Chapter 9. Part II

Sequence-to-Sequence

Profº Drº Ivanovitch Medeiros Dantas da Silva

Antônio Paulo Muniz Lemos

José Lindenberg de Andrade

Maria da Guia Torres da Silva

Summary

I) Self-Attention

II) Target mask

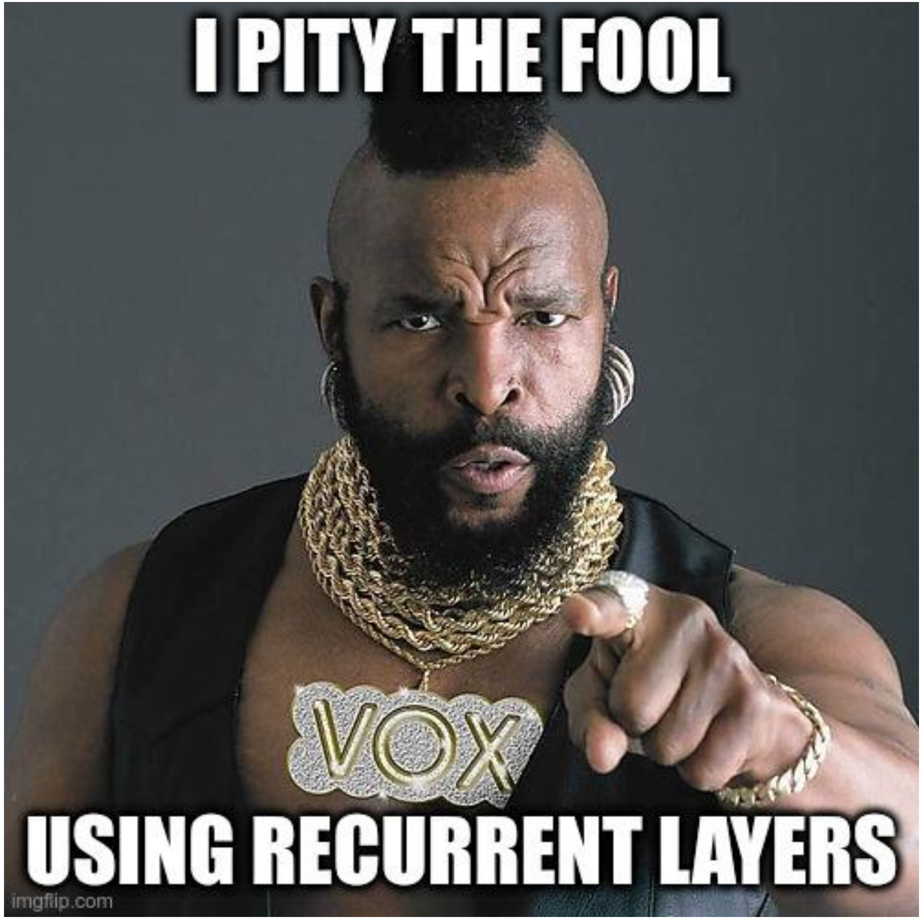
III) Positional encoding

Self-Attention

"What if we replaced the recurrent layer with an attention mechanism?

This is the main proposal of the famous paper 'Attention Is All You Need' by Vaswani, A., et al.

The recurrent layer in the encoder received the source sequence and generated hidden states one by one. But we don’t need to generate hidden states in this manner. We can use a separate attention mechanism to replace the encoder (and, wait, the decoder too!)."



Encoder

An encoder using self-attention!

Each context vector produced by the self-attention mechanism passes through a feed-forward network to generate a 'hidden state' as output.

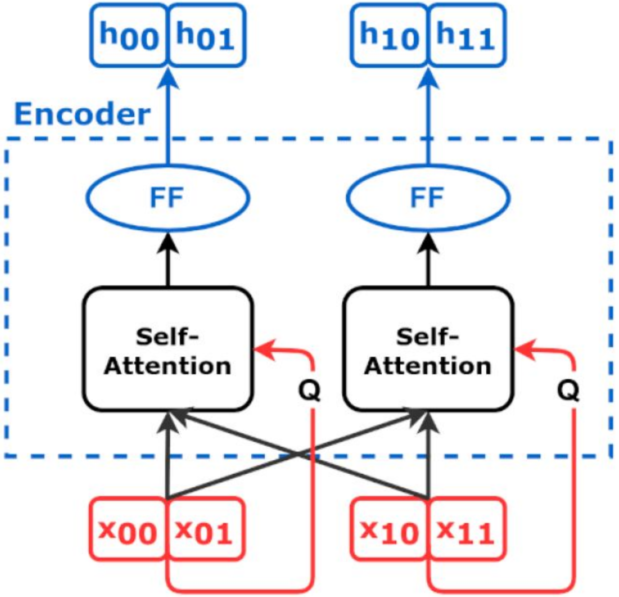


Figure 1 - Encoder with self-attention (simplified)

Encoder

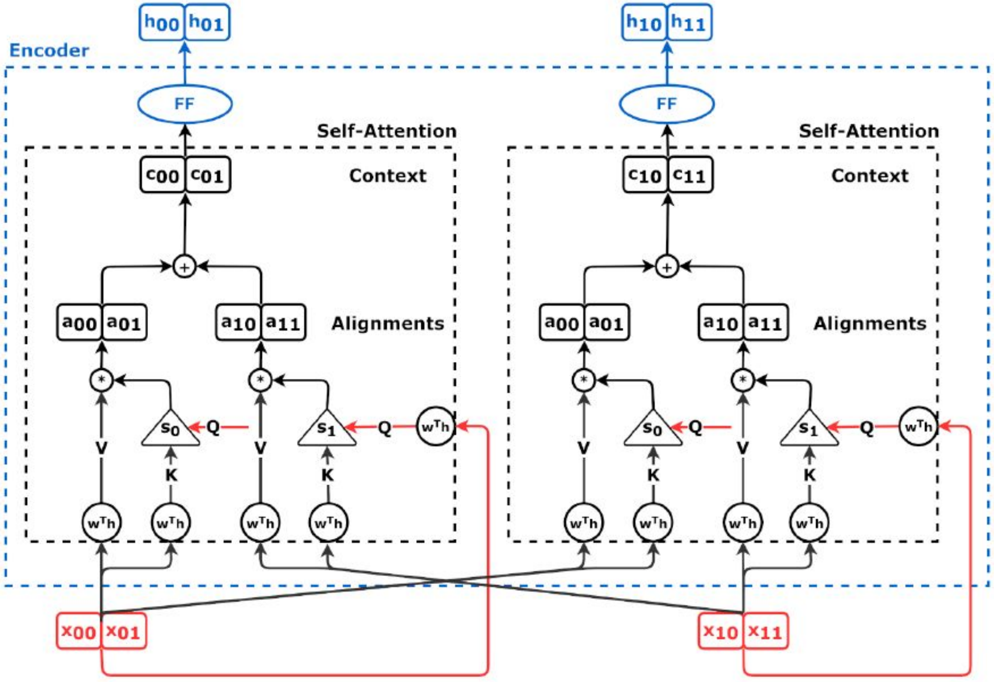


Figure 2 - Encoder with self-attention

Hidden states (h00, h01); Values (V), Query (Q); Keys (K); Data points (x00, x01, x10, x11)

