CSCI E-89B Introduction to Natural Language Processing

Harvard Extension School

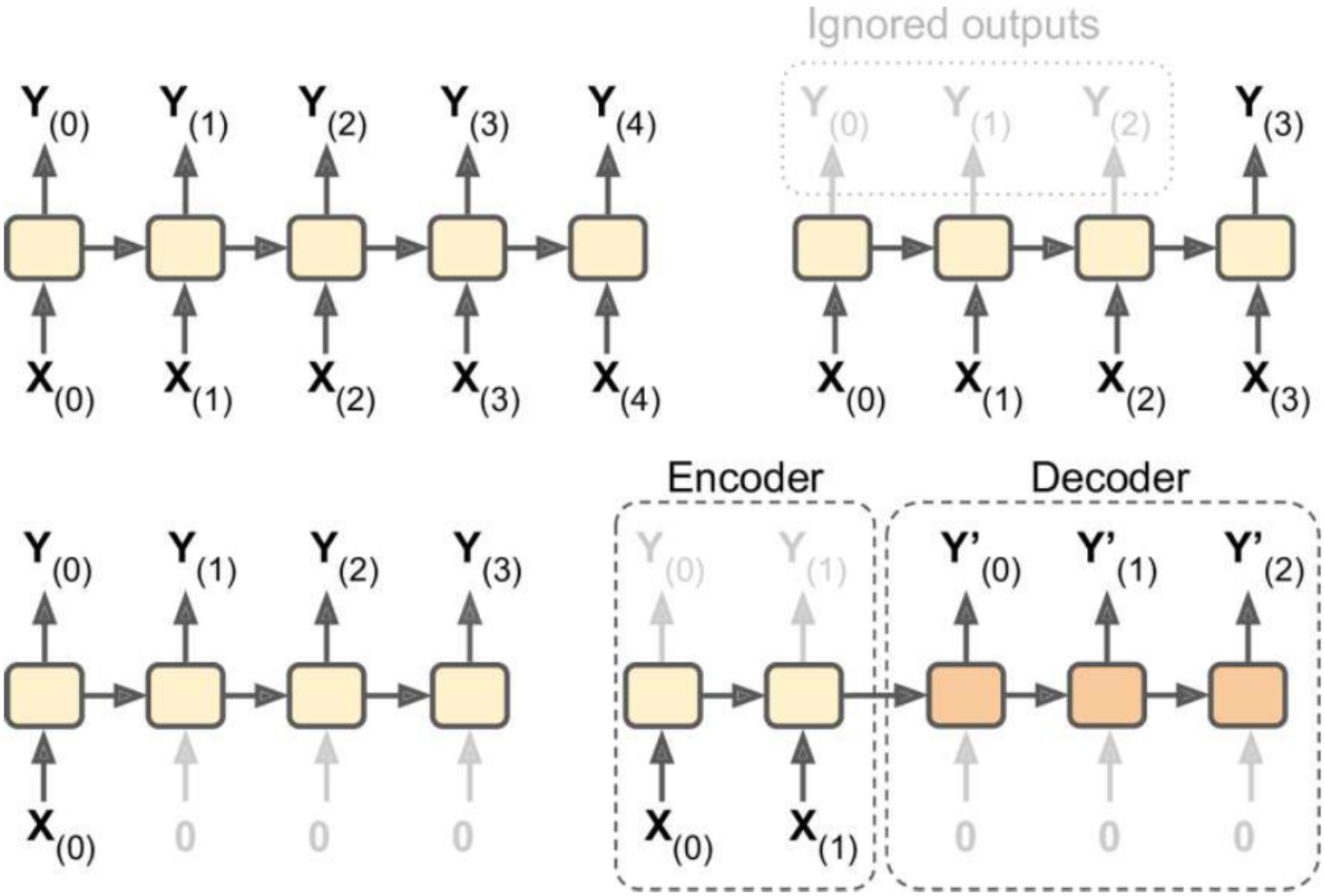
Dmitry Kurochkin

Fall 2025
Lecture 12

Contents

Contents

Challenges Faced by RNNs



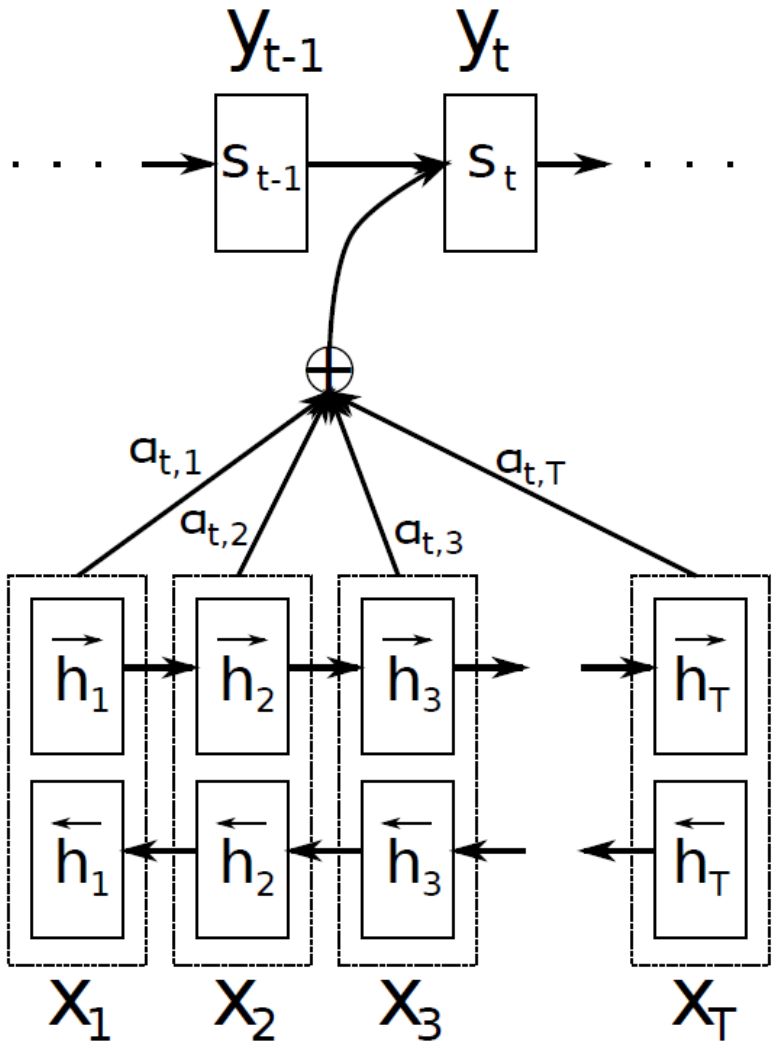
Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Challenges Faced by RNNs (Continued)

- **Long-Range Dependency Challenges**:
  - Difficulty retaining relevant context across long sequences, leading to potential information loss and reduced model effectiveness.
  - Vanishing gradient problem impacts the model's ability to learn and leverage dependencies from distant sequence elements.
- **Sequential Processing Limitations**:
  - The inherent sequential nature of processing restricts the ability to parallelize computations, limiting efficiency.
  - Inefficiencies arise when processing long sequences, leading to increased computational overhead and slower performance, especially as sequence length grows.
- **Fixed-Size Representation Bottleneck**:
  - Represents entire input sequence as a single fixed-size vector in encoder-decoder frameworks.
  - Can lead to loss of detailed information needed for accurate sequence generation.

Contents

Enhancing Recurrent Networks with Attention



Source: *Neural machine translation by jointly learning to align and translate* by Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio

Enhancing Recurrent Networks with Attention

- **Focusing on Relevant Inputs**:
  - Annotations h_j for each word x_j are created by concatenating the forward hidden state $\overrightarrow{h}_j$ and the backward hidden state $\overleftarrow{h}_j$:

$$h_j = [\overrightarrow{h}_j; \overleftarrow{h}_j]$$

  - For a target output y_i, compute context vector c_i as a weighted sum of annotations h_j:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

  - The attention weights α_{ij} are determined by:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

  - Here, e_{ij} is the alignment score calculated as:

$$e_{ij} = a(s_{i-1}, h_j)$$

  - s_{i-1} is the hidden state of the decoder from the previous time step, encapsulating context from previous outputs.

