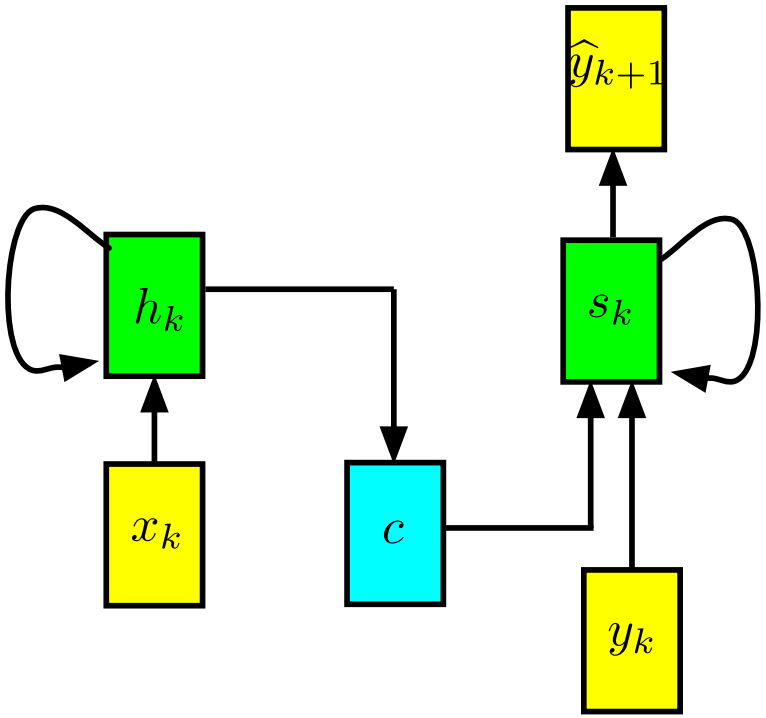
RNN Encoder-Decoder



RNN Encoder-Decoder

$x_k = k$th encoded english
"I am a student"

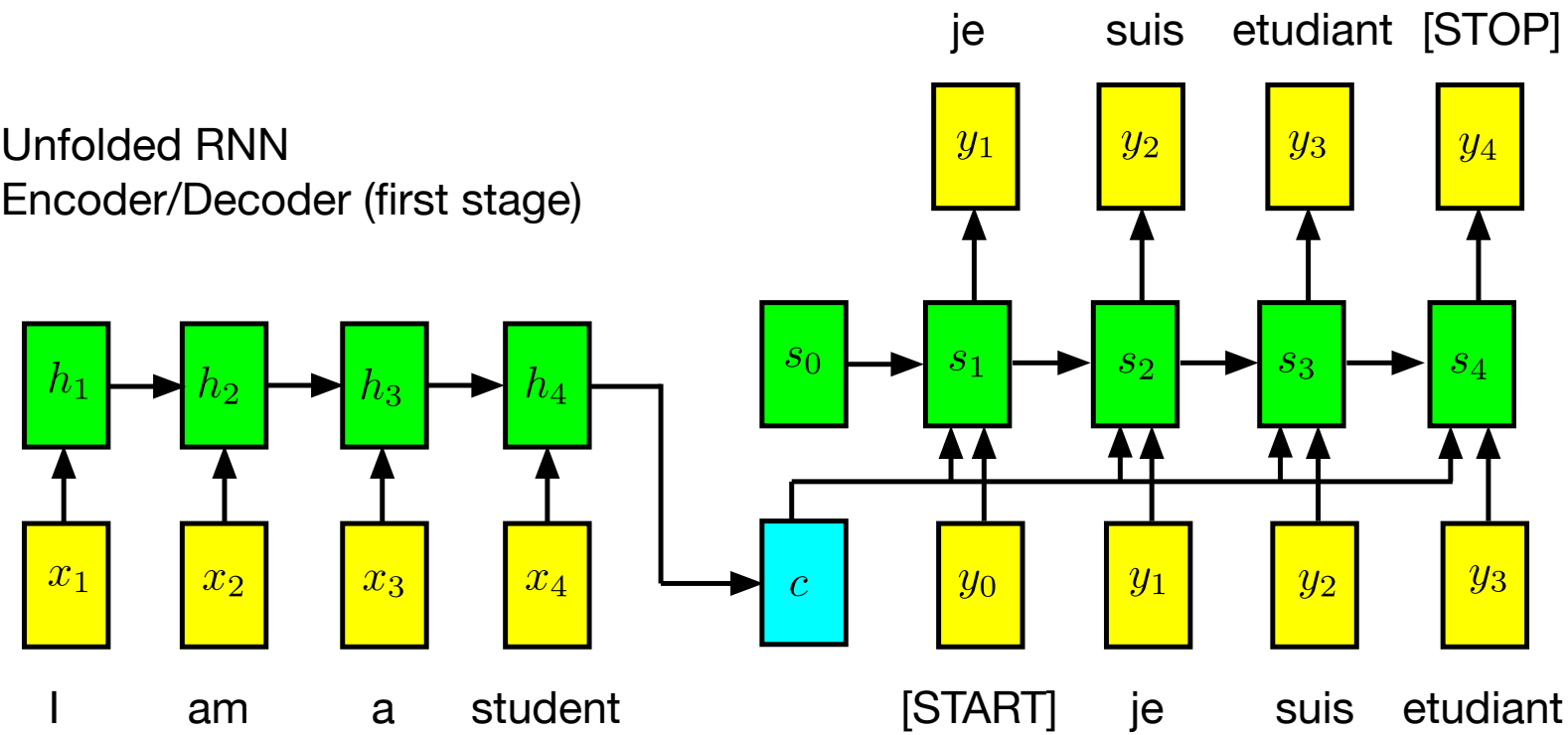
$y_k = k$th encoded French
"[START] je suis etudiant [STOP]"