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

$$\text{context vector}_0 = \alpha_{00} V_0 + \alpha_{01} V_1$$

Equation 1 - Context vector for first input (x_0)

$$\alpha_{10}, \alpha_{11} = \text{softmax}\left(\frac{Q_1 \cdot K_0}{\sqrt{2}}, \frac{Q_1 \cdot K_1}{\sqrt{2}}\right)$$

$$\text{context vector}_1 = \alpha_{10} V_0 + \alpha_{11} V_1$$

Equation 2 - Context vector for second input (x_1)

Encoder

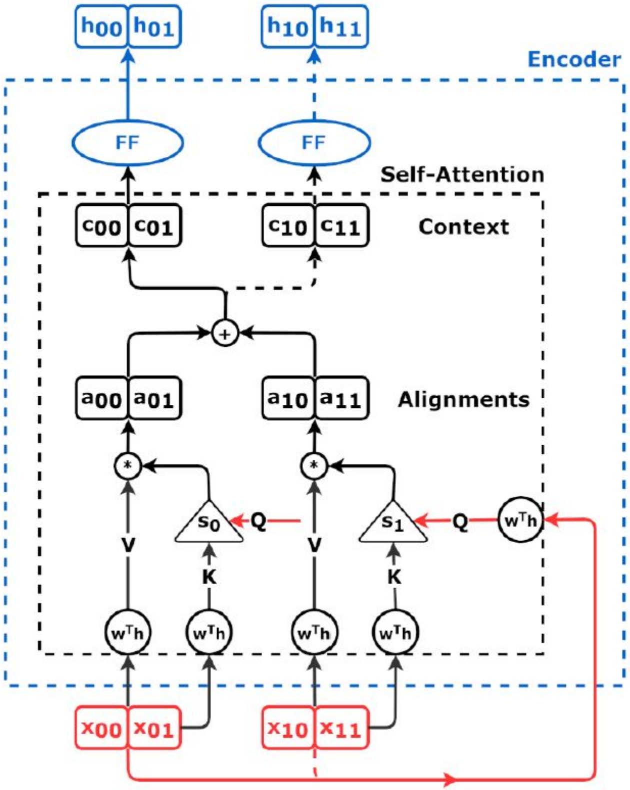


Figure 3 - only one self-attention mechanism, assuming it will be fed a different "query" (Q) every time.

target	source x_0	x_1
h_0	α_{00}	α_{01}
h_1	α_{10}	α_{11}

Equation 3 - Attention scores

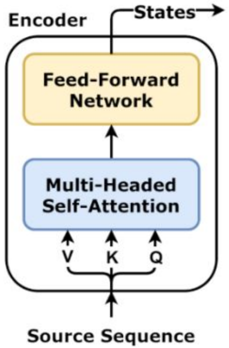


Figure 4 - Encoder with self-attention (diagram)

Encoder + Self-Attention

```python
class EncoderSelfAttn(nn.Module):
    # Classe que implementa um Codificador com Auto-Atenção (Self-Attention)

    def __init__(self, n_heads, d_model, ff_units, n_features=None):
        # Método de inicialização do modelo
        # n_heads: número de cabeças de atenção (multi-head attention)
        # d_model: dimensionalidade do modelo
        # ff_units: número de unidades na camada feed-forward
        # n_features: número de características (features) de entrada, se aplicável

        super().__init__()  # Inicializa a classe pai nn.Module
        self.n_heads = n_heads  # Armazena o número de cabeças de atenção
        self.d_model = d_model  # Armazena a dimensionalidade do modelo
        self.ff_units = ff_units  # Armazena o número de unidades na feed-forward
        self.n_features = n_features  # Armazena o número de características (opcional)

        # Define a camada de Auto-Atenção com múltiplas cabeças (multi-head self-attention)
        self.self_attn_heads = MultiHeadAttention(n_heads, d_model, input_dim=n_features)

        # Define a rede feed-forward usando nn.Sequential
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),  # Primeira camada linear: reduz a dimensão de d_model para ff_units
            nn.ReLU(),  # Função de ativação ReLU
            nn.Linear(ff_units, d_model),  # Segunda camada linear: retorna à dimensão original d_model
        )

    def forward(self, query, mask=None):
        # Método de passagem à frente (forward) do modelo
        # query: entrada do modelo, geralmente o tensor de embeddings
        # mask: máscara opcional para o mecanismo de atenção (usada para ignorar certos elementos)

        # Inicializa as chaves para a atenção com base na entrada (query)
        self.self_attn_heads.init_keys(query)

        # Executa a camada de atenção e obtém o resultado
        att = self.self_attn_heads(query, mask)

        # Passa o resultado da atenção pela rede feed-forward
        out = self.ffn(att)

        # Retorna o resultado final
        return out
```

Create an encoder and feed it with a source sequence:

```python
torch.manual_seed(11)
encself = EncoderSelfAttn(n_heads=3, d_model=2,
                          ff_units=10, n_features=2)
query = source_seq
encoder_states = encself(query)
encoder_states
```

Output:

```
tensor([[[-0.0498,  0.2193],
         [-0.0642,  0.2258]]], grad_fn=<AddBackward0>)
```

Produced a sequence of states that will be the input to the cross-attention mechanism used by the decoder.

Cross-Attention

In the figure, self-attention as the encoder, cross-attention on top of it, and the modifications in the decoder.

