



MACHINE LEARNING
MASTERY

Building Transformer Models

WITH ATTENTION

Implementing a Neural
Machine Translator from
Scratch in Keras

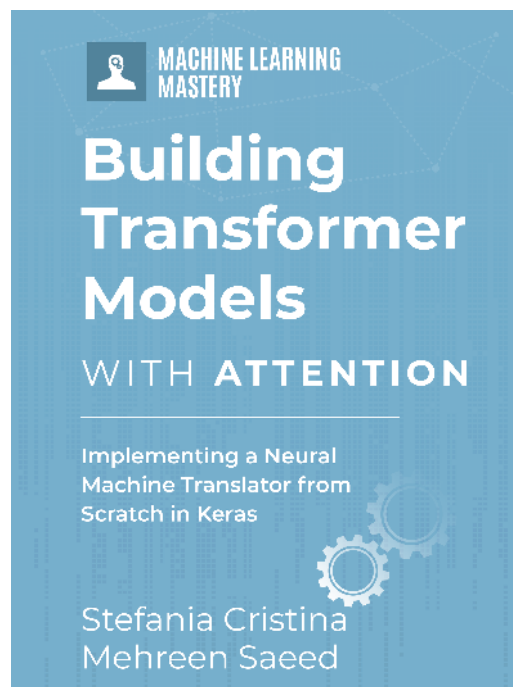
Stefania Cristina
Mehreen Saeed



This is Just a Sample

Thank-you for your interest in **Building Transformer Models with Attention**.

This is just a sample of the full text. You can purchase the complete book online from:
<https://machinelearningmastery.com/transformer-models-with-attention/>



Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Credits

Founder: Jason Brownlee

Authors: Stefania Cristina and Mehreen Saeed

Lead Editor: Adrian Tam

Technical Reviewers: Darci Heikkinen, Devansh Sethi, and Jerry Yiu

Copyright

Building Transformer Models with Attention

© 2022 MachineLearningMastery.com. All Rights Reserved.

Edition: v1.00

Contents

This is Just a Sample	22
Preface	iv
Introduction	v
11 The Transformer Model	1
The Transformer Architecture	1
Sum Up: The Transformer Model	5
Comparison to Recurrent and Convolutional Layers.	6
Further Reading	6
Summary	7
22 Inference with the Transformer Model	8
Inferencing the Transformer Model	8
Testing Out the Code.	13
Further Reading	15
Summary	15
23 A Brief Introduction to BERT	16
From Transformer Model to BERT	16
What Can BERT Do?	18
Using Pre-Trained BERT Model for Summarization.	19
Using Pre-Trained BERT Model for Question-Answering	20
Further Reading	21
Summary	21
This is Just a Sample	22

Preface

It is not an easy task to ask a computer to understand human language. In recent years, we have seen significant progress due to the advance in machine learning techniques. In particular, attention mechanisms and transformers.

Take machine translation as an example. In the past we would consider that as a sequence to sequence transformation problem that a recurrent neural network would fit. But instead of a simple linear transformation, using an attention mechanism was proven to work better with longer sentences. Later, it is discovered that attention without recurrent neural network is not only possible, but also better in many situations.

This book is a guide to lead you to fully understand attention and transformer architecture. We start with the first principle: to build a transformer model in Keras, from scratch. We hope by the time you finish the book, you can appreciate the idea of using attention to extract *context* out of a sequence.

Introduction

Welcome to Building Transformer Models with Attention.

A Recurrent Neural Network (RNN) has been considered magical, and someone even called it unreasonably effective.¹ However, it is not almighty. In machine translation, we have seen that an RNN can give sensible output but not always correct and accurate. While it is a neural network, it is not about the network being too simple or that we didn't train it enough. On the contrary, no matter how hard we train an RNN, there is a ceiling that it cannot breakthrough. Researchers noticed that when using an RNN for translation from one language to another, the neural network reads one word at a time but never sees the entire sentence. Therefore, the traditional way of using an RNN means it will lose the *context*.

Attention is how to mitigate this situation. But it is not as simple as the linear transformation that we usually see in neural networks. Furthermore, researchers found that attention is not necessary to use an RNN. Attention itself can be extended to be a neural network as well. If we do that, we translate words one by one to some encoding or translate the encoding to words. This is a transformer.

However, building an effective transformer for the translation of human languages is not trivial. Partially it is due to the high dimensionality of languages, i.e., any language has thousands of words and can carry a tremendous amount of information. It is also due to the complex architecture of the transformer. But once this hurdle is overcome, you will find the capability of a neural network to deal with human language at a new stage. For example, BERT is an extension of the transformer encoder. We saw that it can be used to build a named entity recognition (NER) system effectively. Another example is GPT-2, which is an extension of the transformer decoder. We saw that it can be used to build a natural language generator that produces realistic-looking paragraphs. These two examples are much larger than the original transformer and very slow to train, but undeniably have their root in attention and transformer.

This book is to guide you through creating a transformer, step-by-step. By doing that, you will learn how to transform a word in a language to an embedding vector, how to implement the attention mechanism, to how a transformer is constructed, and eventually, use it to perform a language translation task.

