Sequence to Sequence Learning with Attention

CITS4012 Natural Language Processing

A/Prof. Wei Liu

wei.liu@uwa.edu.au

Computer Science and Software Engineering
The University of Western Australia

October 20, 2021

What we are going to cover today

1. Attention
   - Attention Mechanism
   - Attention Score Calculation
2. Multi-headed Attention
3. Self-Attention
   - Encoder with Self-Attention
   - Encoder with self- and cross-attentions
   - Decoder with Self-Attention
   - Encoder Decoder with Self Attention

[see Deep Learning with Pytorch Step by Step, Chapter 9]

Attention

Attention Mechanism in A Nutshell

The decoder's role is to generate the translated words.

Paying attention to different elements in the source sequence

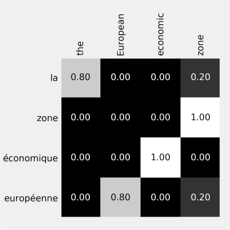
If the decoder is allowed to choose which hidden states from the encoder it will use to generate each output, it means it can choose which English words it will use to generate each translated word.

English: `the European economic zone`
French: `la zone économique européenne`

- From `the` to `la`, the decoder needs to "pay attention" to the last English word `zone`, to determine if it is a **singular feminine noun**.
- For the second French word `zone`, the decoder should again "pay more attention" to the last English word `zone`.

Attention Score Matrix

These weights, or **alphas**, are the **attention scores**.



*The numbers above are made-up and for illustration purpose only.

Since the translated "la" is based on "the" and "zone", we can guess that a sensible decoder would

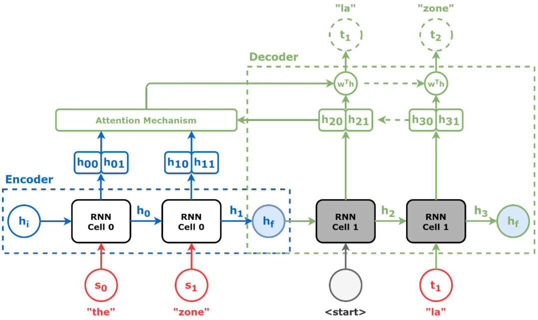
- assigned 80% of its attention to the definite article, and
- the remaining 20% of its attention to the corresponding noun to determine its gender.

The more relevant to the decoder a hidden state from the encoder is, the higher the score.

Encoder-Decoder with Attention

The special token indicating the start of the target sequence

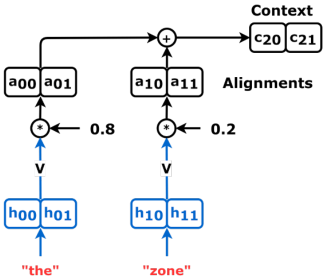
In this translation example, the source and target sequences are **independent**. The first input of the decoder is **not the last element of the source sequence**, but the special token that indicates the start of a new sequence.



Encoder-Decoder with Attention for Translation

A close-up of the Attention Mechanism

Instead of generating predictions solely based on its own hidden states, the decoder will recruit the attention mechanism to help it decide which parts of the source sequence it must pay attention to.



The attention mechanism informed the decoder it should pay 80% of its attention to the encoder's hidden state corresponding to the word "the", and the remaining 20%, to the word "zone".

Let's get the terminology right first

Values

"Values" (V) refer to the encoder's hidden states (or their *affine transformations*).

Alignment Vector

The resulting multiplication of a "value" by its corresponding attention score is called an alignment vector.

$$\textit{context vector} = \underbrace{\alpha_0 * h_0}_{\textit{alignment vector}_0} + \underbrace{\alpha_1 * h_1}_{\textit{alignment vector}_1} = 0.8 * \textit{value}_{the} + 0.2 * \textit{value}_{zone}$$

Context Vector

The sum of all alignment vectors (that is, the weighted average of the hidden states) is called a context vector.

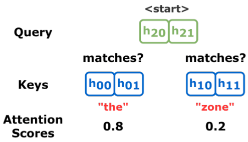
Where do the attention scores come from

The attention scores are based on matching each hidden state of the decoder (h2) to every hidden state of the encoder (h0 and h1).