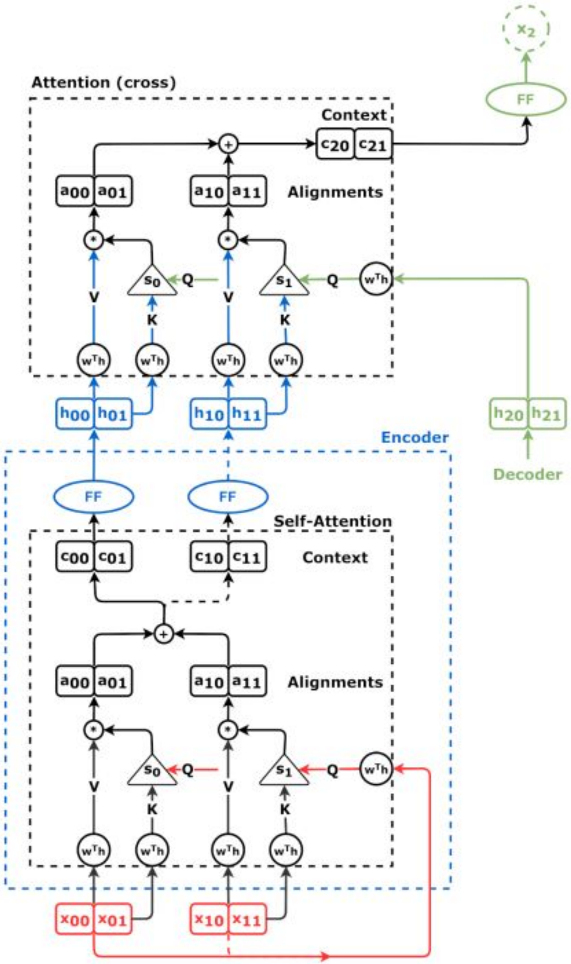


Figure 5 - Encoder with self- and cross-attentions

Decoder

Decoder + Self-Attention

Encoder vs Decoder (self-attention):

- **<u>Encoder: Feed-forward network - self-attention.</u>**
- **<u>Decoder: Feed-forward network - cross-attention, maps the decoder output and produces predictions.</u>**

```python
class DecoderSelfAttn(nn.Module):
    # Classe que implementa um Decodificador com Auto-Atenção e Atenção Cruzada (Self-Attention + Cross-Attention)

    def __init__(self, n_heads, d_model, ff_units, n_features=None):
        # Método de inicialização do modelo
        super().__init__()  # Inicializa a classe pai nn.Module
        self.n_heads = n_heads  # Armazena o número de cabeças de atenção
        self.d_model = d_model  # Armazena a dimensionalidade do modelo
        self.ff_units = ff_units  # Armazena o número de unidades na feed-forward
        self.n_features = d_model if n_features is None else n_features  # Define n_features; se não for especificado, usa d_model como padrão

        # Define a camada de Auto-Atenção (Self-Attention) com múltiplas cabeças
        self.self_attn_heads = MultiHeadAttention(n_heads, d_model, input_dim=self.n_features)

        # Define a camada de Atenção Cruzada (Cross-Attention) para conectar a entrada do decodificador com a saída do codificador
        self.cross_attn_heads = MultiHeadAttention(n_heads, d_model)

        # Define a rede feed-forward usando nn.Sequential
        self.ffn = nn.Sequential(
            nn.Linear(d_model, ff_units),  # Primeira camada linear: reduz a dimensão de d_model para ff_units
            nn.ReLU(),                     # Função de ativação ReLU
            nn.Linear(ff_units, self.n_features),  # Segunda camada linear: retorna à dimensão original (ou n_features)
            )
    def init_keys(self, states):
        # Método para inicializar as chaves na atenção cruzada com base nos estados do codificador
        self.cross_attn_heads.init_keys(states)

    def forward(self, query, source_mask=None, target_mask=None):
        # Método de passagem à frente (forward) do modelo

        # Inicializa as chaves para a auto-atenção com base na entrada (query)
        self.self_attn_heads.init_keys(query)

        # Executa a auto-atenção (Self-Attention) no próprio decodificador
        att1 = self.self_attn_heads(query, target_mask)

        # Executa a atenção cruzada (Cross-Attention), ligando a saída da auto-atenção aos estados do codificador
        att2 = self.cross_attn_heads(att1, source_mask)

        # Passa o resultado da atenção cruzada pela rede feed-forward
        out = self.ffn(att2)
        return out
```

Decoder

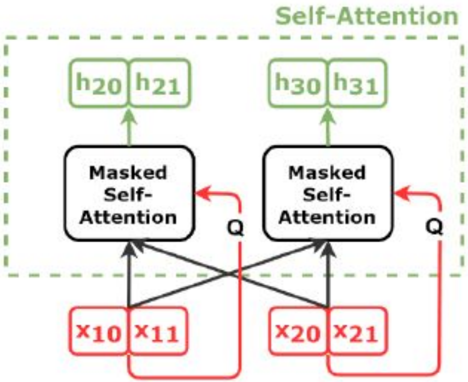


Figure 6 - Decoder with self-attention (simplified)

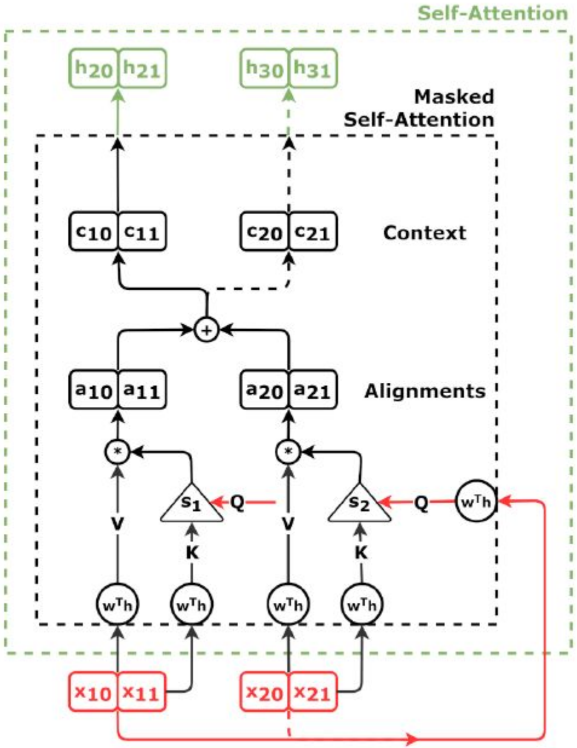


Figure 7 - Decoder with self-attention

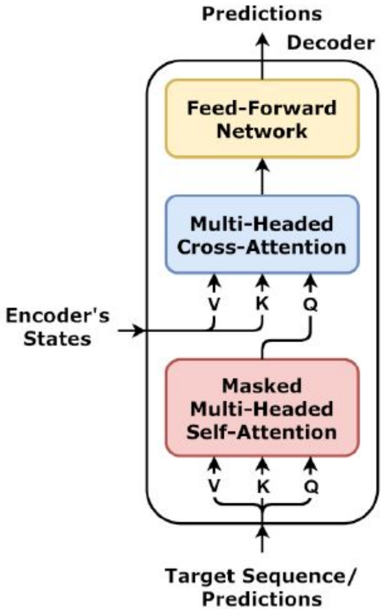


Figure 8 - Decoder with self- and cross-attentions (diagram)

