

EXPERT INSIGHT

Natural Language Processing with TensorFlow

The definitive NLP book to implement the most sought-after machine learning models and tasks

Foreword by:

Andrei Lopatenko,
VP Engineering and Head of Search & NLP at Zillow

Second Edition

Thushan Ganegedara

packt

Natural Language Processing with TensorFlow

Second Edition

The definitive NLP book to implement the most sought-after machine learning models and tasks

Thushan Ganegedara



BIRMINGHAM—MUMBAI

Natural Language Processing with TensorFlow

Second Edition

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Tushar Gupta
Acquisition Editor – Peer Reviews: Saby Dsilva
Project Editor: Parvathy Nair
Content Development Editor: Georgia Daisy van der Post
Copy Editor: Safis Editing
Technical Editor: Tejas Mhasvekar
Proofreader: Safis Editing
Indexer: Subalakshmi Govindhan
Presentation Designer: Rajesh Shirasath

First published: May 2018

Second edition: July 2022

Production reference: 1260722

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83864-135-1

www.packt.com

Foreword

This book addresses the important need for describing how Natural Language Processing (NLP) problems can be solved using TensorFlow-based NLP stacks.

Deep Learning revolutionized NLP recently. Many industrial and academic NLP problems that required a large amount of work in terms of designing new features, tuning models, and finding the best modeling approach (CRF, SVM, Bayesian methods, etc.) can now be solved by NLP scientists in a significantly smaller amount of time. Also, the new deep-learning-based methods typically produce much more accurate models than traditional NLP methods. The big problem is how to make these models work in a modern production setting with operational parameters such as latency and throughput, cloud costs, and operational quality (uptime, etc.). The TensorFlow environment is designed to solve these problems when running NLP models.

In this book, the author teaches the fundamentals of TensorFlow and Keras, a Python-based interface for TensorFlow. Then, the bulk of the book, from *Chapter 3, Word2vec – Learning Word Embeddings*, onward, is focused on NLP problems and solving them using TensorFlow.

This book provides:

- A knowledge of NLP methods in good detail, from their definition to various evaluation methods
- Information about TensorFlow, Keras, and Hugging Face libraries, which are powerful tools to build NLP solutions
- An understanding of neural architectures, which is important to build better models, by building architectures for specific tasks that the reader will encounter in their practice.

The author describes the process of building embeddings and other vector representations that are the basis of most modern deep learning NLP methods. The author also describes popular Neural Network architectures, such as Recurrent Neural Networks, Convolutional Neural Networks, Long Short-Term Memory networks, and Transformer-based architectures, in detail and shows their application in solving various NLP tasks, such as sentence classification, named entity recognition, text generation, machine translation, image caption generation, and more.

In each chapter, the author provides a deep dive into the neural network architecture, with an explanation of why this architecture works; the nature of the NLP problem and why it is an important NLP task; and how the solution to the problem is evaluated. Such deep dives will help readers to address industrial tasks that are reducible to these NLP problems, and to solve other NLP problems through understanding how typical NLP problems are evaluated. These deep dives will also help provide the reader with the knowledge to modify and improve necessary network architectures for particular practical tasks. The author also provides a detailed, step-by-step description of how such models are trained in a TensorFlow/Keras environment.

At the end, the author writes about Transformers, the modern state-of-the-art method to solve NLP problems, with a focus on BERT (a popular transformer method developed by Google). The author provides exercises on how BERT can be used for practical tasks such as answering questions, but the explanations of BERT will also help to solve other tasks with BERT-based networks. The author also dives into Hugging Face, a popular software library for transformer-based NLP solutions.

All of this content makes this book invaluable for practitioners who want to learn how to build TensorFlow-based solutions for NLP problems.

Andrei Lopatenko

VP Engineering and Head of Search & NLP at Zillow

Contributors

About the author

Thushan Ganegedara is a Senior Machine Learning engineer at Canva, an Australian technology unicorn that's democratizing graphic designing and visualizations.. Thushan works with large-scale visual and text data, in order to build and deploy Machine Learning models to make products smarter. Before this, Thushan worked as a Senior Data Scientist at QBE Insurance, helping to solve business problems and make claim processing more efficient using machine learning. Thushan has a PhD from the University of Sydney specializing in Deep Learning.

I would like to acknowledge my parents and my wife, Thushani, for all the support and encouragement provided during the development of this book.

About the reviewers

Arman Cohan is a Research Scientist at the Allen Institute for AI (AI2). His broad research interest is in developing Natural Language Processing methods for addressing information overload. This includes language models for complex document and multi-document tasks, natural language generation and summarization, and information discovery and filtering. His research has been recognized with multiple awards from leading conferences in the field, including a best paper award at EMNLP 2017, an honorable mention at COLING 2018, and the 2019 Harold N. Glassman Distinguished Doctoral Dissertation award.

Pratik Kotian is a Senior Conversation AI engineer with six years of experience in building conversational AI agents and designing products related to conversational design. He is working as a Senior Conversation Bot Engineer (specializing in conversational AI) at Quantiphi, which is an AI company and recognized Google Partner. He has also worked with Packt on reviewing *The TensorFlow Workshop* and *Conversational AI with Rasa*.

I would like to thank my family and friends, who are always supportive and have always believed in me and my talents. It's because of them that I am doing well in my career. And lastly, I would like to thank all the readers of this book: you are definitely going to learn a lot about recent developments in NLP and TensorFlow from this book.

Taeuk Kim is an assistant professor at the Department of Computer Science, Hanyang University. Before joining Hanyang University, he received his Bachelor of Science and PhD from Seoul National University. His expertise lies in the field of Natural Language Processing, where he has been making contributions as an active researcher and a program committee member for related top-tier conferences including ACL and EMNLP.

Dr Pham Quang Nhat Minh is the head of Multimodal AI Lab of Aimesoft JSC, Vietnam. His research field is Language and Speech Processing. Dr Minh has more than fifteen years of experience working in the Natural Language Processing and Machine Learning fields in both academia and industry. He obtained a Bachelor's degree at VNU University of Engineering and Technology in 2006. He obtained a Master's degree in Information Science in 2010 at Japan Advanced Institute of Science and Technology (JAIST) and a PhD in Information Science in 2013 at the same school. He has published several research papers in the NLP field. Currently, he is working in the field of law text processing.

Table of Contents

Preface	xix
<hr/>	
Chapter 1: Introduction to Natural Language Processing	1
What is Natural Language Processing?	2
Tasks of Natural Language Processing	3
The traditional approach to Natural Language Processing	5
Understanding the traditional approach • 5	
<i>Example – generating football game summaries</i> • 7	
Drawbacks of the traditional approach • 10	
The deep learning approach to Natural Language Processing	11
History of deep learning • 12	
The current state of deep learning and NLP • 13	
Understanding a simple deep model – a fully connected neural network • 16	
Introduction to the technical tools	18
Description of the tools • 18	
Installing Anaconda and Python • 18	
<i>Creating a Conda environment</i> • 19	
TensorFlow (GPU) software requirements • 19	
Accessing Jupyter Notebook • 19	
Verifying the TensorFlow installation • 20	
Summary	21
<hr/>	
Chapter 2: Understanding TensorFlow 2	23
What is TensorFlow?	24
Getting started with TensorFlow 2 • 24	
TensorFlow 2 architecture – What happens during graph build? • 29	

TensorFlow architecture – what happens when you execute the graph? • 31	
Café Le TensorFlow 2 – understanding TensorFlow 2 with an analogy • 33	
Flashback: TensorFlow 1 • 35	
Inputs, variables, outputs, and operations	38
Defining inputs in TensorFlow • 39	
<i>Feeding data as NumPy arrays</i> • 39	
<i>Feeding data as tensors</i> • 39	
<i>Building a data pipeline using the tf.data API</i> • 40	
Defining variables in TensorFlow • 43	
Defining outputs in TensorFlow • 45	
Defining operations in TensorFlow • 45	
<i>Comparison operations</i> • 45	
<i>Mathematical operations</i> • 46	
<i>Updating (scattering) values in tensors</i> • 47	
<i>Collecting (gathering) values from a tensor</i> • 49	
Neural network-related operations • 49	
<i>Nonlinear activations used by neural networks</i> • 49	
<i>The convolution operation</i> • 51	
<i>The pooling operation</i> • 54	
<i>Defining loss</i> • 56	
Keras: The model building API of TensorFlow	56
Sequential API • 57	
Functional API • 58	
Sub-classing API • 58	
Implementing our first neural network	59
Preparing the data • 60	
Implementing the neural network with Keras • 61	
<i>Training the model</i> • 62	
<i>Testing the model</i> • 63	
Summary	64