¹<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

Book Organization

This book is in four parts:

Part 1: Foundation of Attention

In this part you will learn about the theoretical background of attention mechanism. In particular, you will see how attention is defined mathematically and the algorithm to get it. You will also see from a high level how attention, a concept to understand a sequence, is incorporated into a larger neural network architecture as well as its application. This part of the book includes the following chapters:

- ▷ What Is Attention?
- ▷ A Bird's Eye View of Research on Attention
- ▷ A Tour of Attention-Based Architectures
- ▷ The Bahdanau Attention Mechanism
- ▷ The Luong Attention Mechanism

Part 2: From Recurrent Neural Networks to Transformer

Since attention was originally designed for recurrent neural networks, we follow the footstep of its history to start from a traditional recurrent neural network and add an attention layer into it. You may have forgotten how a recurrent neural network is structured, therefore we start from an introduction to the RNN. This part of the book includes the following chapter:

- ▷ An Introduction to Recurrent Neural Networks
- ▷ Understanding Simple Recurrent Neural Networks in Keras
- ▷ The Attention Mechanism from Scratch
- ▷ Adding a Custom Attention Layer to Recurrent Neural Network in Keras
- ▷ The Transformer Attention Mechanism
- ▷ The Transformer Model
- ▷ The Vision Transformer Model

At the end of this part, we introduce how an attention mechanism can stand on its own without an RNN. This is how a transformer is created. While the entire story of attention and transformer is motivated by applying neural networks to natural language processing tasks, the last chapter of this part give you an angle from computer vision to show you that the potential of transformer is not limited to NLP.

Part 3: Building a Transformer from Scratch

Unlike other chapters of this book, you are required to read the chapters of this book in its prescribed sequence. The ten chapters in this part lead you into building a fully working transformer model from scratch. We start from the first step, namely, adding positional

encoding to input sequence, and end with using a trained transformer model for inference. This part of the book includes the following chapters:

- ▷ Positional Encoding in Transformer Models
- ▷ Transformer Positional Encoding Layer in Keras
- ▷ Implementing Scaled Dot-Product Attention in Keras
- ▷ Implementing Multi-Head Attention in Keras
- ▷ Implementing the Transformer Encoder in Keras
- ▷ Implementing the Transformer Decoder in Keras
- ▷ Joining the Transformer Encoder and Decoder with Masking
- ▷ Training the Transformer Model
- ▷ Plotting the Training and Validation Loss Curves for the Transformer Model
- ▷ Inference with the Transformer Model

There are a lot to cover in these chapters because of the complexity in transformer architecture. Be patient. But you will find it not difficult to have your own transformer created out of basic Keras functions.

Part 4: Applications

While you already created your own transformer and by the end of last part, your transformer should be able to do sentence to sentence translation between two languages in a reasonable quality. However, the story of transformer does not stop here. There are larger transformer-based architectures proposed with pre-trained model weights made public. We will look into one example and see how we can do some amazing projects with the pre-trained model.

There is only one chapter in this part. It is:

- ▷ A Brief Introduction to BERT

In this chapter you will learn about BERT, which is an extension to transformer's encoder, and its simplified model, DistilBERT. You will see how you can do summarization and question-answering with a pre-trained DistilBERT model.

Requirements for This Book

Python and TensorFlow 2.x

This book covers some advanced topic. You do not need to be a Python expert, but you need to know how to install and setup Python and TensorFlow. You need to be able to install libraries if required and you should be able to navigate through the Python development environment comfortably. You may set up your environment on your workstation or laptop. It can be in a VM or a Docker instance that you run, or it may be a server that you can configure in the cloud.

Appendix A and Appendix B of this book gives you step-by-step guidance on how to set up a Python environment on your own computer and on AWS cloud, respectively.

Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you know how to solve a small machine learning problem, especially a natural language processing task. Basic concepts like cross-validation are described briefly, and you are expected to know how to train and use a neural network in TensorFlow and Keras. You may learn about these in another book, *Deep Learning with Python*.

Training a transformer model can take a long time. It is possible to train it using a CPU but GPU can speed it up significantly. You can access GPU hardware easily and cheaply in the cloud and a step-by-step procedure is taught on how to do this in Appendix B.

Your Outcomes from Reading This Book

This book is a guidebook to help you learn the internal of a transformer model and the attention mechanism. Upon finishing the book, you should be able to explain clearly why attention works and how a transformer can handle a sequence such as a paragraph of words. Specifically, you will know:

- ▷ What is attention, especially the Bahdanau attention and Luong attention
- ▷ What is a multi-head attention and how it is used in transformer models
- ▷ How to build encoder and decoder of a transformer
- ▷ How to combine the encoder and decoder to create a fully working transformer, and how to train it
- ▷ How to use transformer for real-world tasks

From here you can go deeper to investigate other transformer models, for natural language or for computer vision. You also understand how and what a transformer does. Therefore, you can download some pre-trained models and use them for various tasks.

To get the very most from this book, we recommend following each chapter and build upon them. Attempt to improve the results or the implementation. Write up what you tried or learned and share it on your blog, social media or send us an email at jason@MachineLearningMastery.com.

Summary

This book is a bit different from our other books from MachineLearningMastery.com in the sense that there are not a lot of small projects to work with. Instead, this entire book is one big project, to build a transformer model and apply it to NLP. A big project has many small components. By doing this project, you will learn a lot of ideas. Hope this will be eye-opening for you and bring you to a different level of deep learning. We are excited for you. Take your

time, have fun and we're so excited to see where you can take this amazing new technology to.

Next

Let's dive in. Next up is Part I where you will learn the foundation of attention.

The Transformer Model

We have already familiarized ourselves with the concept of self-attention as implemented by the transformer attention mechanism for neural machine translation. We will now be shifting our focus on the details of the transformer architecture itself to discover how self-attention can be implemented without relying on the use of recurrence and convolutions.

In this chapter, you will discover the network architecture of the transformer model. After completing this chapter, you will know:

- ▷ How the transformer architecture implements an encoder-decoder structure without recurrence and convolutions
- ▷ How the transformer encoder and decoder work
- ▷ How the transformer self-attention compares to the use of recurrent and convolutional layers

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ The Transformer Architecture
- ▷ Sum Up: The Transformer Model
- ▷ Comparison to Recurrent and Convolutional Layers

11.1 The Transformer Architecture

The transformer architecture follows an encoder-decoder structure but does not rely on recurrence and convolutions in order to generate an output.