$\widehat{y}_{k+1}$ = predicted french

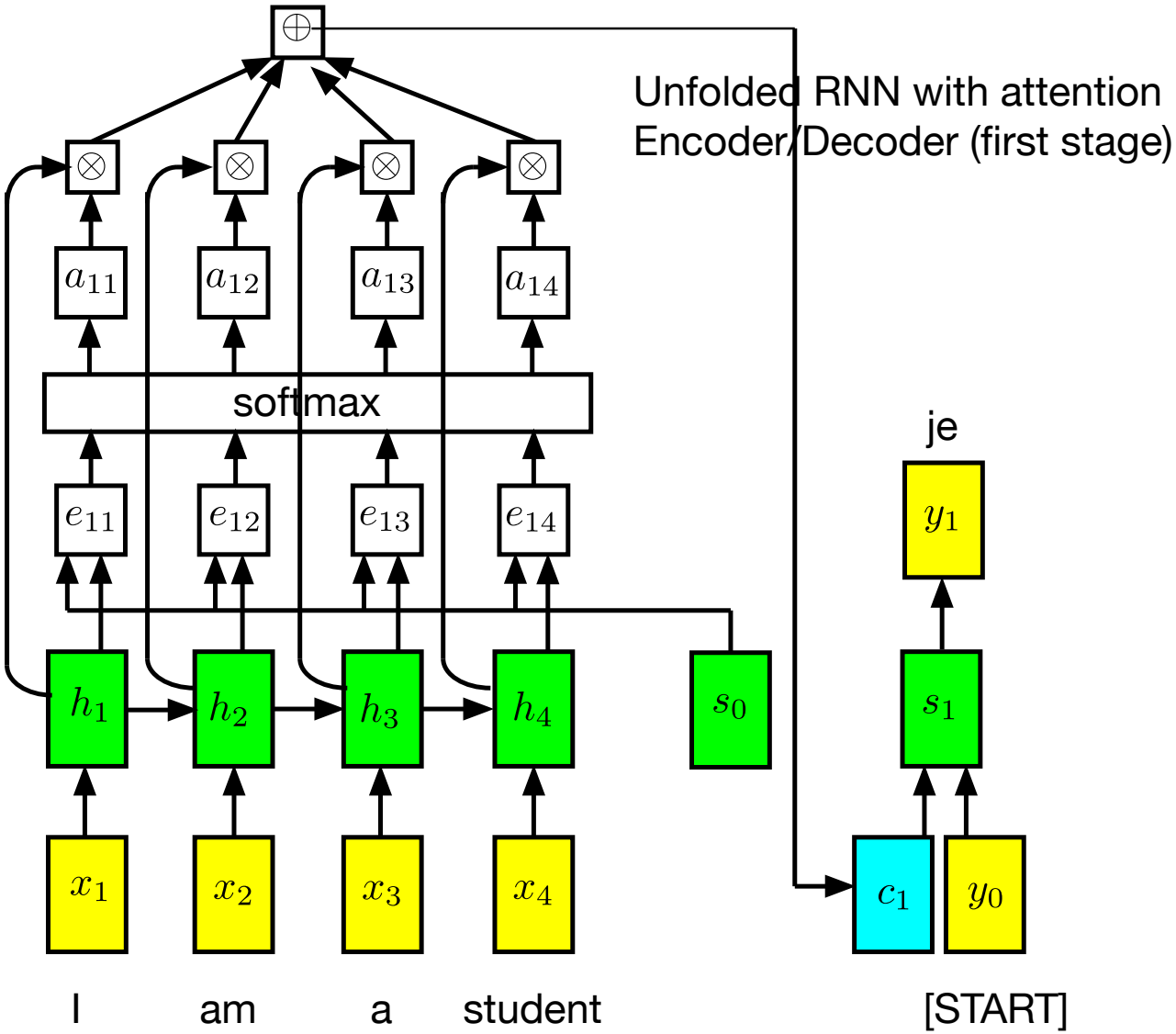
c = single context vector

