)
118     """
119     # Embedding and positional encoding for source sequence
120     src_emb = self.src_embedding.forward(src_seq)
121     src_pos_emb = self.positional_encoding.forward(src_emb)
122
123     # Encoder
124     encoder_output = src_pos_emb
125     for layer in self.encoder_layers:
126         encoder_output = layer.forward(encoder_output, src_mask)
127
128     # Initialize target sequence with SOS token
129     tgt_seq = np.array([[SOS_token]])
130
131     # Generate sequence
132     for _ in range(max_length):
133         # Create target mask
134         tgt_mask = np.tril(np.ones((1, 1, tgt_seq.shape[1], tgt_seq.shape[1])))
135
136         # Embedding and positional encoding for target sequence

```

```

137         tgt_emb = self.tgt_embedding.forward(tgt_seq)
138         tgt_pos_emb = self.positional_encoding.forward(tgt_emb)
139
140         # Decoder
141         decoder_output = tgt_pos_emb
142         for layer in self.decoder_layers:
143             decoder_output = layer.forward(decoder_output, encoder_output, src_mask,
tgt_mask)
144
145         # Final linear layer
146         output = self.fc.forward(decoder_output)
147
148         # Get next token
149         next_token = np.argmax(output[0, -1])
150
151         # Append next token to target sequence
152         tgt_seq = np.concatenate((tgt_seq, [[next_token]]), axis=1)
153
154         # Stop if EOS token is generated
155         if next_token == EOS_token:
156             break
157
158         return tgt_seq
159
160     def train(self):
161         """
162         Set the model to training mode.
163         """
164         self.training = True
165         for layer in self.encoder_layers:
166             # Add method to set training mode for layers that need it (e.g., dropout)
167             if hasattr(layer, 'train'):
168                 layer.train()
169         for layer in self.decoder_layers:
170             if hasattr(layer, 'train'):
171                 layer.train()
172
173     def eval(self):
174         """
175         Set the model to evaluation mode.
176         """
177         self.training = False
178         for layer in self.encoder_layers:
179             if hasattr(layer, 'eval'):
180                 layer.eval()
181         for layer in self.decoder_layers:
182             if hasattr(layer, 'eval'):
183                 layer.eval()

```

Explanation of train() and eval() Methods:

1. **train() Method:**

- Sets the model to training mode.
- This is useful for layers that behave differently during training and evaluation (e.g., dropout layers).
- In this implementation, it sets a flag `self.training` to `True`.
- It also propagates the training mode to encoder and decoder layers that might need it.

2. **eval() Method:**

- Sets the model to evaluation mode.
- This is useful for disabling behaviors specific to training (e.g., dropout).
- In this implementation, it sets the flag `self.training` to `False`.
- It also propagates the evaluation mode to encoder and decoder layers that might need it.

Usage Example:

```
1  # Create model
2  model = TransformerNMT(
3      src_vocab_size=10000,
4      tgt_vocab_size=10000,
5      d_model=512,
6      num_heads=8,
7      d_ff=2048,
8      num_layers=6,
9      max_seq_length=100
10 )
11
12 # Set to training mode
13 model.train()
14
15 # Perform training steps...
16
17 # Set to evaluation mode
18 model.eval()
19
20 # Perform inference
21 generated_seq = model.inference(src_seq, src_mask)
```

These methods allow you to control the behavior of the model during training and evaluation phases, which is particularly important for layers like dropout that need to behave differently in each phase.