Subsequent Inputs and Teacher Forcing

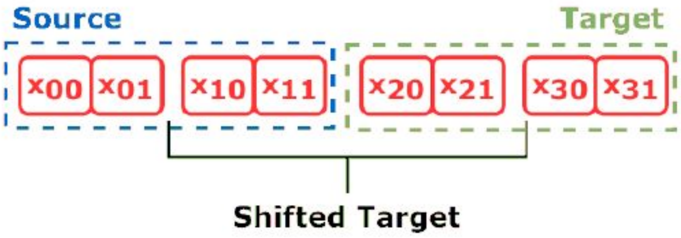


Figure 9 - Shifted target sequence

```
shifted_seq = torch.cat([source_seq[:, -1:],
                         target_seq[:, :-1]], dim=1)
```

One of the advantages of self-attention over recurrent layers is that the operations can be parallelized. There is no longer a need to do anything sequentially, including teacher forcing. This means that we are using the entire target sequence shifted at once as the 'query' input to the decoder.

Attention Scores

To understand the problem, let's look at the context vector that will result in the first 'hidden state' produced by the decoder, which, in turn, will lead to the first prediction:

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

$$\text{context vector}_2 = \alpha_{21} V_1 + \alpha_{22} V_2$$

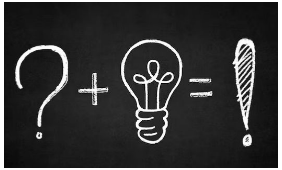
Equation 4 - Context vector for the first target

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.

In other words, the model is being allowed to cheat by peeking into the future because we are giving it all the data points in the target sequence, except for the last one.

Target Mask (Training)

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



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

Equation 6 - Decoder's attention scores

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

Equation 7 - Decoder's (masked) attention scores

Target Mask

Subsequent Mask

```
def subsequent_mask(size):   # Define the function
    attn_shape = (1, size, size)
    subsequent_mask = (
        1 - torch.triu(torch.ones(attn_shape), diagonal=1) # Array
    ).bool()
    return subsequent_mask
```

```
subsequent_mask(2) # 1, L, L
```

Output

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

In practice:

```
torch.manual_seed(13) # Define the random seed
decself = DecoderSelfAttn(n_heads=3, d_model=2,
                          ff_units=10, n_features=2)
decself.init_keys(encoder_states)

query = shifted_seq
out = decself(query, target_mask=subsequent_mask(2))
                                  # Subsequent mask is applied
decself.self_attn_heads.alphas
```

Output

```
tensor([[[[1.0000, 0.0000],
          [0.4011, 0.5989]]],


        [[[1.0000, 0.0000],
          [0.4264, 0.5736]]],


        [[[1.0000, 0.0000],
          [0.6304, 0.3696]]]])
```

Target Mask (Evaluation/Prediction)

```
inputs = source_seq[:, -1:]
trg_masks = subsequent_mask(1)
out = decself(inputs, trg_masks)
out
```

Output

```
tensor([[[0.4132, 0.3728]]], grad_fn=<AddBackward0>)
```

$$1^{\text{st}}\ \text{Step}\left\{\begin{array}{c|c} & \text{source} \\ \text{target} & x_1 \\ \hline h_2 & \alpha_{21} \end{array}\right.$$

Equation 8 - Decoder's (masked) attention scores for the first target

```
inputs = torch.cat([inputs, out[:, -1:, :]], dim=-2)
inputs
```

Output

```
tensor([[[-1.0000,  1.0000],
         [ 0.4132,  0.3728]]], grad_fn=<CatBackward>)
```

$$2^{\text{nd}}\ \text{Step}\left\{\begin{array}{c|cc} & \text{source} & \\ \text{target} & x_1 & x_2 \\ \hline h_2 & \alpha_{21} & 0 \\ h_3 & \alpha_{31} & \alpha_{32} \end{array}\right.$$

Equation 9 - Decoder's (masked) attention scores for the second target

Target Mask (Evaluation/Prediction)

The decoder with self-attention expects the complete sequence as the 'query', then we concatenate the prediction with the previous 'query'.

Target Mask (Evaluation/Prediction)

Predicted coordinates of both x_2 and x_3 :

```
trg_masks = subsequent_mask(2)
out = decself(inputs, trg_masks)
out
```

Output

```
tensor([[[0.4137, 0.3727],
         [0.4132, 0.3728]]], grad_fn=<AddBackward0>)
```

Concatenating prediction to sequence:

```
inputs = torch.cat([inputs, out[:, -1:, :]], dim=-2)
inputs
```

Output

```
tensor([[[-1.0000,  1.0000],
         [ 0.4132,  0.3728],
         [ 0.4132,  0.3728]]], grad_fn=<CatBackward>)
```

Decoder predictions:

```
inputs[:, 1:]
```

Output

```
tensor([[[0.4132, 0.3728],
         [0.4132, 0.3728]]], grad_fn=<SliceBackward>)
```

Encoder + Decoder + Self-Attention

The encoder and decoder come together again, each using self-attention to calculate their corresponding 'hidden states'.

In addition to using masked self-attention, the decoder uses cross-attention to make predictions.