RNN Encoder Decoder Unfolded

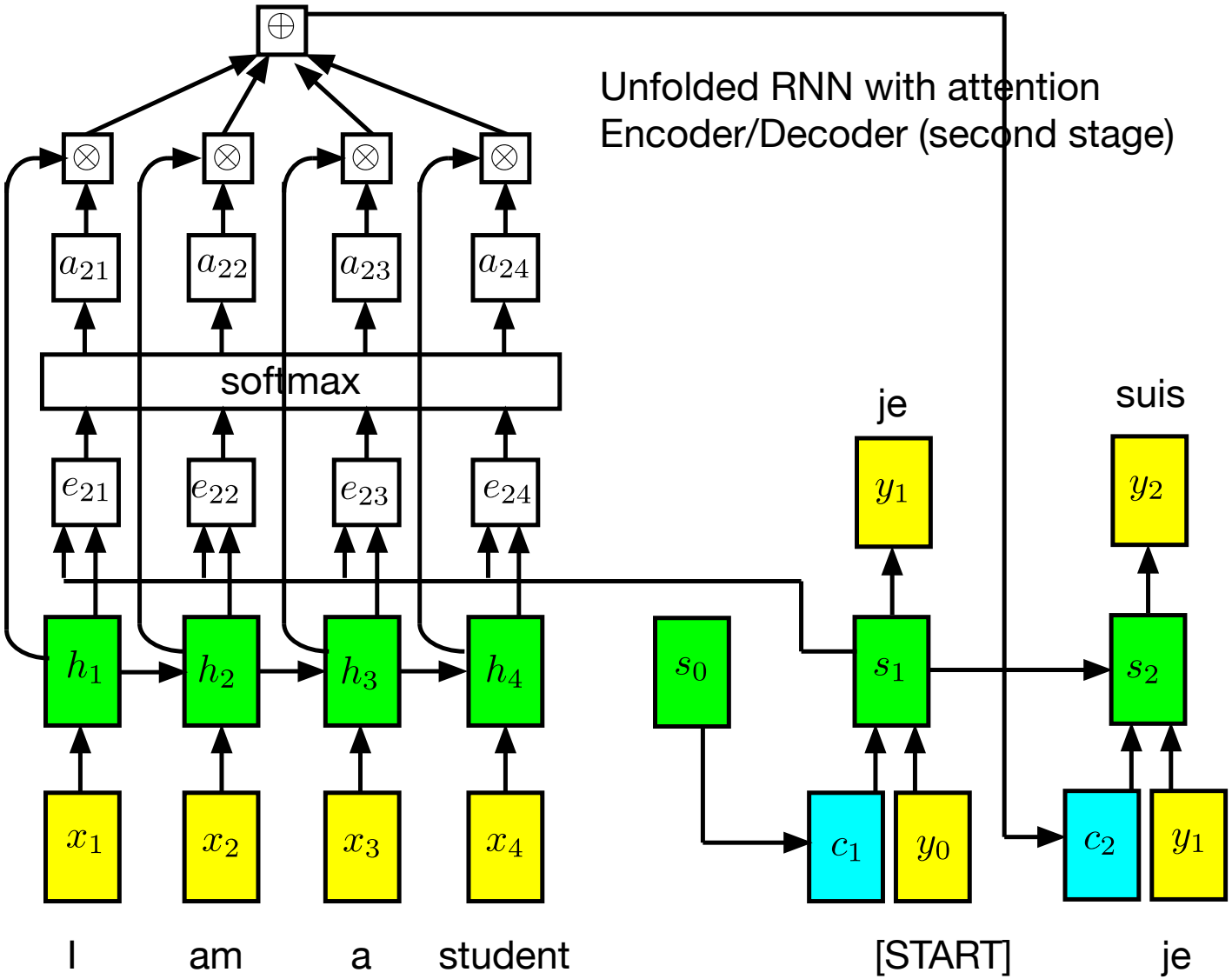
Unfolded RNN
Encoder/Decoder (first stage)



