Encoder and Decoder

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

```
class EncoderLayer(BaseLayer):
    def __init__(self, d_model, num_heads, d_ff):
        """
        Initialize the encoder layer.

        Args:
            d_model: Dimension of the model.
            num_heads: Number of attention heads.
            d_ff: Dimension of the feed-forward network.
        """
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.norm1 = LayerNorm()
        self.ffn = FeedForward(d_model, d_ff)
        self.norm2 = LayerNorm()
        self.residual1 = ResidualConnection()
        self.residual2 = ResidualConnection()

    def forward(self, inputs, mask=None):
        """
        Forward pass of the encoder layer.

        Args:
            inputs: Input tensor of shape (batch_size, seq_length, d_model)
            mask: Optional mask tensor of shape (batch_size, 1, seq_length, seq_length)

        Returns:
            Output tensor of shape (batch_size, seq_length, d_model)
        """
        # Self-attention
        attn_output = self.self_attn.forward(inputs, inputs, inputs, mask)
        # Residual connection and layer normalization
        residual_output1 = self.residual1.forward(inputs, attn_output)
        norm_output1 = self.norm1.forward(residual_output1)

        # Feed-forward network
        ffn_output = self.ffn.forward(norm_output1)
        # Residual connection and layer normalization
        residual_output2 = self.residual2.forward(norm_output1, ffn_output)
        norm_output2 = self.norm2.forward(residual_output2)

        return norm_output2

    def backward(self, upstream_grad):
        """
        Backward pass of the encoder layer.

        Args:
            upstream_grad: Gradient tensor of shape (batch_size, seq_length, d_model)

        Returns:
```

```
            Gradient tensor of shape (batch_size, seq_length, d_model)
        """
        # Backward through layer normalization and residual connection
        grad_norm2 = self.norm2.backward(upstream_grad)
        grad_residual2, grad_ffn = self.residual2.backward(grad_norm2)

        # Backward through feed-forward network
        grad_ffn = self.ffn.backward(grad_ffn)

        # Backward through layer normalization and residual connection
        grad_norm1 = self.norm1.backward(grad_residual2 + grad_ffn)
        grad_residual1, grad_attn = self.residual1.backward(grad_norm1)

        # Backward through self-attention
        grad_attn = self.self_attn.backward(grad_attn + grad_residual1)

        return grad_attn

    def update_parameters(self, learning_rate):
        """
        Update parameters of the encoder layer.

        Args:
            learning_rate: Learning rate for parameter updates.
        """
        self.self_attn.update_parameters(learning_rate)
        self.ffn.update_parameters(learning_rate)


class DecoderLayer(BaseLayer):
    def __init__(self, d_model, num_heads, d_ff):
        """
        Initialize the decoder layer.

        Args:
            d_model: Dimension of the model.
            num_heads: Number of attention heads.
            d_ff: Dimension of the feed-forward network.
        """
        self.masked_attn = MultiHeadAttention(d_model, num_heads)
        self.norm1 = LayerNorm()
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.norm2 = LayerNorm()
        self.ffn = FeedForward(d_model, d_ff)
        self.norm3 = LayerNorm()
        self.residual1 = ResidualConnection()
        self.residual2 = ResidualConnection()
        self.residual3 = ResidualConnection()

    def forward(self, inputs, encoder_output, src_mask=None, tgt_mask=None):
        """
        Forward pass of the decoder layer.

        Args:
            inputs: Input tensor of shape (batch_size, seq_length, d_model)
            encoder_output: Output tensor from the encoder of shape (batch_size,
seq_length, d_model)
            src_mask: Optional source mask tensor of shape (batch_size, 1, 1, seq_length)
```

```
            tgt_mask: Optional target mask tensor of shape (batch_size, 1, seq_length, seq_length)

        Returns:
            Output tensor of shape (batch_size, seq_length, d_model)
        """
        # Masked self-attention
        attn_output = self.masked_attn.forward(inputs, inputs, inputs, tgt_mask)
        # Residual connection and layer normalization
        residual_output1 = self.residual1.forward(inputs, attn_output)
        norm_output1 = self.norm1.forward(residual_output1)

        # Cross-attention
        cross_attn_output = self.cross_attn.forward(norm_output1, encoder_output, encoder_output, src_mask)
        # Residual connection and layer normalization
        residual_output2 = self.residual2.forward(norm_output1, cross_attn_output)
        norm_output2 = self.norm2.forward(residual_output2)

        # Feed-forward network
        ffn_output = self.ffn.forward(norm_output2)
        # Residual connection and layer normalization
        residual_output3 = self.residual3.forward(norm_output2, ffn_output)
        norm_output3 = self.norm3.forward(residual_output3)

        return norm_output3

    def backward(self, upstream_grad):
        """
        Backward pass of the decoder layer.

        Args:
            upstream_grad: Gradient tensor of shape (batch_size, seq_length, d_model)

        Returns:
            Gradient tensor of shape (batch_size, seq_length, d_model)
        """
        # Backward through layer normalization and residual connection
        grad_norm3 = self.norm3.backward(upstream_grad)
        grad_residual3, grad_ffn = self.residual3.backward(grad_norm3)

        # Backward through feed-forward network
        grad_ffn = self.ffn.backward(grad_ffn)

        # Backward through layer normalization and residual connection
        grad_norm2 = self.norm2.backward(grad_residual3 + grad_ffn)
        grad_residual2, grad_cross_attn = self.residual2.backward(grad_norm2)

        # Backward through cross-attention
        grad_cross_attn = self.cross_attn.backward(grad_cross_attn + grad_residual2)

        # Backward through layer normalization and residual connection
        grad_norm1 = self.norm1.backward(grad_cross_attn)
        grad_residual1, grad_masked_attn = self.residual1.backward(grad_norm1)

        # Backward through masked self-attention
        grad_masked_attn = self.masked_attn.backward(grad_masked_attn + grad_residual1)