Chapter 3: Word2vec – Learning Word Embeddings	67
What is a word representation or meaning?	69
Classical approaches to learning word representation	70
One-hot encoded representation • 70	
The TF-IDF method • 71	
Co-occurrence matrix • 72	
An intuitive understanding of Word2vec – an approach to learning word representation	73
Exercise: does queen = king – he + she? • 74	
The skip-gram algorithm	78
From raw text to semi-structured text • 78	
Understanding the skip-gram algorithm • 79	
Implementing and running the skip-gram algorithm with TensorFlow • 82	
<i>Implementing the data generators with TensorFlow</i> • 83	
<i>Implementing the skip-gram architecture with TensorFlow</i> • 95	
<i>Training and evaluating the model</i> • 99	
The Continuous Bag-of-Words algorithm	102
Generating data for the CBOW algorithm • 103	
Implementing CBOW in TensorFlow • 104	
Training and evaluating the model • 106	
Summary	108
Chapter 4: Advanced Word Vector Algorithms	111
GloVe – Global Vectors representation	112
Understanding GloVe • 114	
Implementing GloVe • 116	
Generating data for GloVe • 120	
Training and evaluating GloVe • 123	
ELMo – Taking ambiguities out of word vectors	126
Downloading ELMo from TensorFlow Hub • 128	
Preparing inputs for ELMo • 129	

Generating embeddings with ELMo • 132	
Document classification with ELMo	134
Dataset • 135	
Generating document embeddings • 138	
Classifying documents with document embeddings • 142	
Summary	145
<hr/>	
Chapter 5: Sentence Classification with Convolutional Neural Networks	147
<hr/>	
Introducing CNNs	148
<i>CNN fundamentals</i> • 148	
<i>The power of CNNs</i> • 151	
Understanding CNNs	151
Convolution operation • 151	
<i>Standard convolution operation</i> • 152	
<i>Convolving with stride</i> • 152	
<i>Convolving with padding</i> • 153	
<i>Transposed convolution</i> • 154	
Pooling operation • 155	
<i>Max pooling</i> • 156	
<i>Max pooling with stride</i> • 156	
<i>Average pooling</i> • 157	
Fully connected layers • 158	
Putting everything together • 158	
Exercise – image classification on Fashion-MNIST with CNN	159
About the data • 160	
Downloading and exploring the data • 160	
Implementing the CNN • 162	
Analyzing the predictions produced with a CNN • 167	
Using CNNs for sentence classification	168
How data is transformed for sentence classification • 169	
Implementation – downloading and preparing data • 171	

<i>Implementation – building a tokenizer</i> • 176	
The sentence classification CNN model • 177	
<i>The convolution operation</i> • 177	
<i>Pooling over time</i> • 180	
Implementation – sentence classification with CNNs • 182	
Training the model • 186	
Summary	188
<hr/>	
Chapter 6: Recurrent Neural Networks	191
<hr/>	
Understanding RNNs	192
The problem with feed-forward neural networks • 193	
Modeling with RNNs • 194	
Technical description of an RNN • 196	
Backpropagation Through Time	197
How backpropagation works • 197	
Why we cannot use BP directly for RNNs • 199	
Backpropagation Through Time – training RNNs • 200	
Truncated BPTT – training RNNs efficiently • 200	
Limitations of BPTT – vanishing and exploding gradients • 201	
Applications of RNNs	203
One-to-one RNNs • 203	
One-to-many RNNs • 203	
Many-to-one RNNs • 204	
Many-to-many RNNs • 205	
Named Entity Recognition with RNNs	206
Understanding the data • 206	
Processing data • 212	
Defining hyperparameters • 215	
Defining the model • 216	
<i>Introduction to the TextVectorization layer</i> • 217	
<i>Defining the rest of the model</i> • 219	

Evaluation metrics and the loss function • 223	
Training and evaluating RNN on NER task • 226	
Visually analyzing outputs • 229	
NER with character and token embeddings	231
Using convolution to generate token embeddings • 231	
Implementing the new NER model • 234	
<i>Defining hyperparameters</i> • 235	
<i>Defining the input layer</i> • 235	
<i>Defining the token-based TextVectorization layer</i> • 235	
<i>Defining the character-based TextVectorization layer</i> • 235	
<i>Processing the inputs for the char_vectorize_layer</i> • 235	
<i>Performing convolution on the character embeddings</i> • 237	
Model training and evaluation • 239	
Other improvements you can make • 239	
Summary	240
<hr/>	
Chapter 7: Understanding Long Short-Term Memory Networks	243
<hr/>	
Understanding Long Short-Term Memory Networks	244
What is an LSTM? • 245	
LSTMs in more detail • 246	
How LSTMs differ from standard RNNs • 255	
How LSTMs solve the vanishing gradient problem	256
Improving LSTMs	259
Greedy sampling • 259	
Beam search • 260	
Using word vectors • 261	
Bidirectional LSTMs (BiLSTMs) • 263	
Other variants of LSTMs	265
Peephole connections • 265	
Gated Recurrent Units • 266	
Summary	268

Chapter 8: Applications of LSTM – Generating Text	271
Our data	272
About the dataset • 272	
Generating training, validation, and test sets • 274	
Analyzing the vocabulary size • 275	
Defining the tf.data pipeline • 276	
Implementing the language model	282
Defining the TextVectorization layer • 283	
Defining the LSTM model • 284	
Defining metrics and compiling the model • 286	
Training the model • 288	
Defining the inference model • 288	
Generating new text with the model • 292	
Comparing LSTMs to LSTMs with peephole connections and GRUs	295
Standard LSTM • 295	
<i>Review</i> • 295	
Gated Recurrent Units (GRUs) • 296	
<i>Review</i> • 296	
<i>The model</i> • 297	
LSTMs with peepholes • 298	
<i>Review</i> • 298	
<i>The code</i> • 299	
Training and validation perplexities over time • 300	
Improving sequential models – beam search	301
Implementing beam search • 302	
Generating text with beam search • 304	
Improving LSTMs – generating text with words instead of n-grams	305
The curse of dimensionality • 306	
Word2vec to the rescue • 306	
Generating text with Word2vec • 306	

Summary	308
<hr/>	
Chapter 9: Sequence-to-Sequence Learning – Neural Machine Translation	311
<hr/>	
Machine translation	312
A brief historical tour of machine translation	313
Rule-based translation • 313	
Statistical Machine Translation (SMT) • 315	
Neural Machine Translation (NMT) • 316	
Understanding neural machine translation	320
Intuition behind NMT systems • 320	
NMT architecture • 320	
<i>The embedding layer</i> • 322	
<i>The encoder</i> • 322	
<i>The context vector</i> • 323	
<i>The decoder</i> • 323	
Preparing data for the NMT system	324
The dataset • 324	
<i>Adding special tokens</i> • 326	
<i>Splitting training, validation, and testing datasets</i> • 326	
<i>Defining sequence lengths for the two languages</i> • 327	
<i>Padding the sentences</i> • 328	
Defining the model	330
Converting tokens to IDs • 331	
Defining the encoder • 332	
Defining the decoder • 333	
Attention: Analyzing the encoder states • 336	
<i>Computing Attention</i> • 338	
<i>Implementing Attention</i> • 340	
Defining the final model • 344	
Training the NMT	346
The BLEU score – evaluating the machine translation systems	353

Modified precision • 354	
Brevity penalty • 355	
The final BLEU score • 355	
Visualizing Attention patterns	355
Inference with NMT	360
Other applications of Seq2Seq models – chatbots	361
Training a chatbot • 362	
Evaluating chatbots – the Turing test • 363	
Summary	364
Chapter 10: Transformers	365
Transformer architecture	365
The encoder and the decoder • 366	
Computing the output of the self-attention layer • 370	
Embedding layers in the Transformer • 372	
Residuals and normalization • 375	
Understanding BERT	377
Input processing for BERT • 379	
Tasks solved by BERT • 379	
How BERT is pre-trained • 382	
<i>Masked Language Modeling (MLM)</i> • 383	
<i>Next Sentence Prediction (NSP)</i> • 383	
Use case: Using BERT to answer questions	384
Introduction to the Hugging Face transformers library • 384	
Exploring the data • 385	
Implementing BERT • 387	
<i>Implementing and using the Tokenizer</i> • 387	
<i>Defining a TensorFlow dataset</i> • 391	
<i>BERT for answering questions</i> • 393	
<i>Defining the config and the model</i> • 394	
Training and evaluating the model • 397	