- Some of them will be good matches (higher attention scores)
- Some others will be poor matches (low attention scores)

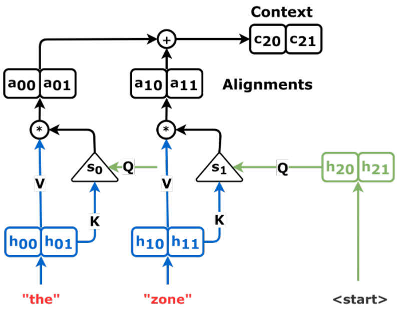
"Keys" and "Queries"

- The encoder's hidden states are called "keys" (K)
- The decoder's hidden state is called a "query" (Q)



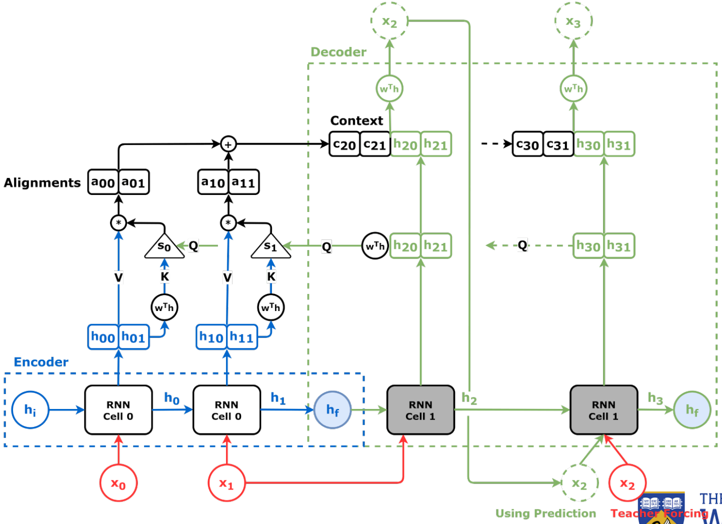
An analogical idea is: the **encoder** works like a *key-value store* that the **decoder** can **query**.

How to match a given "query" (Q) to the "keys" (K)



- The "query" (Q) is matched to both "keys" (K) to compute the attention scores (s) used to compute the context vector (C)
- The context vector (C) is simply the weighted average of the "values" (V).

Encoder + Decoder + Attention



Computing the Context Vector

1. The source sequence is the input of the encoder, and the hidden states it outputs become
   - "values" (V), and
   - affine transformed into and "keys" (K)
2. The encoder-decoder dynamics stay exactly the same: despite that we are sending the entire sequence of hidden states to the decoder, we still use the encoder's final hidden state as the decoder's initial hidden state.
3. In the case of a continuous sequence prediction problem, we still use the last element of the source sequence as input to the first step of the decoder.
4. The first "query" (Q) is the decoder's hidden state, affine transformed.
5. The attention score (s) is worked out from keys (K) and query (Q).
6. The "values" is then scaled to obtain the alignment vector (α).
7. The context vector (C) is then concatenated with the hidden state vector from the Decoder to obtain the final prediction after affine transformation.

The Scoring Method

The role of the scoring method

The scoring method needs to determine if two vectors are a good match or not, or, phrased differently, it needs to determine if two vectors (K and Q) are similar or not.

As both K and Q are vectors of the same dimension (i.e. the size of the hidden state), cosine similarity is the most natural way measuring their similarity.

$$cos\ \theta = \frac{\sum_i q_i k_i}{\sqrt{\sum_j q_j^2}\sqrt{\sum_j k_j^2}}$$

$$cos\ \theta \sqrt{\sum_j q_j^2}\sqrt{\sum_j k_j^2} = \sum_i q_i k_i$$

Since cosine similarity does not consider the norm (magnitude/length) of the vectors, we use the dot product.

$$cos\theta\ ||Q||\ ||K|| = Q \cdot K$$

Many choices of scoring functions

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

The dot product is one of the most common ways to compute alignment (and attention) scores, but it is not the only one.

For more information on different mechanisms, and attention in general, please refer to Lilian Weng's amazing blog post

Scaled Scoring Function

- Dot product is the sum of element-wise multiplication of two vectors. The longer the vectors, the bigger the variance.
- We standardize by scaling the dot product by the inverse of its standard deviation, given by the square root of the dimension of the source hidden state.
- On average, the variance equals the number of dimensions.

$$scaled\ dot\ product = \frac{Q \cdot K}{\sqrt{dim_k}}$$

The `softmax` function ensures that scores add up to 1.

```
def calc_alphas(ks, q):
    dims = q.size(-1)
    # N, 1, H x N, H, L -> N, 1, L
    products = torch.bmm(q, ks.permute(0, 2, 1))
    scaled_products = products / np.sqrt(dims)
    alphas = F.softmax(scaled_products, dim=-1)
    return alphas
```

code/score_function.py

Source Mask

When we pad our sequences to have equal length, we want the attention mechanism to ignore those padded data points.