Enhancing Recurrent Networks with Attention (Continued)

- **Addressing Long-Range Dependencies**:
  - The alignment function a can be defined as:

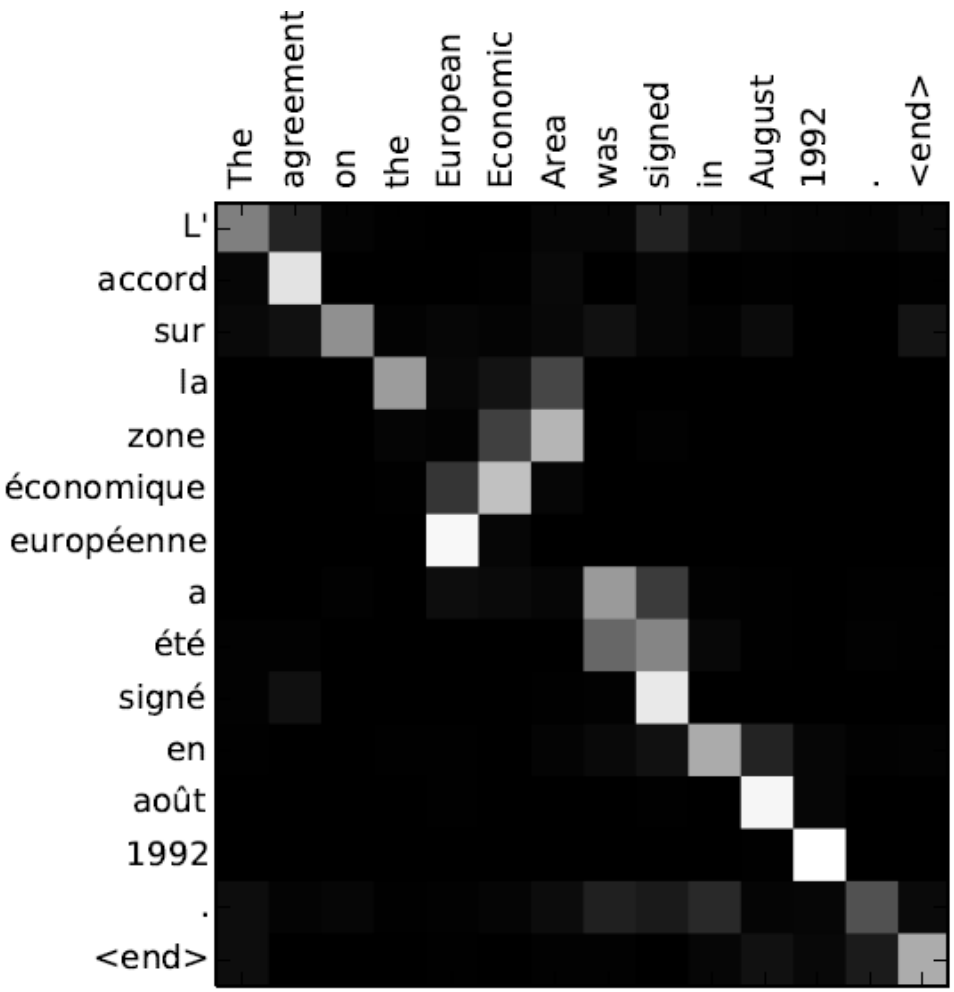
$$a(s_{i-1}, h_j) = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

  - This alignment overcomes the vanishing gradient problem by effectively linking distant inputs and outputs.
- **Dynamic Contextual Information**:
  - Using c_i, the model outputs target word y_i conditioned on previous outputs and c_i:

$$p(y_i \mid y_1, \ldots, y_{i-1}, c_i) = g(y_{i-1}, s_i, c_i)$$

  - s_i is updated to include the new contextual information from c_i, facilitating coherent and informed sequence generation.
  - This process ensures more comprehensive context capture by providing variable-length input alignments.

Sample Alignment Visualization



- This visualization shows the alignment between words in the source sentence (English) and the generated translation (French).
- Each pixel represents the weight α_{ij}, indicating the importance of the j-th source word to the i-th target word in grayscale (0: black, 1: white).