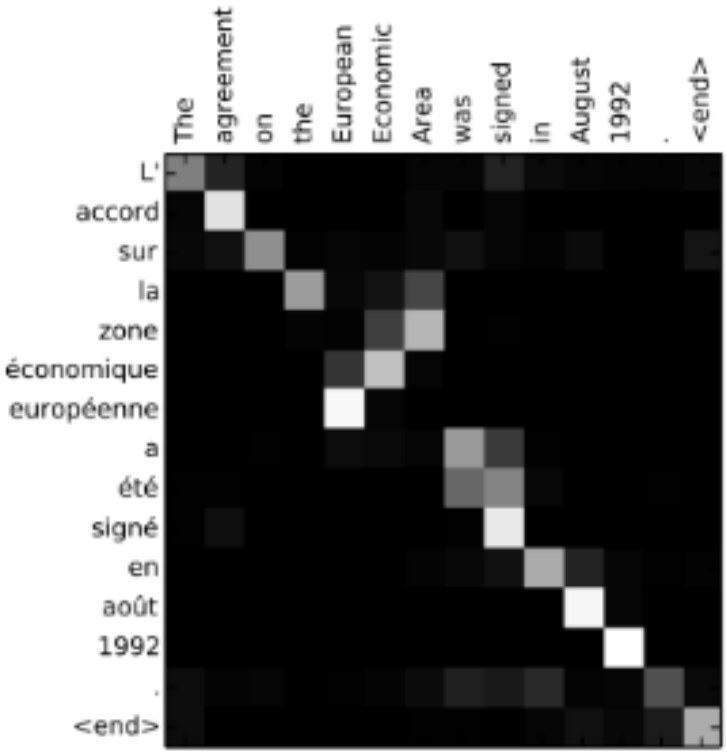
RNN Encoder Decoder with Attention



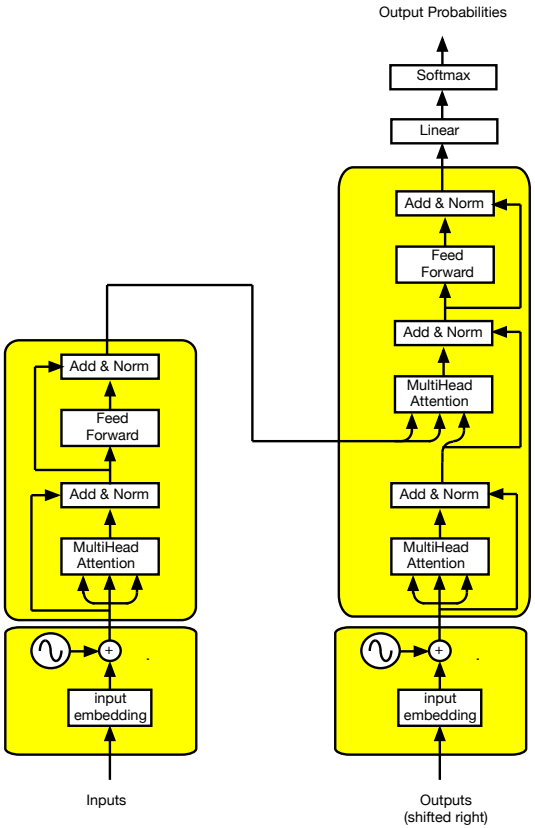
RNN Encoder Decoder with Attention



Visualize Attention Weights



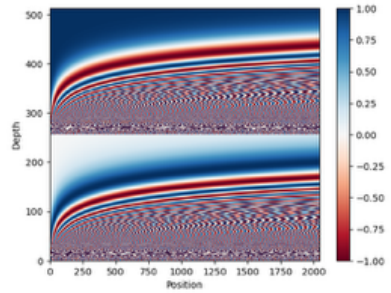
Transformer

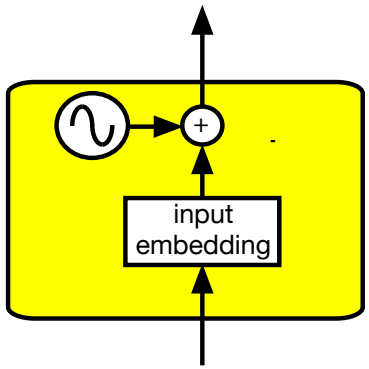


- Positional Embedding Layer
  Embeds token *position* in the representation
- Multi-Head Attention Layers
  Transformers use self,cross, and causal versions of MHA
- Add and Normalization
  Residual connection and Layer Normalization
- FeedForward Network