Answering questions with Bert • 399	
Summary	401
Chapter 11: Image Captioning with Transformers	403
Getting to know the data	404
ILSVRC ImageNet dataset • 405	
The MS-COCO dataset • 406	
Downloading the data	407
Processing and tokenizing data	408
Preprocessing data • 408	
Tokenizing data • 412	
Defining a tf.data.Dataset	414
The machine learning pipeline for image caption generation	420
Vision Transformer (ViT) • 421	
Text-based decoder Transformer • 423	
Putting everything together • 423	
Implementing the model with TensorFlow	424
Implementing the ViT model • 424	
Implementing the text-based decoder • 425	
<i>Defining the self-attention layer</i> • 425	
<i>Defining the Transformer layer</i> • 427	
<i>Defining the full decoder</i> • 429	
Training the model	432
Evaluating the results quantitatively	435
BLEU • 436	
ROUGE • 437	
METEOR • 438	
CIDEr • 439	
Evaluating the model	440
Captions generated for test images	441
Summary	446

Appendix A: Mathematical Foundations and Advanced TensorFlow	449
Basic data structures	449
Scalar • 449	
Vectors • 449	
Matrices • 450	
Indexing of a matrix • 450	
Special types of matrices	451
Identity matrix • 451	
Square diagonal matrix • 452	
Tensors • 452	
Tensor/matrix operations	452
Transpose • 452	
Matrix multiplication • 453	
Element-wise multiplication • 453	
Inverse • 454	
Finding the matrix inverse – Singular Value Decomposition (SVD) • 455	
Norms • 456	
Determinant • 456	
Probability	457
Random variables • 457	
<i>Discrete random variables</i> • 458	
<i>Continuous random variables</i> • 458	
The probability mass/density function • 458	
Conditional probability • 460	
Joint probability • 461	
Marginal probability • 461	
Bayes' rule • 461	
Visualizing word embeddings with TensorBoard	462
Starting TensorBoard • 462	
Saving word embeddings and visualizing via TensorBoard • 463	

Summary	467
Other Books You May Enjoy	471
Index	475

Preface

TensorFlow is at the center of developing **Machine Learning (ML)** solutions. It is an ecosystem that can support all of the different stages in the life cycle of an ML project, from the early prototyping up until the productionization of the model. TensorFlow provides various reusable building blocks, allowing you to build not just the simplest but also the most complex deep neural networks.

Who this book is for

This book is aimed at novice - to intermediate-level users of TensorFlow. The reader may be from academia doing cutting-edge research on ML or an industry practitioner using ML in their job. You will get the most benefit from this book if you have some basic familiarity with TensorFlow (or a similar framework like Pytorch) already. This will help you to grasp the concepts and use cases discussed in the book quicker.

What this book covers

Chapter 1, Introduction to Natural Language Processing, explains what natural language processing is and the kinds of tasks it may entail. We then discuss how an NLP task is solved using traditional methods. This paves the way to discuss how deep learning is used in NLP and what the benefits are. Finally, we discuss the installation and usage of the technical tools in this book.

Chapter 2, Understanding TensorFlow 2, provides you with a sound guide to writing programs and running them in TensorFlow 2. This chapter will first offer an in-depth explanation of how TensorFlow executes a program. This will help you to understand the TensorFlow execution workflow and feel comfortable with TensorFlow terminology. Next, we will discuss various building blocks in TensorFlow and useful operations that are available. We will finally discuss how all this knowledge of TensorFlow can be used to implement a simple neural network to classify images of handwritten digits.

Chapter 3, Word2vec – Learning Word Embeddings, introduces Word2vec—a method to learn numerical representations of words that reflect the semantics of the words. But before diving straight into Word2vec techniques, we first discuss some classical approaches used to represent words, such as one-hot-encoded representations, and the **Term Frequency-Inverse Document Frequency (TF-IDF)** frequency method. Following this, we will move on to a modern tool for learning word vectors known as Word2vec, which uses a neural network to learn word representations. We will discuss two popular Word2vec variants: skip-gram and the **Continuous Bag-of-Words (CBOW)** model. Finally, we will visualize the word representations learned using a dimensionality reduction technique to map the vectors to a more interpretable two-dimensional surface.

Chapter 4, Advanced Word Vector Algorithms, starts with a more recent word embedding learning technique known as GloVe, which incorporates both global and local statistics in text data to find word vectors. Next, we will learn about one of the modern, more sophisticated techniques for generating dynamic word representations based on the context of a word, known as ELMo.

Chapter 5, Sentence Classification with Convolutional Neural Networks, introduces you to **Convolutional Neural Networks (CNNs)**. CNNs are a powerful family of deep models that can leverage the spatial structure of an input to learn from data. In other words, a CNN can process images in their two-dimensional form, whereas a multilayer perceptron needs the image to be unwrapped to a one-dimensional vector. We will first discuss various operations that are undergone in CNNs, such as the convolution and pooling operations, in detail. Then, we will see an example where we will learn to classify images of clothes with a CNN. Then, we will transition into an application of CNNs in NLP. More precisely, we will be investigating how to apply a CNN to classify sentences, where the task is to classify if a sentence is about a person, location, object, and so on.

Chapter 6, Recurrent Neural Networks, focuses on introducing **Recurrent Neural Networks (RNNs)** and using RNNs for language generation. RNNs are different from feed-forward neural networks (for example, CNNs) as RNNs have memory. The memory is stored as a continuously updated system state. We will start with a representation of a feed-forward neural network and modify that representation to learn from sequences of data instead of individual data points. This process will transform the feed-forward network to an RNN. This will be followed by a technical description of the exact equations used for computations within the RNN. Next, we will discuss the optimization process of RNNs that is used to update the RNN’s weights. Thereafter we will iterate through different types of RNNs such as one-to-one RNNs and one-to-many RNNs. We will then discuss a popular application of RNNs, which is to identify named entities in text (for example, Person name, Organization, and so on). Here, we’ll be using a basic RNN model to learn.

Next, we will enhance our model further by incorporating embeddings at different scales (for example, token embeddings and character embeddings). The token embeddings are generated through an embedding layer, where the character embeddings are generated using a CNN. We will then analyze the new model’s performance on the named entity recognition task.

Chapter 7, Understanding Long Short-Term Memory Networks, discusses **Long Short-Term Memory networks (LSTMs)** by initially providing an intuitive explanation of how these models work and progressively diving into the technical details required to implement them on your own. Standard RNNs suffer from the crucial limitation of the inability to persist long-term memory. However, advanced RNN models (for example, LSTMs and **Gated Recurrent Units (GRUs)**) have been proposed, which can remember sequences for a large number of time steps. We will also examine how exactly LSTMs alleviate the problem of persisting long-term memory (this is known as the vanishing gradient problem). We will then discuss several modifications that can be used to improve LSTM models further, such as predicting several time steps ahead at once and reading sequences both forward and backward. Finally, we will discuss several variants of LSTM models such as GRUs and LSTMs with peephole connections.

Chapter 8, Applications of LSTM – Generating Text, explains how to implement the LSTMs, GRUs, and LSTMs with peephole connections discussed in *Chapter 7, Understanding Long Short-Term Memory Networks*. Furthermore, we will compare the performance of these extensions both qualitatively and quantitatively. We will also discuss how to implement some of the extensions examined in *Chapter 7, Understanding Long Short-Term Memory Networks*, such as predicting several time steps ahead (known as beam search) and using word vectors as inputs instead of one-hot-encoded representations.

Chapter 9, Sequence-to-Sequence Learning – Neural Machine Translation, discusses machine translation, which has gained a lot of attention both due to the necessity of automating translation and the inherent difficulty of the task. We start the chapter with a brief historical flashback explaining how machine translation was implemented in the early days. This discussion ends with an introduction to **Neural Machine Translation (NMT)** systems. We will see how well current NMT systems are doing compared to old systems (such as statistical machine translation systems), which will motivate us to learn about NMT systems. Afterward, we will discuss the concepts underpinning the design of NMT systems and continue with the technical details. Then, we will discuss the evaluation metric we use to evaluate our system. Following this, we will investigate how we can implement an English-to-German translator from scratch. Next, we will learn about ways to improve NMT systems. We will look at one of those extensions in detail, called the attention mechanism. The attention mechanism has become essential in sequence-to-sequence learning problems.