Source: *Neural machine translation by jointly learning to align and translate* by Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio

Contents

Implementing Attention in Keras

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Input, LSTM, Dense
from tensorflow.keras.models import Model

# Custom Attention Layer
class AttentionLayer(Layer):
    def __init__(self):
        super(AttentionLayer, self).__init__()

    def build(self, input_shape):
        # Initialize weights and biases for attention mechanism
        self.W = self.add_weight(shape=(input_shape[-1], input_shape[-1]),
                                 initializer='random_normal', trainable=True)
        self.b = self.add_weight(shape=(input_shape[-1],),
                                 initializer='zeros', trainable=True)
        self.u = self.add_weight(shape=(input_shape[-1], 1),
                                 initializer='random_normal', trainable=True)

    def call(self, inputs):
        # Calculate attention scores
        v = tf.tanh(tf.tensordot(inputs, self.W, axes=1) + self.b)
        vu = tf.tensordot(v, self.u, axes=1, name='vu')
        # Compute attention weights
        alphas = tf.nn.softmax(vu, axis=1)
        # Calculate context vector as weighted sum of inputs
        output = tf.reduce_sum(inputs * tf.expand_dims(alphas, -1), axis=1)
        return output
```

Implementing Attention in Keras (Continued)

```
# Specify input shape
input_shape = (30, 64)

# Create the model
inputs = Input(shape=input_shape)
lstm_out = LSTM(128, return_sequences=True)(inputs)
attention = AttentionLayer()(lstm_out)
outputs = Dense(10, activation='softmax')(attention)

# Define and compile the model
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Print the summary of the model
model.summary()
```

Building the Model with Attention (Continued)

```
# Print the summary of the model
model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 30, 64)	0
lstm (LSTM)	(None, 30, 128)	98,816
attention_layer (AttentionLayer)	(None, 30, 128)	16,640
dense (Dense)	(None, 30, 10)	1,290

Total params: 116,746 (456.04 KB)

Trainable params: 116,746 (456.04 KB)

Non-trainable params: 0 (0.00 B)

Contents

Advantages of Attention in RNNs, LSTMs, and GRUs

- **Improved Context Utilization**:
  - Dynamically focuses on relevant sections of the input sequence, enhancing understanding and contextual accuracy.
  - Helps retain important information across long sequences, reducing potential information loss.
- **Enhanced Model Robustness**:
  - Provides adaptability across various tasks by effectively leveraging comprehensive contextual information.
  - Increases model performance stability even with varying sequence lengths.
- **Mitigating Fixed-Size Constraints**:
  - Replaces the need for a single static context vector with dynamic context vectors, facilitating more detailed sequence generation.
  - Allows for a richer representation of input data, improving output accuracy.

Advantages of Attention in RNNs, LSTMs, and GRUs (Continued)

- **Improved Interpretability**:
  - Offers insights into the model's decision-making process by highlighting which parts of the input are most influential for each output.
  - Provides a visual map of attention weights that facilitates understanding and debugging.
- **Scalability and Flexibility**:
  - Enhances scalability for handling increasingly complex and lengthy sequences with consistent performance.
  - Allows models to adapt flexibly to various input structures and requirements.