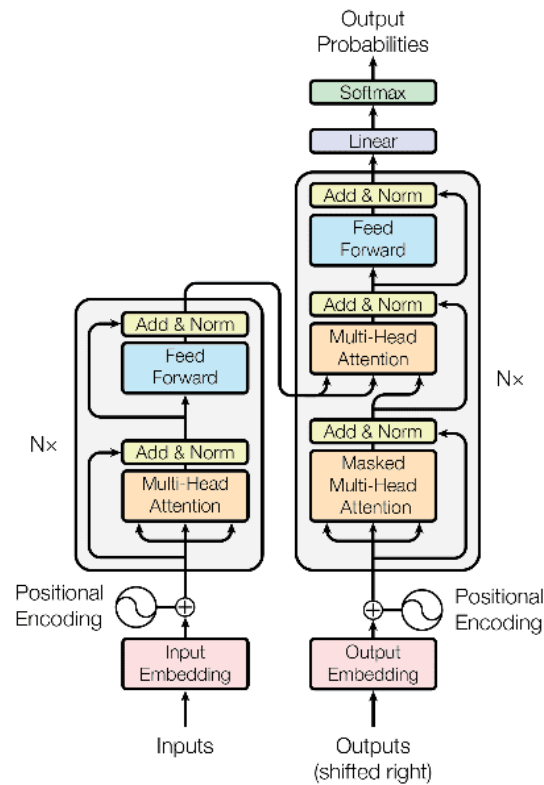


Figure 11.1: The encoder-decoder structure of the transformer architecture. From “Attention Is All You Need”

In a nutshell, the task of the encoder, on the left half of the transformer architecture, is to map an input sequence to a sequence of continuous representations, which is then fed into a decoder. The decoder, on the right half of the architecture, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence.

“ At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

— “Attention Is All You Need”, 2017 ”

The Encoder

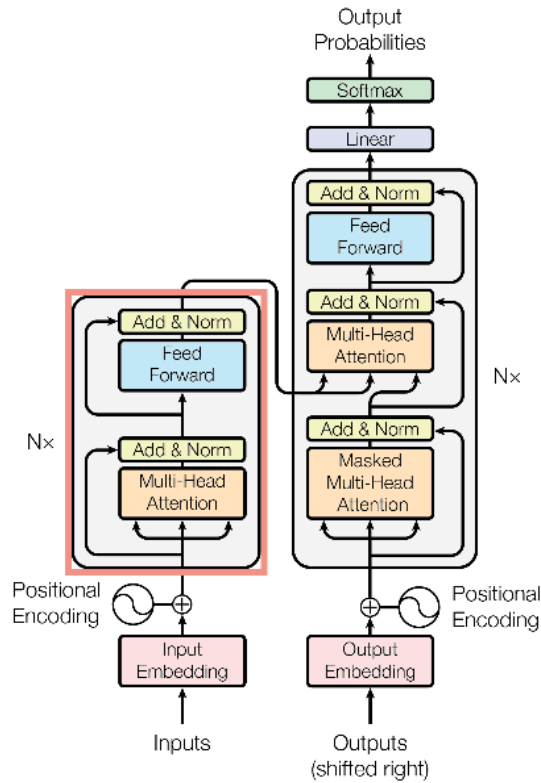


Figure 11.2: The encoder block of the transformer architecture. From “Attention Is All You Need”

The encoder consists of a stack of $N = 6$ identical layers, where each layer is composed of two sublayers:

1. The first sublayer implements a multi-head self-attention mechanism. You have seen that the multi-head mechanism implements h heads that receive a (different) linearly projected version of the queries, keys, and values, each to produce h outputs in parallel that are then used to generate a final result.
2. The second sublayer is a fully connected feedforward network consisting of two linear transformations with Rectified Linear Unit (ReLU) activation in between:

$$\text{FFN}(x) = \text{ReLU}(\mathbf{W}_1 x + b_1) \mathbf{W}_2 + b_2$$

The six layers of the transformer encoder apply the same linear transformations to all the words in the input sequence, but *each* layer employs different weight ($\mathbf{W}_1, \mathbf{W}_2$) and bias (b_1, b_2) parameters to do so.

Furthermore, each of these two sublayers has a residual connection around it. Each sublayer is also succeeded by a normalization layer, $\text{layernorm}(\cdot)$, which normalizes the sum computed between the sublayer input x and the output generated by the sublayer itself, $\text{sublayer}(x)$:

$$\text{layernorm}(x + \text{sublayer}(x))$$

An important consideration to keep in mind is that the transformer architecture cannot inherently capture any information about the relative positions of the words in the sequence since it does not make use of recurrence. This information has to be injected by introducing *positional encodings* to the input embeddings.

The positional encoding vectors are of the same dimension as the input embeddings and are generated using sine and cosine functions of different frequencies. Then, they are simply summed to the input embeddings in order to *inject* the positional information.

