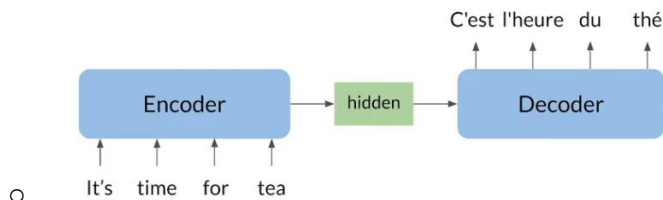


Neural Machine Translation:

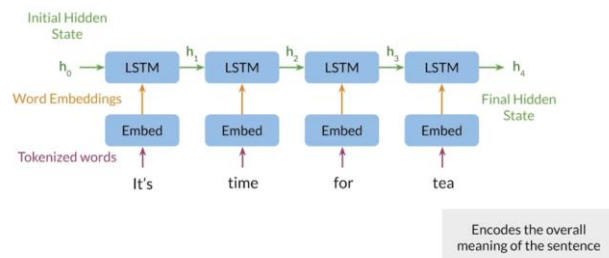
- Basics:

- Concept: Neural machine translation involves an encoder and a decoder.
- Example: "It's time for tea" from English to French ("C'est l'heure du thé").
- Traditional System: Utilizes LSTMs for both encoding and decoding.

- Seq2Seq model:

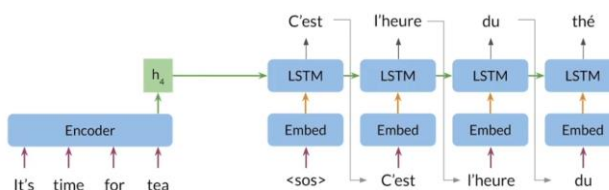


- Encoder:



-
- Consists of an embedding layer and an LSTM module.
- Embeds word tokens into vectors.
- Passes information through LSTM layers, retaining hidden states.
- Outputs the final hidden state (e.g., h_4), encapsulating the entire sentence's meaning.

- Decoder:



-
- Similar structure with an embedding layer and an LSTM layer.
- Generates translated sentences using the output of the encoder.
- Begins with a start of sequence token (SOS).
- Processes each word sequentially, predicting the most probable next word.

- Limitations:

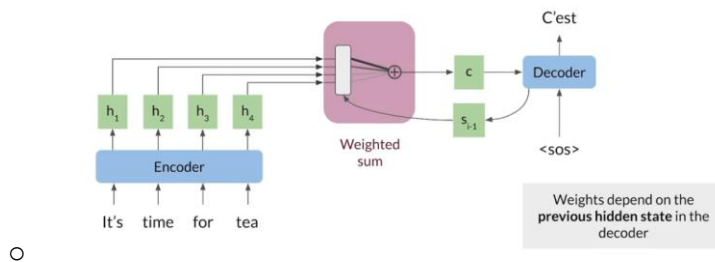
- Information bottleneck:
 - Fixed hidden state size leads to issues with long sequences.
 - Only a fixed amount of information transfers from the encoder to the decoder irrespective of input length.
 - Model performance degrades with increasing sequence size.

- Solutions:

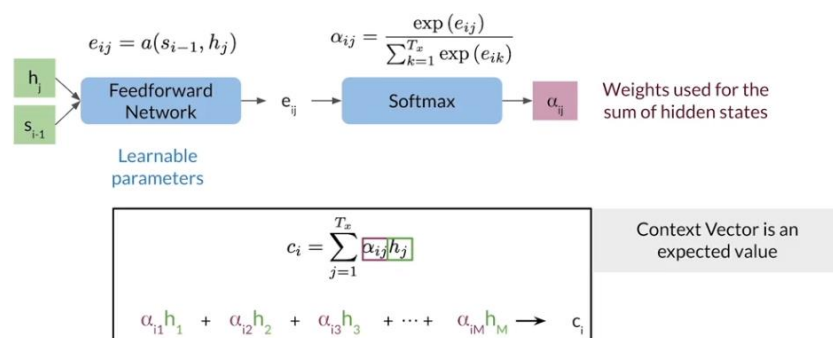
- Idea: use encoder hidden states for each word.
- Challenge: memory efficiency and context handling.
- Solutions: Attention
 - Allows the model to select and focus on the most crucial words at each decoding step.

Seq2seq model with attention

- How to use all the hidden states:



- Previous hidden state in the decoder: s_{i-1}
- Attention layer: (how the weights and context vector are calculated)
 - Goal: to provide a context vector with relevant information from encoder states.
 - Steps:



- Calculate alignment scores e_{ij} between input j and expected output i .
 - Higher match \rightarrow higher score
 - Convert scores to weights (probabilities) using softmax.
 - Multiply encoder states by weights and sum to create the context vector.

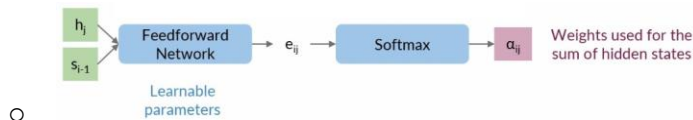
Basic attention implementation

Softmax:

```
1. import numpy as np
2. def softmax(x, axis=0): #axis=0 : column
3.     return np.exp(x) / np.expand_dims(np.sum(np.exp(x), axis=axis), axis)
```

Calculating alignment scores:

- $e_{ij} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)$, $W_a \in \mathbb{R}^{n \times m}$, $U_a \in \mathbb{R}^{n \times m}$, and $v_a \in \mathbb{R}^m$
- A measure of similarity between the decoder hidden state and each encoder hidden state.
- Implementation:



```
1. hidden_size = 16
2. attention_size = 10
3. input_length = 5
4. np.random.seed(42)
5. encoder_states = np.random.randn(input_length, hidden_size)
6. decoder_state = np.random.randn(1, hidden_size)
7. layer_1 = np.random.randn(2 * hidden_size, attention_size)
8. layer_2 = np.random.randn(attention_size, 1)
9.
10. def alignment(encoder_states, decoder_state):
11.     repeated_decoder_state = np.repeat(decoder_state, input_length, axis=0)
12.     inputs = np.concatenate((encoder_states, repeated_decoder_state), axis=1)
13.     activations = np.tanh(np.dot(inputs, layer_1))
14.     scores = np.dot(activations, layer_2)
15.     assert scores.shape == (input_length, 1)
16.     return scores
```

- Turning alignment into weights, then weight the encoder output vectors and sum:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^K \exp(e_{ik})} \rightarrow c_i = \sum_{j=1}^K \alpha_{ij} h_j$$

Attention:

```
1. def attention(encoder_states, decoder_state):
2.     scores = alignment(encoder_states, decoder_state)
3.     weights = softmax(scores)
4.     weighted_scores = encoder_states * weights
5.     context = np.sum(weighted_scores, axis=0)
6.     return context
```

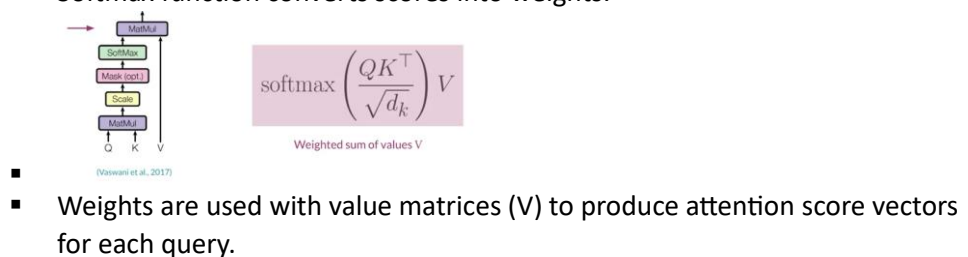
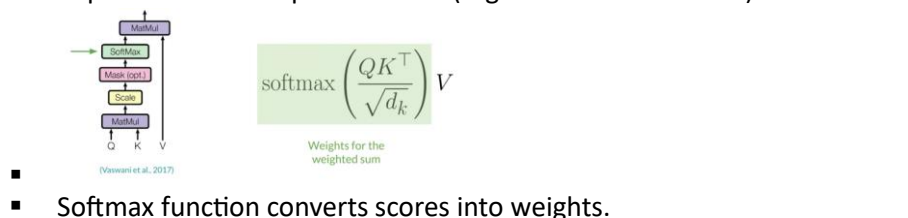
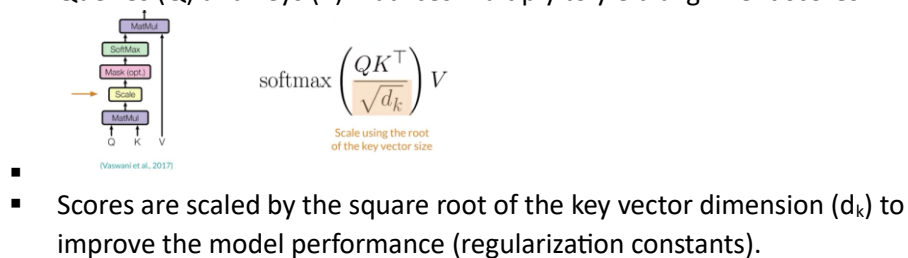
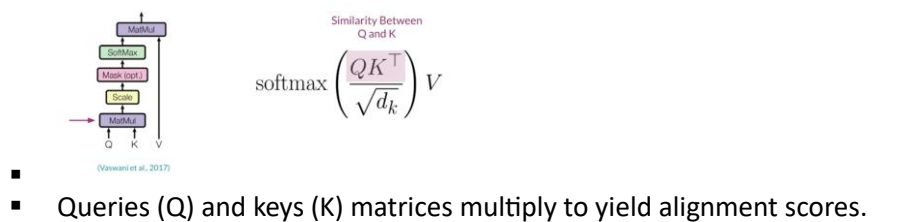
Queries, Keys, Values, and Attention

- Based on information retrieval system.
- Queries, keys, values:

Query	Key	Value
l'heure	It's	[0.5, 0.2, -1.2, ..., ...]
	time	[0.2, -0.7, 0.9, ..., ...]
	for	[1.3, 0.3, 0.8, ..., ...]
	tea	[-0.4, 0.6, -1.1, ..., ...]

- Queries, keys and values in practice are all vector-representations.
- Process: Query and key vectors calculate alignment scores indicating similarity.
- Output: Weighted sum of value vectors resulting in an attention vector.
- Scaled dot-product attention:

- Steps:



- When the attention mechanism assigns a higher attention score to a certain word in the sequence, the next word in the decoder's output will be more strongly influenced by this word than by other words in the sequence.

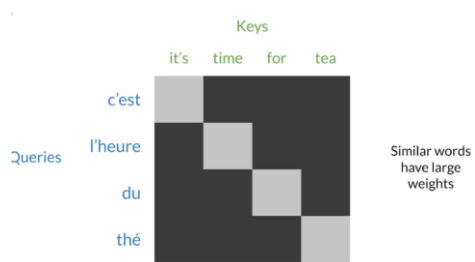
- Benefits:

- Much faster since it involves just two matrix multiplication and softmax.

- Limitation:

- The alignments between the source and target languages must be learned elsewhere, typically in the input embeddings or in other linear layers before the attention layer.

- Closer look in alignment weights:



-
- Learning: The model recognizes similar meaning words (alignment) and encodes this in the query and key vectors.
- Benefit: Effective for languages with different grammatical structures, aligning meaning irrespective of word order.

Scaled Dot-Product Attention Implementation:

Setup:

```
1. import pickle
2. import matplotlib.pyplot as plt
3. import numpy as np
4. with open("./data/word2int_en.pkl", "rb") as f:
5.     en_words = pickle.load(f)
6. with open("./data/word2int_fr.pkl", "rb") as f:
7.     fr_words = pickle.load(f)
8. en_embeddings = np.load("./data/embeddings_en.npz")["embeddings"]
9. fr_embeddings = np.load("./data/embeddings_fr.npz")["embeddings"]
```

Help function:

```
1. def tokenize(sentence, token_mapping):
2.     tokenized = []
3.     for word in sentence.lower().split(" "):
4.         try:
5.             tokenized.append(token_mapping[word])
6.         except KeyError:
7.             # Using -1 to indicate an unknown word
8.             tokenized.append(-1)
9.     return tokenized
10.
11. def embed(tokens, embeddings):
12.     embed_size = embeddings.shape[1]
13.     output = np.zeros((len(tokens), embed_size))
14.     for i, token in enumerate(tokens):
15.         if token == -1:
16.             output[i] = np.zeros((1, embed_size))
17.         else:
18.             output[i] = embeddings[token]
19.     return output
```

```

1. def softmax(x, axis=0):
2.     return np.exp(x) / np.expand_dims(np.sum(np.exp(x), axis=axis), axis)
3.
4. def calculate_weights(queries, keys):
5.     """ Calculate the weights for scaled dot-product attention"""
6.     dot = np.dot(queries, keys.T)
7.     weights = softmax(dot, axis=1)
8.     return weights

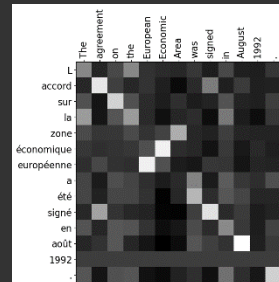
```

Visualization:

```

1. sentence_en = "The agreement on the European Economic Area was signed in August 1992 ."
2. tokenized_en = tokenize(sentence_en, en_words)
3. embedded_en = embed(tokenized_en, en_embeddings)
4. sentence_fr = "L accord sur la zone économique européenne a été signé en août 1992 ."
5. tokenized_fr = tokenize(sentence_fr, fr_words)
6. embedded_fr = embed(tokenized_fr, fr_embeddings)
7. alignment = calculate_weights(embedded_fr, embedded_en)
8.
9. fig, ax = plt.subplots(figsize=(7,7))
10. ax.imshow(alignment, cmap='gray')
11. ax.xaxis.tick_top()
12. ax.set_xticks(np.arange(alignment.shape[1]))
13. ax.set_xticklabels(sentence_en.split(" "), rotation=90, size=16);
14. ax.set_yticks(np.arange(alignment.shape[0]));
15. ax.set_yticklabels(sentence_fr.split(" "), size=16);

```



Attention:

```

1. def attention_qkv(queries, keys, values):
2.     weights = calculate_weights(queries, keys)
3.     return np.matmul(weights, values)

```

Setup for Machine Translation

- Data example:

English	French
I am hungry!	J'ai faim!
...	...
I watched the soccer game.	J'ai regardé le match de football.

- Setup:
 - o Use pre-trained vector embeddings
 - o Otherwise, initially represent words with one-hot vectors
 - o Keep track of index mappings with word2Ind and ind2word dictionaries
 - o Add end of sequence tokens: <EOS>
 - o Pad the token vectors with zeros to match the length of the longest sequence.
- Example:

- Example:

ENGLISH SENTENCE:

Both the ballpoint and the mechanical pencil in the series are equipped with a special mechanism: when the twist mechanism is activated, the lead is pushed forward.

TOKENIZED VERSION OF THE ENGLISH SENTENCE:

[4546 4 11358 362 8 4 23326 20104 1745 8210 9641 5 6 4 3103
 31 2767 30 13 914 4797 64 196 4 22474 5 4797 16 24864 86 2 4
 1060 16 6413 1138 3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0

FRENCH TRANSLATION:

Le stylo à bille et le porte-mine de la série sont équipés d'un mécanisme spécial: lorsque le mécanisme de torsion est activé, le plomb est poussé vers l'avant.