Finally, we will compare the performance improvement obtained with the attention mechanism and analyze the reasons behind the performance gain. This chapter concludes with a section on how the same concept of NMT systems can be extended to implement chatbots. Chatbots are systems that can communicate with humans and are used to fulfill various customer requests.

Chapter 10, Transformers, discusses Transformers, the latest breakthrough in the domain of NLP which have outperformed many other previous state-of-the-art models. In this chapter, we will use the Hugging Face Transformers library to use pre-trained models for downstream tasks with ease. In this chapter, we will learn about the Transformer architecture in depth. This discussion will lead into a popular Transformer model called BERT, which we will use to solve a problem of question answering. We will discuss specific components found in BERT to effectively use it for the application. Next, we will train the model on a popular question-answer dataset known as SQuAD. Finally, we will evaluate the model on a test dataset and use the trained model to generate answers for unseen questions.

Chapter 11, Image Captioning with Transformers, looks at another exciting application, where Transformers are used to generate captions (that is, descriptions) for images. This application is interesting because it shows us how to combine two different types of models as well as how to learn with multimodal data (for example, images and text). Here, we will use a pre-trained Vision Transformer model that generates a rich hidden representation for a given image. This representation, along with caption tokens, is fed to a text-based Transformer model. The text-based Transformer predicts the next caption token, given previous caption tokens. Once the model is trained, we will evaluate the captions generated by our model, both qualitatively and quantitatively. We will also discuss some of the popular metrics used to measure the quality of sequences such as image captions.

Appendix A: Mathematical Foundations and Advanced TensorFlow, introduces various mathematical data structures (for example, matrices) and operations (for example, a matrix inverse). We will also discuss several important concepts in probability. Finally, we will walk you through a guide aimed at teaching you to use TensorBoard to visualize word embeddings. TensorBoard is a handy visualization tool that is shipped with TensorFlow. This can be used to visualize and monitor various variables in your TensorFlow client.

To get the most out of this book

To get the most out of this book you need a basic understanding of TensorFlow or a similar framework such as PyTorch. Familiarity obtained through basic TensorFlow tutorials that are freely available in the web should suffice to get started on this book.

A basic knowledge of mathematics, including an understanding of n-dimensional tensors, matrix multiplication, and so on, will also prove invaluable throughout this book. Finally, you need an enthusiasm for learning about cutting edge machine learning that is setting the stage for modern NLP solutions.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/thushv89/packt_nlp_tensorflow_2. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781838641351_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “After running the `pip install` command, you should have Jupyter Notebook available in the Conda environment.”

A block of code is set as follows:

```
def layer(x, W, b):
    # Building the graph
    h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to perform
    return h
```

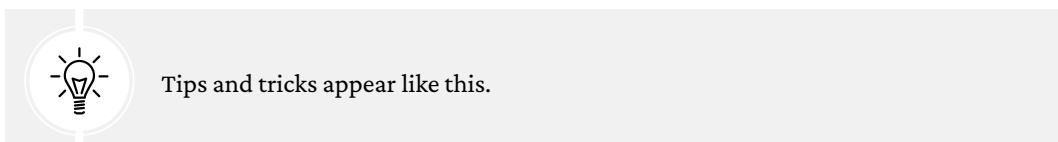
Any command-line input or output is written as follows:

```
<tf.Variable 'ref:0' shape=(3, 2) dtype=float32, numpy=
array([[-1., -9.],
       [ 3., 10.],
       [ 5., 11.]], dtype=float32)>
```

Bold: Indicates a new term or an important word. Words that you see on the screen (such as in menus or dialog boxes) also appear in the text like this, for example: “The feature that builds this computational graph automatically in TensorFlow is known as **AutoGraph**.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book’s title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit <http://www.packtpub.com/submit-errata>, select your book, click on the Errata Submission Form link, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Share your thoughts

Once you've read *Natural Language Processing with TensorFlow, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Introduction to Natural Language Processing

Natural Language Processing (NLP) offers a much-needed set of tools and algorithms for understanding and processing the large volume of unstructured data in today's world. Recently, deep learning has been widely adopted for many NLP tasks because of the remarkable performance deep learning algorithms have shown in a plethora of challenging tasks, such as image classification, speech recognition, and realistic text generation. TensorFlow is one of the most intuitive and efficient deep learning frameworks currently in existence that enables such amazing feats. This book will enable aspiring deep learning developers to handle massive amounts of data using NLP and TensorFlow. This chapter covers the following topics:

- What is Natural Language Processing?
- Tasks of Natural Language Processing
- The traditional approach to Natural Language Processing
- The deep learning approach to Natural Language Processing
- Introduction to the technical tools

In this chapter, we will provide an introduction to NLP and to the rest of the book. We will answer the question, "What is Natural Language Processing?". Also, we'll look at some of its most important use cases. We will also consider the traditional approaches and the more recent deep learning-based approaches to NLP, including a **Fully Connected Neural Network (FCNN)**. Finally, we will conclude with an overview of the rest of the book and the technical tools we will be using.

What is Natural Language Processing?

According to DOMO (<https://www.domo.com/>), an analytics company, there were 1.7MB for every person on earth every second by 2020, with a staggering 4.6 billion active users on the internet. This includes roughly 500,000 tweets sent and 306 billion emails circulated every day. These figures are only going in one direction as this book is being written, and that is up! Of all this data, a large fraction is unstructured text and speech as there are billions of emails and social media content created and phone calls made every day.

These statistics provide a good basis for us to define what NLP is. Simply put, the goal of NLP is to make machines understand our spoken and written languages. Moreover, NLP is ubiquitous and is already a large part of human life. **Virtual Assistants (VAs)**, such as Google Assistant, Cortana, Alexa, and Apple Siri, are largely NLP systems. Numerous NLP tasks take place when one asks a VA, “*Can you show me a good Italian restaurant nearby?*” First, the VA needs to convert the utterance to text (that is, speech-to-text). Next, it must understand the semantics of the request (for example, identify the most important keywords like restaurant and Italian) and formulate a structured request (for example, cuisine = Italian, rating = 3–5, distance < 10 km). Then, the VA must search for restaurants filtering by the location and cuisine, and then, rank the restaurants by the ratings received. To calculate an overall rating for a restaurant, a good NLP system may look at both the rating and text description provided by each user. Finally, once the user is at the restaurant, the VA might assist the user by translating various menu items from Italian to English. This example shows that NLP has become an integral part of human life.

It should be understood that NLP is an extremely challenging field of research as words and semantics have a highly complex nonlinear relationship, and it is even more difficult to capture this information as a robust numerical representation. To make matters worse, each language has its own grammar, syntax, and vocabulary. Therefore, processing textual data involves various complex tasks such as text parsing (for example, tokenization and stemming), morphological analysis, word sense disambiguation, and understanding the underlying grammatical structure of a language. For example, in these two sentences, *I went to the bank* and *I walked along the river bank*, the word *bank* has two entirely different meanings, due to the context it’s used in. To distinguish or (disambiguate) the word *bank*, we need to understand the context in which the word is being used. Machine learning has become a key enabler for NLP, helping to accomplish the aforementioned tasks through machines. Below we discuss some of the important tasks that fall under NLP.

Tasks of Natural Language Processing

NLP has a multitude of real-world applications. A good NLP system is one that performs many NLP tasks. When you search for today's weather on Google or use Google Translate to find out how to say, "*How are you?*" in French, you rely on a subset of such tasks in NLP. We will list some of the most ubiquitous tasks here, and this book covers most of these tasks:

- **Tokenization:** Tokenization is the task of separating a text corpus into atomic units (for example, words or characters). Although it may seem trivial for a language like English, tokenization is an important task. For example, in the Japanese language, words are not delimited by spaces or punctuation marks.
- **Word-Sense Disambiguation (WSD):** WSD is the task of identifying the correct meaning of a word. For example, in the sentences, *The dog barked at the mailman* and *Tree bark is sometimes used as a medicine*, the word *bark* has two different meanings. WSD is critical for tasks such as question answering.
- **Named Entity Recognition (NER):** NER attempts to extract entities (for example, person, location, and organization) from a given body of text or a text corpus. For example, the sentence, *John gave Mary two apples at school on Monday* will be transformed to *[John]name gave [Mary]name [two]number apples at [school]organization on [Monday]time*. NER is an imperative topic in fields such as information retrieval and knowledge representation.
- **Part-of-Speech (PoS) tagging:** PoS tagging is the task of assigning words to their respective parts of speech. It can either be basic tags such as noun, verb, adjective, adverb, and preposition, or it can be granular such as proper noun, common noun, phrasal verb, verb, and so on. The Penn Treebank project, a popular project focusing PoS, defines a comprehensive list of PoS tags at https://www.ling.upenn.edu/courses/ling001/penn_treebank_pos.html.
- **Sentence/synopsis classification:** Sentence or synopsis (for example, movie reviews) classification has many use cases such as spam detection, news article classification (for example, political, technology, and sport), and product review ratings (that is, positive or negative). This is achieved by training a classification model with labeled data (that is, reviews annotated by humans, with either a positive or negative label).
- **Text generation:** In text generation, a learning model (for example, a neural network) is trained with text corpora (a large collection of textual documents), and it then predicts new text that follows. For example, language modeling can output an entirely new science fiction story by using existing science fiction stories for training.