```

```python
        return grad_masked_attn

    def update_parameters(self, learning_rate):
        """
        Update parameters of the decoder layer.

        Args:
            learning_rate: Learning rate for parameter updates.
        """
        self.masked_attn.update_parameters(learning_rate)
        self.cross_attn.update_parameters(learning_rate)
        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:

```python
# Hyperparameters
d_model = 512
num_heads = 8
d_ff = 2048
batch_size = 32
seq_len = 50

# Create encoder and decoder layers
encoder_layer = EncoderLayer(d_model, num_heads, d_ff)
decoder_layer = DecoderLayer(d_model, num_heads, d_ff)
```

```
# Example inputs
encoder_input = np.random.randn(batch_size, seq_len, d_model)
decoder_input = np.random.randn(batch_size, seq_len, d_model)
encoder_output = np.random.randn(batch_size, seq_len, d_model)

# Create masks
src_mask = np.random.randint(0, 2, (batch_size, 1, 1, seq_len))
tgt_mask = np.random.randint(0, 2, (batch_size, 1, seq_len, seq_len))

# Forward pass for encoder
encoder_output = encoder_layer.forward(encoder_input, src_mask)

# Forward pass for decoder
decoder_output = decoder_layer.forward(decoder_input, encoder_output, src_mask, tgt_mask)

# Simulate upstream gradients
encoder_upstream_grad = np.random.randn(batch_size, seq_len, d_model)
decoder_upstream_grad = np.random.randn(batch_size, seq_len, d_model)

# Backward pass for encoder
encoder_grad = encoder_layer.backward(encoder_upstream_grad)

# Backward pass for decoder
decoder_grad = decoder_layer.backward(decoder_upstream_grad)

# Update parameters
learning_rate = 0.001
encoder_layer.update_parameters(learning_rate)
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

```
# Hyperparameters
num_epochs = 30
batch_size = 128
learning_rate = 1e-4
warmup_steps = 4000

# Optimizer
optimizer = AdamOptimizer(model.parameters(), lr=learning_rate, beta1=0.9, beta2=0.999, epsilon=1e-8)
```

```python

# Learning rate scheduler
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=warmup_steps,
    num_training_steps=num_epochs * num_batches
)

# Training loop
for epoch in range(num_epochs):
    model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        inputs, targets = batch
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        scheduler.step()

    # Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for batch in val_loader:
            inputs, targets = batch
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            val_loss += loss.item()

    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:

```python
class AdamOptimizer:
    def __init__(self, params, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.params = params
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = None
        self.v = None
        self.t = 0

    def update(self):
        if self.m is None:
            self.m = [np.zeros_like(param) for param in self.params]
            self.v = [np.zeros_like(param) for param in self.params]

        self.t += 1
        for i, param in enumerate(self.params):
```

```python
            if param.dW is not None:
                self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * param.dW
                self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (param.dW ** 2)

                m_hat = self.m[i] / (1 - self.beta1 ** self.t)
                v_hat = self.v[i] / (1 - self.beta2 ** self.t)

                param.W -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)
                param.dW = None

            if param.db is not None:
                self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * param.db
                self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (param.db ** 2)

                m_hat = self.m[i] / (1 - self.beta1 ** self.t)
                v_hat = self.v[i] / (1 - self.beta2 ** self.t)

                param.b -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)
                param.db = None


def train_model(model, train_loader, val_loader, num_epochs, optimizer, criterion):
    for epoch in range(num_epochs):
        model.train()
        total_loss = 0.0
        for batch in train_loader:
            src_seq, tgt_seq = batch

            optimizer.zero_grad()
            outputs = model.forward(src_seq, tgt_seq[:, :-1])
            loss = criterion(outputs, tgt_seq[:, 1:])
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_train_loss = total_loss / len(train_loader)