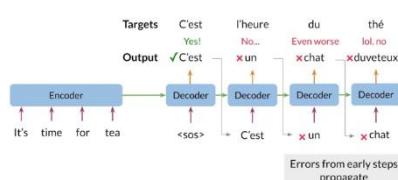
TOKENIZED VERSION OF THE FRENCH TRANSLATION:

[7 29587 9 18240 8 7 420 5 3440 2 6 156 39 7941 14 19 5548 2648
562 7 5548 2 23194 18 20114 1 7 5695 18 8865 149 12 137 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] <FOS>

Padding

Training Approach:

- Traditional Method:

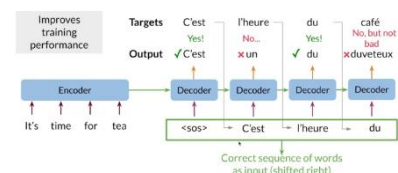


Compare the decoder output sequence with the target sequence to calculate loss.

Use cross-entropy for each step and sum for total loss.

Challenge: Early training stages yield naïve prediction → compounding errors.

- Solution: (Teacher Forcing)

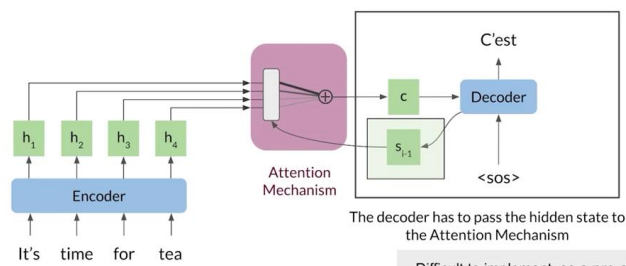


Use ground truth words as decoder inputs rather than the model's predictions.

Benefits: Continues training as if correct predictions were made, the training process is accelerated.

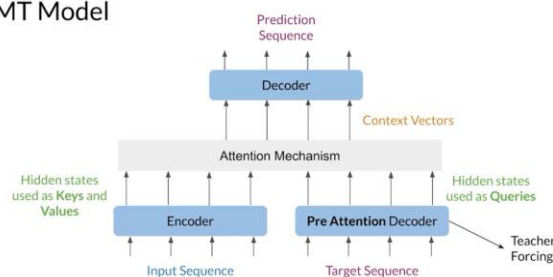
- Variations: Curriculum Learning
 - o Gradual Shift: Slowly transitioning from ground truth inputs to using the model's decoder outputs during training.
 - o End Goal: Reduce reliance on target words, leading to more autonomous model predictions.

Neural Machine Translation(NMT) Model with Attention:



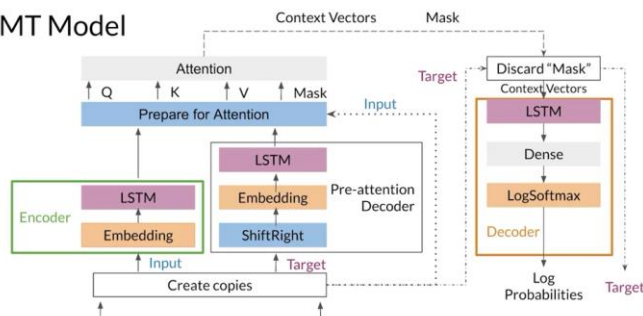
- Difficult to implement the pass of the hidden states from the decoder to the Attention Mechanism. → We use two decoders: Pre-attention decoder to provide hidden states; post-attention decoder to provide the translation.
- Modified:

NMT Model




- Model Implementation steps:

NMT Model



- - **Token Duplication:** Make two copies of the input and target tokens.
 - **Parallel Processing:**
 - Encoder: Transforms input tokens into key and value vectors.
 - Pre-attention Decoder ShiftRight: Processes a copy of target tokens with a start-of-sentence token.
 - **Embedding and LSTM Layers:** Input and target sequences are embedded and passed through LSTMs in both encoder and pre-attention decoder.
 - **Preparation for Attention:**
 - Obtain query, key, and value vectors.
 - Use a padding mask function with input tokens to aid attention layer token recognition.
 - **Attention Layer:**
 - Inputs: Queries, keys, values, and mask.
 - Outputs context vectors and disregards the mask.
 - **Post-attention Decoder:**
 - Components: LSTM, dense layer, and LogSoftmax.
 - Outputs log probabilities.

Model Evaluation: BiLingual Evaluation Understudy (BLEU) Score

-  0 1
- The closer to 1, the better
- Method: Count matches and divide by total words in candidate.
- Example:

Candidate	I	I	am	I	
Reference 1	Younes	said	I	am	hungry
Reference 2	He	said	I	am	hungry

Count: $\frac{1+1+1+1}{4} = 1$

A model that always outputs common words will do great! ← Bad

- Modified BLEU Score:

Candidate	I	I	am	I	
Reference 1	Younes	said			hungry
Reference 2	He	said			hungry

Count: $\frac{1+1}{4} = 0.5$

Better than the previous implementation version!

- Issues:
 - o Doesn't consider semantic meaning.
 - o Doesn't consider sentence structure.
 - o Example:
 - "Ate I was hungry because!"
 - "I ate because I was hungry!"

BLEU score Implementation:

```
1. import numpy as np                # import numpy to make numerical computations.
2. import nltk                       # import NLTK to handle simple NL tasks like tokenization.
3. nltk.download("punkt")
4. from nltk.util import ngrams
5. from collections import Counter   # import a counter.
6. !pip3 install 'sacrebleu'         # install the sacrebleu package.
7. import sacrebleu                 # import sacrebleu in order to compute the BLEU score.
8. import matplotlib.pyplot as plt   # import pyplot in order to make some illustrations.
```

Definition and formulas:

$$BLEU = BP \times \left(\prod_{i=1}^n precision_i \right)^{(1/n)}$$

- BP: Brevity Penalty

$$BP = \min \left(1, e^{(1 - (\text{len}(\text{ref}) / \text{len}(\text{cand})))} \right)$$

- o $\text{len}(\text{ref})$ and $\text{len}(\text{cand})$: the length or count of words in the reference and candidate translation.
- o The brevity penalty helps to handle very short translations.

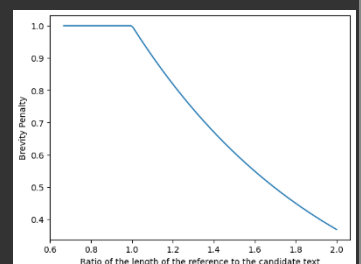
- Precision:

$$precision_i = \frac{\sum_{s_i \in \text{cand}} \min(C(s_i, \text{cand}), C(s_i, \text{ref}))}{\sum_{s_i \in \text{cand}} C(s_i, \text{cand})}$$

- o $C(s_i, \text{cand})$ and $C(s_i, \text{ref})$: the counts of the i-grams in the candidate and reference sentences respectively.
- o The sum counts all the n-grams in the candidate sentence that also appear in the reference sentence, but only counts them as many times as they appear in the reference sentence and not more

Visualizing the BLEU score:

```
1. reference_length = 1
2. candidate_length = np.linspace(1.5, 0.5, 100)
3. length_ratio = reference_length / candidate_length
4. BP = np.minimum(1, np.exp(1 - length_ratio))
5. fig, ax = plt.subplots(1)
6. lines = ax.plot(length_ratio, BP)
7. ax.set(
8.     xlabel="Ratio of the length of the reference to the candidate text",
9.     ylabel="Brevity Penalty",
10. )
11. plt.show()
```



Example Calculations of the BLEU Score

```
1. reference = "The NASA Opportunity rover is battling a massive dust storm on planet Mars."
2. candidate_1 = "The Opportunity rover is combating a big sandstorm on planet Mars."
3. candidate_2 = "A NASA rover is fighting a massive storm on planet Mars."
4.
5. tokenized_ref = nltk.word_tokenize(reference.lower())
6. tokenized_cand_1 = nltk.word_tokenize(candidate_1.lower())
7. tokenized_cand_2 = nltk.word_tokenize(candidate_2.lower())

1. def brevity_penalty(candidate, reference):
2.     reference_length = len(reference)
3.     candidate_length = len(candidate)
4.     if reference_length < candidate_length:
5.         BP = 1
6.     else:
7.         penalty = 1 - (reference_length / candidate_length)
8.         BP = np.exp(penalty)
9.     return BP
```

Compute the clipped Precision

- This function calculates how many of the n-grams in the candidate sentence actually appear in the reference sentence. The clipping takes care of overcounting. For example if a certain n-gram appears five times in the candidate sentence, but only twice in the reference, the value is clipped to two.

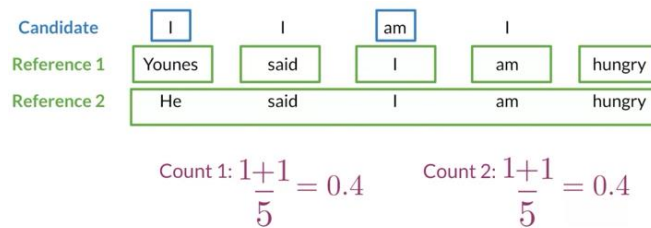
```
1. def average_clipped_precision(candidate, reference):
2.     clipped_precision_score = []
3.     for n_gram_length in range(1, 5):
4.         reference_n_gram_counts = Counter(ngrams(reference, n_gram_length))
5.         candidate_n_gram_counts = Counter(ngrams(candidate, n_gram_length))
6.         total_candidate_ngrams = sum(candidate_n_gram_counts.values())
7.         for ngram in candidate_n_gram_counts:
8.             if ngram in reference_n_gram_counts:
9.                 if candidate_n_gram_counts[ngram] > reference_n_gram_counts[ngram]:
10.                    candidate_n_gram_counts[ngram] = reference_n_gram_counts[ngram] # t
11.             else:
12.                 candidate_n_gram_counts[ngram] = 0
13.         clipped_candidate_ngrams = sum(candidate_n_gram_counts.values())
14.         clipped_precision_score.append(clipped_candidate_ngrams / total_candidate_ngrams)
15.     s = np.exp(np.mean(np.log(clipped_precision_score)))
16.     return s
```

BLEU score:

```
1. def bleu_score(candidate, reference):
2.     BP = brevity_penalty(candidate, reference)
3.     geometric_average_precision = average_clipped_precision(candidate, reference)
4.     return BP * geometric_average_precision
```

Another performance metric: ROUGE-N score

- Recall-Oriented Understudy of Gisting Evaluation(ROUGE)
- Cares about how much of the human created references appear in the candidate translation.
- Example:



F1 score (Using both BLEU and ROUGE-N):

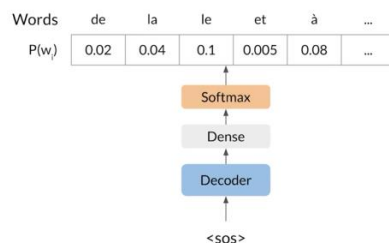
$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \rightarrow F1 = 2 \times \frac{\text{BLEU} \times \text{ROUGE-N}}{\text{BLEU} + \text{ROUGE-N}}$$

$$F1 = 2 \times \frac{0.5 \times 0.4}{0.5 + 0.4} = \frac{4}{9} \approx 0.44$$

- Consideration: all metrics above ignore sentence structure and semantics, focusing only on n-gram matches.

Sampling and Decoding

- Seq2Seq model:



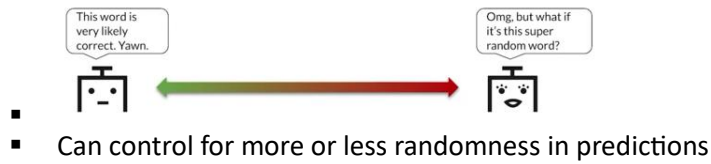
- The decoder outputs probability distribution over words in target language which is produced from a dense layer and a softmax/log softmax operation.
- The final output of the model depends on how you choose the words using the probability distribution at each step.
- Greedy decoding:
 - Select the most probable word at each step.
 - Limitation:
 - But the best word at each step may not be the best for longer sequences.
 - Can be fine for shorter sequences, but limited by inability to look further down the sequence.

- Random sampling:

am	full	hungry	I	the
0.05	0.3	0.15	0.25	0.25

- Provides probabilities for each word and sample accordingly.
- Problem:
 - Often a little too random for accurate translation.
- Solution:
 - Assign more weight to more probable words, and less weight to less probable words

- Temperature



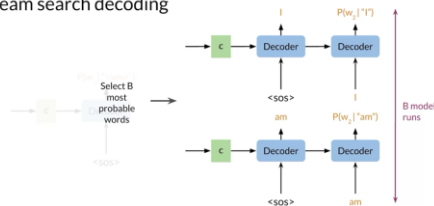
Beam Search

- Beam search decoding:
 - Calculate probability of multiple possible sequences at each step
 - Beam width B determines number of sequences you keep until all B most probable sequences end with <EOS>
 - Beam search with B=1 is greedy decoding.

- Example:



Beam search decoding



- Steps:
 - Start with the SOS (Start Of Sentence) token.
 - Compute probabilities for the first word.
 - Keep the top B=2 probabilities ('I' and 'am').
 - Expand sequences and calculate conditional probabilities for the next words.
 - Multiply the new probabilities with previous step probabilities to find overall sequence probabilities.
 - Keep only the top B=2 sequences and discard the rest.
 - Iterate until the sequences end with an EOS token.
 - Choose the sequence with the highest final probability.
- Considerations:
 - Memory Consumption: Stores B most probable sequences at each step.
 - Computational Cost: Calculates probabilities for all words in the vocabulary for B sequences.
 - Normalization: Normalize probabilities by sequence length to avoid penalizing longer sequences.
- Disadvantages
 - Penalizes Longer Sequences: Without normalization, longer sequences are less likely due to the multiplicative nature of probabilities.
 - Computational Expense: Requires significant memory and computational resources, especially with larger B.