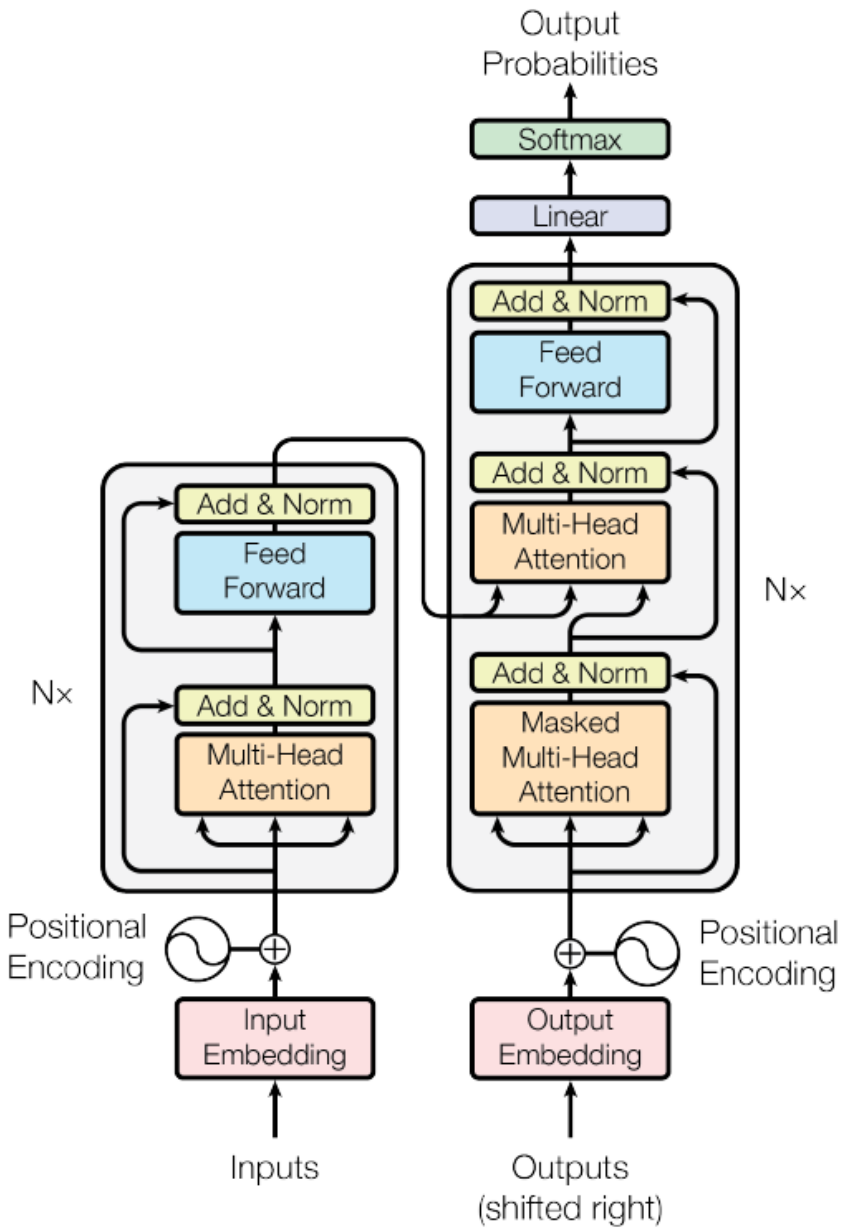
Contents

Limitations Despite Attention in RNNs, LSTMs, and GRUs

- **Sequential Processing Constraint**:
  - Despite improved context handling, RNNs process sequences step-by-step, limiting potential speed and efficiency gains.
  - Sequential nature restricts parallel computation, resulting in longer training times for long sequences.
- **Persistent Vanishing Gradient Problem**:
  - While mitigated, gradient issues still pose challenges, particularly in deep networks or very long sequences.
  - Difficulties in maintaining gradient flow can impact the learning of long-range dependencies.
- **Transfer Learning Limitations**:
  - Models require extensive retraining and fine-tuning when adapting to new tasks or domains.
  - Architecture is less aligned with modern transfer learning paradigms centered on large-scale pre-training.