The Decoder

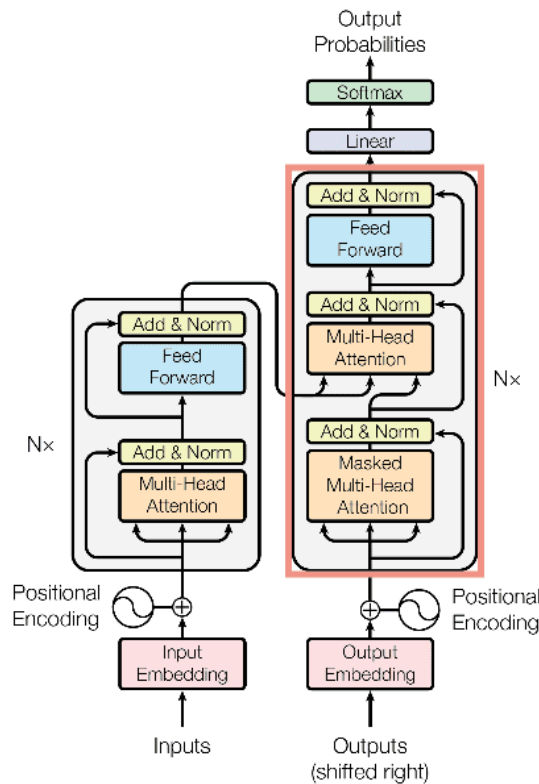


Figure 11.3: The decoder block of the transformer architecture. From “Attention Is All You Need”

The decoder shares several similarities with the encoder. The decoder also consists of a stack of $N = 6$ identical layers that are each composed of three sublayers:

1. The first sublayer receives the previous output of the decoder stack, augments it with positional information, and implements multi-head self-attention over it. While the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding words. Hence, the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence. In the multi-head attention mechanism (which implements multiple, single attention functions in parallel), this is achieved by introducing a mask over the values produced by the scaled multiplication

of matrices $\mathbf{Q}$ and $\mathbf{K}$. This masking is implemented by suppressing the matrix values that would otherwise correspond to illegal connections:

$$\text{mask}(\mathbf{QK}^\top) = \text{mask} \left(\begin{bmatrix} e_{11} & e_{12} & \dots & e_{1n} \\ e_{21} & e_{22} & \dots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix} \right) = \begin{bmatrix} e_{11} & -\infty & \dots & -\infty \\ e_{21} & e_{22} & \dots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix}$$

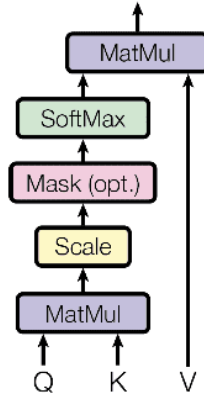


Figure 11.4: The multi-head attention in the decoder implements several masked, single attention functions. From “Attention Is All You Need”

“ The masking makes the decoder unidirectional (unlike the bidirectional encoder). ”

— *Advanced Deep Learning with Python*, 2019 ”

2. The second layer implements a multi-head self-attention mechanism similar to the one implemented in the first sublayer of the encoder. On the decoder side, this multi-head mechanism receives the queries from the previous decoder sublayer and the keys and values from the output of the encoder. This allows the decoder to attend to all the words in the input sequence.
3. The third layer implements a fully connected feedforward network, similar to the one implemented in the second sublayer of the encoder.

Furthermore, the three sublayers on the decoder side also have residual connections around them and are succeeded by a normalization layer. Positional encodings are also added to the input embeddings of the decoder in the same manner as previously explained for the encoder.

11.2 Sum Up: The Transformer Model

The transformer model runs as follows:

1. Each word forming an input sequence is transformed into a d_{model} -dimensional embedding vector.

2. Each embedding vector representing an input word is augmented by summing it (element-wise) to a positional encoding vector of the same d_{model} length, hence introducing positional information into the input.
3. The augmented embedding vectors are fed into the encoder block consisting of the two sublayers explained above. Since the encoder attends to all words in the input sequence, irrespective if they precede or succeed the word under consideration, then the transformer encoder is *bidirectional*.
4. The decoder receives as input its own predicted output word at timestep $t - 1$.
5. The input to the decoder is also augmented by positional encoding in the same manner done on the encoder side.
6. The augmented decoder input is fed into the three sublayers comprising the decoder block explained above. Masking is applied in the first sublayer in order to stop the decoder from attending to succeeding words. At the second sublayer, the decoder also receives the output of the encoder, which now allows the decoder to attend to all the words in the input sequence.
7. The output of the decoder finally passes through a fully connected layer, followed by a softmax layer, to generate a prediction for the next word of the output sequence.

11.3 Comparison to Recurrent and Convolutional Layers

Vaswani et al. (2017) explain that their motivation for abandoning the use of recurrence and convolutions was based on several factors:

1. Self-attention layers were found to be faster than recurrent layers for shorter sequence lengths and can be restricted to consider only a neighborhood in the input sequence for very long sequence lengths.
2. The number of sequential operations required by a recurrent layer is based on the sequence length, whereas this number remains constant for a self-attention layer.
3. In convolutional neural networks, the kernel width directly affects the long-term dependencies that can be established between pairs of input and output positions. Tracking long-term dependencies would require using large kernels or stacks of convolutional layers that could increase the computational cost.

11.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

11.5 Summary

In this chapter, you discovered the network architecture of the transformer model. Specifically, you learned:

- ▷ How the transformer architecture implements an encoder-decoder structure without recurrence and convolutions
- ▷ How the transformer encoder and decoder work
- ▷ How the transformer self-attention compares to recurrent and convolutional layers

Before we wrap up this part, we will digress a bit from text sequences to see, in the next chapter, how transformer can also be applied to images.

Inference with the Transformer Model

22

We have seen how to train the transformer model on a dataset of English and German sentence pairs and how to plot the training and validation loss curves to diagnose the model's learning performance and decide at which epoch to run inference on the trained model. We are now ready to run inference on the trained transformer model to translate an input sentence.

In this chapter, you will discover how to run inference on the trained transformer model for neural machine translation. After completing this chapter, you will know:

- ▷ How to run inference on the trained transformer model
- ▷ How to generate text translations

Let's get started.

Overview

This chapter is divided into two parts; they are:

- ▷ Inferencing the Transformer Model
- ▷ Testing Out the Code

22.1 Inferencing the Transformer Model

Let's start by creating a new instance of the `TransformerModel` class that was previously implemented in Chapter 19.