Minimum Bayes Risk (MBR)

- finding a consensus between all candidate translations.
- Step:
 - Generate several random samples.
 - Generate several candidate translations.
 - Compare each sample against each other using a similarity score or a loss function.
 - Assign a similarity to every pair using a similarity score (e.g. ROUGE).
 - Select the sample with the highest average similarity.
 - The translation that you get using this method is the closest to all candidate translations.
- Formula:

$$\arg \max_E \frac{1}{n} \sum_{E'} \text{ROUGE}(E, E')$$

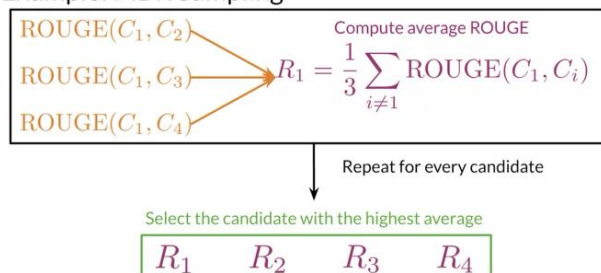
Find the candidate translation that maximizes

Compare with every other candidate

ROUGE score between pair of candidates

- Example:

Example: MBR Sampling



Neural Machine Translation Assignment:

Setup:

```
1. import os
2. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
3. import numpy as np
4. import tensorflow as tf
5. from collections import Counter
6. from utils import (sentences, train_data, val_data, english_vectorizer, portuguese_vectorizer,
7.                    masked_loss, masked_acc, tokens_to_text)
8. VOCAB_SIZE = 12000
9. UNITS = 256
```

Data Demo:

- **portuguese_sentences, english_sentences = sentences**
 - o **portuguese_sentences[-5]:**
 - Não importa o quanto você tenta convencer os outros de que ...
 - o **english_sentences[-5]:**
 - No matter how much you try to convince people that chocolate is ...
- **english_vectorizer.get_vocabulary()[:10]:**
 - o [' ', '[UNK]', '[SOS]', '[EOS]', '.', 'tom', 'i', 'to', 'you', 'the']
- **portuguese_vectorizer.get_vocabulary()[:10]:**
 - o [' ', '[UNK]', '[SOS]', '[EOS]', '.', 'tom', 'que', 'o', 'nao', 'eu']

```
1. vocab_size = portuguese_vectorizer.vocabulary_size() #12000
2. word_to_id = tf.keras.layers.StringLookup(
3.     vocabulary=portuguese_vectorizer.get_vocabulary(),
4.     mask_token="",
5.     oov_token="[UNK]"
6. )
7. id_to_word = tf.keras.layers.StringLookup(
8.     vocabulary=portuguese_vectorizer.get_vocabulary(),
9.     mask_token="",
10.    oov_token="[UNK]",
11.    invert=True,
12. )
13. unk_id = word_to_id("[UNK]") #1
14. sos_id = word_to_id("[SOS]") #2
15. eos_id = word_to_id("[EOS]") #3
16. baunilha_id = word_to_id("baunilha") #7079
```

- **(to_translate, sr_translation), translation in train_data.take(1): first batch**
 - o **to_translate[0, :].numpy():**
 - [2 210 9 146 123 38 9 1672 4 3 0 0 0 0]
 - o **sr_translation[0, :].numpy():**
 - [2 1085 7 128 11 389 37 2038 4 0 0 0 0 0 0]
 - o **translation[0, :].numpy():**
 - [1085 7 128 11 389 37 2038 4 3 0 0 0 0 0 0]

Details:

- Padding has already been applied to the tensors and the value used for this is 0
- Each example consists of 3 different tensors:
 - o The sentence to translate
 - o The shifted-to-the-right translation
 - o The translation

NMT model with attention

- Scaled Dot Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Encoder

```
1. class Encoder(tf.keras.layers.Layer):
2.     def __init__(self, vocab_size, units):
3.         super(Encoder, self).__init__()
4.         self.embedding = tf.keras.layers.Embedding(
5.             input_dim=vocab_size,
6.             output_dim=units,
7.             mask_zero=True #let it know you are using 0 padding
8.         )
9.         self.rnn = tf.keras.layers.Bidirectional(
10.            merge_mode="sum",
11.            layer=tf.keras.layers.LSTM(
12.                units=units,
13.                return_sequences=True
14.            ),
15.        )
16.     def call(self, context):
17.         x = self.embedding(context)
18.         x = self.rnn(x)
19.         return x
20.
21. encoder = Encoder(VOCAB_SIZE, UNITS)
22. encoder_output = encoder(to_translate)
```

Cross Attention between the original sentences and the translations

- Use MultiHeadAttention layer but with head = 1 rather than Attention for simpler code during the forward pass.

```
1. class CrossAttention(tf.keras.layers.Layer):
2.     def __init__(self, units):
3.         super().__init__()
4.         self.mha = (
5.             tf.keras.layers.MultiHeadAttention(
6.                 key_dim=units,
7.                 num_heads=1
8.             )
9.         )
10.        self.layernorm = tf.keras.layers.LayerNormalization()
11.        self.add = tf.keras.layers.Add()
12.     def call(self, context, target):
13.         attn_output = self.mha(
14.             query=target,
15.             value=context
16.         )
17.         x = self.add([target, attn_output])
18.         x = self.layernorm(x)
19.         return x
```


Decoder

```
1. class Decoder(tf.keras.layers.Layer):
2.     def __init__(self, vocab_size, units):
3.         super(Decoder, self).__init__()
4.         self.embedding = tf.keras.layers.Embedding(
5.             input_dim=vocab_size,
6.             output_dim=units,
7.             mask_zero=True
8.         )
9.         self.pre_attention_rnn = tf.keras.layers.LSTM(
10.             units=units,
11.             return_sequences=True,
12.             return_state=True
13.         )
14.         self.attention = CrossAttention(units)
15.         self.post_attention_rnn = tf.keras.layers.LSTM(
16.             units=units,
17.             return_sequences=True
18.         )
19.         self.output_layer = tf.keras.layers.Dense(
20.             units=vocab_size,
21.             activation=tf.nn.log_softmax
22.         )
23.     def call(self, context, target, state=None, return_state=False):
24.         x = self.embedding(target)
25.         x, hidden_state, cell_state = self.pre_attention_rnn(x, initial_state=state)
26.         x = self.attention(context, x)
27.         x = self.post_attention_rnn(x)
28.         logits = self.output_layer(x)
29.         if return_state:
30.             return logits, [hidden_state, cell_state]
31.         return logits
32.
33. decoder = Decoder(VOCAB_SIZE, UNITS)
34. logits = decoder(encoder_output, sr_translation)
```

Translator

```
1. class Translator(tf.keras.Model):
2.     def __init__(self, vocab_size, units):
3.         super().__init__()
4.         self.encoder = Encoder(vocab_size, units)
5.         self.decoder = Decoder(vocab_size, units)
6.     def call(self, inputs):
7.         context, target = inputs
8.         encoded_context = self.encoder(context)
9.         logits = self.decoder(encoded_context, target)
10.        return logits
11.
12. translator = Translator(VOCAB_SIZE, UNITS)
13. logits = translator((to_translate, sr_translation))
```

Model Training:

```
1. def compile_and_train(model, epochs=20, steps_per_epoch=500):
2.     model.compile(optimizer="adam", loss=masked_loss, metrics=[masked_acc, masked_loss])
3.     history = model.fit(
4.         train_data.repeat(),
5.         epochs=epochs,
6.         steps_per_epoch=steps_per_epoch,
7.         validation_data=val_data,
8.         validation_steps=50,
9.         callbacks=[tf.keras.callbacks.EarlyStopping(patience=3)],
10.    )
11.    return model, history
```

Generate next token:

```
1. def generate_next_token(decoder, context, next_token, done, state, temperature=0.0):
2.     logits, state = decoder(context, next_token, state=state, return_state=True)
3.     logits = logits[:, -1, :]
4.     if temperature == 0.0:
5.         next_token = tf.argmax(logits, axis=-1)
6.     else:
7.         logits = logits / temperature
8.         next_token = tf.random.categorical(logits, num_samples=1)
9.     logits = tf.squeeze(logits)
10.    next_token = tf.squeeze(next_token)
11.    logit = logits[next_token].numpy()
12.    next_token = tf.reshape(next_token, shape=(1,1))
13.    if next_token == eos_id:
14.        done = True
15.    return next_token, logit, state, done
```

Translate function:

```
1. def translate(model, text, max_length=50, temperature=0.0):
2.     tokens, logits = [], []
3.     text = tf.convert_to_tensor([text])
4.     context = english_vectorizer(text).to_tensor()
5.     context = model.encoder(context)
6.     next_token = tf.fill((1, 1), sos_id)
7.     state = [tf.zeros((1, UNITS)), tf.zeros((1, UNITS))]
8.     done = False
9.     for i in range(max_length):
10.        next_token, logit, state, done = generate_next_token(
11.            decoder=model.decoder,
12.            context=context,
13.            next_token=next_token,
14.            done=done,
15.            state=state,
16.            temperature=temperature
17.        )
18.        if done:
19.            break
20.
21.        tokens.append(next_token)
22.        logits.append(logit)
23.    tokens = tf.concat(tokens, axis=-1)
24.    translation = tf.squeeze(tokens_to_text(tokens, id_to_word))
25.    translation = translation.numpy().decode()
26.    return translation, logits[-1], tokens
```

Generate samples:

```
1. def generate_samples(model, text, n_samples=4, temperature=0.6):
2.     samples, log_probs = [], []
3.     for _ in range(n_samples):
4.         _, logp, sample = translate(model, text, temperature=temperature)
5.         samples.append(np.squeeze(sample.numpy()).tolist())
6.         log_probs.append(logp)
7.     return samples, log_probs
```

Comparing overlaps:

- Comparing each sample against each other.

Jaccard similarity

- Gets the intersection over union of two sets.

```
1. def jaccard_similarity(candidate, reference):
2.     candidate_set = set(candidate)
3.     reference_set = set(reference)
4.     common_tokens = candidate_set.intersection(reference_set)
5.     all_tokens = candidate_set.union(reference_set)
6.     overlap = len(common_tokens) / len(all_tokens)
7.     return overlap
```

```
1. l1 = [1, 2, 3]
2. l2 = [1, 2, 3, 4]
3. js = jaccard_similarity(l1, l2) # 0.750
```

ROUGE Similarity (more common)

$$score = 2 * \frac{(precision * recall)}{(precision + recall)}$$

```
1. from collections import Counter
2.
3. def rouge1_similarity(candidate, reference):
4.     candidate_word_counts = Counter(candidate)
5.     reference_word_counts = Counter(reference)
6.     overlap = 0
7.     for token in candidate_word_counts.keys():
8.         token_count_candidate = candidate_word_counts[token]
9.         token_count_reference = reference_word_counts[token]
10.        overlap += min(token_count_candidate, token_count_reference)
11.    precision = overlap / len(candidate)
12.    recall = overlap / len(reference)
13.    if precision + recall != 0:
14.        f1_score = 2 * ((precision * recall) / (precision + recall))
15.        return f1_score
16.    return 0
```

Take the average of the overlap.

```
1. def average_overlap(samples, similarity_fn):
2.     scores = {}
3.     for index_candidate, candidate in enumerate(samples):
4.         overlap = 0
5.         for index_sample, sample in enumerate(samples):
6.             if index_candidate == index_sample:
7.                 continue
8.             sample_overlap = similarity_fn(candidate, sample)
9.             overlap += sample_overlap
10.        score = overlap / (len(samples) - 1)
11.        score = round(score, 3)
12.        scores[index_candidate] = score
13.    return scores
```

In practice, It is also common to see the weighted mean begin used:

```
1. def weighted_avg_overlap(samples, log_probs, similarity_fn):
2.     scores = {}
3.     for index_candidate, candidate in enumerate(samples):
4.         overlap, weight_sum = 0.0, 0.0
5.         for index_sample, (sample, logp) in enumerate(zip(samples, log_probs)):
6.             if index_candidate == index_sample:
7.                 continue
8.             sample_p = float(np.exp(logp))
9.             weight_sum += sample_p
10.            sample_overlap = similarity_fn(candidate, sample)
11.            overlap += sample_p * sample_overlap
12.        score = overlap / weight_sum
13.        score = round(score, 3)
14.        scores[index_candidate] = score
15.    return scores
```

Putting everything together:

```
1. def mbr_decode(model, text, n_samples=5, temperature=0.6, similarity_fn=jaccard_similarity):
2.     samples, log_probs = generate_samples(model, text, n_samples=n_samples, temperature=temperature)
3.     scores = weighted_avg_overlap(samples, log_probs, similarity_fn)
4.     decoded_translations = [tokens_to_text(s, id_to_word).numpy().decode('utf-8') for s in samples]
5.     max_score_key = max(scores, key=lambda k: scores[k])
6.     translation = decoded_translations[max_score_key]
7.     return translation, decoded_translations
```

```
1. english_sentence = "I love languages"
2. translation, candidates = mbr_decode(trained_translator, english_sentence, n_samples=10,
temperature=0.6)
3. print("Translation candidates:")
4. for c in candidates:
5.     print(c)
6. print(f"\nSelected translation: {translation}")
```

Limitations of RNNs:

- **Sequential Processing:** RNNs process inputs in a step-by-step manner, which limits parallel computation.
 - **Time Steps:** Denoted by capital T, representing the number of steps to process input. For a 5-word sentence, T equals 5.
 - **Scalability:** The longer the input sequence, the longer the processing time, affecting efficiency.
- **Information Bottleneck:** Long sequences can lead to difficulty in learning and remembering information through the sequence.
 - **Information Loss:** Through long sequences, pertinent information may be lost.
- **Improvements:** LSTM and GRU architectures mitigate some problems but still struggle with very long sequences. Attention previously implemented in seq2seq model but still reliant on RNNs.

Transformers (Attention is all you need)

- **Attention-Based:** only use attention mechanisms without the need for RNNs.
- **Components:** Consist of linear and non-linear transformations in addition to attention.
- **Advantages:** Overcomes RNN limitations with speed and context handling.

Transformer Architecture

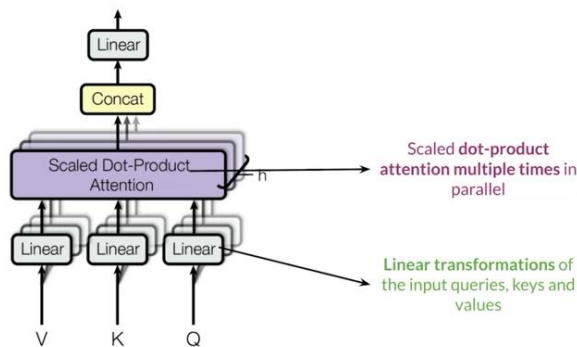
- **Core Mechanism:** Scale dot-product attention.



○ (Vaswani et al., 2017)

- Provides computational and memory efficiency through matrix multiplications.

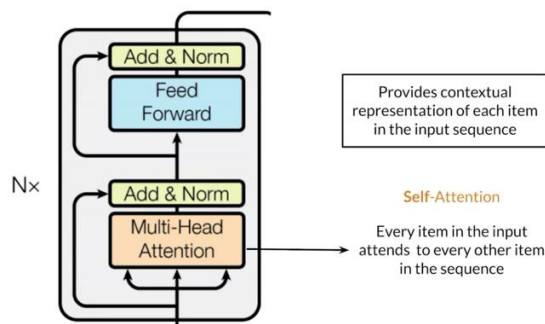
- **Multi-head Attention Layer:**



-
- **Parallel Processing:** Runs multiple attention mechanisms in parallel.
- **Components:** Consists of scale dot-product attention with linear transformations for input queries, keys, and values.
- **Parameters:** Linear transformations are learnable within the model.

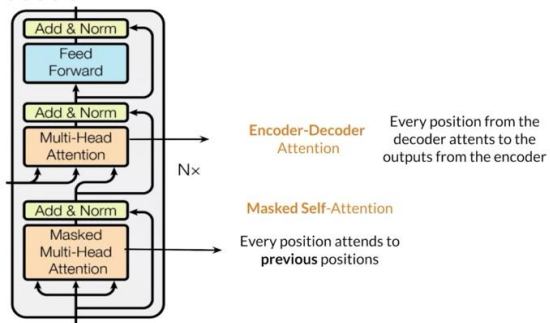
Encoder and Decoder

- **Encoder:**



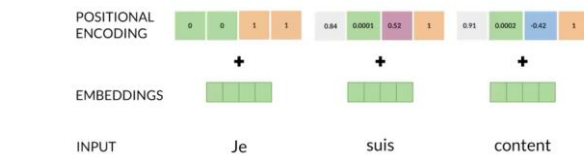
-
- **Structure:** Begins with multi-head attention for self-attention on the input sequence.
- **Processes:**
 - Self-attention within input sequence.
 - Followed by residual connection and normalization.
 - Includes a feed-forward layer.
 - Ends with another residual connection and normalization.
- **Repetition:** Multiple encoder layers are stacked (repeated N times).
- **Function:** Provides contextual representation for each input.