Recently, OpenAI released a language model known as OpenAI-GPT-2, which can generate incredibly realistic text. Furthermore, this task plays a very important role in understanding language, which helps a downstream decision-support model get off the ground quickly.

- **Question Answering (QA):** QA techniques possess a high commercial value, and such techniques are found at the foundation of chatbots and VA (for example, Google Assistant and Apple Siri). Chatbots have been adopted by many companies for customer support. Chatbots can be used to answer and resolve straightforward customer concerns (for example, changing a customer's monthly mobile plan), which can be solved without human intervention. QA touches upon many other aspects of NLP such as information retrieval and knowledge representation. Consequently, all this makes developing a QA system very difficult.
- **Machine Translation (MT):** MT is the task of transforming a sentence/phrase from a source language (for example, German) to a target language (for example, English). This is a very challenging task, as different languages have different syntactical structures, which means that it is not a one-to-one transformation. Furthermore, word-to-word relationships between languages can be one-to-many, one-to-one, many-to-one, or many-to-many. This is known as the **word alignment problem** in MT literature.

Finally, to develop a system that can assist a human in day-to-day tasks (for example, VA or a chatbot) many of these tasks need to be orchestrated in a seamless manner. As we saw in the previous example where the user asks, “*Can you show me a good Italian restaurant nearby?*” several different NLP tasks, such as speech-to-text conversion, semantic and sentiment analyses, question answering, and machine translation, need to be completed. In *Figure 1.1*, we provide a hierarchical taxonomy of different NLP tasks categorized into several different types. It is a difficult task to attribute an NLP task to a single classification. Therefore, you can see some tasks spanning multiple categories. We will split the categories into two main types: language-based (light-colored with black text) and problem formulation-based (dark-colored with white text). The linguistic breakdown has two categories: syntactic (structure-based) and semantic (meaning-based). The problem formulation-based breakdown has three categories: preprocessing tasks (tasks that are performed on text data before feeding to a model), discriminative tasks (tasks where we attempt to assign an input text to one or more categories from a set of predefined categories) and generative tasks (tasks where we attempt to generate a new textual output). Of course, this is one classification among many. But it will show how difficult it is to assign a specific NLP task to a specific category.

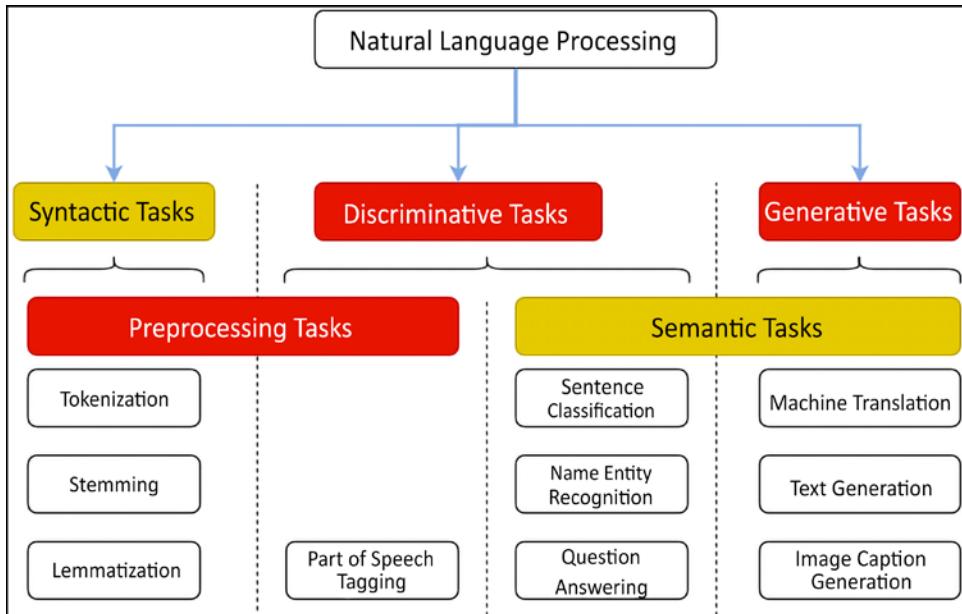


Figure 1.1: A taxonomy of the popular tasks of NLP categorized under broader categories

Having understood the various tasks in NLP, let us now move on to understand how we can solve these tasks with the help of machines. We will discuss both the traditional method and the deep-learning-based approach.

The traditional approach to Natural Language Processing

The traditional or classical approach to solving NLP is a sequential flow of several key steps, and it is a statistical approach. When we take a closer look at a traditional NLP learning model, we will be able to see a set of distinct tasks taking place, such as preprocessing data by removing unwanted data, feature engineering to get good numerical representations of textual data, learning to use machine learning algorithms with the aid of training data, and predicting outputs for novel, unseen data. Of these, feature engineering was the most time-consuming and crucial step for obtaining good performance on a given NLP task.

Understanding the traditional approach

The traditional approach to solving NLP tasks involves a collection of distinct subtasks. First, the text corpora need to be preprocessed, focusing on reducing the vocabulary and *distractions*.

By *distractions*, I refer to the things that distract the algorithm (for example, punctuation marks and stop word removal) from capturing the vital linguistic information required for the task.

Next come several feature engineering steps. The main objective of feature engineering is to make learning easier for the algorithms. Often the features are hand-engineered and biased toward the human understanding of a language. Feature engineering was of the utmost importance for classical NLP algorithms, and consequently, the best-performing systems often had the best-engineered features. For example, for a sentiment classification task, you can represent a sentence with a parse tree and assign positive, negative, or neutral labels to each node/subtree in the tree to classify that sentence as positive or negative. Additionally, the feature engineering phase can use external resources such as WordNet (a lexical database that can provide insights into how different words are related to each other – e.g. synonyms) to develop better features. We will soon look at a simple feature engineering technique known as *bag-of-words*.

Next, the learning algorithm learns to perform well at the given task using the obtained features and, optionally, the external resources. For example, for a text summarization task, a parallel corpus containing common phrases and succinct paraphrases would be a good external resource. Finally, prediction occurs. Prediction is straightforward, where you will feed a new input and obtain the predicted label by forwarding the input through the learning model. The entire process of the traditional approach is depicted in *Figure 1.2*:

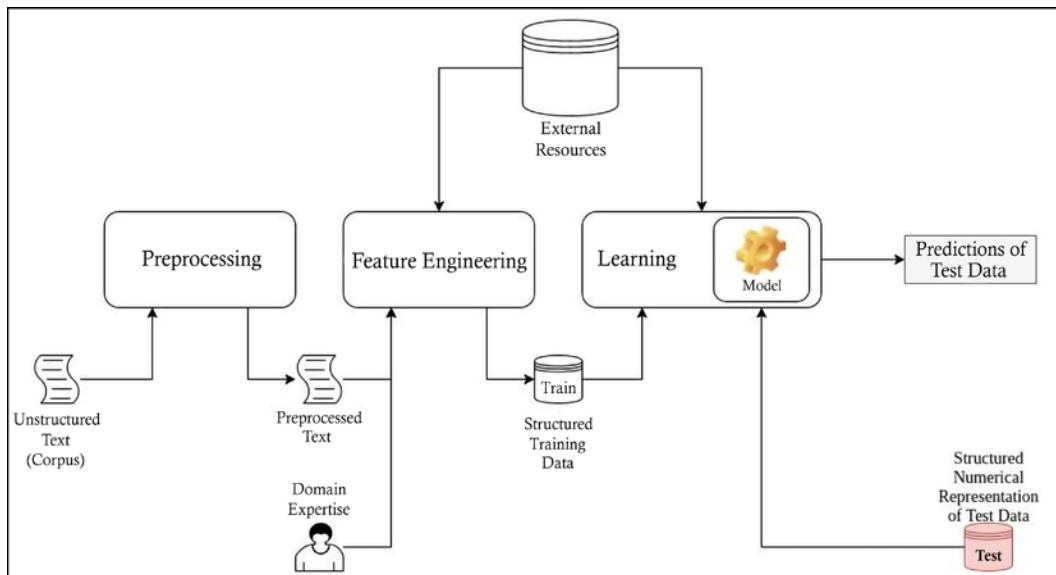


Figure 1.2: The general approach of classical NLP

Next, let's discuss a use case where we use NLP to generate football game summaries.