Positional Embedding

- Attention scores are invariant with respect to a word's position.
- Position Embedding attaches info about word position by adding sines and cosines the word's embedding vector
- Sines and and cosines vibrate at different frequencies along the depth of the embedding vector. Oscillation is across position in sequence





$$\begin{aligned} \mathrm{PE}_{\mathrm{pos},2i} &= \sin(\mathrm{pos}/10000^{2i/d}) \\ \mathrm{PE}_{\mathrm{pos},2i+1} &= \cos(\mathrm{pos}/10000^{2i}/d) \end{aligned}$$

Self Attention Layer

Base Attention Class

```
class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()
```

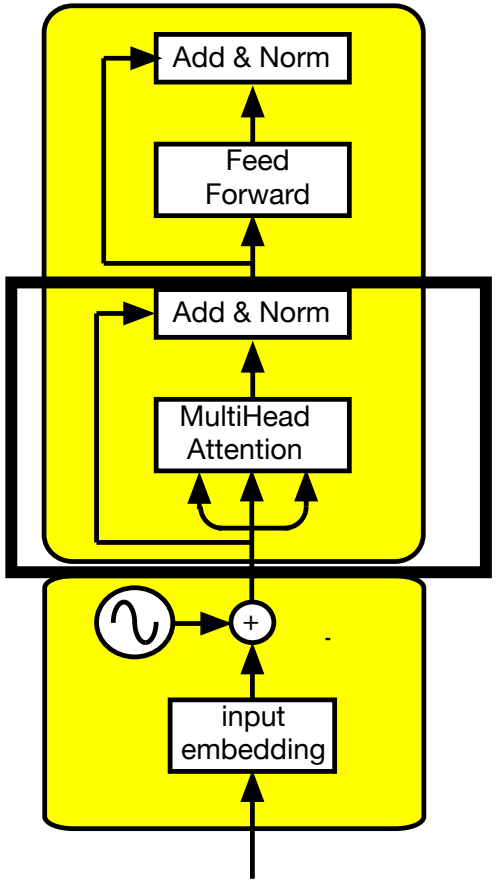
CrossAttention Class

```
class GlobalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

attention layer

residual connection

layer normalization



Causal Attention Layer

Base Attention Class

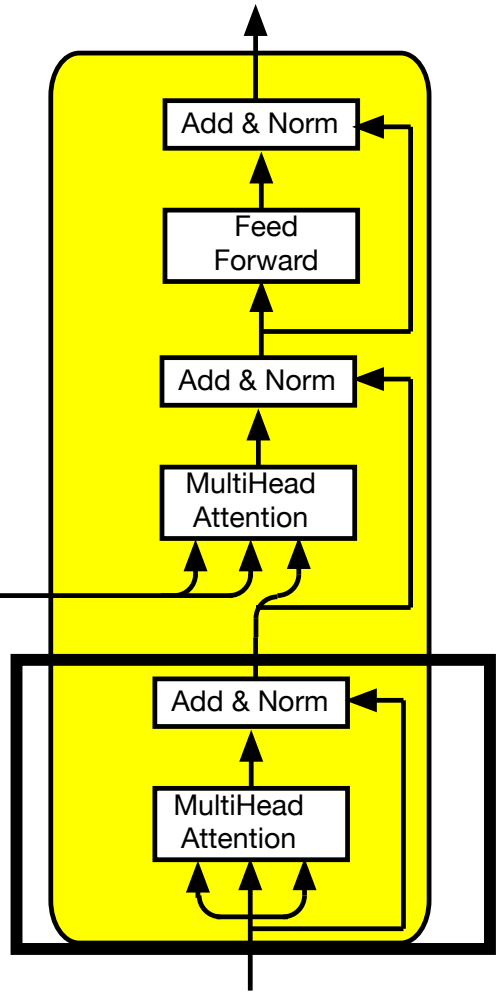
```
class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()
```

CausalAttention Class