- Decoder:



-
- **Masked Attention:** First module uses masked attention to prevent forward-looking information flow.
- **Integration:** Second module integrates encoder outputs for full sequence attention.
- **Repetition:** Decoder layers are also repeated sequentially.

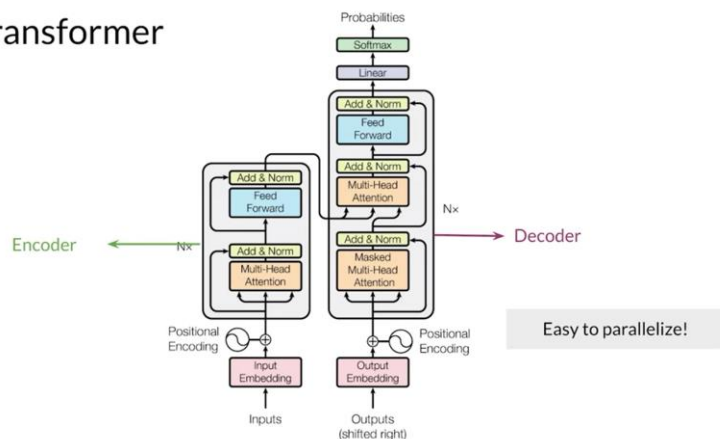
Positional Encoding



- Purpose: Encodes the position of each input in the sequence, essential due to the non-recurrent nature of transformers.
- Implementation: Can be either learned or fixed, akin to word embeddings.
- Example: French phrase translation using positional encoding for sequence order.

Transformer Architecture Overview:

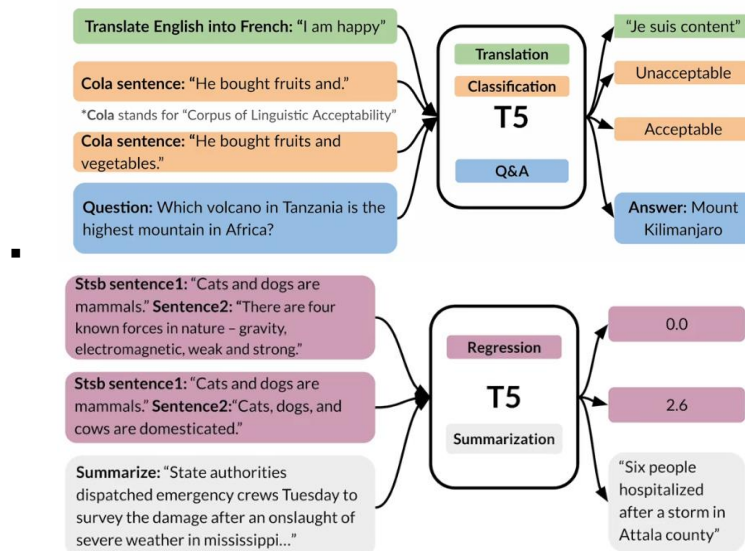
The Transformer



- **Input Processing:**
 - Sentence embedding followed by positional encoding.
 - Passes through multiple layers of encoder multi-head attention modules.
- **Output Processing:**
 - Decoder takes the encoder's output.
 - Shifts output sentence one step to the right for prediction.
 - Uses linear layer with softmax activation for output probabilities.
- **Advantages:**
 - Allows easy parallelization and efficient training on GPUs, scalable for multiple tasks and large datasets.

Transformer Applications

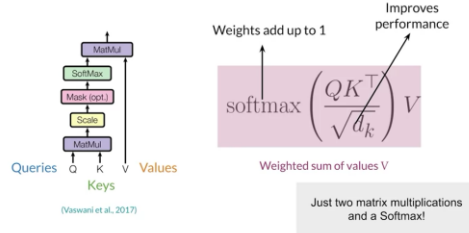
- NLP Tasks:
 - o Automatic text summarization.
 - o Autocompletion.
 - o Named entity recognition (NER).
 - o Automatic question answering (QA).
 - o Machine translation.
 - o Chatbot development.
 - o Sentiment analysis.
 - o Market intelligence.
- State-of-the-Art Transformer Models
 - o GPT-2:
 - o BERT:
 - o T5 (Text-to-Text Transfer Transformer):
- Deep Dive into T5
 - o Multitask Learning:
 - A single T5 model can perform multiple tasks, substituting the need for individual models for each task.
 - Tasks are specified via input strings that include both the task directive and the data.
 - o Examples of T5 Tasks:



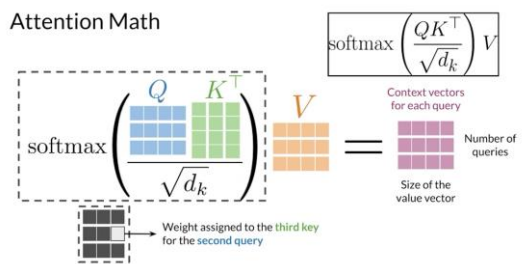
- Translation: Input "translate English to French: I am happy" yields "Je suis content."
- Classification: Classifies sentences as "acceptable" or "unacceptable."
- Question Answering: Answers questions like "which volcano in Tanzania is the highest mountain in Africa?" with "Mount Kilimanjaro."

Scaled and Dot-Product Attention

Scaled dot-product attention

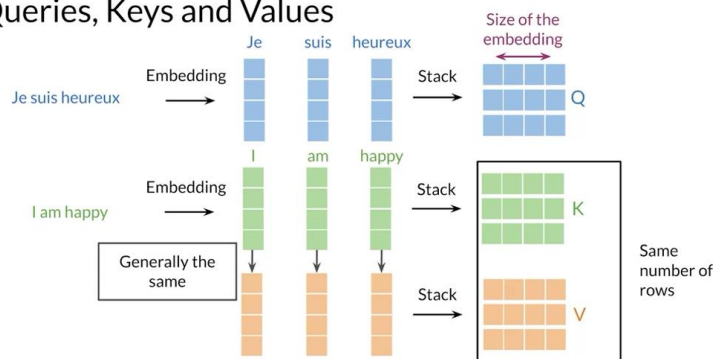


Attention Math



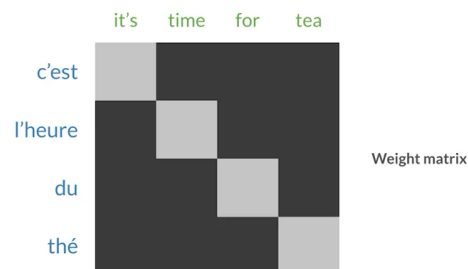
Queries, Keys and Values:

Queries, Keys and Values



Encoder-Decoder Attention:

Queries from one sentence, keys and values from another



Self-Attention:

Queries, keys and values come from the same sentence



- Every word attends to every other word in the sequence → A contextual representation of the meaning of each word within the sentence.

Masked Self-Attention

Queries, keys and values come from the **same sentence**. Queries don't attend to future positions.

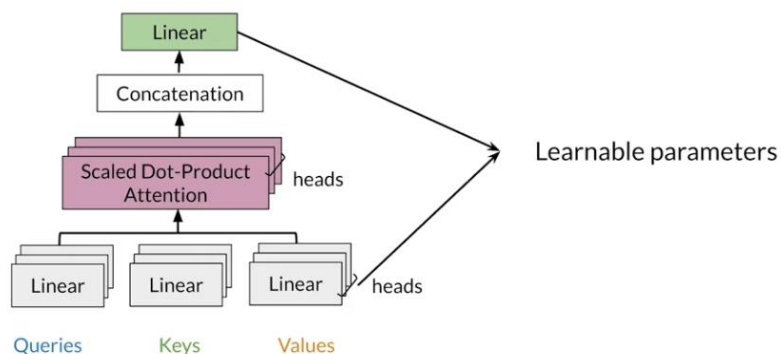


- Ensures that predictions at each position depend only on the known outputs.
- Math:

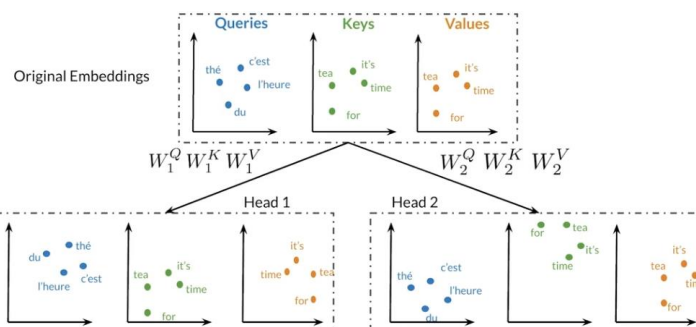
$$\text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} + \begin{matrix} \text{Minus infinity} \\ \begin{bmatrix} 0 & \text{Minus infinity} & \text{Minus infinity} \\ 0 & 0 & \text{Minus infinity} \\ 0 & 0 & 0 \end{bmatrix} \end{matrix} \right) V$$

Weights assigned to future positions are equal to 0

Multi-head Attention

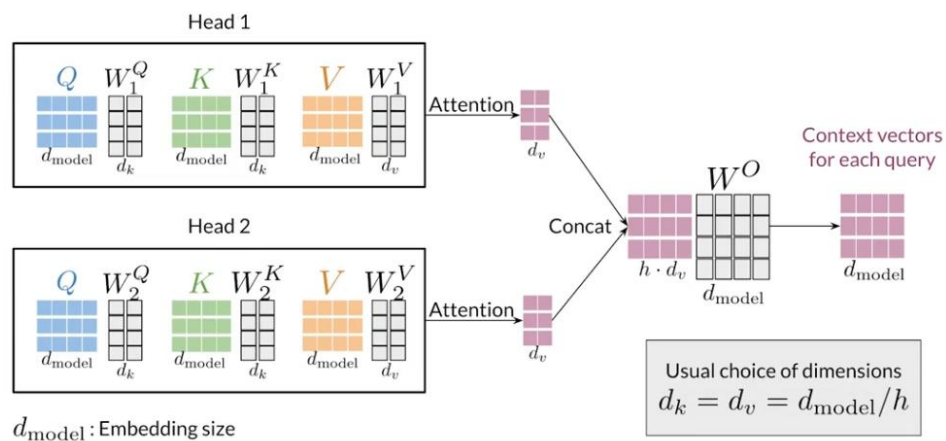


- Intuition:



- to enable the model to jointly attend to information from different representation subspaces at different positions. With single-head attention, the model is limited to focusing on one set of weights at a time. In contrast, multi-head attention allows the model to look at different parts of the sequence differently, which is akin to having multiple independent attention mechanisms focusing on various parts of the input.

Multi-Head Attention in more details:



- **Input Embeddings:** Start with embeddings for queries (Q), keys (K), and values (V).
- **Parallel Transformations:** Apply linear transformations to Q, K, and V for each attention head, using different weight matrices W
- **Scaled Dot-Product Attention:** Perform the scaled dot-product attention in parallel for each transformed set of Q, K, and V.
- **Concatenation:** Concatenate the outputs of each head's attention.
- **Final Linear Transformation:** Apply a final linear transformation using weights W_O to the concatenated output.

Mathematical details for one head:

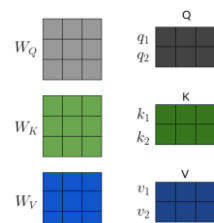
- Given a word:

Word 1

Word 2

○

- Take its embedding then multiply it by W_Q , W_K , W_V matrix to get the corresponding queries, keys and values.



○

- Calculate scores with those embedding:

$$\text{Softmax} \left[\frac{Q \cdot K^T}{\sqrt{d_k}} \right] \times V = Z$$

○

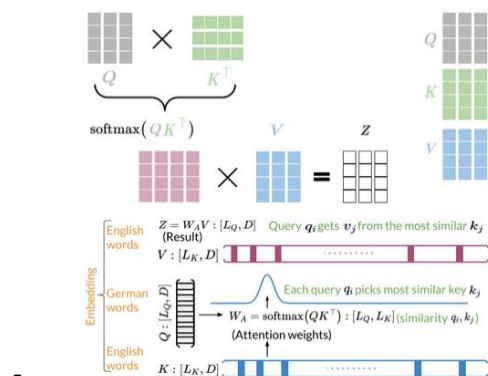
- For n heads, concatenate them and multiply by a W_O matrix:

$$\begin{bmatrix} Z_1 & Z_2 & \dots & Z_n \end{bmatrix} \cdot W_O = Z$$

○

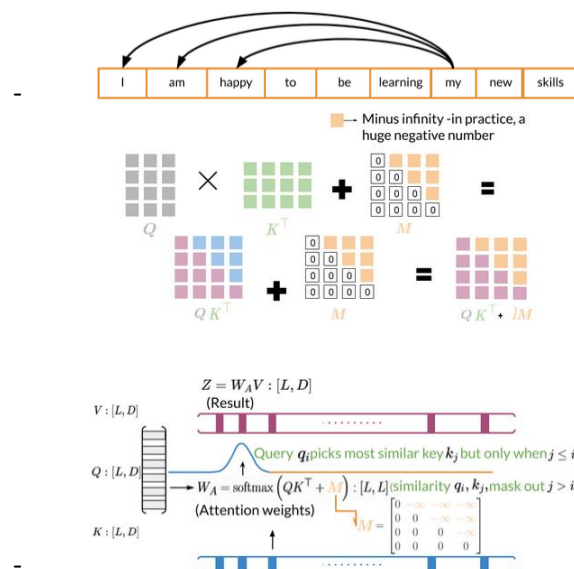
Attention Lab:

Dot product attention:



Bi-directional self-attention captures interactions between every word embedding in your query and every word in your key in the same sentences.

Causal attention considers only words which have come before the current one, with a mask.



Lab setup:

```
1. import os
2. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
3. import sys
4. import tensorflow as tf
5. import textwrap
6. wrapper = textwrap.TextWrapper(width=70)
```

Helper function:

```
1. def display_tensor(t, name):
2.     print(f'{name} shape: {t.shape}\n')
3.     print(f'{t}\n')
4.
5. q = tf.constant([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
6. k = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
7. v = tf.constant([[0.0, 1.0, 0.0], [1.0, 0.0, 1.0]])
8. m = tf.constant([[1.0, 0.0], [1.0, 1.0]])
```

Dot product attention:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d}} + M\right)V$$

```
1. def dot_product_attention(q, k, v, mask, scale=True):
2.     matmul_qk = tf.matmul(q, k, transpose_b=True) # (... , seq_len_q, seq_len_k)
3.     if scale:
4.         dk = tf.cast(tf.shape(k)[-1], tf.float32)
5.         matmul_qk = matmul_qk / tf.math.sqrt(dk)
6.     if mask is not None:
7.         matmul_qk = matmul_qk + (1. - mask) * -1e9
8.     attention_weights = tf.keras.activations.softmax(matmul_qk)
9.     attention_output = tf.matmul(attention_weights, v) # (... , seq_len_q, depth_v)
10.    return attention_output
```

Causal dot product attention:

```
1. def causal_dot_product_attention(q, k, v, scale=True):
2.     mask_size = q.shape[-2]
3.     mask = tf.experimental.numpy.tril(tf.ones((mask_size, mask_size)))
4.     # Creates a matrix with ones below the diagonal and 0s above. It should have shape (1, mask_size, mask_size)
5.     return dot_product_attention(q, k, v, mask, scale=scale)

1. result = causal_dot_product_attention(q, k, v)
```

Masking Lab:

Masking help the softmax computation give the appropriate weights to the words in the input sentence.