You will feed into it the relevant input arguments as specified in the paper of “Attention Is All You Need” and the relevant information about the dataset in use:

```
# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack
```

```
# Define the dataset parameters
enc_seq_length = 7 # Encoder sequence length
dec_seq_length = 12 # Decoder sequence length
enc_vocab_size = 2405 # Encoder vocabulary size
dec_vocab_size = 3858 # Decoder vocabulary size

# Create model
inferencing_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
                                     dec_seq_length, h, d_k, d_v, d_model, d_ff, n, 0)
```

Listing 22.1: Parameters from “Attention Is All You Need”

Here, note that the last input being fed into the `TransformerModel` corresponded to the dropout rate for each of the `Dropout` layers in the transformer model. These `Dropout` layers will not be used during model inferencing (you will eventually set the `training` argument to `False`), so you may safely set the dropout rate to 0. Furthermore, the `TransformerModel` class was already saved into a separate script named `model.py`. Hence, to be able to use the `TransformerModel` class, you need to include “`from model import TransformerModel`”.

Next, let’s create a class `Translate` that inherits from the `Module` base class in Keras and assign the initialized inferencing model to the variable `transformer`:

```
class Translate(Module):
    def __init__(self, inferencing_model, **kwargs):
        super().__init__(**kwargs)
        self.transformer = inferencing_model
    ...
```

Listing 22.2: The `Translate` class

When you trained the transformer model, you saw that you first needed to tokenize the sequences of text that were to be fed into both the encoder and decoder. You achieved this by creating a vocabulary of words and replacing each word with its corresponding vocabulary index. You will need to implement a similar process during the inferencing stage before feeding the sequence of text to be translated into the transformer model.

For this purpose, you will include within the class the following `load_tokenizer` method, which will serve to load the encoder and decoder tokenizers that you would have generated and saved during the training stage in Chapter 21:

```
def load_tokenizer(self, name):
    with open(name, 'rb') as handle:
        return load(handle)
```

Listing 22.3: Function to load the trained tokenizers

It is important that you tokenize the input text at the inferencing stage using the same tokenizers generated at the training stage of the transformer model since these tokenizers would have already been trained on text sequences similar to your testing data. The next step is to create the class method, `call()`, that will take care to:

- ▷ Append the start (<START>) and end-of-string (<EOS>) tokens to the input sentence:

```
def __call__(self, sentence):
    sentence[0] = "<START> " + sentence[0] + " <EOS>"
```

- ▷ Load the encoder and decoder tokenizers (in this case, saved in the `enc_tokenizer.pkl` and `dec_tokenizer.pkl` pickle files from Chapter 21, respectively):

```
enc_tokenizer = self.load_tokenizer('enc_tokenizer.pkl')
dec_tokenizer = self.load_tokenizer('dec_tokenizer.pkl')
```

- ▷ Prepare the input sentence by tokenizing it first, then padding it to the maximum phrase length, and subsequently converting it to a tensor:

```
encoder_input = enc_tokenizer.texts_to_sequences(sentence)
encoder_input = pad_sequences(encoder_input,
                             maxlen=enc_seq_length, padding='post')
encoder_input = convert_to_tensor(encoder_input, dtype=int64)
```

- ▷ Repeat a similar tokenization and tensor conversion procedure for the `<START>` and `<EOS>` tokens at the output:

```
output_start = dec_tokenizer.texts_to_sequences(["<START>"])
output_start = convert_to_tensor(output_start[0], dtype=int64)

output_end = dec_tokenizer.texts_to_sequences(["<EOS>"])
output_end = convert_to_tensor(output_end[0], dtype=int64)
```

- ▷ Prepare the output array that will contain the translated text. Since you do not know the length of the translated sentence in advance, you will initialize the size of the output array to 0, but set its `dynamic_size` parameter to `True` so that it may grow past its initial size. You will then set the first value in this output array to the `<START>` token:

```
decoder_output = TensorArray(dtype=int64, size=0, dynamic_size=True)
decoder_output = decoder_output.write(0, output_start)
```

- ▷ Iterate, up to the decoder sequence length, each time calling the transformer model to predict an output token. Here, the `training` input, which is then passed on to each of the transformer's `Dropout` layers, is set to `False` so that no values are dropped during inference. The prediction with the highest score is then selected and written at the next available index of the output array. The `for` loop is terminated with a `break` statement as soon as an `<EOS>` token is predicted:

```
for i in range(dec_seq_length):
    prediction = self.transformer(encoder_input, transpose(decoder_output.stack()),
                                  training=False)
```

```

prediction = prediction[:, -1, :]

predicted_id = argmax(prediction, axis=-1)
predicted_id = predicted_id[0][newaxis]

decoder_output = decoder_output.write(i + 1, predicted_id)

if predicted_id == output_end:
    break

```

▷ Decode the predicted tokens into an output list and return it:

```

output = transpose(decoder_output.stack())[0]
output = output.numpy()

output_str = []

# Decode the predicted tokens into an output list
for i in range(output.shape[0]):
    key = output[i]
    translation = dec_tokenizer.index_word[key]
    output_str.append(translation)

return output_str

```

The complete code listing, so far, is as follows:

```

from pickle import load
from tensorflow import Module
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow import convert_to_tensor, int64, TensorArray, argmax, newaxis, transpose
from model import TransformerModel

# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack

# Define the dataset parameters
enc_seq_length = 7 # Encoder sequence length
dec_seq_length = 12 # Decoder sequence length
enc_vocab_size = 2404 # Encoder vocabulary size
dec_vocab_size = 3864 # Decoder vocabulary size

# Create model
inferencing_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
                                     dec_seq_length, h, d_k, d_v, d_model, d_ff, n, 0)

