Attention Mechanism in Seq2Seq Models

We decide to use an attention mechanism. Instead of forcing the decoder to rely on just one compressed representation of the input, we let it decide automatically which parts of the input to focus on at each step. Our proposed model learns attention weights that tell it how much importance to give to different words in the input. This way, the decoder can focus as it generates each word, making the sequence-to-sequence model more accurate and improving performance on longer sequences.

In simple terms, attention helps the model "look back" at relevant words instead of trying to remember everything.

Mathematical Demonstration

Let h_i be the encoder hidden state and s_t be the decoder state.

We use dot product attention to compute a score that measures the alignment between the decoder state and each encoder hidden state:

$$a(s_t, h_i) = s_t^\top h_i$$

The attention weights are computed using the softmax function:

$$\alpha_{t,i} = \frac{\exp\left(a(s_t, h_i)\right)}{\sum_j \exp\left(a(s_t, h_j)\right)}$$

The context vector is obtained as a weighted sum of encoder hidden states:

$$c_t = \sum_i \alpha_{t,i} h_i$$

Finally, the decoder state is updated using:

$$s_t = \text{Decoder}(y_{t-1}, s_{t-1}, c_t)$$

Note: As an encoder, we usually use a transformer (which is a complex architecture) or an LSTM for minimal complexity. This type of architecture is used in various domains, such as machine translation.

Code Implementation

```
class Attention(nn.Module):
    def __init__(self, hidden_dim):
        super(Attention, self).__init__()

    def forward(self, decoder_state, encoder_outputs):
        # Compute dot product attention scores
        scores = torch.bmm(encoder_outputs, decoder_state.unsqueeze(2)).squeeze(2)
        attn_weights = torch.softmax(scores, dim=1)

        # Compute context vector
        context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs).squeeze(1)
        return context, attn_weights
```