Setup:

```
1. import os
2. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
3. import tensorflow as tf
```

Padding mask:

- If the maximum length of a sequence your network can process is 5, truncate the sequences or pad the sequences with zeros with meet the same length 5 as it is important that they are of uniform length when passing into a transformer model:

```
[["Do", "you", "know", "when", "Jane", "is", "going", "to", "visit", "Africa"],
 ["Jane", "visits", "Africa", "in", "September" ],
 ["Exciting", "!"]]
- ]
→
[[ 71, 121, 4, 56, 99, 2344, 345, 1284, 15],
 [ 56, 1285, 15, 181, 545],
 [ 87, 600]]
- ]
→
[[ 71, 121, 4, 56, 99],
 [ 2344, 345, 1284, 15, 0],
 [ 56, 1285, 15, 181, 545],
 [ 87, 600, 0, 0, 0]]
- ]
```

Create the mask of an already padded sequence:

```
1. def create_padding_mask(decoder_token_ids):
2.     seq = 1 - tf.cast(tf.math.equal(decoder_token_ids, 0), tf.float32)
3.     return seq[:, tf.newaxis, :]

1. x = tf.constant([[7., 6., 0., 0., 0.], [1., 2., 3., 0., 0.], [3., 0., 0., 0., 0.]])
2. print(create_padding_mask(x))
3. #tf.Tensor(
4. # [[1. 1. 0. 0. 0.]]
5. # [[1. 1. 1. 0. 0.]]
6. # [[1. 0. 0. 0. 0.]]], shape=(3, 1, 5), dtype=float32)
```

Usage:

```
1. mask = create_padding_mask(x)
2. x_extended = x[:, tf.newaxis, :]
3. res = tf.keras.activations.softmax(x_extended + (1 - mask) * -1.0e9)
```

Look-ahead Mask:

- It helps your model pretend that it correctly predicted a part of the output and see if, without looking ahead, it can correctly predict the next output.
- Example:
 - o The expected correct output is [1,2,3], mask out the values and input the masked sequence [1, -1e9, -1e9] and see if it could generate [1, 2, -1e9]

```
1. def create_look_ahead_mask(sequence_length):
2.     mask = tf.linalg.band_part(tf.ones((1, sequence_length, sequence_length)), -1, 0)
3.     return mask

1. x = tf.random.uniform((1, 3))
2. temp = create_look_ahead_mask(x.shape[1])
3. #temp:
4. #<tf.Tensor: shape=(1, 3, 3), dtype=float32, numpy=
5. #array([[[1., 0., 0.],
6. #       [1., 1., 0.],
7. #       [1., 1., 1.]])], dtype=float32)>
```

Position Encoding

Setup:

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

Encode the positions of your inputs and pass them into the network using the sine and cosine formulas:

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

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

- d: the dimension of the word embedding and positional embedding
- pos: the position of the word
- k: refers to each of the different dimensions in the positional encodings, with $i = k // 2$

Intuition:

- think of positional encodings broadly as a feature that contains the information about the relative positions of words.
- The values of the sine and cosine equations are small enough (between -1 and 1) that when adding the positional encoding to a word embedding, the word embedding is not significantly distorted, and is instead enriched.

Sine and Cosine Angles:

Consider the inner term as you calculate the positional encoding for a word in a sequence:

$$\theta(pos, i, d) = \frac{pos}{10000^{\frac{2i}{d}}}$$

$$PE_{(pos, 0)} = \sin\left(\frac{pos}{10000^{\frac{0}{d}}}\right)$$

$$PE_{(pos, 1)} = \cos\left(\frac{pos}{10000^{\frac{0}{d}}}\right)$$

- The angle is the same for both, and so for $PE(pos, 2)$ and $PE(pos, 3)$ as $i = 1$ in both cases.

This relationship holds true for all paired sine and cosine curves:

k	0	1	2	3	...	d - 2	d - 1
encoding(0) =	$[\sin(\theta(0, 0, d))$	$\cos(\theta(0, 0, d))$	$\sin(\theta(0, 1, d))$	$\cos(\theta(0, 1, d))$...	$\sin(\theta(0, d/2, d))$	$\cos(\theta(0, d/2, d))]$
encoding(1) =	$[\sin(\theta(1, 0, d))$	$\cos(\theta(1, 0, d))$	$\sin(\theta(1, 1, d))$	$\cos(\theta(1, 1, d))$...	$\sin(\theta(1, d/2, d))$	$\cos(\theta(1, d/2, d))]$
...							
encoding(pos) =	$[\sin(\theta(pos, 0, d))$	$\cos(\theta(pos, 0, d))$	$\sin(\theta(pos, 1, d))$	$\cos(\theta(pos, 1, d))$...	$\sin(\theta(pos, d/2, d))$	$\cos(\theta(pos, d/2, d))]$

Calculate the angle:

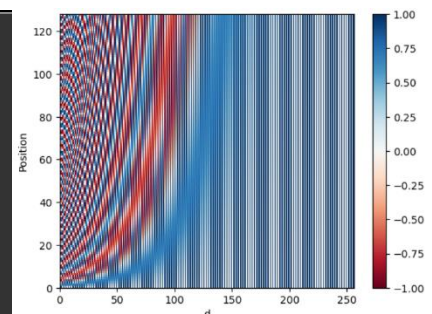
```
1. def get_angles(position, k, d_model):
2.     i = k // 2
3.     angle_rates = 1 / np.power(10000, (2 * i) / np.float32(d_model))
4.     return position * angle_rates
```

Since and Cosine Positional Encodings:

```
1. def positional_encoding(positions, d):
2.     """
3.     Precomputes a matrix with all the positional encodings
4.     Arguments:
5.         positions (int): Maximum number of positions to be encoded
6.         d (int): Encoding size
7.     Returns:
8.         pos_encoding (tf.Tensor): A matrix of shape (1, position, d_model) with the
positional encodings
9.     """
10.    # initialize a matrix angle_rads of all the angles
11.    angle_rads = get_angles(np.arange(positions)[:, np.newaxis],
12.                             np.arange(d)[np.newaxis, :],
13.                             d)
14.    # apply sin to even indices in the array; 2i
15.    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
16.    # apply cos to odd indices in the array; 2i+1
17.    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
18.    pos_encoding = angle_rads[np.newaxis, ...]
19.    return tf.cast(pos_encoding, dtype=tf.float32)
```

Visualizign the positional encodings:

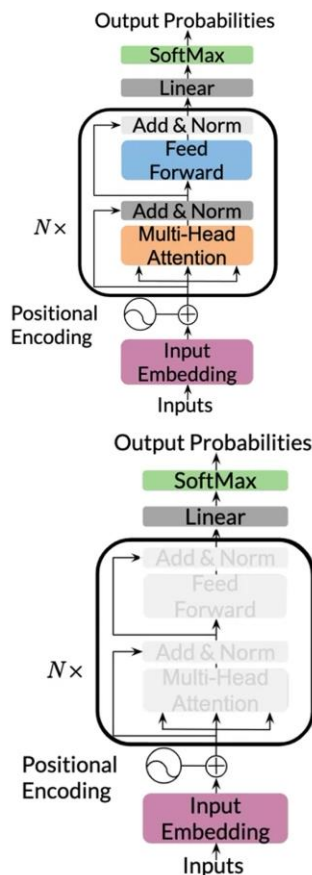
```
1. pos_encoding = positional_encoding(128, 256)
2.
3. plt.pcolormesh(pos_encoding[0], cmap='RdBu')
4. plt.xlabel('d')
5. plt.xlim((0, 256))
6. plt.ylabel('Position')
7. plt.colorbar()
8. plt.show()
```



Each row represents a positional encoding, none of the rows are identical, there will be a unique positional encoding for each of the words.

Transformer decoder

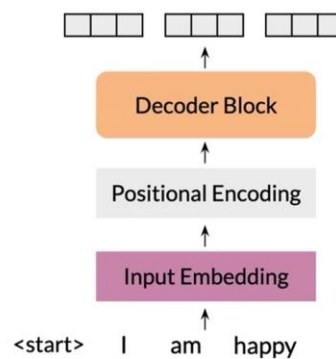
Transformer decoder



Overview

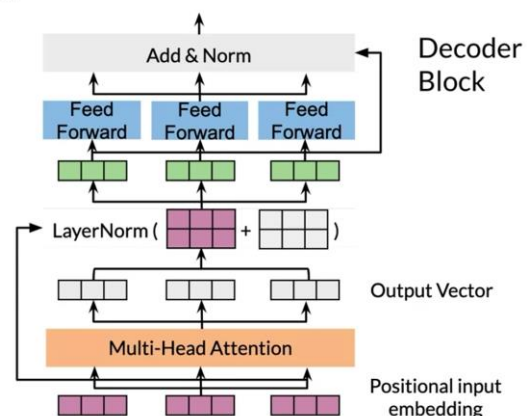
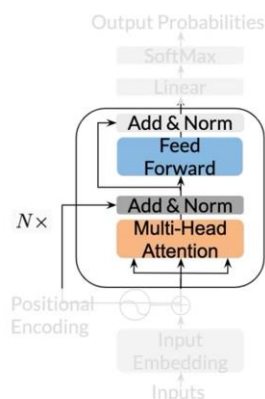
- input: sentence or paragraph
 - we predict the next word
- sentence gets embedded, add positional encoding
 - (vectors representing $\{0, 1, 2, \dots, K\}$)
- multi-head attention looks at previous words
- feed-forward layer with ReLU
 - that's where most parameters are!
- residual connection with layer normalization
- repeat N times
- dense layer and softmax for output

Explanation



- Once get the embeddings, append to it the positional encoding. Then, do multi-head attention \rightarrow feedforward layer with a ReLU \rightarrow a residual connection with layer normalization (repeat the block shown above N times) \rightarrow a linear layer with a softmax.
- The block that gets repeated N times:

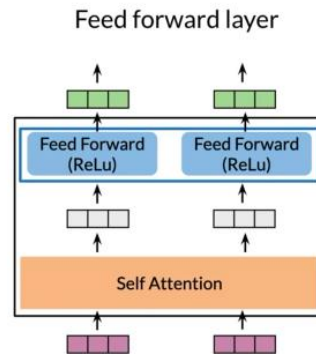
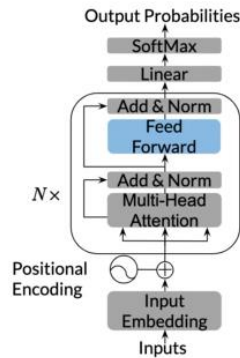
The Transformer decoder



- Positional Input Embedding:
 - Combination of input sequence vectors and positional coding vectors.
- Multi-Headed Attention:
 - Processes words while considering relationships across positions.
- Residual Connection and Layer Normalization:
 - Follow each attention layer.

- Feedforward block:

The Transformer decoder



Technical details for data processing:

Model Input:

ARTICLE TEXT <EOS> SUMMARY <EOS> <pad> ...

Tokenized version:

[2,3,5,2,1,3,4,7,8,2,5,1,2,3,6,2,1,0,0]

- **Weighted Loss:** Essential to focus on summary words during training.
- **Weights:** Zero for article words, one (or other values for sparse data) for summary words.
- **Cost Function:** Cross-entropy that ignores words from the article, targeting only the summary.

Inference with a language model:

Model input:

[Article] <EOS> [Summary] <EOS>

Inference:

- Provide: [Article] <EOS>
- Generate summary word-by-word
 - until the final <EOS>
- Pick the next word by random sampling
 - each time you get a different summary!

Transformer Summarizer

Setup:

```
1. import os
2. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
3. import numpy as np
4. import pandas as pd
5. import tensorflow as tf
6. import matplotlib.pyplot as plt
7. import time
8. import utils
9. import textwrap
10. wrapper = textwrap.TextWrapper(width=70)
11. tf.keras.utils.set_random_seed(10)
```

Dataset:

```
1. data_dir = "data/corpus"
2. train_data, test_data = utils.get_train_test_data(data_dir)
3. example_summary, example_dialogue = train_data.iloc[10]
4. print(f"Dialogue:\n{example_dialogue}")
5. print(f"\nSummary:\n{example_summary}")
```

```
Dialogue:
Lucas: Hey! How was your day?
Demi: Hey there!
Demi: It was pretty fine, actually, thank you!
Demi: I just got promoted! :D
Lucas: Whoa! Great news!
Lucas: Congratulations!
Lucas: Such a success has to be celebrated.
Demi: I agree! :D
Demi: Tonight at Death & Co.?
Lucas: Sure!
Lucas: See you there at 10pm?
Demi: Yeah! See you there! :D

Summary:
Demi got promoted. She will celebrate that with Lucas at Death & Co at 10 pm.
```

Preprocessing:

```
1. document, summary = utils.preprocess(train_data)
2. document_test, summary_test = utils.preprocess(test_data)

1. # The [ and ] from default tokens cannot be removed as they mark the SOS and EOS token.
2. filters = '!\"#$%&()*+,-./:;<=>?@\\^_`{|}~\t\n'
3. oov_token = '[UNK]'
4. tokenizer = tf.keras.preprocessing.text.Tokenizer(filters=filters, oov_token=oov_token, lower=False)
5. documents_and_summary = pd.concat([document, summary], ignore_index=True)
6. tokenizer.fit_on_texts(documents_and_summary)
7. inputs = tokenizer.texts_to_sequences(document)
8. targets = tokenizer.texts_to_sequences(summary)
9. vocab_size = len(tokenizer.word_index) + 1
10. print(f'Size of vocabulary: {vocab_size}') #34250

1. # Limit the size of the input and output data
2. encoder_maxlen = 150
3. decoder_maxlen = 50
4. # Pad the sequences.
5. inputs = tf.keras.preprocessing.sequence.pad_sequences(inputs, maxlen=encoder_maxlen,
padding='post', truncating='post')
6. targets = tf.keras.preprocessing.sequence.pad_sequences(targets, maxlen=decoder_maxlen,
padding='post', truncating='post')
7. inputs = tf.cast(inputs, dtype=tf.int32)
8. targets = tf.cast(targets, dtype=tf.int32)
9. # Create the final training dataset.
10. BUFFER_SIZE = 10000
11. BATCH_SIZE = 64
12. dataset = tf.data.Dataset.from_tensor_slices((inputs, targets)).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Positional Encoding:

```

1. def positional_encoding(positions, d_model):
2.     position = np.arange(positions)[:, np.newaxis]
3.     k = np.arange(d_model)[np.newaxis, :]
4.     i = k // 2
5.     angle_rates = 1 / np.power(10000, (2 * i) / np.float32(d_model))
6.     angle_rads = position * angle_rates
7.     angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
8.     angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
9.     pos_encoding = angle_rads[np.newaxis, ...]
10.    return tf.cast(pos_encoding, dtype=tf.float32)

```

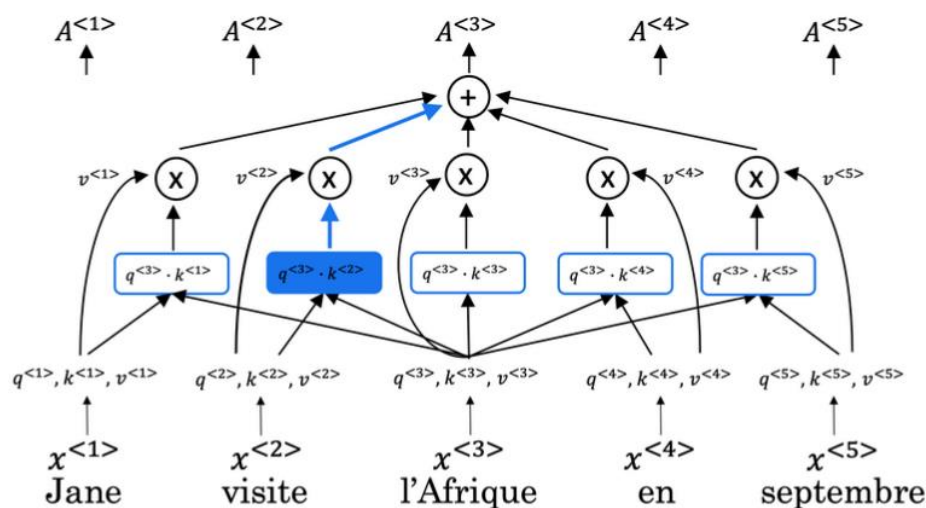
Masking:

```

1. def create_padding_mask(decoder_token_ids):
2.     seq = 1 - tf.cast(tf.math.equal(decoder_token_ids, 0), tf.float32)
3.     return seq[:, tf.newaxis, :]
4.
5. def create_look_ahead_mask(sequence_length):
6.     mask = tf.linalg.band_part(tf.ones((1, sequence_length, sequence_length)), -1, 0)
7.     return mask

```

Self Attention (Scaled dot product attention)



$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

```

1. def scaled_dot_product_attention(q, k, v, mask):
2.     matmul_qk = tf.matmul(q, k, transpose_b=True)
3.     dk = tf.cast(tf.shape(k)[-1], tf.float32)
4.     scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
5.     if mask is not None:
6.         output = tf.matmul(tf.keras.activations.softmax(scaled_attention_logits + (1 - mask) * -1.0e9), v)
7.         return output, tf.keras.activations.softmax(scaled_attention_logits + (1 - mask) * -1.0e9)
8.     else:
9.         output = tf.matmul(tf.keras.activations.softmax(scaled_attention_logits), v)
10.    return output, tf.keras.activations.softmax(scaled_attention_logits)

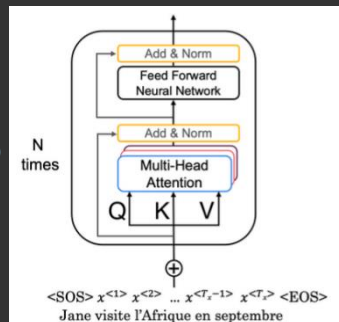
```

Full Encoder

```

1. class Encoder(tf.keras.layers.Layer):
2.     def __init__(self, num_layers, embedding_dim, num_heads, fully_connected_dim, input_vocab_size,
3.                   maximum_position_encoding, dropout_rate=0.1, layernorm_eps=1e-6):
4.         super(Encoder, self).__init__()
5.         self.embedding_dim = embedding_dim
6.         self.num_layers = num_layers
7.         self.embedding = tf.keras.layers.Embedding(input_vocab_size, self.embedding_dim)
8.         self.pos_encoding = positional_encoding(maximum_position_encoding,
9.                                                  self.embedding_dim)
10.        self.enc_layers = [EncoderLayer(embedding_dim=self.embedding_dim,
11.                                         num_heads=num_heads,
12.                                         fully_connected_dim=fully_connected_dim,
13.                                         dropout_rate=dropout_rate,
14.                                         layernorm_eps=layernorm_eps)
15.                            for _ in range(self.num_layers)]
16.        self.dropout = tf.keras.layers.Dropout(dropout_rate)
17.
18.    def call(self, x, training, mask):
19.        seq_len = tf.shape(x)[1]
20.        x = self.embedding(x) # (batch_size, input_seq_len, embedding_dim)
21.        x *= tf.math.sqrt(tf.cast(self.embedding_dim, tf.float32))
22.        x += self.pos_encoding[:, :seq_len, :]
23.        x = self.dropout(x, training=training)
24.        for i in range(self.num_layers):
25.            x = self.enc_layers[i](x, training, mask)
26.        return x # (batch_size, input_seq_len, embedding_dim)

```



Decoder:

```

1. class DecoderLayer(tf.keras.layers.Layer):
2.     def __init__(self, embedding_dim, num_heads, fully_connected_dim, dropout_rate=0.1,
3.                   layernorm_eps=1e-6):
4.         super(DecoderLayer, self).__init__()
5.         self.mha1 = tf.keras.layers.MultiHeadAttention(
6.             num_heads=num_heads,
7.             key_dim=embedding_dim,
8.             dropout=dropout_rate
9.         )
10.        self.mha2 = tf.keras.layers.MultiHeadAttention(
11.            num_heads=num_heads,
12.            key_dim=embedding_dim,
13.            dropout=dropout_rate
14.        )
15.        self.ffn = FullyConnected(
16.            embedding_dim=embedding_dim,
17.            fully_connected_dim=fully_connected_dim
18.        )
19.        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)
20.        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)
21.        self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=layernorm_eps)
22.        self.dropout_ffn = tf.keras.layers.Dropout(dropout_rate)
23.    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
24.        mult_attn_out1, attn_weights_block1 = self.mha1(
25.            query=x, key=x, value=x, attention_mask=look_ahead_mask, return_attention_scores=True,
26.            training=training
27.        )
28.        Q1 = self.layernorm1(mult_attn_out1+x)
29.        mult_attn_out2, attn_weights_block2 = self.mha2(
30.            query=Q1, key=enc_output, value=enc_output, attention_mask=padding_mask,
31.            return_attention_scores=True, training=training
32.        )
33.        mult_attn_out2 = self.layernorm2(mult_attn_out2 + Q1)
34.        ffn_output = self.ffn(mult_attn_out2)
35.        ffn_output = self.dropout_ffn(ffn_output, training=training)
36.        out3 = self.layernorm3(ffn_output+mult_attn_out2)
37.        return out3, attn_weights_block1, attn_weights_block2

```

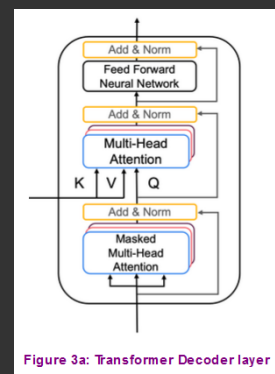


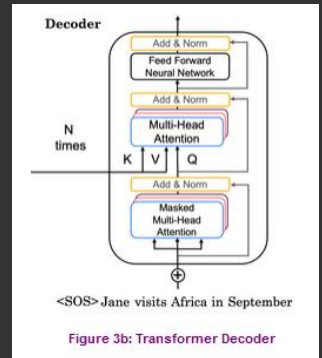
Figure 3a: Transformer Decoder layer

Full Decoder:

```

1. class Decoder(tf.keras.layers.Layer):
2.     def __init__(self, num_layers, embedding_dim, num_heads, fully_connected_dim, target_vocab_size,
3.         maximum_position_encoding, dropout_rate=0.1, layernorm_eps=1e-6):
4.         super(Decoder, self).__init__()
5.         self.embedding_dim = embedding_dim
6.         self.num_layers = num_layers
7.         self.embedding = tf.keras.layers.Embedding(target_vocab_size, self.embedding_dim)
8.         self.pos_encoding = positional_encoding(maximum_position_encoding, self.embedding_dim)
9.         self.dec_layers = [DecoderLayer(embedding_dim=self.embedding_dim,
10.             num_heads=num_heads,
11.             fully_connected_dim=fully_connected_dim,
12.             dropout_rate=dropout_rate,
13.             layernorm_eps=layernorm_eps)
14.             for _ in range(self.num_layers)]
15.         self.dropout = tf.keras.layers.Dropout(dropout_rate)
16.     def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
17.         seq_len = tf.shape(x)[1]
18.         attention_weights = {}
19.         x = self.embedding(x)
20.         x *= tf.math.sqrt(tf.cast(self.embedding_dim, tf.float32))
21.         x += self.pos_encoding[:, :seq_len, :]
22.         x = self.dropout(x, training=training)
23.         for i in range(self.num_layers):
24.             x, block1, block2 = self.dec_layers[i](x, enc_output, training, look_ahead_mask, padding_mask)
25.             attention_weights['decoder_layer{}_block1_self_att'.format(i+1)] = block1
26.             attention_weights['decoder_layer{}_block2_decenc_att'.format(i+1)] = block2
27.         return x, attention_weights

```

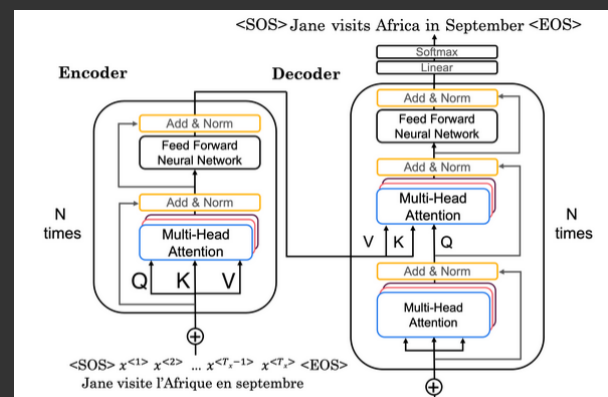


Transformer:

```

1. class Transformer(tf.keras.Model):
2.     def __init__(self, num_layers, embedding_dim, num_heads, fully_connected_dim, input_vocab_size,
3.         target_vocab_size, max_positional_encoding_input,
4.         max_positional_encoding_target, dropout_rate=0.1, layernorm_eps=1e-6):
5.         super(Transformer, self).__init__()
6.         self.encoder = Encoder(num_layers=num_layers,
7.             embedding_dim=embedding_dim,
8.             num_heads=num_heads,
9.             fully_connected_dim=fully_connected_dim,
10.             input_vocab_size=input_vocab_size,
11.             maximum_position_encoding=max_positional_encoding_input,
12.             dropout_rate=dropout_rate,
13.             layernorm_eps=layernorm_eps)
14.         self.decoder = Decoder(num_layers=num_layers,
15.             embedding_dim=embedding_dim,
16.             num_heads=num_heads,
17.             fully_connected_dim=fully_connected_dim,
18.             target_vocab_size=target_vocab_size,
19.             maximum_position_encoding=max_positional_encoding_target,
20.             dropout_rate=dropout_rate,
21.             layernorm_eps=layernorm_eps)
22.         self.final_layer = tf.keras.layers.Dense(target_vocab_size, activation='softmax')
23.     def call(self, input_sentence, output_sentence, training, enc_padding_mask, look_ahead_mask, dec_padding_mask):
24.         enc_output = self.encoder(input_sentence, training, enc_padding_mask)
25.         dec_output, attention_weights = self.decoder(output_sentence, enc_output, training, look_ahead_mask,
26.             dec_padding_mask)
27.         final_output = self.final_layer(dec_output)
28.         return final_output, attention_weights

```



Initialize the model:

```
1. num_layers = 2
2. embedding_dim = 128
3. fully_connected_dim = 128
4. num_heads = 2
5. positional_encoding_length = 256
6. transformer = Transformer(
7.     num_layers,
8.     embedding_dim,
9.     num_heads,
10.    fully_connected_dim,
11.    vocab_size,
12.    vocab_size,
13.    positional_encoding_length,
14.    positional_encoding_length,
15. )
```

Prepare for training the model:

```
1. class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
2.     def __init__(self, d_model, warmup_steps=4000):
3.         super(CustomSchedule, self).__init__()
4.         self.d_model = tf.cast(d_model, dtype=tf.float32)
5.         self.warmup_steps = warmup_steps
6.     def __call__(self, step):
7.         step = tf.cast(step, dtype=tf.float32)
8.         arg1 = tf.math.rsqrt(step)
9.         arg2 = step * (self.warmup_steps ** -1.5)
10.        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
11. learning_rate = CustomSchedule(embedding_dim)
12. optimizer = tf.keras.optimizers.Adam(0.0002, beta_1=0.9, beta_2=0.98, epsilon=1e-9)
```

Loss:

```
1. loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False, reduction='none')
2. def masked_loss(real, pred):
3.     mask = tf.math.logical_not(tf.math.equal(real, 0))
4.     loss_ = loss_object(real, pred)
5.     mask = tf.cast(mask, dtype=loss_.dtype)
6.     loss_ *= mask
7.     return tf.reduce_sum(loss_)/tf.reduce_sum(mask)
8. train_loss = tf.keras.metrics.Mean(name='train_loss')
9. losses = []
```

Training the model:

```
1. @tf.function
2. def train_step(model, inp, tar):
3.     tar_inp = tar[:, :-1]
4.     tar_real = tar[:, 1:]
5.     enc_padding_mask = create_padding_mask(inp)
6.     look_ahead_mask = create_look_ahead_mask(tf.shape(tar_inp)[1])
7.     dec_padding_mask = create_padding_mask(inp)
8.     with tf.GradientTape() as tape:
9.         predictions, _ = model(
10.            inp,
11.            tar_inp,
12.            True,
13.            enc_padding_mask,
14.            look_ahead_mask,
15.            dec_padding_mask
16.        )
17.        loss = masked_loss(tar_real, predictions)
18.        gradients = tape.gradient(loss, transformer.trainable_variables)
19.        optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))
20.        train_loss(loss)
```


Summarization: predict next word

```
1. def next_word(model, encoder_input, output):
2.     enc_padding_mask = create_padding_mask(encoder_input)
3.     look_ahead_mask = create_look_ahead_mask(tf.shape(output)[1])
4.     dec_padding_mask = create_padding_mask(output)
5.     predictions, attention_weights = model(
6.         encoder_input,
7.         output,
8.         False, # False because it's prediction, not training
9.         enc_padding_mask,
10.        look_ahead_mask,
11.        dec_padding_mask
12.    )
13.    predictions = predictions[:, -1:, :]
14.    predicted_id = tf.cast(tf.argmax(predictions, axis=-1), tf.int32)
15.    return predicted_id
```

Summarize:

```
1. def summarize(model, input_document):
2.     input_document = tokenizer.texts_to_sequences([input_document])
3.     input_document = tf.keras.preprocessing.sequence.pad_sequences(input_document,
4.        maxlen=encoder_maxlen, padding='post', truncating='post')
5.     encoder_input = tf.expand_dims(input_document[0], 0)
6.     output = tf.expand_dims([tokenizer.word_index["[SOS]"]], 0)
7.     for i in range(decoder_maxlen):
8.         predicted_id = next_word(model, encoder_input, output)
9.         output = tf.concat([output, predicted_id], axis=-1)
10.        if predicted_id == tokenizer.word_index["[EOS]"]:
11.            break
12.    return tokenizer.sequences_to_texts(output.numpy())[0]
```

Train the model

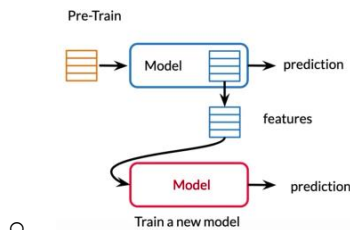
```
1. # Take an example from the test set, to monitor it during training
2. test_example = 0
3. true_summary = summary_test[test_example]
4. true_document = document_test[test_example]
5. # Define the number of epochs
6. epochs = 20
7. # Training loop
8. for epoch in range(epochs):
9.     start = time.time()
10.    train_loss.reset_states()
11.    number_of_batches=len(list(enumerate(dataset)))
12.    for (batch, (inp, tar)) in enumerate(dataset):
13.        print(f'Epoch {epoch+1}, Batch {batch+1}/{number_of_batches}', end='\n')
14.        train_step(transformer, inp, tar)
15.    print(f'Epoch {epoch+1}, Loss {train_loss.result():.4f}')
16.    losses.append(train_loss.result())
17.    print(f'Time taken for one epoch: {time.time() - start} sec')
18.    print('Example summarization on the test set:')
19.    print(' True summarization:')
20.    print(f' {true_summary}')
21.    print(' Predicted summarization:')
22.    print(f' {summarize(transformer, true_document)}\n')
```


Question answering:

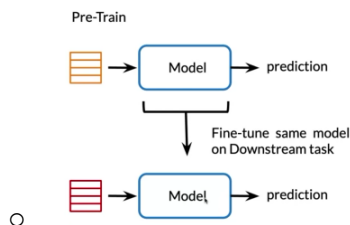
- Question-based:
 - o Given a context and a question, model tells you where the answer is inside the context.
- Closed book:
 - o Given a question, model gives its own answer.

Transfer Learning in NLP

- Feature-based: (word-embedding)



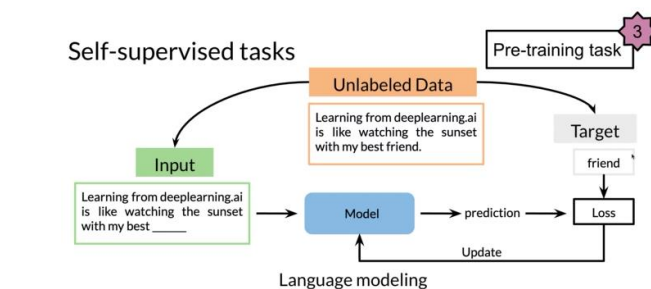
- Fine-tuning:



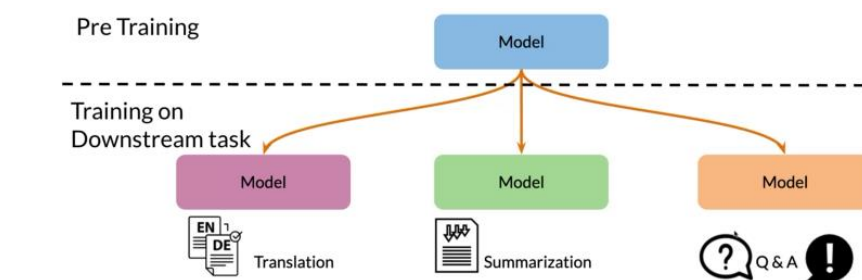
Transfer learning with unlabeled data:

- Pre-train data: usually there are more unlabeled data than labeled data.

Self-supervised tasks:

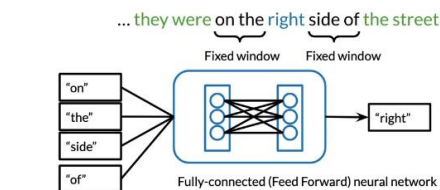


Fine-tuning a model for each downstream task:



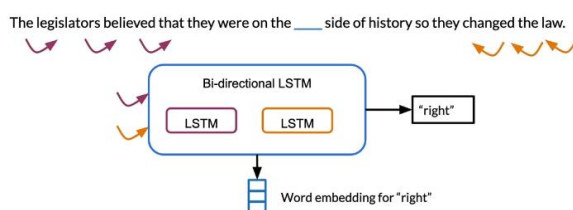
Different models:

Continuous Bag of Words Model:



- This early model predicted a word based on the context of surrounding words within a fixed window size. However, it was limited by the inability to consider the full sentence context beyond the fixed window.

ELMo (Embeddings from Language Models):



- ELMo addressed the limitations of the continuous bag of words by using bi-directional LSTMs (Long Short-Term Memory networks) to incorporate context from both sides of a word, thus creating richer word embeddings.

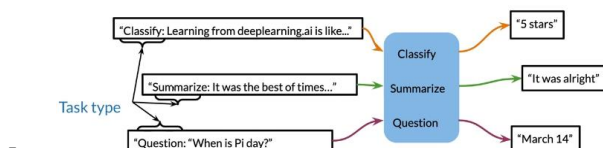
OpenAI GPT (Generative Pre-trained Transformer):

- GPT marked a departure from RNNs, using a stack of decoders from the transformer architecture. It was adept at generating text in a uni-directional context but was limited by its inability to incorporate future context in predictions.

BERT (Bidirectional Encoder Representations from Transformers):

- BERT introduced the use of transformer encoders to provide deep bi-directional context, allowing the model to consider both past and future context in text. It also employed novel pre-training tasks such as multi-mask language modeling and next sentence prediction.

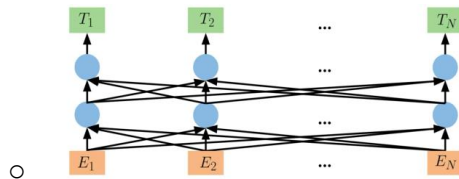
T5 (Text-to-Text Transfer Transformer):



- T5 expanded upon previous models by using both encoder and decoder stacks from the original transformer architecture. It showcased improved performance and the ability to handle multiple tasks through a multi-task training strategy. The model could be prompted to perform specific tasks, like classification or summarization, by appending task-specific prompts (e.g., "classify:" or "summarize:") to the input.

Bidirectional Encoder Representations from Transformers (BERT)

- Makes use of transfer learning/pre-training:



- Architecture:

- o A multi layer bidirectional transformer
- o Positional embeddings
- o BERT_base:
 - 12 layers (12 transformer blocks)
 - 12 attentions heads
 - 110 million parameters

- Pre-training: (from unlabeled text)

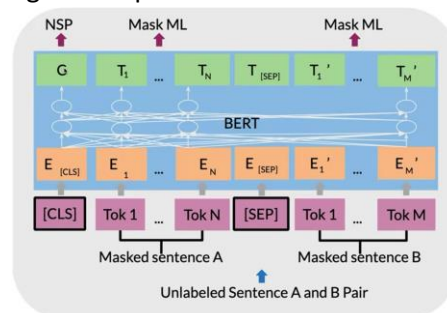
- o Masked Language modeling (MLM)
 - Choose 15% of the tokens at random:
 - Replaced with a mask token 80% of the time.
 - Replaced with a random token 10% of the time.
 - Left unchanged 10% of the time.
 - There could be multiple masked spans in a sentence.
 - Next sentence prediction is also used when pre-training.

- Objective:

- o Formalizing the input:

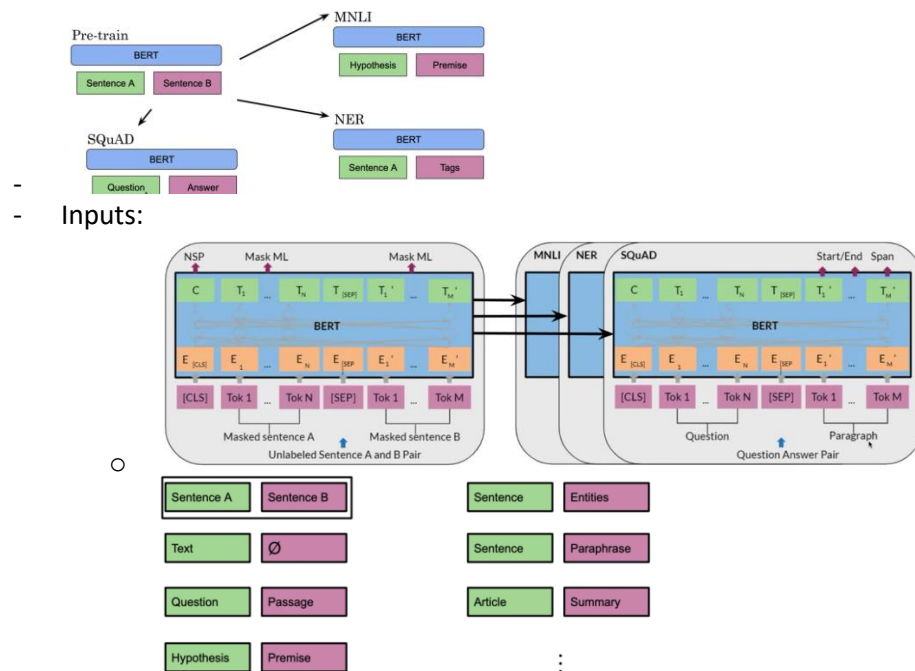
Input	[CLS] my dog is cute [SEP] he likes play ##ing [SEP]										
Token	E [CLS]	E my	E dog	E is	E cute	E [SEP]	E he	E likes	E play	E ##ing	E [SEP]
Embeddings	*	*	*	*	*	*	*	*	*	*	*
Segment	E _A	E _A	E _A	E _A	E _A	E _A	E _B	E _B	E _B	E _B	E _B
Embeddings	*	*	*	*	*	*	*	*	*	*	*
Position	E ₀	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	E ₇	E ₈	E ₉	E ₁₀
Embeddings	*	*	*	*	*	*	*	*	*	*	*

- Segment embeddings Indicate whether it is sentence A or sentence B.
- o Visualizing the output:



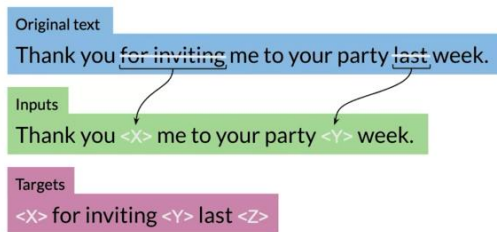
- [CLS]: a special classification symbol added in front of every input
- [SEP]: a special separator token
- o Objective 1: Multi-Mask LM
 - Loss: Cross Entropy Loss
- o Objective 2: Next Sentence Prediction
 - Loss: Binary Loss

Fine tuning BERT:

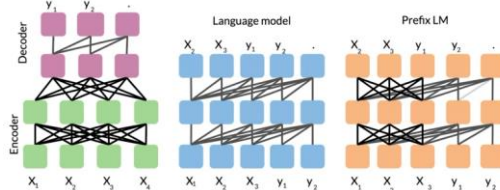


Transformer: T5

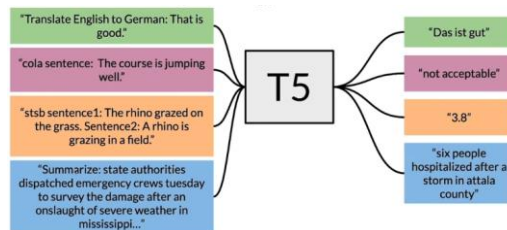
- Data:



- Architecture:



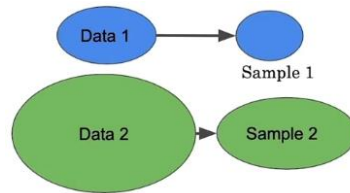
- Multi-Task Training Strategy:



Fine-tuning method	GLUE	CNN3M	SQuAD	SGLUE	EnDe	EnFr	EnRo
* All parameters	83.28	19.24	80.88	71.36	26.98	39.82	27.65
Adapter layers, $d = 32$	80.52	15.08	79.32	60.40	13.84	17.88	15.54
Adapter layers, $d = 128$	81.51	16.62	79.47	63.03	19.83	27.50	22.63
Adapter layers, $d = 512$	81.54	17.78	79.18	64.30	23.45	33.98	25.81
Adapter layers, $d = 2048$	81.51	16.62	79.47	63.03	19.83	27.50	22.63
Gradual unfreezing	82.50	18.95	79.17	70.79	26.71	39.02	26.93

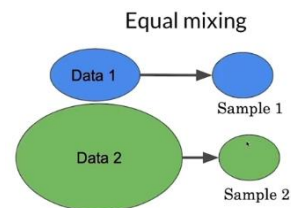
- Data Training strategy:

- Examples-proportional mixing:



- E.g. 10% for each data

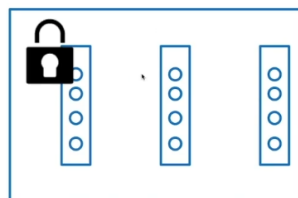
- Equal mixing:



- Temperature-scaled mixing:

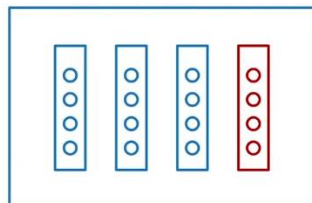
- Play with the parameters to get something in between.

- Gradual unfreezing vs. Adapter layers



Gradual unfreezing

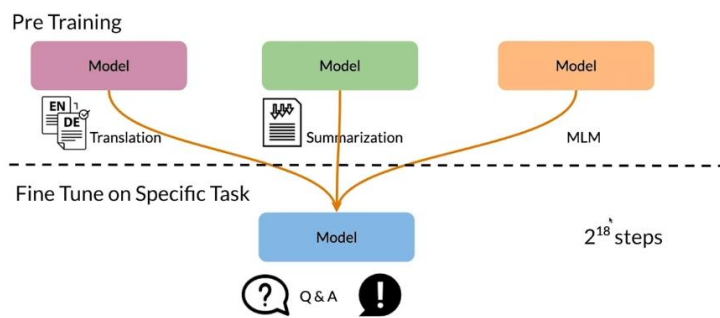
- unfreeze one layer at a time.
- E.g. Unfreeze → fine tune → so on



Adapter layers

- Add a neural network to each feed-forward in each block of the transformer, these new feed-forward networks, they're designed so that the output dimension matches the input. This allows them to be inserted without having any structural change. When fine-tuning only these new adapter layers and the layer normalization parameters are being updated.

Fine-tuning T5:



General Language Understanding Evaluation (GLUE) Benchmark:

- A collections of resources used to train, evaluate, analyze natural language understanding systems.
- Datasets with different genres, and of different sizes and difficulties.
- Leaderboard
- Tasks evaluated on
 - o Sentence grammatical or not
 - o Sentiment
 - o Paraphrase
 - o Similarity
 - o Questions duplicates
 - o Answerable
 - o Contradiction
 - o Entailment
 - o Winograd (co-ref)

SentencePiece and BPE Lab:

SentencePiece: unicode characters are grouped together using either a unigram language model or byte-pair encoding (BPE).