Contents

Transformer Architecture



Source: *Attention Is All You Need* by Vaswani et al., 2017

Components of the Transformer: Attention Mechanisms

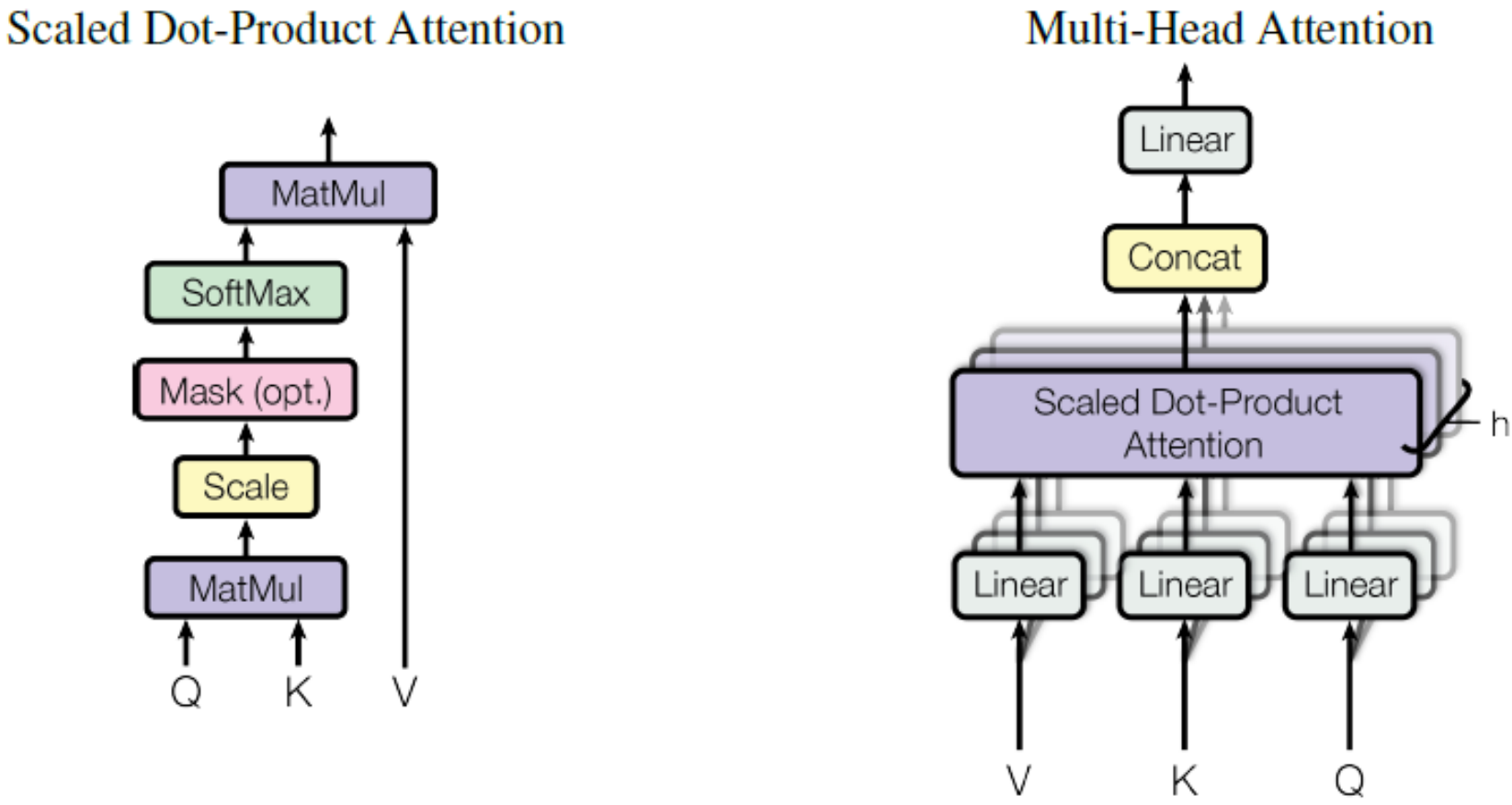


Figure: (Left) Scaled Dot-Product Attention. (Right) Multi-Head Attention.

Source: *Attention Is All You Need* by Vaswani et al., 2017

Scaled Dot-Product Attention

The attention is computed as:

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

where

- Queries (Q): Generated by applying a learned weight matrix W^Q to the input embedding.
- Keys (K): Generated by applying a learned weight matrix W^K to the input embedding.
- Values (V): Generated by applying a learned weight matrix W^V to the input embedding.
- The scaling factor $\sqrt{d_k}$ is used to stabilize gradients during training, preventing large dot products and ensuring efficient learning.
- The softmax function computes a probability distribution over the keys, determining how much focus to assign to each value, facilitating a contextualized output.

Multi-Head Attention

Multi-Head Attention

- Instead of a single attention function, multiple attention heads operate in parallel:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- The outputs of all heads are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

- Enables the model to focus on different parts of the input sequence simultaneously.