```
class CausalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x,
        use_causal_mask = True)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

attention layer

residual connection
layer normalization



Cross Attention Layer

Base Attention Class

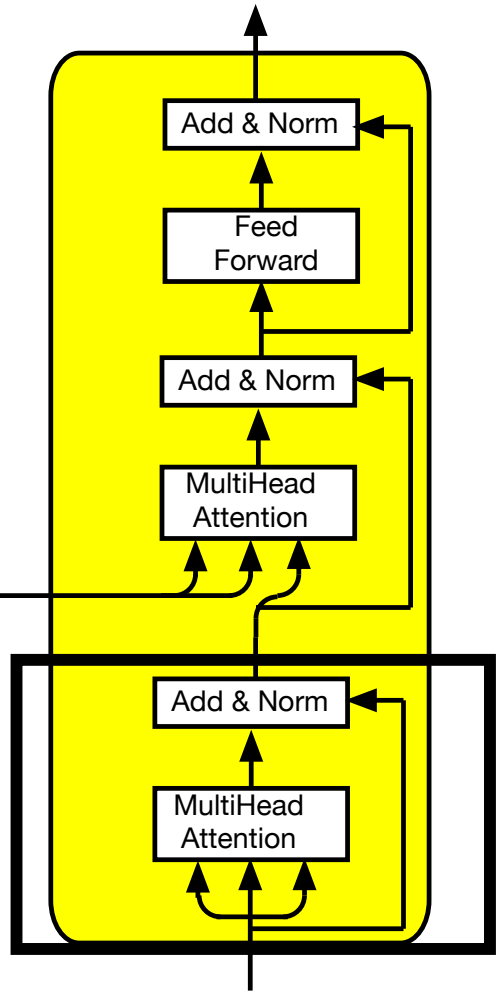
```
class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()
```

CausalAttention Class

attention layer

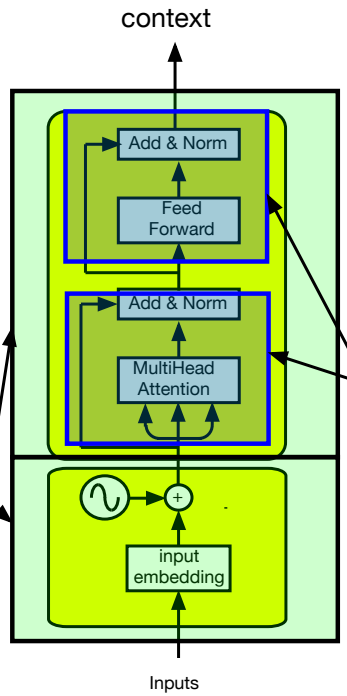
residual connection
layer normalization

```
class CrossAttention(BaseAttention):
  def call(self, x, context):
    attn_output, attn_scores = self.mha(
        query=x,
        key=context,
        value=context,
        return_attention_scores=True)
    self.last_attn_scores = attn_scores

    x = self.add([x, attn_output])
    x = self.layernorm(x)

    return x
```



Encoder Block

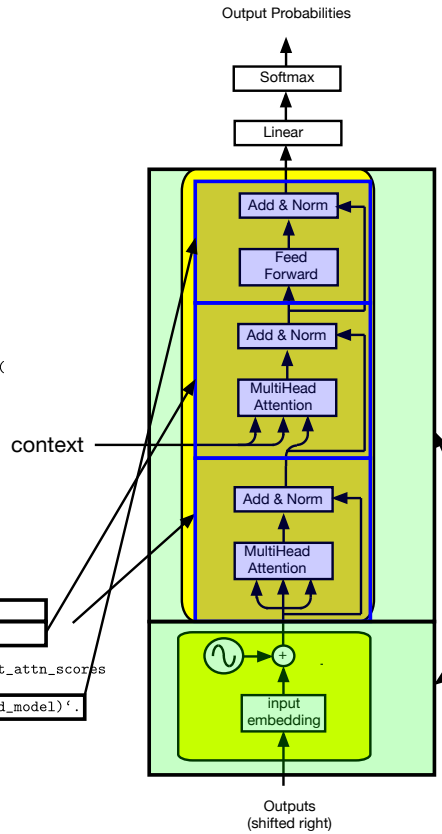
```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.
```




```
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self,*, d_model, num_heads, dff, dropout_rate=0.1):
    super().__init__()

    self.self_attention = GlobalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x):
    x = self.self_attention(x)
    x = self.ffn(x)
    return x
```