```
1. from unicodedata import normalize
2. norm_eaccent = normalize('NFKC', '\u00E9')
3. norm_eaccent = normalize('NFKC', '\u0065\u0301')
4. print(f'{norm_eaccent} = {norm_eaccent} : {norm_eaccent == norm_eaccent}') #True

1. def get_hex_encoding(s):
2.     return ' '.join(hex(ord(c)) for c in s)
3. def print_string_and_encoding(s):
4.     print(f'{s} : {get_hex_encoding(s)}')
```

Lossless Tokenization (white space is preserved):

```
1. s = 'Tokenization is hard.'
2. s_ = s.replace(' ', '\u2581')
3. s_n = normalize('NFKC', 'Tokenization is hard.')
```

```
1. print(get_hex_encoding(s))
2. print(get_hex_encoding(s_))
3. print(get_hex_encoding(s_n))
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72
0x64 0x2e
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x2581 0x69 0x73 0x2581 0x68 0x61
0x72 0x64 0x2e
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72
0x64 0x2e
```

```
1. s = 'Tokenization is hard.'
2. sn = normalize('NFKC', 'Tokenization is hard.')
3. sn_ = s.replace(' ', '\u2581')
```

```
1. print(get_hex_encoding(s))
2. print(get_hex_encoding(sn))
3. print(get_hex_encoding(sn_))
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72
0x64 0x2e
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72
0x64 0x2e
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x2581 0x69 0x73 0x2581 0x68 0x61
0x72 0x64 0x2e
```

Data preparation:

```
1. import ast
2. def convert_json_examples_to_text(filepath):
3.     example_jsons = list(map(ast.literal_eval, open(filepath)))
4.     texts = [example_json['text'].decode('utf-8') for example_json in example_jsons]
5.     text = '\n\n'.join(texts) # Separate different articles by two newlines
6.     text = normalize('NFKC', text) # Normalize the text
7.     with open('example.txt', 'w') as fw:
8.         fw.write(text)
9.     return text
10.
11. text = convert_json_examples_to_text('./data/data.txt')
```

Vocabulary:

```
1. from collections import Counter
2.
3. vocab = Counter(['\u2581' + word for word in text.split()])
4. vocab = {' '.join([l for l in word]): freq for word, freq in vocab.items()}
```

```
1. def show_vocab(vocab, end='\n', limit=20):
2.     shown = 0
3.     for word, freq in vocab.items():
4.         print(f'{word}: {freq}', end=end)
5.         shown += 1
6.         if shown > limit:
7.             break
8.
9. show_vocab(vocab)
```

```
1. _ B e g i n n e r s : 1
2. _ B B Q : 3
3. _ C l a s s : 2
4. _ T a k i n g : 1
5. ...
```

BPE Algorithm:

```
1. import re, collections
2.
3. def get_stats(vocab):
4.     pairs = collections.defaultdict(int)
5.     for word, freq in vocab.items():
6.         symbols = word.split()
7.         for i in range(len(symbols) - 1):
8.             pairs[symbols[i], symbols[i+1]] += freq
9.     return pairs
10.
11. def merge_vocab(pair, v_in):
12.     v_out = {}
13.     bigram = re.escape(' '.join(pair))
14.     p = re.compile(r'(?!\S)' + bigram + r'(!\S)')
15.     for word in v_in:
16.         w_out = p.sub(' '.join(pair), word)
17.         v_out[w_out] = v_in[word]
18.     return v_out
19.
20. def get_sentence_piece_vocab(vocab, frac_merges=0.60):
21.     sp_vocab = vocab.copy()
22.     num_merges = int(len(sp_vocab)*frac_merges)
23.
24.     for i in range(num_merges):
25.         pairs = get_stats(sp_vocab)
26.         best = max(pairs, key=pairs.get)
27.         sp_vocab = merge_vocab(best, sp_vocab)
28.
29.     return sp_vocab
```


Question Answering:

Set up:

```
1. import os
2. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
3. import traceback
4. import time
5. import json
6. from termcolor import colored
7. import string
8. import textwrap
9. import itertools
10. import numpy as np
11. import tensorflow_text as tf_text
12. import tensorflow as tf
13. import transformer_utils
14. import utils
15. wrapper = textwrap.TextWrapper(width=70)
16. np.random.seed(42)
```

C4 Dataset:

```
1. with open('data/c4-en-10k.jsonl', 'r') as file:
2.     example_jsons = [json.loads(line.strip()) for line in file]
```

Process C4:

```
1. natural_language_texts = [example_json['text'] for example_json in example_jsons]
```

Decode to natural language with SentencePieceTokenizer:

```
1. with open("./models/sentencepiece.model", "rb") as f:
2.     pre_trained_tokenizer = f.read()
3. tokenizer = tf_text.SentencePieceTokenizer(pre_trained_tokenizer, out_type=tf.int32)
4. eos = tokenizer.string_to_id("</s>").numpy()[# EOS: 1]
5. tokenized_text = [(list(tokenizer.tokenize(word).numpy()), word) for word in
    natural_language_texts[2].split())]
```

```
1. Tokenized_text: element[1] --> element[0]
2.   Foil          -->      [4452, 173]
3.   plaid         -->      [30772]
4.   lycra         -->      [3, 120, 2935]
5.   and           -->      [11]
6.   ...
```

```
1. def get_sentinels(tokenizer, display=False): #for creating input and target pairs
2.     sentinels = {}
3.     vocab_size = tokenizer.vocab_size(name=None)
4.     for i, char in enumerate(reversed(string.ascii_letters), 1):
5.         decoded_text = tokenizer.detokenize([vocab_size - i]).numpy().decode("utf-8")
6.         # Sentinels, ex: <Z> - <a>
7.         sentinels[decoded_text] = f'<{char}>'
8.         if display:
9.             print(f'The sentinel is <{char}> and the decoded token is:', decoded_text)
10.    return sentinels
11. def pretty_decode(encoded_str_list, sentinels, tokenizer):
12.     # If already a string, just do the replacements.
13.     if tf.is_tensor(encoded_str_list) and encoded_str_list.dtype == tf.string:
14.         for token, char in sentinels.items():
15.             encoded_str_list = tf.strings.regex_replace(encoded_str_list, token, char)
16.         return encoded_str_list
17.     # We need to decode and then prettyfy it.
18.     return pretty_decode(tokenizer.detokenize(encoded_str_list), sentinels, tokenizer)
19.
20. pretty_decode(tf.constant("I want to dress up as an Intellectual this halloween."), sentinels, tokenizer)
21. # <tf.Tensor: shape=(), dtype=string, numpy=b'I want to dress up as an <b> this <b>.'>
```

```
1. sentinels = get_sentinels(tokenizer, display=True)
2. #The sentinel is <b> and the decoded token is: halloween...
```

Tokenize and mask

- Tokenizes and masks input words based on a given probability controlled by noise.
- Generate two lists of tokenized sequences following the algorithm outlined below:

Function:

```
1. def tokenize_and_mask(text,
2.                       noise=0.15,
3.                       randomizer=np.random.uniform,
4.                       tokenizer=None):
5.     cur_sentinel_num = 0
6.     inps, targs = [], []
7.     vocab_size = int(tokenizer.vocab_size())
8.     eos = tokenizer.string_to_id("</s>").numpy()
9.     prev_no_mask = True
10.    for token in tokenizer.tokenize(text).numpy():
11.        rnd_val = randomizer()
12.        if rnd_val < noise:
13.            if prev_no_mask:
14.                cur_sentinel_num += 1
15.                end_id = vocab_size - cur_sentinel_num
16.                targs.append(end_id)
17.                inps.append(end_id)
18.            targs.append(token)
19.            prev_no_mask = False
20.        else:
21.            inps.append(token)
22.            prev_no_mask = True
23.    targs.append(eos)
24.    return inps, targs
```

Create the pairs:

```
1. # Apply tokenize_and_mask
2. inputs_targets_pairs = [tokenize_and_mask(text.encode('utf-8',
    errors='ignore').decode('utf-8'), tokenizer=tokenizer)
    for text in natural_language_texts[0:2000]]

1. def display_input_target_pairs(inputs_targets_pairs, sentinels,
    wrapper=textwrap.TextWrapper(width=70), tokenizer=tokenizer):
2.     for i, inp_tgt_pair in enumerate(inputs_targets_pairs, 1):
3.         inps, tgts = inp_tgt_pair
4.         inps = str(pretty_decode(inps, sentinels, tokenizer).numpy(), encoding='utf-8')
5.         tgts = str(pretty_decode(tgts, sentinels, tokenizer).numpy(), encoding='utf-8')
6.         print(f'[{i}]\n\n')
7.         f'inputs:\n{wrapper.fill(text=inps)}\n\n'
8.         f'targets:\n{wrapper.fill(text=tgts)}\n\n\n')
9.
10. # Print the first 5 samples
11. display_input_target_pairs(inputs_targets_pairs[0:5], sentinels, wrapper, tokenizer)

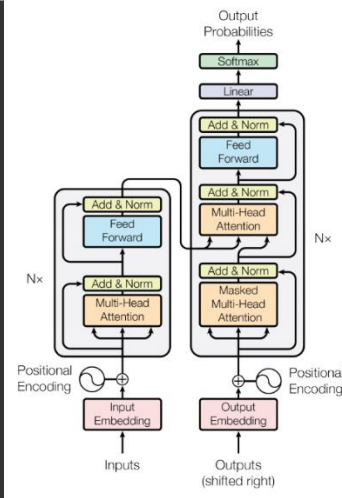
1. [1]
2. inputs:
3. Beginners BBQ Class Taking <Z> in Missoul <Y>! Do you want to ...
4. Targets: <Z>: Place <Y>a ...
```

Pretrain a T5 model using C4

```

1. num_layers = 2
2. embedding_dim = 128
3. fully_connected_dim = 128
4. num_heads = 2
5. positional_encoding_length = 256
6.
7. encoder_vocab_size = int(tokenizer.vocab_size())
8. decoder_vocab_size = encoder_vocab_size
9.
10. transformer = transformer_utils.Transformer(
11.     num_layers,
12.     embedding_dim,
13.     num_heads,
14.     fully_connected_dim,
15.     encoder_vocab_size,
16.     decoder_vocab_size,
17.     positional_encoding_length,
18.     positional_encoding_length,
19. )
20.
21. learning_rate = transformer_utils.CustomSchedule(embedding_dim)
22. optimizer = tf.keras.optimizers.Adam(0.0001, beta_1=0.9, beta_2=0.98, epsilon=1e-9)
23.
24. loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none')
25. train_loss = tf.keras.metrics.Mean(name='train_loss')
26.
27. losses = []

```



Ensure that all inputs and targets have the same length by truncating the longer sequences and padding the shorter ones with 0.

```

1. encoder_maxlen = 150
2. decoder_maxlen = 50
3. inputs = tf.keras.preprocessing.sequence.pad_sequences([x[0] for x in inputs_targets_pairs],
maxlen=encoder_maxlen, padding='post', truncating='post')
4. targets = tf.keras.preprocessing.sequence.pad_sequences([x[1] for x in inputs_targets_pairs],
maxlen=decoder_maxlen, padding='post', truncating='post')
5. inputs = tf.cast(inputs, dtype=tf.int32)
6. targets = tf.cast(targets, dtype=tf.int32)
7. BUFFER_SIZE = 10000
8. BATCH_SIZE = 64
9. dataset = tf.data.Dataset.from_tensor_slices((inputs, targets)).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

```

Train the model for 10 epochs:

```

1. epochs = 10
2.
3. for epoch in range(epochs):
4.     start = time.time()
5.     train_loss.reset_states()
6.     number_of_batches=len(list(enumerate(dataset)))
7.     for (batch, (inp, tar)) in enumerate(dataset):
8.         print(f'Epoch {epoch+1}, Batch {batch+1}/{number_of_batches}', end='\r')
9.         transformer_utils.train_step(inp, tar, transformer, loss_object, optimizer, train_loss)
10.    print(f'Epoch {epoch+1}, Loss {train_loss.result():.4f}')
11.    losses.append(train_loss.result())
12.    print(f'Time taken for one epoch: {time.time() - start} sec')
13. # Save the pretrained model
14. # transformer.save_weights('./model_c4_temp')

1. transformer.load_weights('./pretrained_models/model_c4')

```

Fine tune the T5 model for Question Answering:

```
1. with open('data/train-v2.0.json', 'r') as f:
2.     example_jsons = json.load(f)
3. example_jsons = example_jsons['data']
```

Iterates over all the articles, paragraphs and questions in the SQuAD dataset(Json). Extract pairs of input and targets for the QA model using the provided code template.

```
1. # GRADED FUNCTION: parse_squad
2. def parse_squad(dataset):
3.     inputs, targets = [], []
4.     for article in dataset:
5.         for paragraph in article['paragraphs']:
6.             context = paragraph['context']
7.             for qa in paragraph['qas']:
8.                 if len(qa['answers']) > 0 and not(qa['is_impossible']):
9.                     question_context = 'question: ' + qa['question'] + ' context: ' + context
10.                    answer = 'answer: ' + qa['answers'][0]['text']
11.                    inputs.append(question_context)
12.                    targets.append(answer)
13.     return inputs, targets

1. inputs, targets = parse_squad(example_jsons)
```

Divide data:

```
1. # 50K pairs for training
2. inputs_train = inputs[0:40000]
3. targets_train = targets[0:40000]
4. # 5K pairs for testing
5. inputs_test = inputs[40000:45000]
6. targets_test = targets[40000:45000]
```

Truncating/padding:

```
1. # Limit the size of the input and output data so this can run in this environment
2. encoder_maxlen = 150
3. decoder_maxlen = 50
4. inputs_str = [tokenizer.tokenize(s) for s in inputs_train]
5. targets_str = [tf.concat([tokenizer.tokenize(s), [1]], 0) for s in targets_train]
6. inputs = tf.keras.preprocessing.sequence.pad_sequences(inputs_str, maxlen=encoder_maxlen,
padding='post', truncating='post')
7. targets = tf.keras.preprocessing.sequence.pad_sequences(targets_str,
maxlen=decoder_maxlen, padding='post', truncating='post')
8. inputs = tf.cast(inputs, dtype=tf.int32)
9. targets = tf.cast(targets, dtype=tf.int32)
10. # Create the final training dataset.
11. BUFFER_SIZE = 10000
12. BATCH_SIZE = 64
13. dataset = tf.data.Dataset.from_tensor_slices((inputs,
targets)).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Fine-Tune the T5 model:

```
1. epochs = 2
2. losses = []
3. for epoch in range(epochs):
4.     start = time.time()
5.     train_loss.reset_states()
6.     number_of_batches=len(list(enumerate(dataset)))
7.     for (batch, (inp, tar)) in enumerate(dataset):
8.         print(f'Epoch {epoch+1}, Batch {batch+1}/{number_of_batches}', end='\n')
9.         transformer_utils.train_step(inp, tar, transformer, loss_object, optimizer, train_loss)
10.    losses.append(train_loss.result())
11.    print(f'Time taken for one epoch: {time.time() - start} sec')
12.    #if epoch % 15 == 0:
13.        #transformer.save_weights('./pretrained_models/model_qa_temp')
14. #transformer.save_weights('./pretrained_models/model_qa_temp')
```

Loading the weights:

```
1. transformer.load_weights('./pretrained_models/model_qa3')
```

Implement the Question Answering model:

```
1. example_question = "question: What color is the sky? context: Sky is blue"
2. tokenized_padded_question = tf.constant([[822, 10, 363, 945, 19, 8, 5796, 58, 2625, 10,
5643, 19, 1692, 0, 0]])
3. tokenized_answer = tf.constant([[1525, 10]])
4. next_word = transformer_utils.next_word(tokenized_padded_question, tokenized_answer,
transformer)
5. print(f"Predicted next word is: '{tokenizer.detokenize(next_word).numpy()[0].decode('utf-
8')}'")
6. answer_so_far = tf.concat([tokenized_answer, next_word], axis=-1)
7. print(f"Answer so far: '{tokenizer.detokenize(answer_so_far).numpy()[0].decode('utf-8')}'")
```

Answer question:

```
1. def answer_question(question, model, tokenizer, encoder_maxlen=150, decoder_maxlen=50):
2.     tokenized_question = tokenizer.tokenize(question)
3.     tokenized_question = tf.expand_dims(tokenized_question, 0)
4.     padded_question = tf.keras.preprocessing.sequence.pad_sequences(tokenized_question,
5.                                                                     maxlen=encoder_maxlen,
6.                                                                     padding='post',
7.                                                                     truncating='post')
8.     tokenized_answer = tokenizer.tokenize("answer: ")
9.     tokenized_answer = tf.expand_dims(tokenized_answer, 0)
10.    eos = tokenizer.string_to_id("</s>")
11.    for i in range(decoder_maxlen):
12.        next_word = transformer_utils.next_word(padded_question, tokenized_answer, model)
13.        tokenized_answer = tf.concat([tokenized_answer, next_word], axis=-1)
14.        if next_word.numpy()[0][0] == eos:
15.            break
16.    return tokenized_answer
```