The **source mask** should be **False** for every padded data point, and its shape should be (N, 1, L) where L is the length of the source sequence.

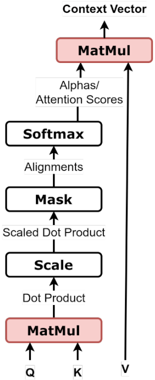
```
source_seq = torch.tensor([[[-1., 1.], [0., 0.]]])
# pretend there's an encoder here...
keys = torch.tensor([[[-.38, .44], [.85, -.05]]])
query = torch.tensor([[[-1., 1.]]])

source_mask = (source_seq != 0).all(axis=2).unsqueeze(1)
source_mask # N, 1, L
```

code/mask.py

```
tensor([[[ True, False]]])
```

Attention in PyTorch code I

The job of the attention layer is to caculate context vector as shown below:



Scaled Dot-Product Attention

Attention in PyTorch code II

```
class Attention(nn.Module):
    def __init__(self, hidden_dim, input_dim=None,
                proj_values=False):
        super().__init__()
        self.d_k = hidden_dim
        self.input_dim = hidden_dim
                        if input_dim is None
                        else input_dim
        self.proj_values = proj_values
        # Affine transformations for Q, K, and V
        self.linear_query = nn.Linear(self.input_dim,
                                      hidden_dim)
        self.linear_key = nn.Linear(self.input_dim,
                                      hidden_dim)
        self.linear_value = nn.Linear(self.input_dim,
                                      hidden_dim)
        self.alphas = None
```

Attention in PyTorch code III

```python

    def init_keys(self, keys):
        self.keys = keys
        self.proj_keys = self.linear_key(self.keys)
        self.values = self.linear_value(self.keys) \
                        if self.proj_values else self.keys

    def score_function(self, query):
        proj_query = self.linear_query(query)
        # scaled dot product
        # N, 1, H x N, H, L -> N, 1, L
        dot_products = torch.bmm(proj_query,
                         self.proj_keys.permute(0, 2, 1))
        scores =  dot_products / np.sqrt(self.d_k)
        return scores
```

Attention in PyTorch code IV

```

    def forward(self, query, mask=None):
        # Query is batch-first N, 1, H
        scores = self.score_function(query) # N, 1, L
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, 1, L
        self.alphas = alphas.detach()

        # N, 1, L x N, L, H -> N, 1, H
        context = torch.bmm(alphas, self.values)
        return context
```

code/attention.py

Decoder with Attention in PyTorch code I

```python
class DecoderAttn(nn.Module):
    def __init__(self, n_features, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.hidden = None
        self.basic_rnn = nn.GRU(self.n_features,
                                self.hidden_dim, batch_first=True)
        self.attn = Attention(self.hidden_dim)
        self.regression = nn.Linear(2 * self.hidden_dim,
                                    self.n_features)

    def init_hidden(self, hidden_seq):
        # the output of the encoder is N, L, H
        # and init_keys expects batch-first as well
        self.attn.init_keys(hidden_seq)
        hidden_final = hidden_seq[:, -1:]
        self.hidden = hidden_final.permute(1, 0, 2)# L, N, H


```

Decoder with Attention in PyTorch code II

```
    def forward(self, X, mask=None):
        # X is N, 1, F
        batch_first_output, self.hidden =
                                self.basic_rnn(X, self.hidden)

        query = batch_first_output[:, -1:]
        # Attention
        context = self.attn(query, mask=mask)
        concatenated = torch.cat([context, query], axis=-1)
        out = self.regression(concatenated)

        # N, 1, F
        return out.view(-1, 1, self.n_features)
```

code/decoder_attn.py

Visualising Attention Scores I

Our former `EncoderDecoder` class works seamlessly with an instance of `DecoderAttn`. We can however modify it to visualise the attention scores.

