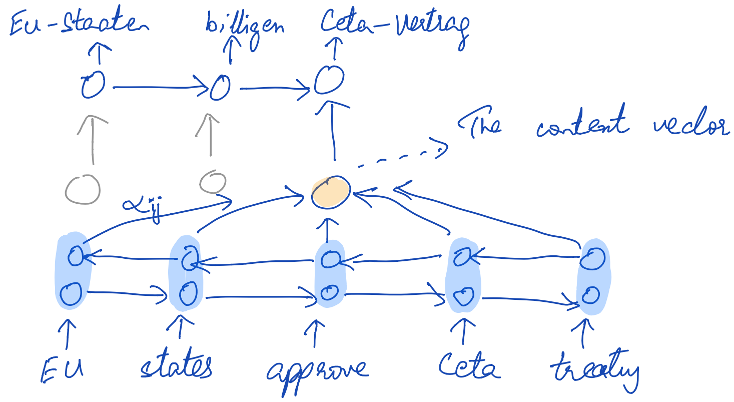
Goal: Model the probability $P(Y|X)$ of a target sequence
$$Y = (y_1, \ldots, y_N)$$
given a source $X = (x_1, \ldots, x_M)$



Encoder produces: Sequence of hidden states $h_1, \ldots, h_M$ for each source word

Recursive formula:
$$h_j = GRU(x_j, h_{j-1})$$

- A normal GRU would know about the preceding words but not the following.
- So use a Bi-GRU and concatenate the two hidden states shown highlighted in blue above.

Decoder: has a hidden state s_i
$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$
y_{i-1} is the previously generated target word.

c_i is the context vector.

- After computing the decoder state s_i, apply a non-linear function g (which applies a softmax) to calculate the probability of the target word, y_i

$$p(y_i \mid y_{<i}, x_1^m) = g(s_i, c_i, y_{i-1})$$

- Since g applies softmax, it provides a vector of the size of the output vocab that sums to 1.0

$\boxed{X}$

When outputting the final hidden state, manually concatenate the final states for both directions since it is a Bi-GRU.

$\boxed{X}$

Teacher forcing: During training, we simply feed the correct previous target word embedding to the GRU.

```python
class Decoder(nn.Module):
    """A conditional RNN decoder with attention."""

    def __init__(self, emb_size, hidden_size, attention, num_layers=1, dropout=0.5,
                 bridge=True):
        super(Decoder, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.attention = attention
        self.dropout = dropout

        self.rnn = nn.GRU(emb_size + 2*hidden_size, hidden_size, num_layers,
                          batch_first=True, dropout=dropout)

        # to initialize from the final encoder state
        self.bridge = nn.Linear(2*hidden_size, hidden_size, bias=True) if bridge else None

        self.dropout_layer = nn.Dropout(p=dropout)
        self.pre_output_layer = nn.Linear(hidden_size + 2*hidden_size + emb_size,
                                          hidden_size, bias=False)

    def forward_step(self, prev_embed, encoder_hidden, src_mask, proj_key, hidden):
        """Perform a single decoder step (1 word)"""

        # compute context vector using attention mechanism
        query = hidden[-1].unsqueeze(1)  # [#layers, B, D] -> [B, 1, D]
        context, attn_probs = self.attention(
            query=query, proj_key=proj_key,
            value=encoder_hidden, mask=src_mask)

        # update rnn hidden state
        rnn_input = torch.cat([prev_embed, context], dim=2)
        output, hidden = self.rnn(rnn_input, hidden)

        pre_output = torch.cat([prev_embed, output, context], dim=2)
        pre_output = self.dropout_layer(pre_output)
        pre_output = self.pre_output_layer(pre_output)

        return output, hidden, pre_output

    def forward(self, trg_embed, encoder_hidden, encoder_final,
                src_mask, trg_mask, hidden=None, max_len=None):
        """Unroll the decoder one step at a time."""
```

This is from the final encoder state so 2 x hidden size

$$p(y_i \mid y_{<i}, x_1^m) = g(s_i, c_i, y_{i-1})$$

hidden size

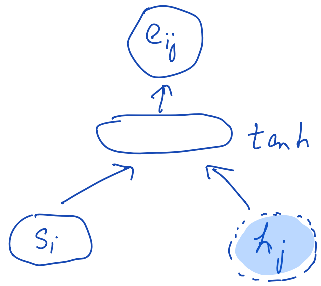
2 x hidden size

emb_size

Attention

- At every step, decoder has access to all source word representations $h_1, \ldots, h_M$

Remember that the state of decoder is represented by GRU hidden state $s_i \rightarrow$ so we want to know which source word representations h_j are most relevant



$\boxed{x}$

- the query : the current decoder state s_i
- the key : each encoder state h_j
- Project this to a single value(i.e. scalar) to get the attention energy e_{ij}

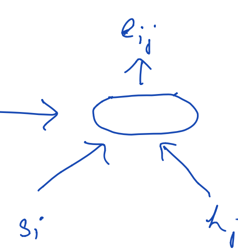
Normalize by a softmax :→ $\sum_j \alpha_{ij} = 1.0$