Example – generating football game summaries

To gain an in-depth understanding of the traditional approach to NLP, let's consider a task of automatic text generation from the statistics of a game of football. We have several sets of game statistics (for example, the score, penalties, and yellow cards) and the corresponding articles generated for that game by a journalist, as the training data. Let's also assume that for a given game, we have a mapping from each statistical parameter to the most relevant phrase of the summary for that parameter. Our task here is that, given a new game, we need to generate a natural-looking summary of the game. Of course, this can be as simple as finding the best-matching statistics for the new game from the training data and retrieving the corresponding summary. However, there are more sophisticated and elegant ways of generating text.

If we were to incorporate machine learning to generate natural language, a sequence of operations, such as preprocessing the text, feature engineering, learning, and prediction, is likely to be performed.

Preprocessing: The text involves operations, such as tokenization (for example, splitting “*I went home*” into “*I*”, “*went*”, “*home*”), stemming (for example, converting *listened* to *listen*), and removing punctuation (for example, ! and ;), in order to reduce the vocabulary (that is, the features), thus reducing the dimensionality of the data. Tokenization might appear trivial for a language such as English, as the words are isolated; however, this is not the case for certain languages such as Thai, Japanese, and Chinese, as these languages are not consistently delimited. Next, it is important to understand that stemming is not a trivial operation either. It might appear that stemming is a simple operation that relies on a simple set of rules such as removing *ed* from a verb (for example, the stemmed result of *listened* is *listen*); however, it requires more than a simple rule base to develop a good stemming algorithm, as stemming certain words can be tricky (for example, using rule-based stemming, the stemmed result of *argued* is *argu*). In addition, the effort required for proper stemming can vary in complexity for other languages.

Feature engineering is used to transform raw text data into an appealing numerical representation so that a model can be trained on that data, for example, converting text into a bag-of-words representation or using n-gram representation, which we will discuss later. However, remember that state-of-the-art classical models rely on much more sophisticated feature engineering techniques.

The following are some of the feature engineering techniques:

Bag-of-words: This is a feature engineering technique that creates feature representations based on the word occurrence frequency. For example, let's consider the following sentences:

- *Bob went to the market to buy some flowers*
- *Bob bought the flowers to give to Mary*

The vocabulary for these two sentences would be:

[“Bob”, “went”, “to”, “the”, “market”, “buy”, “some”, “flowers”, “bought”, “give”, “Mary”]

Next, we will create a feature vector of size V (vocabulary size) for each sentence, showing how many times each word in the vocabulary appears in the sentence. In this example, the feature vectors for the sentences would respectively be as follows:

[1, 1, 2, 1, 1, 1, 1, 0, 0, 0]

[1, 0, 2, 1, 0, 0, 0, 1, 1, 1]

A crucial limitation of the bag-of-words method is that it loses contextual information as the order of words is no longer preserved.

n-gram: This is another feature engineering technique that breaks down text into smaller components consisting of n letters (or words). For example, 2-gram would break the text into two-letter (or two-word) entities. For example, consider this sentence:

Bob went to the market to buy some flowers

The letter level n-gram decomposition for this sentence is as follows:

[“Bo”, “ob”, “b”, “w”, “we”, “en”, ..., “me”, “e”, “f”, “fl”, “lo”, “ow”, “we”, “er”, “rs”]

The word-based n-gram decomposition is this:

[“Bob went”, “went to”, “to the”, “the market”, ..., “to buy”, “buy some”, “some flowers”]

The advantage in this representation (letter level) is that the vocabulary will be significantly smaller than if we were to use words as features for large corpora.

Next, we need to structure our data to be able to feed it into a learning model. For example, we will have data tuples of the form (*a statistic, a phrase explaining the statistic*) as follows:

Total goals = 4, “The game was tied with 2 goals for each team at the end of the first half”

Team 1 = Manchester United, “The game was between Manchester United and Barcelona”

Team 1 goals = 5, “Manchester United managed to get 5 goals”

The learning process may comprise three sub-modules: a **Hidden Markov Model (HMM)**, a sentence planner, and a discourse planner. An HMM is a recurrent model that can be used to solve time-series problems. For example, generating text is a time-series problem as the order of generated words matters. In our example, an HMM might learn to model language (i.e. generate meaningful text) by training on a corpus of statistics and related phrases. We will train the HMM so that it produces a relevant sequence of text, given the statistics as the starting input. Once trained, the HMM can be used for inference in a recursive manner, where we start with a seed (e.g. a statistic) and predict the first word of the description, then use the predicted word to generate the next word, and so on.

Next, we can have a sentence planner that corrects any syntactical or grammatical errors, that might have been introduced by the model. For example, a sentence planner might take in the phrase, *I go house* and output *I go home*. For this, it can use a database of rules, which contains the correct way of conveying meanings, such as the need for a preposition between a verb and the word *house*.

Using the HMM and the sentence planner, we will have syntactically grammatically correct sentences. Then, we need to collate these phrases in such a way that the essay made from the collection of phrases is human readable and flows well. For example, consider the three phrases, *Player 10 of the Barcelona team scored a goal in the second half*, *Barcelona played against Manchester United*, and *Player 3 from Manchester United got a yellow card in the first half*; having these sentences in this order does not make much sense. We like to have them in this order: *Barcelona played against Manchester United*, *Player 3 from Manchester United got a yellow card in the first half*, and *Player 10 of the Barcelona team scored a goal in the second half*. To do this, we use a discourse planner; discourse planners can organize a set of messages so that the meaning of them is conveyed properly.

Now, we can get a set of arbitrary test statistics and obtain an essay explaining the statistics by following the preceding workflow, which is depicted in *Figure 1.3*:

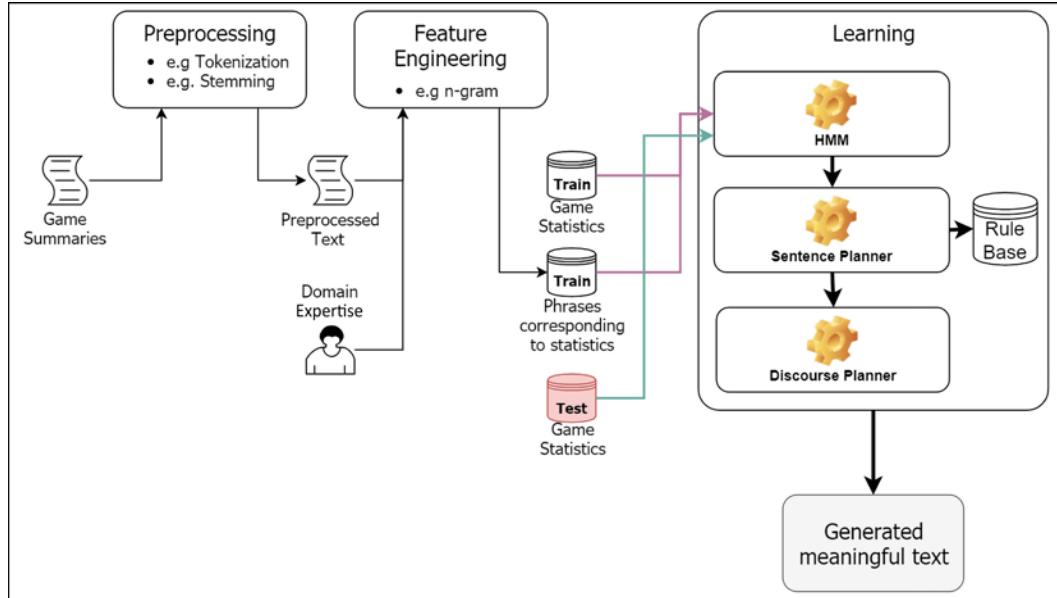


Figure 1.3: The classical approach to solving a language modeling task

Here, it is important to note that this is a very high-level explanation that only covers the main general-purpose components that are most likely to be included in traditional NLP. The details can largely vary according to the particular application we are interested in solving. For example, additional application-specific crucial components might be needed for certain tasks (a rule base and an alignment model in machine translation). However, in this book, we do not stress about such details as the main objective here is to discuss more modern ways of Natural Language Processing.