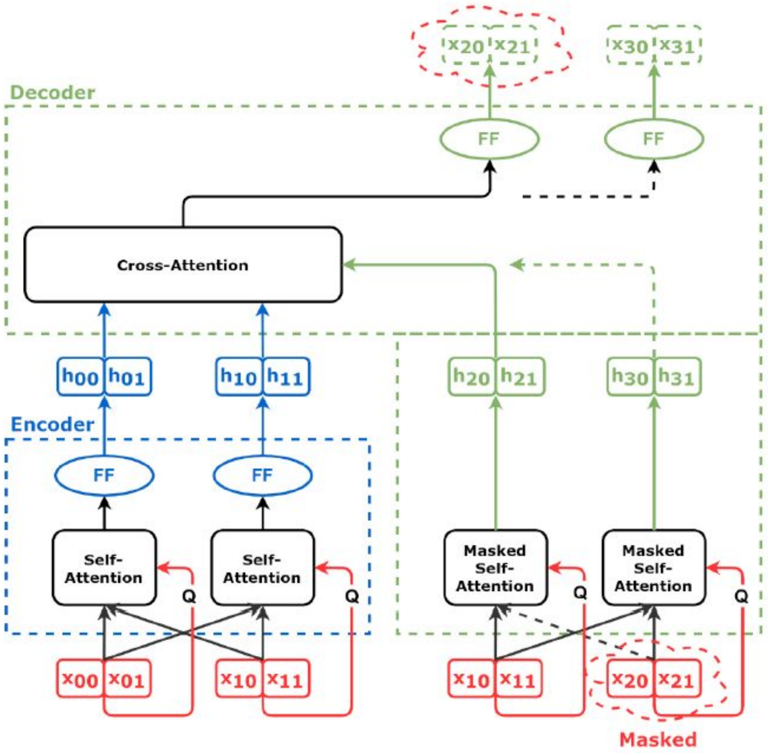
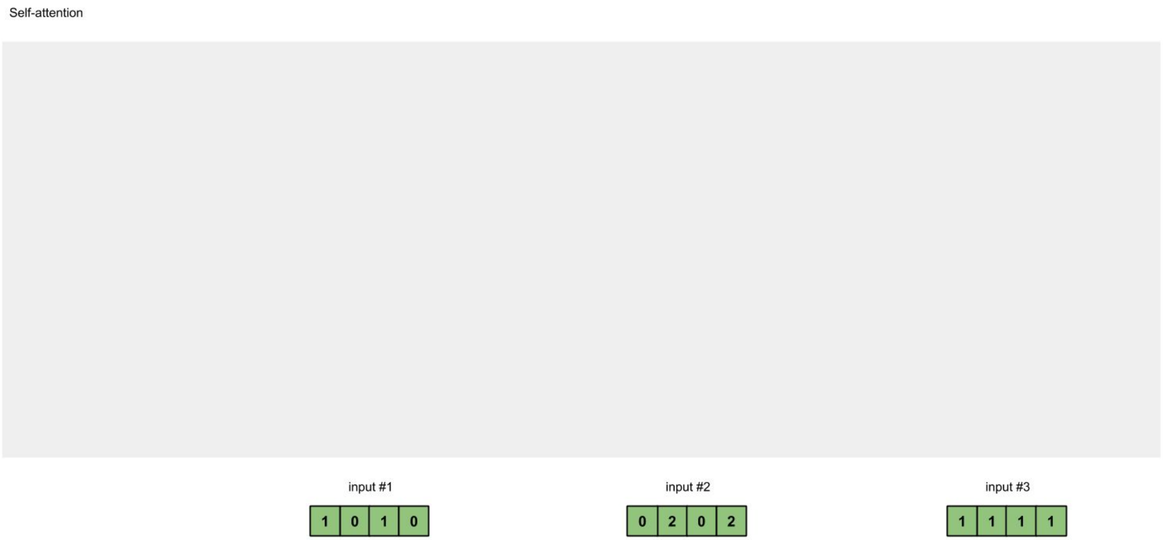


Figure 10 - Encoder + decoder + attention (simplified)

Self-Attention



Model Configuration & Training

Model Configuration

```
torch.manual_seed(23) # Define the random seed
encself = EncoderSelfAttn(n_heads=3, d_model=2, # Create encoder
                          ff_units=10, n_features=2)
decself = DecoderSelfAttn(n_heads=3, d_model=2,  # Create decoder
                          ff_units=10, n_features=2)
model = EncoderDecoderSelfAttn(encself, decself,  # Match e/d
                               input_len=2, target_len=2)
loss = nn.MSELoss() # Loss function
optimizer = optim.Adam(model.parameters(), lr=0.01) # Algorithm
```

Model Training

```
sbs_seq_selfattn = StepByStep(model, loss, optimizer)
sbs_seq_selfattn.set_loaders(train_loader, test_loader)
sbs_seq_selfattn.train(100)
```

```
fig = sbs_seq_selfattn.plot_losses() # graph of losses
```

Losses

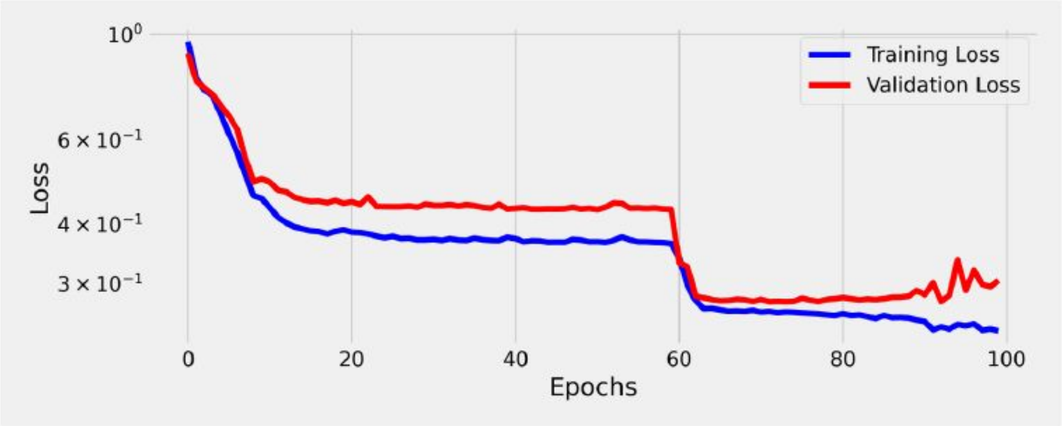


Figure 12 - Losses-encoder + decoder + self-attention

Visualizing Predictions

```
fig = sequence_pred(sbs_seq_selfattn, full_test, test_directions)
```