- Content vector c_i is the weighted sum of the encoder hidden states

$$c_i = \sum_j \alpha_{ij} h_j$$

This sums to 1

```python
class BahdanauAttention(nn.Module):
    """Implements Bahdanau (MLP) attention"""

    def __init__(self, hidden_size, key_size=None, query_size=None):
        super(BahdanauAttention, self).__init__()

        # We assume a bi-directional encoder so key_size is 2*hidden_size
        key_size = 2 * hidden_size if key_size is None else key_size
        query_size = hidden_size if query_size is None else query_size

        self.key_layer = nn.Linear(key_size, hidden_size, bias=False)
        self.query_layer = nn.Linear(query_size, hidden_size, bias=False)
        self.energy_layer = nn.Linear(hidden_size, 1, bias=False)

        # to store attention scores
        self.alphas = None

    def forward(self, query=None, proj_key=None, value=None, mask=None):
        assert mask is not None, "mask is required"

        # We first project the query (the decoder state).
        # The projected keys (the encoder states) were already pre-computed.
        query = self.query_layer(query)

        # Calculate scores.
        scores = self.energy_layer(torch.tanh(query + proj_key))
        scores = scores.squeeze(2).unsqueeze(1)

        # Mask out invalid positions.
        # The mask marks valid positions so we invert it using `mask & 0`.
        scores.data.masked_fill_(mask == 0, -float('inf'))

        # Turn scores to probabilities.
        alphas = F.softmax(scores, dim=-1)
        self.alphas = alphas

        # The context vector is the weighted sum of the values.
        context = torch.bmm(alphas, value)

        # context shape: [B, 1, 2D], alphas shape: [B, 1, M]
        return context, alphas
```



c_i α_{ij}

Decoder again

```python
class Decoder(nn.Module):
    """A conditional RNN decoder with attention."""

    def __init__(self, emb_size, hidden_size, attention, num_layers=1, dropout=0.5,
                 bridge=True):
        super(Decoder, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.attention = attention
        self.dropout = dropout

        self.rnn = nn.GRU(emb_size + 2*hidden_size, hidden_size, num_layers,
                          batch_first=True, dropout=dropout)

        # to initialize from the final encoder state
        self.bridge = nn.Linear(2*hidden_size, hidden_size, bias=True) if bridge else None

        self.dropout_layer = nn.Dropout(p=dropout)
        self.pre_output_layer = nn.Linear(hidden_size + 2*hidden_size + emb_size,
                                          hidden_size, bias=False)

    def forward_step(self, prev_embed, encoder_hidden, src_mask, proj_key, hidden):
        """Perform a single decoder step (1 word)"""

        # compute context vector using attention mechanism
        query = hidden[-1].unsqueeze(1)  # [#layers, B, D] -> [B, 1, D]
        context, attn_probs = self.attention(
            query=query, proj_key=proj_key,
            value=encoder_hidden, mask=src_mask)

        # update rnn hidden state
        rnn_input = torch.cat([prev_embed, context], dim=2)
        output, hidden = self.rnn(rnn_input, hidden)

        pre_output = torch.cat([prev_embed, output, context], dim=2)
        pre_output = self.dropout_layer(pre_output)
        pre_output = self.pre_output_layer(pre_output)

        return output, hidden, pre_output

    def forward(self, trg_embed, encoder_hidden, encoder_final,
                src_mask, trg_mask, hidden=None, max_len=None):
        """Unroll the decoder one step at a time."""

        # the maximum number of steps to unroll the RNN
        if max_len is None:
            max_len = trg_mask.size(-1)

        # initialize decoder hidden state
        if hidden is None:
            hidden = self.init_hidden(encoder_final)

        # pre-compute projected encoder hidden states
        # (the "keys" for the attention mechanism)
        # this is only done for efficiency
        proj_key = self.attention.key_layer(encoder_hidden)

        # here we store all intermediate hidden states and pre-output vectors
        decoder_states = []
        pre_output_vectors = []

        # unroll the decoder RNN for max_len steps
        for i in range(max_len):
            prev_embed = trg_embed[:, i].unsqueeze(1)
            output, hidden, pre_output = self.forward_step(
              prev_embed, encoder_hidden, src_mask, proj_key, hidden)
            decoder_states.append(output)
            pre_output_vectors.append(pre_output)

        decoder_states = torch.cat(decoder_states, dim=1)
        pre_output_vectors = torch.cat(pre_output_vectors, dim=1)
        return decoder_states, hidden, pre_output_vectors  # [B, N, D]

    def init_hidden(self, encoder_final):
        """Returns the initial decoder state,
        conditioned on the final encoder state."""

        if encoder_final is None:
            return None  # start with zeros

        return torch.tanh(self.bridge(encoder_final))
```

s_i

c_i (2 x hidden size)

y_{i-1}

s_{i-1}

$$p(y_i | y_{<i}, x_1^M) = g(\overbrace{s_i, c_i, y_{i-1}}^{\text{pre-output}})$$

Note that we don't apply softmax on training

output: Don't think it is further used.

hidden would be the next s_i in the next iteration of the loop