Drawbacks of the traditional approach

Let's list several key drawbacks of the traditional approach as this would lay a good foundation for discussing the motivation for deep learning:

- The preprocessing steps used in traditional NLP forces a trade-off of potentially useful information embedded in the text (for example, punctuation and tense information) in order to make the learning feasible by reducing the vocabulary. Though preprocessing is still used in modern deep-learning-based solutions, it is not as crucial for them as it is for the traditional NLP workflow due to the large representational capacity of deep networks and their ability to optimize high-end hardware like GPUs.

- Feature engineering is very labor-intensive. In order to design a reliable system, good features need to be devised. This process can be very tedious as different feature spaces need to be extensively explored and evaluated. Additionally, in order to effectively explore robust features, domain expertise is required, which can be scarce and expensive for certain NLP tasks.
- Various external resources are needed for it to perform well, and there are not many freely available ones. Such external resources often consist of manually created information stored in large databases. Creating one for a particular task can take several years, depending on the severity of the task (for example, a machine translation rule base).

Now, let's discuss how deep learning can help to solve NLP problems.

The deep learning approach to Natural Language Processing

I think it is safe to say that deep learning revolutionized machine learning, especially in fields such as computer vision, speech recognition, and of course, NLP. Deep models created a wave of paradigm shifts in many of the fields in machine learning, as deep models learned rich features from raw data instead of using limited human-engineered features. This consequentially caused the pesky and expensive feature engineering to be obsolete. With this, deep models made the traditional workflow more efficient, as deep models perform feature learning and task learning, simultaneously. Moreover, due to the massive number of parameters (that is, weights) in a deep model, it can encompass significantly more features than a human could've engineered. However, deep models are considered a black box due to the poor interpretability of the model. For example, understanding the "how" and "what" features learned by deep models for a given problem is still an active area of research. But it is important to understand that there is a lot more research focusing on "model interpretability of deep learning models".

A deep neural network is essentially an artificial neural network that has an input layer, many interconnected hidden layers in the middle, and finally, an output layer (for example, a classifier or a regressor). As you can see, this forms an end-to-end model from raw data to predictions. These hidden layers in the middle give the power to deep models as they are responsible for learning the good features from raw data, eventually succeeding at the task at hand. Let's now understand the history of deep learning briefly.

History of deep learning

Let's briefly discuss the roots of deep learning and how the field evolved to be a very promising technique for machine learning. In 1960, Hubel and Weisel performed an interesting experiment and discovered that a cat's visual cortex is made of simple and complex cells, and that these cells are organized in a hierarchical form. Also, these cells react differently to different stimuli. For example, simple cells are activated by variously oriented edges while complex cells are insensitive to spatial variations (for example, the orientation of the edge). This kindled the motivation for replicating a similar behavior in machines, giving rise to the concept of artificial neural networks.

In the years that followed, neural networks gained the attention of many researchers. In 1965, a neural network trained by a method known as the **Group Method of Data Handling (GMDH)** and based on the famous *Perceptron* by Rosenblatt, was introduced by Ivakhnenko and others. Later, in 1979, Fukushima introduced the *Neocognitron*, which planted the seeds for one of the most famous variants of deep models—Convolutional Neural Networks (CNNs). Unlike the perceptrons, which always took in a 1D input, a Neocognitron was able to process 2D inputs using convolution operations.

Artificial neural networks used to backpropagate the error signal to optimize the network parameters by computing the gradients of the weights of a given layer with regards to the loss. Then, the weights are updated by pushing them in the opposite direction of the gradient, in order to minimize the loss. For a layer further away from the output layer (i.e. where the loss is computed), the algorithm uses the chain rule to compute gradients. The chain rule used with many layers led to a practical problem known as the vanishing gradients problem, strictly limiting the potential number of layers (depth) of the neural network. The gradients of layers closer to the inputs (i.e. further away from the output layer), being very small, cause the model training to stop prematurely, leading to an underfitted model. This is known as the **vanishing gradients phenomenon**.

Then, in 2006, it was found that *pretraining* a deep neural network by minimizing the *reconstruction error* (obtained by trying to compress the input to a lower dimensionality and then reconstructing it back into the original dimensionality) for each layer of the network provides a good initial starting point for the weight of the neural network; this allows a consistent flow of gradients from the output layer to the input layer. This essentially allowed neural network models to have more layers without the ill effects of the vanishing gradient. Also, these deeper models were able to surpass traditional machine learning models in many tasks, mostly in computer vision (for example, test accuracy for the MNIST handwritten digit dataset). With this breakthrough, deep learning became the buzzword in the machine learning community.

Things started gaining progressive momentum when, in 2012, AlexNet (a deep convolutional neural network created by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton) won the **Large Scale Visual Recognition Challenge (LSVRC)** 2012 with an error decrease of 10% from the previous best. During this time, advances were made in speech recognition, wherein state-of-the-art speech recognition accuracies were reported using deep neural networks. Furthermore, people began to realize that **Graphical Processing Units (GPUs)** enable more parallelism, which allows for faster training of larger and deeper networks compared with **Central Processing Units (CPUs)**.

Deep models were further improved with better model initialization techniques (for example, Xavier initialization), making the time-consuming pretraining redundant. Also, better nonlinear activation functions, such as **Rectified Linear Units (ReLUs)**, were introduced, which alleviated the adversities of the vanishing gradient in deeper models. Better optimization (or learning) techniques, such as the Adam optimizer, automatically tweaked individual learning rates of each parameter among the millions of parameters that we have in the neural network model, which rewrote the state-of-the-art performance in many different fields of machine learning, such as object classification and speech recognition. These advancements also allowed neural network models to have large numbers of hidden layers. The ability to increase the number of hidden layers (that is, to make the neural networks deep) is one of the primary contributors to the significantly better performance of neural network models compared with other machine learning models. Furthermore, better intermediate regularizers, such as batch normalization layers, have improved the performance of deep nets for many tasks.

Later, even deeper models such as ResNets, Highway Nets, and Ladder Nets were introduced, which had hundreds of layers and billions of parameters. It was possible to have such an enormous number of layers with the help of various empirically and theoretically inspired techniques. For example, ResNets use shortcut connections or skip connections to connect layers that are far apart, which minimizes the diminishing of gradients layer to layer, as discussed earlier.

The current state of deep learning and NLP

Many different deep models have seen the light since their inception in early 2000. Even though they share a resemblance, such as all of them using nonlinear transformation of the inputs and parameters, the details can vary vastly. For example, a **CNN** can learn from two-dimensional data (for example, RGB images) as it is, while a multilayer perceptron model requires the input to be unwrapped to a one-dimensional vector, causing the loss of important spatial information.

When processing text, as one of the most intuitive interpretations of text is to perceive it as a sequence of characters, the learning model should be able to do time-series modeling, thus requiring the *memory* of the past. To understand this, think of a language modeling task; the next word for the word *cat* should be different from the next word for the word *climbed*. One such popular model that encompasses this ability is known as a **Recurrent Neural Network (RNN)**. We will see in *Chapter 6, Recurrent Neural Networks*, how exactly RNNs achieve this by going through interactive exercises.

It should be noted that *memory* is not a trivial operation that is inherent to a learning model. Conversely, ways of persisting memory should be carefully designed.

Also, the term *memory* should not be confused with the learned weights of a non-sequential deep network that only looks at the current input, where a sequential model (for example, an RNN) will look at both the learned weights and the previous element of the sequence to predict the next output.

One prominent drawback of RNNs is that they cannot remember more than a few (approximately seven) time steps, thus lacking long-term memory. **Long Short-Term Memory (LSTM)** networks are an extension of RNNs that encapsulate long-term memory. Therefore, often LSTMs are preferred over standard RNNs, nowadays. We will peek under the hood in *Chapter 7, Understanding Long Short-Term Memory Networks*, to understand them better.