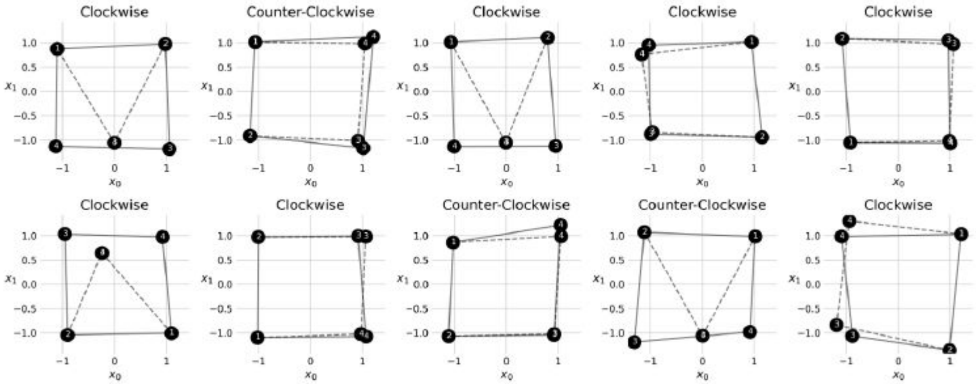
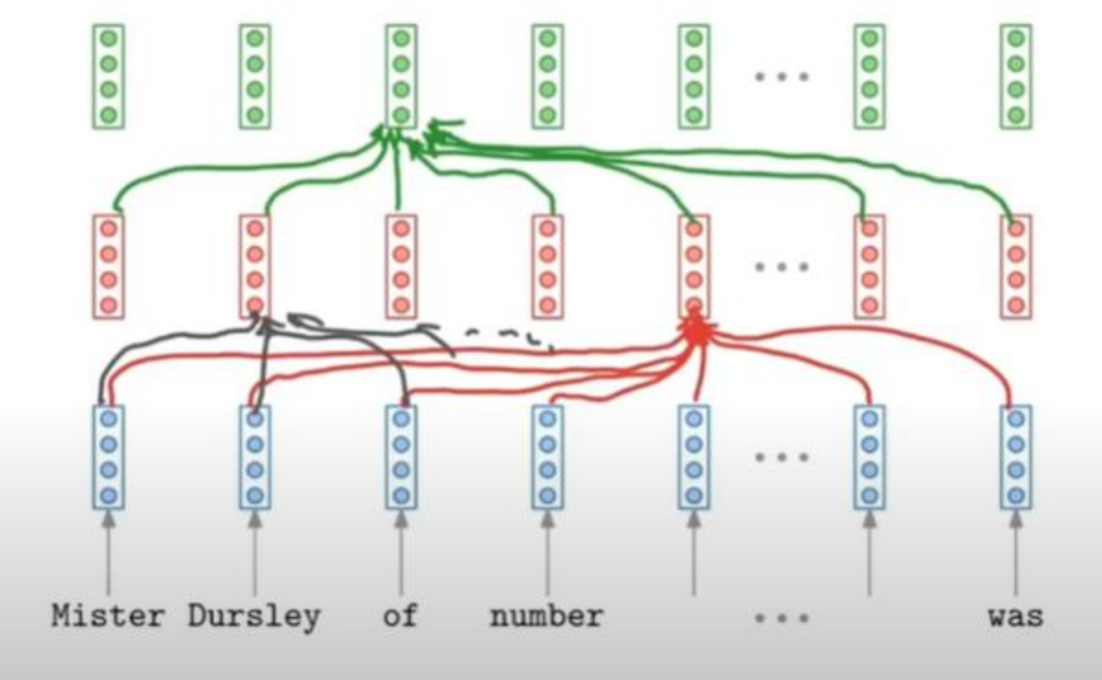


Figure 13 - Predictions-encoder + decoder + self-attention

"*Can we fix it?*"

Positional Encoding (PE)



- Order

Positional Encoding (PE)

Position	0	1	2	3	4	5	6	7
(Pos mod 4)/4	0.00	0.25	0.50	0.75	0.00	0.25	0.50	0.75
(Pos mod 5)/5	0.00	0.20	0.40	0.60	0.80	0.00	0.20	0.40
(Pos mod 7)/7	0.00	0.14	0.29	0.43	0.57	0.71	0.86	0.00

Figure 9.41 - Combining results for different modules

$$\begin{array}{ccccc} 3 & & 2 & & \text{Diff} \\ \begin{bmatrix} 0.75 \\ 0.60 \\ 0.43 \end{bmatrix} & - & \begin{bmatrix} 0.50 \\ 0.40 \\ 0.29 \end{bmatrix} & = & \begin{bmatrix} 0.25 \\ 0.20 \\ 0.14 \end{bmatrix} \end{array}$$

Distance = ||Diff|| = 0.35

$$\begin{array}{ccccc} 4 & & 3 & & \text{Diff} \\ \begin{bmatrix} 0.00 \\ 0.80 \\ 0.57 \end{bmatrix} & - & \begin{bmatrix} 0.75 \\ 0.60 \\ 0.43 \end{bmatrix} & = & \begin{bmatrix} -0.75 \\ 0.20 \\ 0.14 \end{bmatrix} \end{array}$$

Distance = ||Diff|| = 0.79

Figure 9.42 - Inconsistent distances

Positional Encoding (PE)

Position	0	1	2	3	4	5	6	7
sine (base 4)	0.00	1.00	0.00	-1.00	0.00	1.00	0.00	-1.00
cosine (base 4)	1.00	0.00	-1.00	0.00	1.00	0.00	-1.00	0.00
sine (base 5)	0.00	0.95	0.59	-0.59	-0.95	0.00	0.95	0.59
cosine (base 5)	1.00	0.31	-0.81	-0.81	0.31	1.00	0.31	-0.81
sine (base 7)	0.00	0.78	0.97	0.43	-0.43	-0.97	-0.78	0.00
cosine (base 7)	1.00	0.62	-0.22	-0.90	-0.90	-0.22	0.62	1.00

Figure 9.45 - Representing degrees using sine and cosine

$$PE_{pos,\ 2d} = \sin\left(\frac{1}{10000^{\frac{2d}{d_{model}}}} pos\right)$$

$$PE_{pos,\ 2d+1} = \cos\left(\frac{1}{10000^{\frac{2d}{d_{model}}}} pos\right)$$

Equation 9.21 - Positional encoding

$$\begin{array}{c} 3 \\ \begin{bmatrix} -1.00 \\ 0.00 \\ -0.59 \\ -0.81 \\ 0.43 \\ -0.90 \end{bmatrix} \end{array} - \begin{array}{c} 2 \\ \begin{bmatrix} 0.00 \\ -1.00 \\ 0.59 \\ -0.81 \\ 0.97 \\ -0.22 \end{bmatrix} \end{array} = \begin{array}{c} \text{Diff} \\ \begin{bmatrix} -1.00 \\ 1.00 \\ -1.18 \\ 0.00 \\ -0.54 \\ -0.68 \end{bmatrix} \end{array}$$