> The easiest way is to create a new class that inherits from `EncoderDecoder` and to override the `init_outputs` and `store_outputs` methods

```python
class EncoderDecoderAttn(EncoderDecoder):
    def __init__(self, encoder, decoder,
                 input_len, target_len,
                 teacher_forcing_prob=0.5):
        super().__init__(encoder, decoder, input_len,
                 target_len, teacher_forcing_prob)
        self.alphas = None

    def init_outputs(self, batch_size):
        device = next(self.parameters()).device
        # N, L (target), F
        self.outputs = torch.zeros(batch_size,
```

Visualising Attention Scores II

```
                                   self.target_len,
                                   self.encoder.n_features)
                                                   .to(device)
        # N, L (target), L (source)
        self.alphas = torch.zeros(batch_size,
                                       self.target_len,
                                       self.input_len)
                                                   .to(device)

    def store_output(self, i, out):
        # Stores the output
        self.outputs[:, i:i+1, :] = out
        self.alphas[:, i:i+1, :] = self.decoder.attn.alphas
```

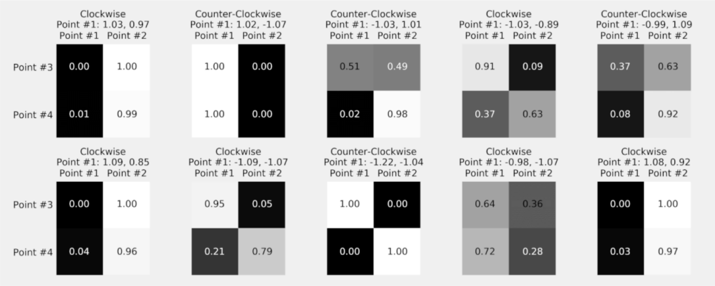
code/EncoderDecoderAttn.py

Visualising Attention Scores III

$target$	$source$ x_0	x_1
x_2	α_{20}	α_{21}
x_3	α_{30}	α_{31}

$$\Longrightarrow$$

$target$	$source$ x_0	x_1
x_2	0.0011	0.9989
x_3	0.0146	0.9854



Multi-headed Attention

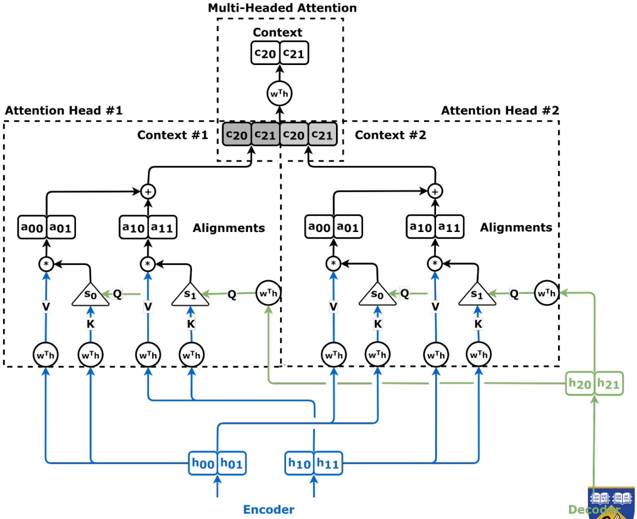
Multi-headed Attention

We can use several attention mechanisms at once, each one being referred to as an **attention head**.

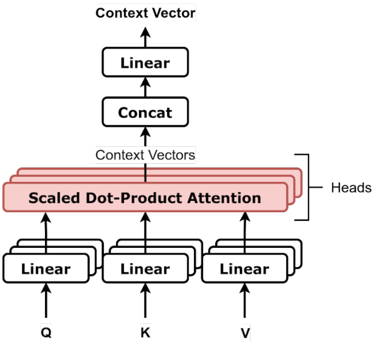
1. Each attention head will output its own context vector,
2. The context vectors will all get concatenated together and combined using a linear layer.
3. In the end, the multi-headed attention mechanism will still output a single context vector.

THE UNIVERSITY OF WESTERN AUSTRALIA

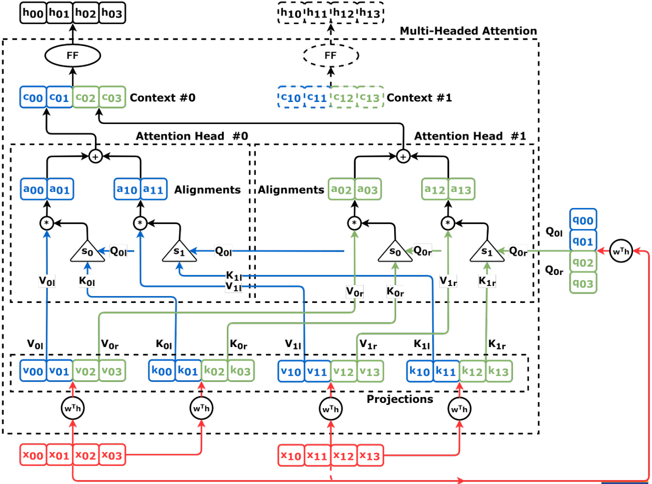
Multi-headed attention flow



A simplified illustration of Multi-headed Attention



Narrow Attention with Two Heads



Wide vs. Narrow Attention

Wide Attention

Each attention head gets the full "hidden state" and produces a context vector of the same size.