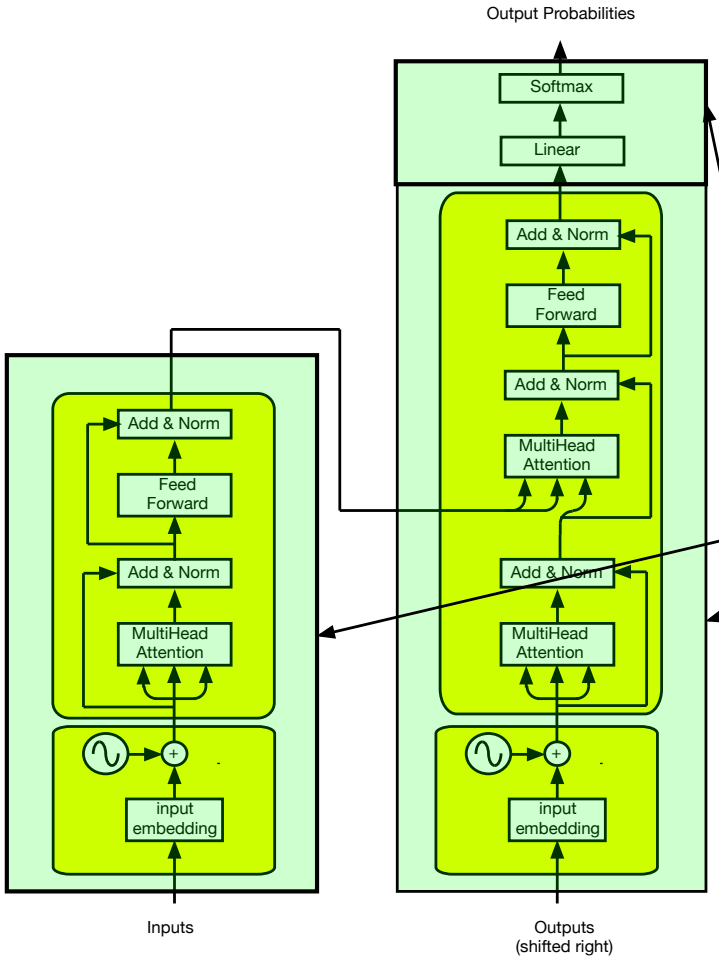
Decoder Block

```
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self,
               *,
               d_model,
               num_heads,
               dff,
               dropout_rate=0.1):
    super(DecoderLayer, self).__init__()

    self.causal_self_attention = CausalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.cross_attention = CrossAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x, context):
    x = self.causal_self_attention(x=x)
    x = self.cross_attention(x=x, context=context)

    self.last_attn_scores = self.cross_attention.last_attn_scores

    x = self.ffn(x)  # Shape `(batch_size, seq_len, d_model)`.
    return x
```




```
lass Decoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads, dff, vocab_size,
               dropout_rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(vocab_size=vocab_size,
                                             d_model=d_model)
    self.dropout = tf.keras.layers.Dropout(dropout_rate)
    self.dec_layers = [
        DecoderLayer(d_model=d_model, num_heads=num_heads,
                     dff=dff, dropout_rate=dropout_rate)
        for _ in range(num_layers)]

    self.last_attn_scores = None

  def call(self, x, context):
    # `x` is token-IDs shape (batch, target_seq_len)
    x = self.pos_embedding(x)  # (batch_size, target_seq_len, d_model)

    x = self.dropout(x)

    for i in range(self.num_layers):
      x  = self.dec_layers[i](x, context)

    self.last_attn_scores = self.dec_layers[-1].last_attn_scores

    # The shape of x is (batch_size, target_seq_len, d_model).
    return x
```

Transformer Complete




```
class Transformer(tf.keras.Model):
  def __init__(self, *, num_layers, d_model, num_heads, dff,
               input_vocab_size, target_vocab_size, dropout_rate=0.1):
    super().__init__()
    self.encoder = Encoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=input_vocab_size,
                           dropout_rate=dropout_rate)

    self.decoder = Decoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=target_vocab_size,
                           dropout_rate=dropout_rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inputs):
    # To use a Keras model with `.fit` you must pass all your inputs in the
    # first argument.
    context, x  = inputs

    context = self.encoder(context)  # (batch_size, context_len, d_model)

    x = self.decoder(x, context)  # (batch_size, target_len, d_model)

    # Final linear layer output.
    logits = self.final_layer(x)  # (batch_size, target_len, target_vocab_size)

    try:
      # Drop the keras mask, so it doesn't scale the losses/metrics.
      # b/250038731
      del logits._keras_mask
    except AttributeError:
      pass

    # Return the final output and the attention weights.
    return logits
```