Distance = ||Diff|| = 2.03

$$\begin{array}{c} 4 \\ \begin{bmatrix} 0.00 \\ 1.00 \\ -0.95 \\ 0.31 \\ -0.43 \\ -0.90 \end{bmatrix} \end{array} - \begin{array}{c} 3 \\ \begin{bmatrix} -1.00 \\ 0.00 \\ -0.59 \\ -0.81 \\ 0.43 \\ -0.90 \end{bmatrix} \end{array} = \begin{array}{c} \text{Diff} \\ \begin{bmatrix} 1.00 \\ 1.00 \\ -0.36 \\ 1.12 \\ -0.87 \\ 0.00 \end{bmatrix} \end{array}$$

Distance = ||Diff|| = 2.03

Figure 9.46 - Consistent distances

Positional Encoding (PE)

```
max_len = 10
d_model = 8
position = torch.arange(0, max_len).float().unsqueeze(1)
angular_speed = torch.exp(
  torch.arange(0, d_model, 2).float() * (-np.log(10000.0) / d_model)
)
encoding = torch.zeros(max_len, d_model)
encoding[:, 0::2] = torch.sin(angular_speed * position)
encoding[:, 1::2] = torch.cos(angular_speed * position)
```

Positional Encoding (PE)

```
np.round(encoding[0:4], 4)  # first four positions
```

Output

```
tensor([[ 0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000,
0.0000,  1.0000],
        [ 0.8415,  0.5403,  0.0998,  0.9950,  0.0100,  1.0000,
0.0010,  1.0000],
        [ 0.9093, -0.4161,  0.1987,  0.9801,  0.0200,  0.9998,
0.0020,  1.0000],
        [ 0.1411, -0.9900,  0.2955,  0.9553,  0.0300,  0.9996,
0.0030,  1.0000]])
```

```python
class PositionalEncoding(nn.Module):
    def __init__(self, max_len, d_model):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).float().unsqueeze(1)
        angular_speed = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-np.log(10000.0) / d_model)
        )
        # even dimensions
        pe[:, 0::2] = torch.sin(position * angular_speed)
        # odd dimensions
        pe[:, 1::2] = torch.cos(position * angular_speed)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x is N, L, D
        # pe is 1, maxlen, D
        scaled_x = x * np.sqrt(self.d_model)
        encoded = scaled_x + self.pe[:, :x.size(1), :]
        return encoded
```

ENCODER + DECODER + PE

```python
class EncoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_feat
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = EncoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

    def forward(self, query, mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, mask)
        return out
```

```python
class DecoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = DecoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

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

    def forward(self, query, source_mask=None, target_mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, source_mask, target_mask)
        return out
```

Model Configuration & Training

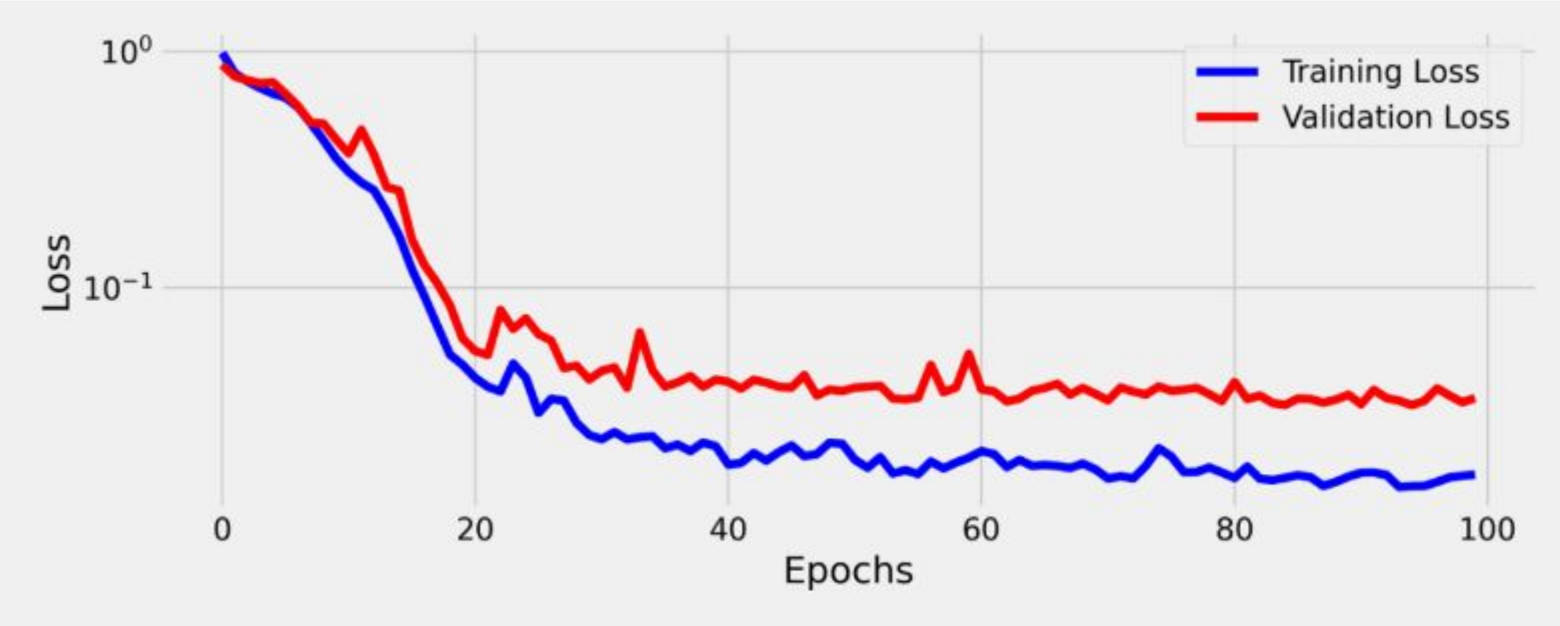
```
torch.manual_seed(43)
encpe = EncoderPe(n_heads=3, d_model=2, ff_units=10, n_features=2)
decpe = DecoderPe(n_heads=3, d_model=2, ff_units=10, n_features=2)

model = EncoderDecoderSelfAttn(encpe, decpe,
                               input_len=2, target_len=2)
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

```
sbs_seq_selfattnpe = StepByStep(model, loss, optimizer)
sbs_seq_selfattnpe.set_loaders(train_loader, test_loader)
sbs_seq_selfattnpe.train(100)
```