```

```

class Translate(Module):
    def __init__(self, inferencing_model, **kwargs):
        super().__init__(**kwargs)
        self.transformer = inferencing_model

    def load_tokenizer(self, name):
        with open(name, 'rb') as handle:
            return load(handle)

    def __call__(self, sentence):
        # Append start and end of string tokens to the input sentence
        sentence[0] = "<START> " + sentence[0] + " <EOS>"

        # Load encoder and decoder tokenizers
        enc_tokenizer = self.load_tokenizer('enc_tokenizer.pkl')
        dec_tokenizer = self.load_tokenizer('dec_tokenizer.pkl')

        # Prepare the input sentence by tokenizing, padding and converting to tensor
        encoder_input = enc_tokenizer.texts_to_sequences(sentence)
        encoder_input = pad_sequences(encoder_input,
                                     maxlen=enc_seq_length, padding='post')
        encoder_input = convert_to_tensor(encoder_input, dtype=int64)

        # Prepare the output <START> token by tokenizing, and converting to tensor
        output_start = dec_tokenizer.texts_to_sequences(["<START>"])
        output_start = convert_to_tensor(output_start[0], dtype=int64)

        # Prepare the output <EOS> token by tokenizing, and converting to tensor
        output_end = dec_tokenizer.texts_to_sequences(["<EOS>"])
        output_end = convert_to_tensor(output_end[0], dtype=int64)

        # Prepare the output array of dynamic size
        decoder_output = TensorArray(dtype=int64, size=0, dynamic_size=True)
        decoder_output = decoder_output.write(0, output_start)

        for i in range(dec_seq_length):
            # Predict an output token
            prediction = self.transformer(encoder_input, transpose(decoder_output.stack()),
                                         training=False)
            prediction = prediction[:, -1, :]

            # Select the prediction with the highest score
            predicted_id = argmax(prediction, axis=-1)
            predicted_id = predicted_id[0][newaxis]

            # Write the selected prediction to the output array at the next
            # available index
            decoder_output = decoder_output.write(i + 1, predicted_id)

            # Break if an <EOS> token is predicted
            if predicted_id == output_end:
                break

        output = transpose(decoder_output.stack())[0]

```

```

output = output.numpy()

output_str = []

# Decode the predicted tokens into an output string
for i in range(output.shape[0]):
    key = output[i]
    output_str.append(dec_tokenizer.index_word[key])

return output_str

```

Listing 22.4: Complete code for inference

22.2 Testing Out the Code

In order to test out the code, let's have a look at the `test_dataset.txt` file that you would have saved when preparing the dataset for training in Chapter 21. This text file contains a set of English-German sentence pairs that you have been reserved for testing, from which you can select a couple of sentences to test.

Let's start with the first sentence:

```

# Sentence to translate
sentence = ['im thirsty']

```

Listing 22.5: A sentence for testing

The corresponding ground truth translation in German for this sentence, including the `<START>` and `<EOS>` decoder tokens, should be: “`<START> ich bin durstig <EOS>`”. If you have a look at the plotted training and validation loss curves for this model (here, you are training for 20 epochs), you may notice that the validation loss curve slows down considerably and starts plateauing at around epoch 16.

So let's proceed to load the saved model's weights at the 16th epoch and check out the prediction that is generated by the model:

```

# Load the trained model's weights at the specified epoch
inferencing_model.load_weights('weights/wgths16.ckpt')

# Create a new instance of the 'Translate' class
translator = Translate(inferencing_model)

# Translate the input sentence
print(translator(sentence))

```

Listing 22.6: Loading the model for inference

Running the lines of code above produces the following translated list of words:

```
['start', 'ich', 'bin', 'durstig', 'eos']
```

Output 22.1: Output of the model using weights from the 16th epoch

Which is equivalent to the ground truth German sentence that was expected.



Always keep in mind that since you are training the transformer model from scratch, you may arrive at different results depending on the random initialization of the model weights.

Let's check out what would have happened if you had, instead, loaded a set of weights corresponding to a much earlier epoch, such as the 4th epoch. In this case, the generated translation is the following:

```
['start', 'ich', 'bin', 'nicht', 'nicht', 'eos']
```

Output 22.2: Output of the model using weights from the 4th epoch

In English, this translates to “*I in not not*”, which is clearly far off from the input English sentence, but which is expected since, at this epoch, the learning process of the transformer model is still at the very early stages.

Let's try again with a second sentence from the test dataset:

```
# Sentence to translate
sentence = ['are we done']
```

Listing 22.7: Another sentence for testing

The corresponding ground truth translation in German for this sentence, including the <START> and <EOS> decoder tokens, should be: “<START> sind wir dann durch <EOS>”. The model's translation for this sentence, using the weights saved at epoch 16, is:

```
['start', 'ich', 'war', 'fertig', 'eos']
```

Output 22.3: Output of the model using weights from the 16th epoch

Which, instead, translates to: “*I was ready*”. While this is also not equal to the ground truth, it is *close* to its meaning.

What the last test suggests, however, is that the transformer model might have required many more data samples to train effectively. This is also corroborated by the validation loss at which the validation loss curve plateaus remain relatively high. Indeed, transformer models are notorious for being very data hungry. Vaswani et al. (2017), for example, trained their English-to-German translation model using a dataset containing around 4.5 million sentence pairs.

“ We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs ... For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences ...

— “Attention Is All You Need”, 2017 ”

They reported that it took them 3.5 days on eight P100 GPUs to train the English-to-German translation model. In comparison, you have only trained on a dataset comprising 10,000 data samples here, split between training, validation, and test sets. So the next task is actually for

you. If you have the computational resources available, try to train the transformer model on a much larger set of sentence pairs and see if you can obtain better results than the translations obtained here with a limited amount of data.