Training Phase

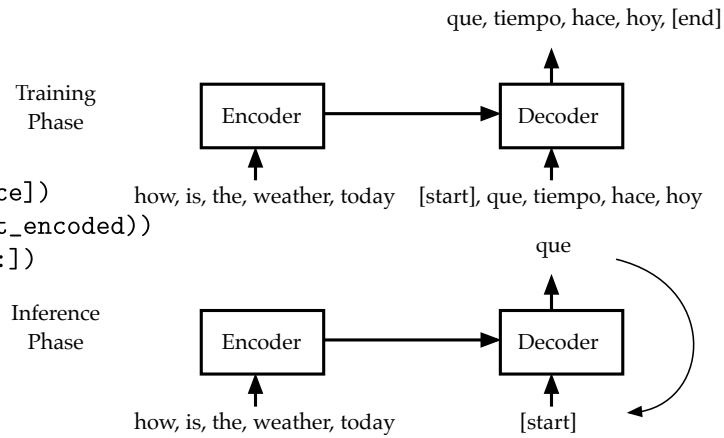
- Instantiate the transformer
- Compile the model using a customized schedule for the learning rate and a special masked loss function
- Fit the model in the usual manner on the ptrain and validation dataset objects.

```
transformer.compile(
    loss=masked_loss,
    optimizer=optimizer,
    metrics=[masked_accuracy])

transformer.fit(ptrain_ds,epochs=20,validation_data = val_ds)
```

```
num_layers = 4
d_model = 128
dff = 512
num_heads = 8
dropout_rate = 0.1

transformer = Transformer(
    num_layers=num_layers,
    d_model=d_model,
    num_heads=num_heads,
    dff=dff,
    input_vocab_size= eng_vocab_size,
    target_vocab_size=spa_vocab_size,
    dropout_rate=dropout_rate)

class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self, d_model, warmup_steps=4000):
    super().__init__()

    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)

    self.warmup_steps = warmup_steps

  def __call__(self, step):
    step = tf.cast(step, dtype=tf.float32)
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps ** -1.5)

    return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)

learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)

def masked_loss(label, pred):
  mask = label != 0
  loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
  loss = loss_object(label, pred)

  mask = tf.cast(mask, dtype=loss.dtype)
  loss *= mask

  loss = tf.reduce_sum(loss)/tf.reduce_sum(mask)
```

Inference Phase

```
def decode_sequence(input):
    input_encoded = eng_vectorization([input])
    decoded_sentence = "[start]"
    for word in range(max_decoded_sentence_length):
        target_encoded = spa_vectorization([decoded_sentence])
        next_token_pred = transformer((input_encoded,target_encoded))
        next_token_indx = np.argmax(next_token_pred[0, i, :])
        next_token = spa_index_lookup(next_token_indx)
        decoded_sentence += " " + next_token
        if next_token == "[end]":
            break
    return decoded_sentence
```

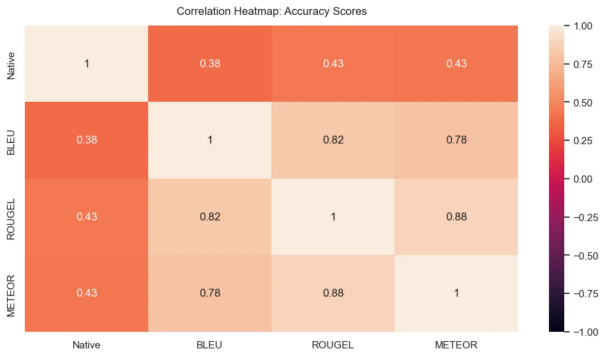



```
This is a song I learned when I was a kid.
[start] Esta es una cancion que aprendi cuando era chico [end]
#Google Translate: Esta es una canción que aprendí cuando era niño

She can lay the piano.
[start] ella puede tocar piano [end]
#Google Translate: Ella puede tocar el piano.

It may have rained a little last night.
[start] puede que llueve un poco el pasado [end]
#Google Translate: Puede que haya llovido un poco anoche
```

Evalutation - Metrics

- Traditional accuracy metrics are not good at evaluating the quality of language translation.
- Common metrics used for language translation are
  - BLEU (BiLingual Evaluation Understudy)
  - ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation)
  - METEOR (Metric for Evaluation of Translation with Explicit Ordering)

Heatmap shows correlation of accuracy scores for 3 metrics and a native speaker.



Reference (human):

I am very happy to say that I am getting a big boost of energy.

Generated translation:

I am very happy to say that I am getting a big boost of energy. -> METEOR 0.99 | BLEU 1.0

I am very happy that I am getting a boost of energy. -> METEOR 0.82 | BLEU 0.54

I am quite glad to tell that I am getting a huge boost of energy. -> METEOR 0.75 | BLEU 0.34

RNN w/o Attention vs Transformer

RNN w/o Attention

1. (+) Does reasonably well on long sequences
2. (-) Requires inputs to be ordered
3. (-) Can only operate on input samples sequentially
4. (-) Cannot be parallelized

Transformer

1. (+) Excellent performance on long sequences
2. (+) Does not require ordered inputs if we use Positional Embedding
3. (+) Parallel Operation
4. (-) Requires a lot of memory