- **Pros:** likely yield better models compared to using narrow attention on the same number of dimensions
- **Cons:** cannot deal with large dimensions

Narrow Attention

Each attention head will get a chunk of the affine transformation of the "hidden state" to work with. Note it is not a chunk of the original "hidden state", but of its transformation.

- Pros: can deal with large dimensions
- Cons: may not be as good as Wide Attention, but Transformer uses narrow attention.

Multi-headed Attention in Code I

```
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, d_model,
                    input_dim=None, proj_values=True):
        super().__init__()
        self.linear_out = nn.Linear(n_heads*d_model,d_model)
        self.attn_heads = nn.ModuleList(
            [Attention(d_model, input_dim=input_dim,
                        proj_values=proj_values)
                            for _ in range(n_heads)])

    def init_keys(self, key):
        for attn in self.attn_heads:
            attn.init_keys(key)

    @property
    def alphas(self):
        # Shape: n_heads, N, 1, L (source)
        return torch.stack([attn.alphas
                        for attn in self.attn_heads], dim=0)

    def output_function(self, contexts):
```

Multi-headed Attention in Code II

```
        # N, 1, n_heads * D
        concatenated = torch.cat(contexts, axis=-1)
        # Linear transf. to go back to original dimension
        out = self.linear_out(concatenated) # N, 1, D
        return out

    def forward(self, query, mask=None):
        contexts = [attn(query, mask=mask)
                    for attn in self.attn_heads]
        out = self.output_function(contexts)
        return out
```

code/multi–headed.py

Self-Attention

Attention Is All You Need

Some Radical Notion!

what about replacing the recurrent layer with an attention mechanism?

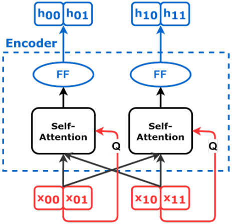
- This is the main proposition of the famous "Attention Is All You Need" from Google Brain [2].
- The 2017 paper introduced the **Transformer** architecture, based on a self attention mechanism, that has completely dominate the NLP landscape and now take over Computer Vision.

With self-attention, We can use another, separate, attention mechanism to replace the encoder and the decoder too!)

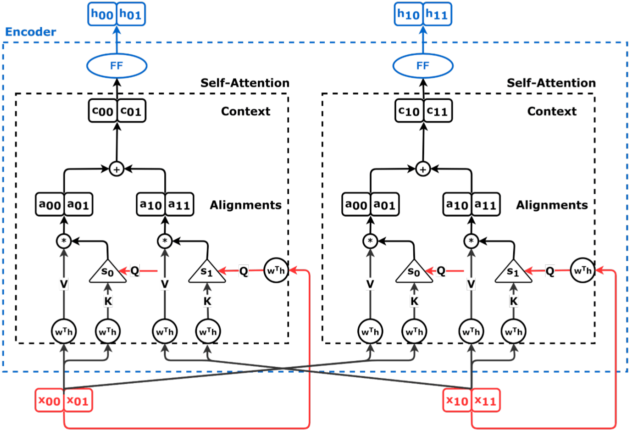
"I pity the fool using recurrent layers." 😄

Encoder with Self-Attention

- the source sequence (in red) works as "keys" (K), "values" (V), and "queries (Q)" as well.
- Now we have "queries", plural, instead of "query".
  - In the encoder, each data point is a "query" (the red arrow entering the self-attention mechanism from the side), and
  - each produces its own context vector using alignment vectors for every data point in the source sequence, including itself.
  - This means it is possible for a data point to generate a context vector that is only paying attention to itself.