22.3 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

22.4 Summary

In this chapter, you discovered how to inference the trained transformer model for neural machine translation. Specifically, you learned:

- ▷ How to inference the trained transformer model
- ▷ How to generate text translations

In the next chapter, you will see how you can use a pre-trained model.

A Brief Introduction to BERT

As we learned what a Transformer is and how we might train the Transformer model, we notice that it is a great tool to make a computer understand human language. However, the Transformer was originally designed as a model to translate one language to another. If we repurpose it for a different task, we would likely need to retrain the whole model from scratch. Given the time it takes to train a Transformer model is enormous, we would like to have a solution that enables us to readily reuse the trained Transformer for many different tasks. BERT is such a model. It is an extension of the encoder part of a Transformer.

In this chapter, you will learn what BERT is and discover what it can do. After completing this chapter, you will know:

- ▷ What is a Bidirectional Encoder Representations from Transformer (BERT)
- ▷ How a BERT model can be reused for different purposes
- ▷ How you can use a pre-trained BERT model

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ▷ From Transformer Model to BERT
- ▷ What Can BERT Do?
- ▷ Using Pre-Trained BERT Model for Summarization
- ▷ Using Pre-Trained BERT Model for Question-Answering

23.1 From Transformer Model to BERT

In the transformer model, the encoder and decoder are connected to make a seq2seq model in order for you to perform a translation, such as from English to German, as you saw before.

Recall that the attention equation says:

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

But each of the Q , K , and V above is an embedding vector transformed by a weight matrix in the transformer model. Training a transformer model means finding these weight matrices. Once the weight matrices are learned, the transformer becomes a *language model*, which means it represents a way to understand the language that you used to train it.

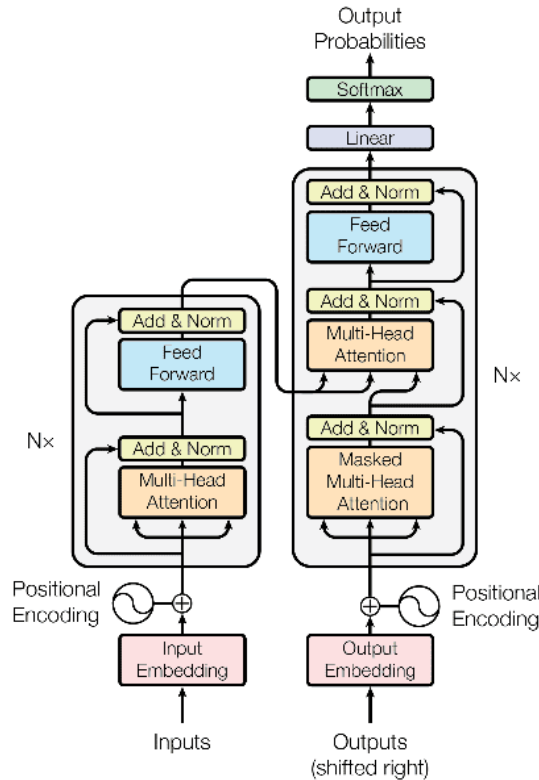


Figure 23.1: The encoder-decoder structure of the Transformer architecture. From “Attention Is All You Need”

A transformer has encoder and decoder parts. As the name implies, the encoder transforms sentences and paragraphs into an internal format (a numerical matrix) that understands the context, whereas the decoder does the reverse. Combining the encoder and decoder allows a transformer to perform seq2seq tasks, such as translation. If you take out the encoder part of the transformer, it can tell you something about the context, which can do something interesting.

The Bidirectional Encoder Representation from Transformer (BERT) leverages the attention model to get a deeper understanding of the language context. BERT is a stack of many encoder blocks. The input text is separated into tokens as in the transformer model, and each token will be transformed into a vector at the output of BERT.

23.2 What Can BERT Do?

A BERT model is trained using the *masked language model* (MLM) and *next sentence prediction* (NSP) simultaneously.

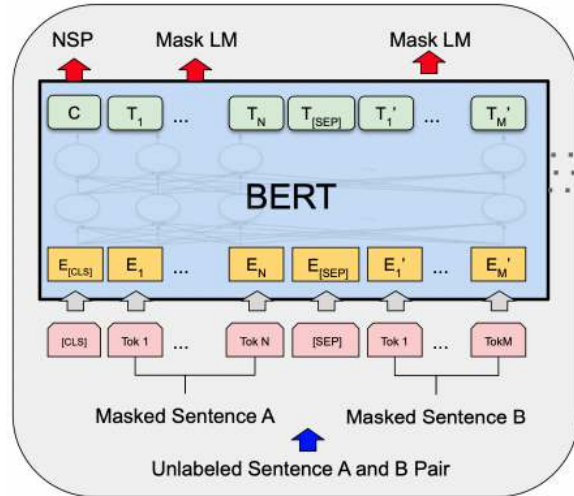


Figure 23.2: The BERT model

Each training sample for BERT is a pair of sentences from a document. The two sentences can be consecutive in the document or not. There will be a [CLS] token prepended to the first sentence (to represent the *class*) and a [SEP] token appended to each sentence (as a *separator*). Then, the two sentences will be concatenated as a sequence of tokens to become a training sample. A small percentage of the tokens in the training sample is *masked* with a special token [MASK] or replaced with a random token.

Before it is fed into the BERT model, the tokens in the training sample will be transformed into embedding vectors, with the positional encodings added, and particular to BERT, with *segment embeddings* added as well to mark whether the token is from the first or the second sentence.

Each input token to the BERT model will produce one output vector. In a well-trained BERT model, we expect:

- ▷ output corresponding to the masked token can reveal what the original token was
- ▷ output corresponding to the [CLS] token at the beginning can reveal whether the two sentences are consecutive in the document

Then, the weights trained in the BERT model can understand the language context well.