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for batch in val_loader:
                src_seq, tgt_seq = batch
                outputs = model.forward(src_seq, tgt_seq[:, :-1])
                loss = criterion(outputs, tgt_seq[:, 1:])
                val_loss += loss.item()

        avg_val_loss = val_loss / len(val_loader)

        print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss}, Val Loss: {avg_val_loss}")


# Example usage:
if __name__ == "__main__":
    # Hyperparameters
    src_vocab_size = 10000
    tgt_vocab_size = 10000
    d_model = 512
```

```
    num_heads = 8
    d_ff = 2048
    num_layers = 6
    max_seq_length = 100
    batch_size = 32
    num_epochs = 30
    learning_rate = 0.0001

    # Create model
    model = TransformerNMT(
        src_vocab_size,
        tgt_vocab_size,
        d_model,
        num_heads,
        d_ff,
        num_layers,
        max_seq_length
    )

    # Create optimizer
    optimizer = AdamOptimizer(
        [param for param in model.__dict__.values() if isinstance(param, BaseLayer)],
        lr=learning_rate
    )

    # Create loss function
    criterion = lambda outputs, targets: np.mean(
        np.cross_entropy(outputs.reshape(-1, tgt_vocab_size), targets.reshape(-1))
    )

    # Create dummy data loaders
    train_loader = [
        (
            np.random.randint(0, src_vocab_size, size=(batch_size, max_seq_length)),
            np.random.randint(0, tgt_vocab_size, size=(batch_size, max_seq_length))
        )
        for _ in range(100)
    ]
    val_loader = [
        (
            np.random.randint(0, src_vocab_size, size=(batch_size, max_seq_length)),
            np.random.randint(0, tgt_vocab_size, size=(batch_size, max_seq_length))
        )
        for _ in range(20)
    ]

    # Train the model
    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:

```
class TransformerNMT(BaseLayer):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads, d_ff, num_layers, max_seq_length):
        """
        Initialize the Transformer NMT model.

        Args:
            src_vocab_size: Size of the source vocabulary.
            tgt_vocab_size: Size of the target vocabulary.
            d_model: Dimension of the model.
            num_heads: Number of attention heads.
            d_ff: Dimension of the feed-forward network.
            num_layers: Number of encoder and decoder layers.
            max_seq_length: Maximum sequence length.
        """
        self.src_embedding = EmbeddingLayer(src_vocab_size, d_model)
        self.tgt_embedding = EmbeddingLayer(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEmbeddingLayer(max_seq_length, d_model)

        self.encoder_layers = [EncoderLayer(d_model, num_heads, d_ff) for _ in range(num_layers)]
        self.decoder_layers = [DecoderLayer(d_model, num_heads, d_ff) for _ in range(num_layers)]

        self.fc = LinearLayer(d_model, tgt_vocab_size)
        self.training = True  # Flag to indicate training mode