Encoder with Self-Attention Details



Encoder with Self-Attention Details Explained

1. The transformed coordinates of the first data point $(x_{00}, x_{01})$are used as "query" (Q).
2. This "query" (Q) is paired, independently, with each one of the two "keys" (K).
3. One of the keys (K) corresponding to a different transformation of the same input data point (x_{00}, x_{01}), the other a transformation of the second data point (x_{10}, x_{11}).
4. The pairing will result in two attention scores (alphas) that, multiplied by their corresponding "values" (V), are added up to become the context vector.

$$\alpha_{00}, \alpha_{01} = softmax(\frac{Q_0 \cdot K_0}{\sqrt{2}}, \frac{Q_0 \cdot K_1}{\sqrt{2}})$$

$$context\ vector_0 = \alpha_{00} V_0 + \alpha_{01} V_1$$

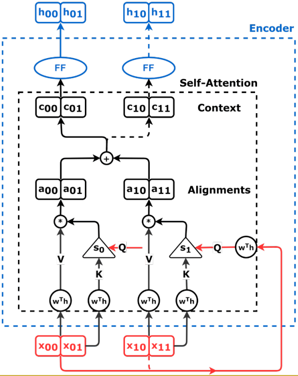
5. Next, the context vector goes through the feed-forward network, and the first hidden state is born

A Simplified Illustration

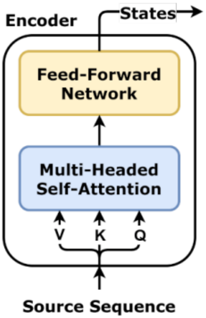
Context Vector - A function of Q

The context vector (and thus the "hidden state") associated with a data point is basically a function of the corresponding "query" (Q).

This is because everything else ("keys" (K), "values" (V), and the parameters of the self-attention mechanism) is held constant for all queries.



Even more Simplified Illustration for Self-Attention



One great news is that the process is not sequential!

These operations can be parallelized to generate all "hidden states" at once, which is much more efficient than using a recurrent layer that is sequential in nature.

Encoder with Self Attention in code I

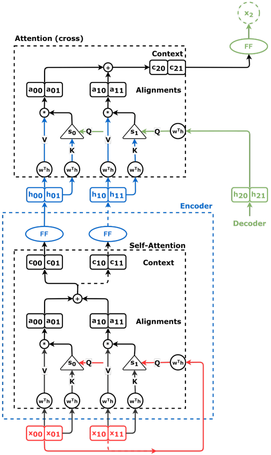
```python
class EncoderSelfAttn(nn.Module):


    def __init__(self, n_heads, d_model, ff_units,
                                            n_features=None):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.n_features = n_features
        self.self_attn_heads = MultiHeadAttention(
                                            n_heads, d_model,
                                            input_dim=n_features)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Linear(ff_units, d_model),
        )
```

Encoder with Self Attention in code II

```python

    def forward(self, query, mask=None):
        self.self_attn_heads.init_keys(query)
        att = self.self_attn_heads(query, mask)
        out = self.ffn(att)
        return out
```

code/encoder_self_attn.py

Encoder with self- and cross-attentions

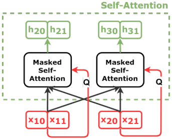


Encoder with self- and cross-attention

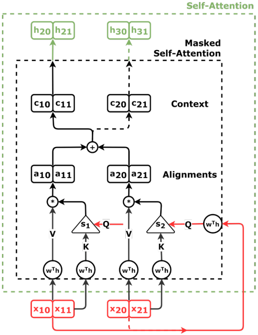
Cross-Attention

The attention mechanism that we have seen before self-attention, where the the decoder provided a "query" (Q) which served not only as input but also got concatenated to the resulting context vector, is called **cross-attention**.

Decoder with Self-Attention



Decoder with Self-Attention - Simplified



Decoder with Self-Attention in Code I

There is one main difference (in the code) between the encoder and the decoder:

The decoder includes a cross-attention mechanism

```
class DecoderSelfAttn(nn.Module):


    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.ff_units = ff_units
        self.n_features = d_model
                 if n_features is None else n_features
        self.self_attn_heads = MultiHeadAttention(
                                  n_heads, d_model,
                          input_dim=self.n_features)
        self.cross_attn_heads = MultiHeadAttention(
                                  n_heads, d_model)
```

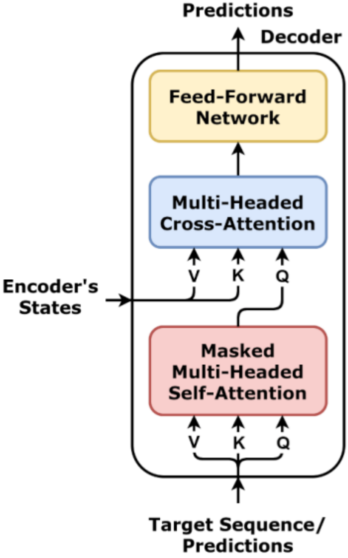
Decoder with Self-Attention in Code II

```
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),
            nn.ReLU(),
            nn.Linear(ff_units, self.n_features),
        )

    def init_keys(self, states):
        self.cross_attn_heads.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        self.self_attn_heads.init_keys(query)
        att1 = self.self_attn_heads(query, target_mask)
        att2 = self.cross_attn_heads(att1, source_mask)
        out = self.ffn(att2)
        return out
```