```
fig = sbs_seq_selfattnpe.plot_losses()
```

Losses



ENCODER+DECODER+PE

Losses

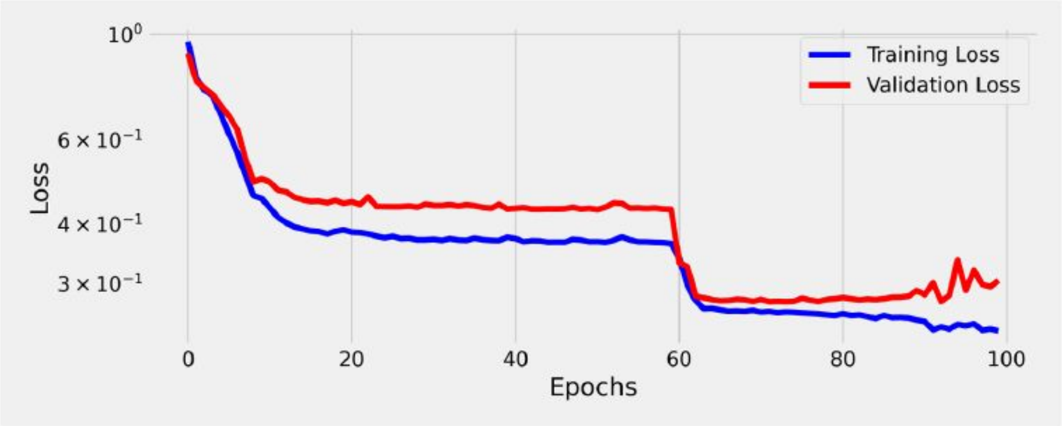
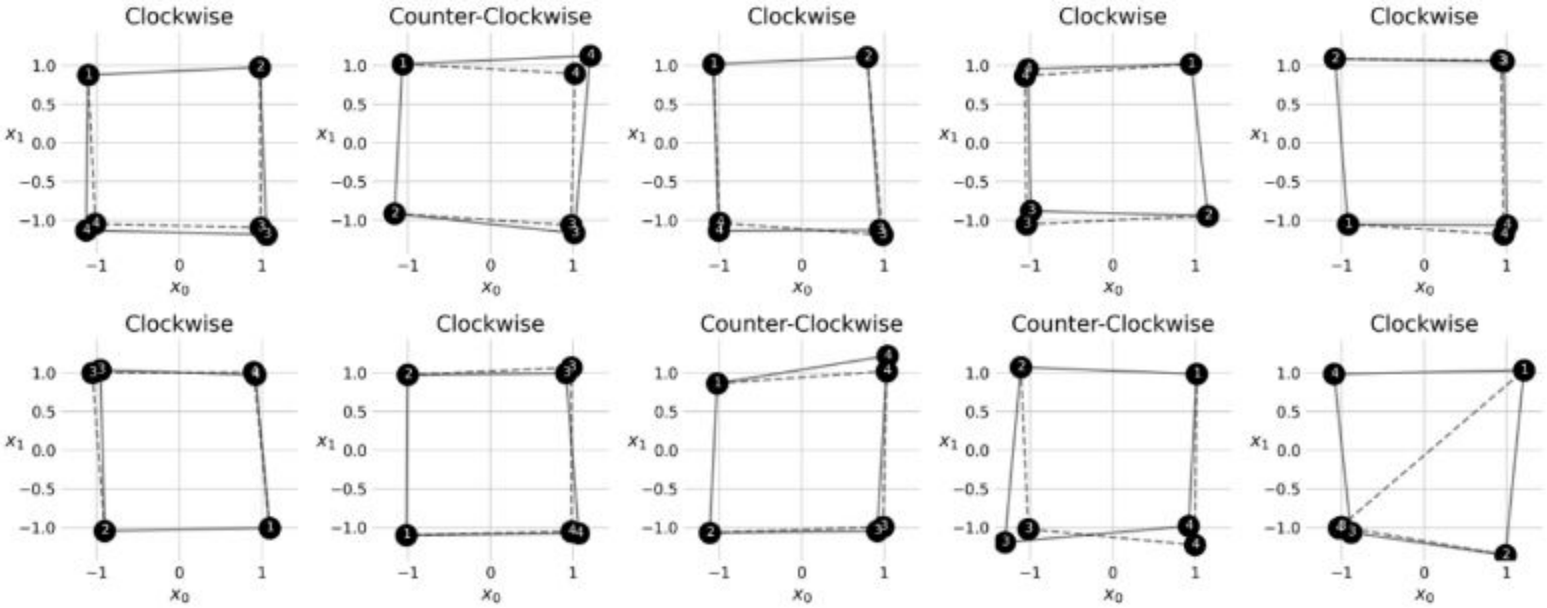


Figure 12 - Losses-encoder + decoder + self-attention

Visualizing Predictions



Model assembly

```
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)
        ).bool()
        return subsequent_mask
```

```
    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
        # (masked) target sequence - used in TRAIN mode
        outputs = self.decoder(shifted_target_seq,
                               source_mask=source_mask,
                               target_mask=target_mask)
        return outputs
```

Model assembly

```python
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
        shifted_target_seq = X[:, self.input_len-1:-1, :]
        # Decodes using the mask to prevent cheating
        outputs = self.decode(shifted_target_seq,
                              source_mask,
```

```python
                              self.trg_masks)
    else:
        # Decodes using its own predictions
        outputs = self.predict(source_seq, source_mask)

    return outputs
```

Model assembly

```python
class PositionalEncoding(nn.Module):
    def __init__(self, max_len, d_model):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).float().unsqueeze(1)
        angular_speed = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-np.log(10000.0) / d_model)
        )
        # even dimensions
        pe[:, 0::2] = torch.sin(position * angular_speed)
        # odd dimensions
        pe[:, 1::2] = torch.cos(position * angular_speed)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x is N, L, D
        # pe is 1, maxlen, D
        scaled_x = x * np.sqrt(self.d_model)
        encoded = scaled_x + self.pe[:, :x.size(1), :]
        return encoded
```

```python
class EncoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = EncoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

    def forward(self, query, mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, mask)
        return out
```

```python
class DecoderPe(nn.Module):
    def __init__(self, n_heads, d_model, ff_units,
                 n_features=None, max_len=100):
        super().__init__()
        pe_dim = d_model if n_features is None else n_features
        self.pe = PositionalEncoding(max_len, pe_dim)
        self.layer = DecoderSelfAttn(n_heads, d_model,
                                     ff_units, n_features)

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

    def forward(self, query, source_mask=None, target_mask=None):
        query_pe = self.pe(query)
        out = self.layer(query_pe, source_mask, target_mask)
        return out
```

References

GODOY, Daniel Voigt. *Deep Learning with PyTorch Step-by-Step: A Beginner’s Guide*. Leanpub, 2022. Disponível em: https://leanpub.com/deep-learning-with-pytorch-step-by-step.