Once you have such a BERT model, you can use it for many *downstream tasks*. For example, by adding an appropriate classification layer on top of an encoder and feeding in only one sentence to the model instead of a pair, you can take the class token [CLS] as input for sentiment classification. It works because the output of the class token is trained to aggregate the attention for the entire input.

Another example is to take a question as the first sentence and the text (e.g., a paragraph) as the second sentence, then the output token from the second sentence can mark the position

where the answer to the question rested. It works because the output of each token reveals some information about that token in the context of the entire input.

23.3 Using Pre-Trained BERT Model for Summarization

A transformer model takes a long time to train from scratch. The BERT model would take even longer. But the purpose of BERT is to create one model that can be reused for many different tasks.

There are pre-trained BERT models that you can use readily. In the following, you will see a few use cases. The text used in the following example is from:

▷ <https://www.project-syndicate.org/commentary/bank-of-england-gilt-purchases-necessary-but-mistakes-made-by-willem-h-buiter-and-anne-c-sibert-2022-10>

Theoretically, a BERT model is an encoder that maps each input token to an output vector, which can be extended to an infinite length sequence of tokens. In practice, there are limitations imposed in the implementation of other components that limit the input size. Mostly, a few hundred tokens should work, as not every implementation can take thousands of tokens in one shot. You can save the entire article in `article.txt`. In case your model needs a smaller text, you can use only a few paragraphs from it.

First, let's explore the task for summarization. Using BERT, the idea is to *extract* a few sentences from the original text that represent the entire text. You can see this task is similar to next sentence prediction, in which if given a sentence and the text, you want to classify if they are related.

To do that, you need to use the Python module `bert-extractive-summarizer`

```
pip install bert-extractive-summarizer
```

Listing 23.1: Installing Python module

It is a wrapper to some Hugging Face models to provide the summarization task pipeline. Hugging Face is a platform that allows you to publish machine learning models, mainly on NLP tasks.

Once you have installed `bert-extractive-summarizer`, producing a summary is just a few lines of code:

```
from summarizer import Summarizer
text = open("article.txt").read()
model = Summarizer('distilbert-base-uncased')
result = model(text, num_sentences=3)
print(result)
```

Listing 23.2: Generate extractive summary from text using BERT

This gives the output:

```
Amid the political turmoil of outgoing British Prime Minister Liz Truss's
short-lived government, the Bank of England has found itself in the
fiscal-financial crossfire. Whatever government comes next, it is vital
that the BOE learns the right lessons. According to a statement by the BOE's Deputy Governor for
Financial Stability, Jon Cunliffe, the MPC was merely "informed of the
issues in the gilt market and briefed in advance of the operation,
including its financial-stability rationale and the temporary and targeted
nature of the purchases."
```

Output 23.1: Extractive summary produced

That's the complete code! Behind the scene, spaCy was used on some preprocessing, and Hugging Face was used to launch the model. The model used was named `distilbert-base-uncased`. DistilBERT is a simplified BERT model that can run faster and use less memory. The model is an "uncased" one, which means the uppercase or lowercase in the input text is considered the same once it is transformed into embedding vectors.

The output from the summarizer model is a string. As you specified `num_sentences=3` in invoking the model, the summary is three selected sentences from the text. This approach is called the *extractive summary*. The alternative is an *abstractive summary*, in which the summary is generated rather than extracted from the text. This would need a different model than BERT.

23.4 Using Pre-Trained BERT Model for Question-Answering

The other example of using BERT is to match questions to answers. You will give both the question and the text to the model and look for the output of the beginning *and* the end of the answer from the text.

A quick example would be just a few lines of code as follows, reusing the same example text as in the previous example:

```
from transformers import pipeline
text = open("article.txt").read()
question = "What is BOE doing?"

answering = pipeline("question-answering",
                      model='distilbert-base-uncased-distilled-squad')
result = answering(question=question, context=text)
print(result)
```

Listing 23.3: Question-answering using BERT

Here, Hugging Face is used directly. If you have installed the module used in the previous example, the Hugging Face Python module is a dependence that you already installed. Otherwise, you may need to install it with `pip`:

```
pip install transformers
```

Listing 23.4: Installing Hugging Face module

And to actually use a Hugging Face model, you should have *both* PyTorch and TensorFlow installed as well:

```
pip install torch tensorflow
```

Listing 23.5: Installing PyTorch and TensorFlow modules

The output of the code above is a Python dictionary, as follows:

```
{'score': 0.42369240522384644,
 'start': 1261,
 'end': 1344,
 'answer': 'to maintain or restore market liquidity in systemically
  important\financial markets'}
```

Output 23.2: Question-answering result

This is where you can find the answer (which is a sentence from the input text), as well as the begin and end position in the token order where this answer was from. The score can be regarded as the confidence score from the model that the answer could fit the question.

Behind the scenes, what the model did was generate a probability score for the best beginning in the text that answers the question, as well as the text for the best ending. Then the answer is extracted by finding the location of the highest probabilities.

23.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proc. NAACL*. Vol. 1. June 2019, pp. 4171–4186. DOI: [10.18653/v1/N19-1423](https://arxiv.org/abs/1810.04805).

<https://arxiv.org/abs/1810.04805>

23.6 Summary

In this chapter, you discovered what BERT is and how to use a pre-trained BERT model. Specifically, you learned:

- ▷ How is BERT created as an extension to Transformer models
- ▷ How to use pre-trained BERT models for extractive summarization and question answering

This marks the end of this book.

This is Just a Sample

Thank-you for your interest in **Building Transformer Models with Attention**.

This is just a sample of the full text. You can purchase the complete book online from:
<https://machinelearningmastery.com/transformer-models-with-attention/>