code/decoder_self_attn.py

Decoder with Self-Attention Schematic Diagram

- There is one small difference in the self-attention architecture between encoder and decoder: the feed-forward network sits atop the cross-attention mechanism (not depicted in the figure above) instead of the self-attention mechanism.
- The feed-forward network also maps the decoder's output from the dimensionality of the model (`d_model`) back to the number of features, thus yielding predictions.



How Decoder with Self-Attention Works

- **First input** is (x_{10}, x_{11}), the last known element of the source sequence, as usual.
- **Source mask** is the same mask used to ignore padded data points in the encoder.

What we also need are:

- Subsequent Inputs and Teacher Forcing
- Target Mask

Subsequent Inputs and Teacher Forcing

The shifted target sequence

The shifted target sequence is defined as a sequence that includes the last known element of the source sequence and all elements in the target sequence **but the last one**.

The **shifted target sequence** was already used when we discussed *teacher forcing*.

- In teacher forcing, at every step (after the first one), it randomly chose as the input to the subsequent step either an actual element from that sequence or a prediction.
- It worked very well with recurrent layers that were sequential in nature.

Problem with Attention Scores in Decoder

We are using the whole *shifted target sequence* at once as the "query" argument of the decoder.

$$\alpha_{21}, \alpha_{22} = softmax(\frac{Q_1 \cdot K_1}{\sqrt{2}}, \frac{Q_1 \cdot K_2}{\sqrt{2}})$$

The problem is that it is using a "key" (K2) and a "value" (V2) that are transformations of the data point it is trying to predict. We should not allow the model to **cheat** by peeking into the future.

$$context\ \ vector_2 = \alpha_{21} V_1 + \alpha_{22} V_2$$

$$\alpha_{31}, \alpha_{32} = softmax(\frac{Q_2 \cdot K_1}{\sqrt{2}}, \frac{Q_2 \cdot K_2}{\sqrt{2}})$$

$$context\ \ vector_3 = \alpha_{31} V_1 + \alpha_{32} V_2$$

Decoder Attention Scores and Target Mask

- For the decoder, the shape of the alphas attribute is given by $(N, L_{target}, L_{target})$ since it is looking at itself.
- Any alphas above the diagonal are, literally, cheating codes.

	source	
target	x_1	x_2
h_2	α_{21}	α_{22}
h_3	α_{31}	α_{32}

We need to force the self-attention mechanism to ignore them.

Target Mask

The purpose of the target mask is to zero attention scores for "future" data points.

	source	
target	x_1	x_2
h_2	α_{21}	0
h_3	α_{31}	α_{32}

THE UNIVERSITY OF WESTERN AUSTRALIA

Frame Title

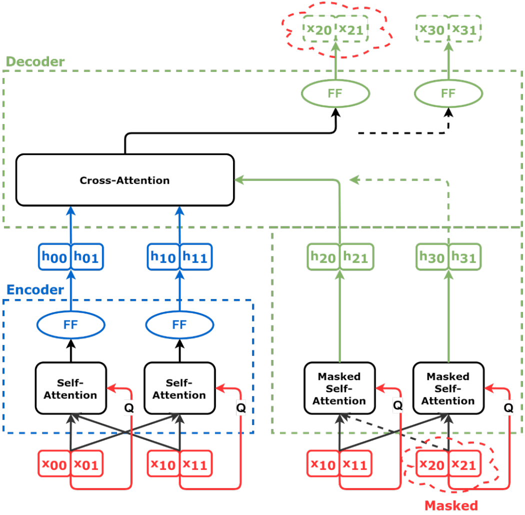
```python
def subsequent_mask(size):
    attn_shape = (1, size, size)
    # torch.triu returns the upper triangular part of a matrix
    subsequent_mask = (1 - torch.triu(torch.ones(attn_shape),
                              diagonal=1)).bool()
    return subsequent_mask
```