Contents

Transformer Block in Python

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, MultiHeadAttention, LayerNormalization,
    Dropout, Add
from tensorflow.keras.models import Model

# Define a simple Transformer block
def transformer_block(input_shape, num_heads, ff_dim, dropout_rate=0.1):
    inputs = Input(shape=input_shape)

    # Multi-Head Attention
    attention_output = MultiHeadAttention(num_heads=num_heads,
                          key_dim=input_shape[-1])(inputs, inputs)
    attention_output = Dropout(dropout_rate)(attention_output)
    attention_output = Add()([inputs, attention_output])  # Residual connection
    attention_output = LayerNormalization(epsilon=1e-6)(attention_output)

    # Feedforward Network
    ffn_output = Dense(ff_dim, activation='relu')(attention_output)
    ffn_output = Dense(input_shape[-1])(ffn_output)
    ffn_output = Dropout(dropout_rate)(ffn_output)
    ffn_output = Add()([attention_output, ffn_output])  # Residual connection
    ffn_output = LayerNormalization(epsilon=1e-6)(ffn_output)

    return Model(inputs, ffn_output, name='TransformerBlock')
```

Transformer Block in Python (Continued)

```
# Model definition
seq_length = 10
embed_dim = 512  # Embedding dimension
num_heads = 8
ff_dim = 2048  # Feed-forward dimension

inputs = Input(shape=(seq_length, embed_dim))
x = transformer_block(input_shape=(seq_length, embed_dim), num_heads=num_heads, ff_dim=ff_dim
x = Dense(10, activation='softmax')(x)  # Assuming 10 classes for classification

model = Model(inputs=inputs, outputs=x)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Display the model architecture
model.summary()
```

Transformer Block in Python (Continued)

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 10, 512)	0
TransformerBlock (Functional)	(None, 10, 512)	10,503,168
dense_3 (Dense)	(None, 10, 10)	5,130

Total params: 10,508,298 (40.09 MB)

Trainable params: 10,508,298 (40.09 MB)

Non-trainable params: 0 (0.00 B)

Figure: (Left) Scaled Dot-Product Attention. (Right) Multi-Head Attention.

Contents

Advantages of Transformer-Based Models

- **Parallelization Capability**:
  - Self-attention allows for parallel processing of sequences, leading to faster training times compared to RNNs and LSTMs.
  - Exploits modern hardware efficiently, enhancing scalability for large datasets.
- **Handling Long-Range Dependencies**:
  - Effectively models relationships between distant tokens in a sequence, overcoming limitations of traditional sequential architectures.
  - Provides nuanced context representation crucial for complex tasks like translation and summarization.
- **Flexibility and Expandability**:
  - Easily adapts to various tasks, including text, image, and multimodal inputs, leading to breakthroughs in several domains.
- **Transfer Learning and Pre-Training**:
  - Supports pre-training on large corpora, which transfers knowledge effectively to new tasks, reducing data and resource requirements.
  - Models like BERT and GPT leverage this capability to excel in diverse applications with minimal fine-tuning.

Advantages of Transformer-Based Models (Continues)

- **State-of-the-Art Performance**:
  - Achieves superior results across many natural language processing benchmarks.
  - Transformer-based models are central to the recent progress in AI research, setting new performance standards.

Bidirectional Encoder Representations from Transformers (BERT)

- **Introduction:**
  - Developed by Google AI Language in 2018.
  - Built on the Transformer architecture, focusing on bidirectional context.
- **Key Features:**
  - **Bidirectional Contextual Understanding:**
    - ★ Unlike traditional models that read text sequentially, BERT analyzes context from both directions.
  - **Pre-training and Fine-tuning:**
    - ★ Pre-trained on a large corpus with tasks like Masked Language Modeling (MLM) and Next Sentence Prediction (NSP).
    - ★ Fine-tuned on specific downstream tasks for better task-specific performance.
- **Applications:**
  - Widely used in NLP tasks like sentiment analysis, question answering, and text classification.
  - Performs exceptionally well on benchmarks for understanding natural language.