```

```python
    def forward(self, src_seq, tgt_seq, src_mask=None, tgt_mask=None):
        """
        Forward pass of the Transformer NMT model.

        Args:
            src_seq: Source sequence tensor of shape (batch_size, src_seq_length)
            tgt_seq: Target sequence tensor of shape (batch_size, tgt_seq_length)
            src_mask: Optional source mask tensor of shape (batch_size, 1, 1, src_seq_length)
            tgt_mask: Optional target mask tensor of shape (batch_size, 1, tgt_seq_length, tgt_seq_length)

        Returns:
            Output tensor of shape (batch_size, tgt_seq_length, tgt_vocab_size)
        """
        # Embedding and positional encoding for source sequence
        src_emb = self.src_embedding.forward(src_seq)
        src_pos_emb = self.positional_encoding.forward(src_emb)

        # Encoder
        encoder_output = src_pos_emb
        for layer in self.encoder_layers:
            encoder_output = layer.forward(encoder_output, src_mask)

        # Embedding and positional encoding for target sequence
        tgt_emb = self.tgt_embedding.forward(tgt_seq)
        tgt_pos_emb = self.positional_encoding.forward(tgt_emb)

        # Decoder
        decoder_output = tgt_pos_emb
        for layer in self.decoder_layers:
            decoder_output = layer.forward(decoder_output, encoder_output, src_mask, tgt_mask)

        # Final linear layer
        output = self.fc.forward(decoder_output)

        return output

    def backward(self, upstream_grad):
        """
        Backward pass of the Transformer NMT model.

        Args:
            upstream_grad: Gradient tensor of shape (batch_size, tgt_seq_length, tgt_vocab_size)

        Returns:
            Gradient tensor of shape (batch_size, tgt_seq_length, d_model)
        """
        grad_fc = self.fc.backward(upstream_grad)

        grad_decoder = grad_fc
        for layer in reversed(self.decoder_layers):
            grad_decoder = layer.backward(grad_decoder)

        grad_encoder = None
        for layer in reversed(self.encoder_layers):
```

```
            if grad_encoder is None:
                grad_encoder = layer.backward(grad_decoder)
            else:
                grad_encoder = layer.backward(grad_encoder)

        return grad_encoder

    def update_parameters(self, learning_rate):
        """
        Update parameters of the Transformer NMT model.

        Args:
            learning_rate: Learning rate for parameter updates.
        """
        self.src_embedding.update_parameters(learning_rate)
        self.tgt_embedding.update_parameters(learning_rate)
        self.positional_encoding.update_parameters(learning_rate)

        for layer in self.encoder_layers:
            layer.update_parameters(learning_rate)

        for layer in self.decoder_layers:
            layer.update_parameters(learning_rate)

        self.fc.update_parameters(learning_rate)

    def inference(self, src_seq, src_mask, max_length=100, SOS_token=1, EOS_token=2):
        """
        Inference function for the Transformer NMT model.

        Args:
            src_seq: Source sequence tensor of shape (1, src_seq_length)
            src_mask: Source mask tensor of shape (1, 1, 1, src_seq_length)
            max_length: Maximum length of the generated sequence.
            SOS_token: Start-of-sequence token.
            EOS_token: End-of-sequence token.

        Returns:
            Generated sequence tensor of shape (1, tgt_seq_length)
        """
        # Embedding and positional encoding for source sequence
        src_emb = self.src_embedding.forward(src_seq)
        src_pos_emb = self.positional_encoding.forward(src_emb)

        # Encoder
        encoder_output = src_pos_emb
        for layer in self.encoder_layers:
            encoder_output = layer.forward(encoder_output, src_mask)

        # Initialize target sequence with SOS token
        tgt_seq = np.array([[SOS_token]])

        # Generate sequence
        for _ in range(max_length):
            # Create target mask
            tgt_mask = np.tril(np.ones((1, 1, tgt_seq.shape[1], tgt_seq.shape[1])))

            # Embedding and positional encoding for target sequence
```

```
            tgt_emb = self.tgt_embedding.forward(tgt_seq)
            tgt_pos_emb = self.positional_encoding.forward(tgt_emb)

            # Decoder
            decoder_output = tgt_pos_emb
            for layer in self.decoder_layers:
                decoder_output = layer.forward(decoder_output, encoder_output, src_mask, tgt_mask)

            # Final linear layer
            output = self.fc.forward(decoder_output)

            # Get next token
            next_token = np.argmax(output[0, -1])

            # Append next token to target sequence
            tgt_seq = np.concatenate((tgt_seq, [[next_token]]), axis=1)

            # Stop if EOS token is generated
            if next_token == EOS_token:
                break

        return tgt_seq

    def train(self):
        """
        Set the model to training mode.
        """
        self.training = True
        for layer in self.encoder_layers:
            # Add method to set training mode for layers that need it (e.g., dropout)
            if hasattr(layer, 'train'):
                layer.train()
        for layer in self.decoder_layers:
            if hasattr(layer, 'train'):
                layer.train()

    def eval(self):
        """
        Set the model to evaluation mode.
        """
        self.training = False
        for layer in self.encoder_layers:
            if hasattr(layer, 'eval'):
                layer.eval()
        for layer in self.decoder_layers:
            if hasattr(layer, 'eval'):
                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:

```
# Create model
model = TransformerNMT(
    src_vocab_size=10000,
    tgt_vocab_size=10000,
    d_model=512,
    num_heads=8,
    d_ff=2048,
    num_layers=6,
    max_seq_length=100
)

# Set to training mode
model.train()

# Perform training steps...

# Set to evaluation mode
model.eval()

# Perform inference
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.