code/subsequent_mask.py

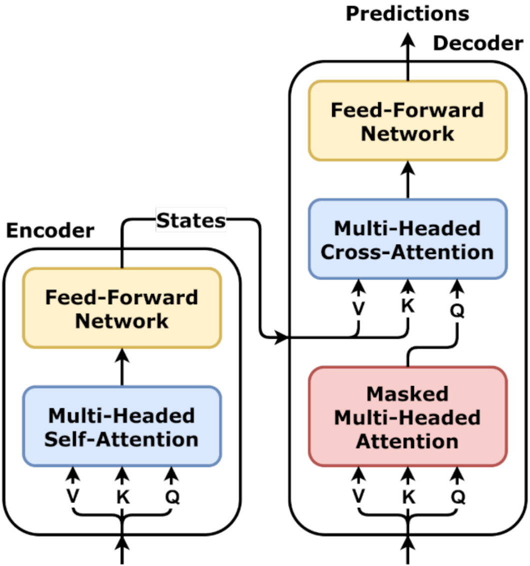
```
tensor([[[ True, False],
         [ True,  True]]])
```

- We must use this mask while querying the decoder to prevent it from cheating.
- You can choose to use an additional mask to "hide" more data from the decoder if you wish, but the subsequent mask is a strong requirement of the self-attention decoder.

Encoder Decoder with Self Attention



Encoder Decoder Self-Attention Schematic Diagram



Encoder Decoder Self-Attention Model in Code I

```
class EncoderDecoderSelfAttn(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.subsequent_mask(self.target_len)

    @staticmethod
    def subsequent_mask(size):
        attn_shape = (1, size, size)
        subsequent_mask = (1-torch.triu(torch.ones(attn_shape),
                                        diagonal=1))
        return subsequent_mask
```

Encoder Decoder Self-Attention Model in Code II

```
    def encode(self, source_seq, source_mask):
        # Encodes the source sequence and uses the result
        # to initialize the decoder
        encoder_states = self.encoder(source_seq, source_mask)
        self.decoder.init_keys(encoder_states)

    def decode(self, shifted_target_seq, source_mask=None,
                                         target_mask=None):
        # Decodes/generates a sequence using the shifted
        # target sequence (masked) - used in TRAIN mode
        outputs = self.decoder(shifted_target_seq,
                               source_mask=source_mask,
                               target_mask=target_mask)
        return outputs
```

Encoder Decoder Self-Attention Model in Code III

```
    def predict(self, source_seq, source_mask):
        # Decodes/generates a sequence using one input
        # at a time - used in EVAL mode
        inputs = source_seq[:, -1:]
        for i in range(self.target_len):
            out = self.decode(inputs, source_mask, self.trg_masks[:, :i+1, :i+1])
            out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
            inputs = out.detach()
        outputs = inputs[:, 1:, :]
        return outputs

    def forward(self, X, source_mask=None):
        # Sends the mask to the same device as the inputs
        self.trg_masks = self.trg_masks.type_as(X).bool()
        # Slices the input to get source sequence
        source_seq = X[:, :self.input_len, :]
        # Encodes source sequence AND initializes decoder
        self.encode(source_seq, source_mask)
        if self.training:
            # Slices the input to get the shifted target seq
```

Encoder Decoder Self-Attention Model in Code IV

```
                shifted_target_seq = X[:, self.input_len-1:-1, :]
                # Decodes using the mask to prevent cheating
                outputs = self.decode(shifted_target_seq,
                        source_mask, self.trg_masks)
            else:
                # Decodes using its own predictions
                outputs = self.predict(source_seq, source_mask)

            return outputs
```

code/EncoderDecoderSelfAttn.py

Code Explained

- **encode**: takes the source sequence and mask and encodes it into a sequence of states that is immediately used to initialize the ”keys” (and ”values”) in the decoder
- **decode**: takes the shifted target sequence and both source and target masks to generate a target sequence - it is used for training only
- **predict**: takes the source sequence, the source mask, and uses a subset of the target mask to actually predict an unknown target sequence - it is used for evaluation/prediction only
- **forward**: it splits the input into the source and shifted target sequences (if available), encodes the source sequence, and calls either decode or predict according to the model's mode (train or eval).

Take Aways

- Understand the attention mechanism
- Understand how context vectors are calculated
- Cross-Attention vs. Self-Attention

References

[1] Daniel Voigt Godoy. **Deep Learning with PyTorch Step-by-Step: A Beginner's Guide**. Published at: http://leanpub.com/pytorch, Github Repo: https://github.com/dvgodoy/PyTorchStepByStep, 2021.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, **Advances in Neural Information Processing Systems**, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

https://github.com/vdumoulin/conv$_a$*rithmetic/blob/master/README.md*