Finally, a model known as a **Transformer** has been introduced by Google fairly recently, which has outperformed many of the previous state-of-the-art models such as LSTMs on a plethora of NLP tasks. Previously, both recurrent models (e.g. LSTMs) and convolutional models (e.g. CNNs) dominated the NLP domain. For example, CNNs have been used for sentence classification, machine translation, and sequence-to-sequence learning tasks. However, Transformers use an entirely different approach where they use neither recurrence nor convolution, but an attention mechanism. The attention mechanism allows the model to look at the entire sequence at once, to produce a single output. For example, consider the sentence “*The animal didn’t cross the road because it was tired.*” While generating intermediate representations for the word “*it*,” it would be useful for the model to learn that “*it*” refers to the “*animal*”. The attention mechanism allows the Transformer model to learn such relationships. This capability cannot be replicated with standard recurrent models or convolutional models. We will investigate these models further in *Chapter 10, Transformers* and *Chapter 11, Image Captioning with Transformers*.

In summary, we can mainly separate deep networks into three categories: the non-sequential models that deal with only a single input at a time for both training and prediction (for example, image classification), the sequential models that cope with sequences of inputs of arbitrary length (for example, text generation where a single word is a single input), and finally, attention-based models that look at the sequence at once such as the Transformer, BERT, and XLNet, which are pretrained models based on the Transformer architecture. We can categorize non-sequential (also called feed-forward) models into deep (approximately less than 20 layers) and very deep networks (can be greater than hundreds of layers). The sequential models are categorized into short-term memory models (for example, RNNs), which can only memorize short-term patterns, and long-term memory models, which can memorize longer patterns. In *Figure 1.4*, we outline the discussed taxonomy. You don't have to understand these different deep learning models fully at this point, but it illustrates the diversity of the deep learning models:

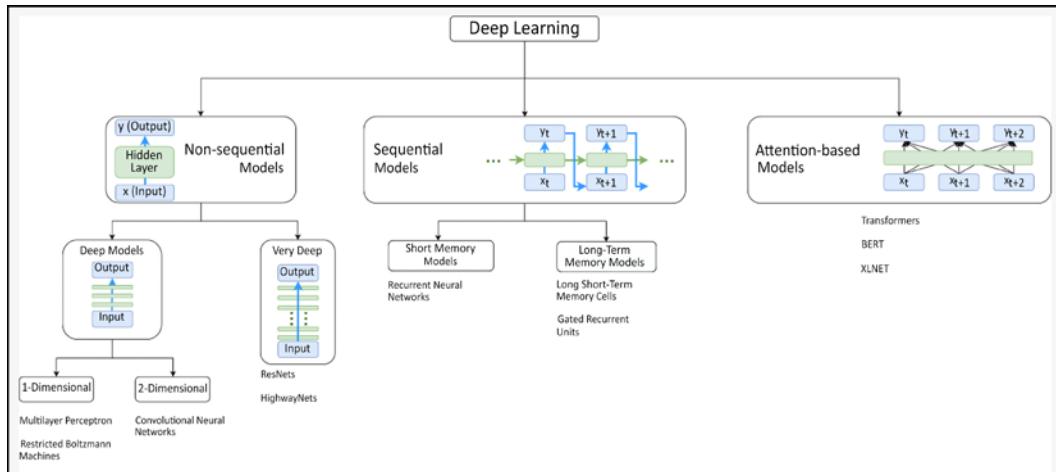


Figure 1.4: A general taxonomy of the most commonly used deep learning methods, categorized into several classes

Now, let's take our first steps toward understanding the inner workings of a neural network.

Understanding a simple deep model – a fully connected neural network

Now, let's have a closer look at a deep neural network in order to gain a better understanding. Although there are numerous different variants of deep models, let's look at one of the earliest models (dating back to 1950–60), known as a **fully connected neural network (FCNN)**, sometimes called a multilayer perceptron. *Figure 1.5* depicts a standard three-layered FCNN.

The goal of an FCNN is to map an input (for example, an image or a sentence) to a certain label or annotation (for example, the object category for images). This is achieved by using an input x to compute h – a hidden representation of x – using a transformation such as $h = \sigma(W * x + b)$; here, W and b are the weights and bias of the FCNN, respectively, and σ is the sigmoid activation function. Neural networks use non-linear activation functions at every layer. Sigmoid activation is one such activation. It is an element-wise transformation applied to the output of a layer, where the sigmoidal output of x is given by, $\sigma(x) = 1/(1 + e^{-x})$. Next, a classifier is placed on top of the FCNN that gives the ability to leverage the learned features in hidden layers to classify inputs. The classifier is a part of the FCNN and yet another hidden layer with some weights, W_s and a bias, b_s . Also, we can calculate the final output of the FCNN as $output = softmax(W_s * h + b_s)$. For example, a softmax classifier can be used for multi-label classification problems. It provides a normalized representation of the scores output by the classifier layer. That is, it will produce a valid probability distribution over the classes in the classifier layer. The label is considered to be the output node with the highest softmax value. Then, with this, we can define a classification loss that is calculated as the difference between the predicted output label and the actual output label. An example of such a loss function is the mean squared loss. You don't have to worry if you don't understand the actual intricacies of the loss function. We will discuss quite a few of them in later chapters. Next, the neural network parameters, W , b , W_s , and b_s , are optimized using a standard stochastic optimizer (for example, the stochastic gradient descent) to reduce the classification loss of all the inputs. *Figure 1.5* depicts the process explained in this paragraph for a three-layer FCNN. We will walk through the details on how to use such a model for NLP tasks, step by step, in *Chapter 3, Word2vec – Learning Word Embeddings*.

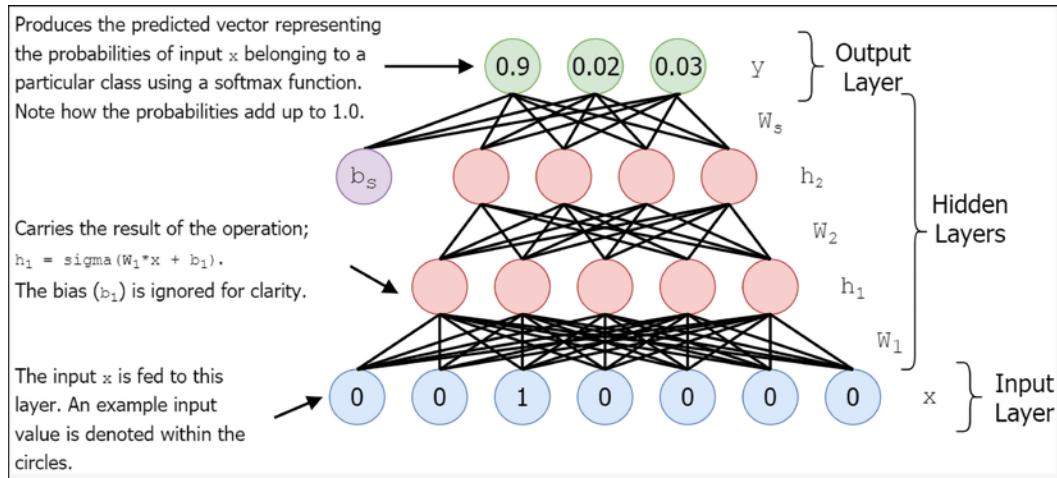


Figure 1.5: An example of a fully connected neural network (FCNN)

Let's look at an example of how to use a neural network for a sentiment analysis task. Consider that we have a dataset where the input is a sentence expressing a positive or negative opinion about a movie and a corresponding label saying if the sentence is actually positive (1) or negative (0). Then, we are given a test dataset, where we have single-sentence movie reviews, and our task is to classify these new sentences as positive or negative.

It is possible to use a neural network (which can be deep or shallow, depending on the difficulty of the task) for this task by adhering to the following workflow:

1. Tokenize the sentence by words.
2. Convert the sentences into a fixed sized numerical representation (for example, Bag-of-Words representation). A fixed sized representation is needed as fully connected neural networks require a fixed sized input.
3. Feed the numerical inputs to the neural network, predict the output (positive or negative), and compare that with the true target.
4. Optimize the neural network using a desired loss function.

In this section we looked at deep learning in more detail. We looked at the history and the current state of NLP. Finally, we looked at a fully connected neural network (a type of deep learning model) in more detail.

Now that we've introduced NLP, its tasks, and how approaches to it have evolved over the years, let's take a moment to look the technical tools required for the rest of this book.

Introduction to the technical tools

In this section, you will be introduced to the technical tools that will be used in the exercises of the following chapters. First, we will present a brief introduction to the main tools provided. Next, we will present a rough guide on how to install each tool along with hyperlinks to detailed guides provided by the official websites. Additionally, we will share tips on how to make sure that the tools were installed properly.

Description of the tools

We will use Python as the coding/scripting language. Python is a very versatile, easy-to-set-up coding language that is heavily used by the scientific and machine learning communities.

Additionally, there are numerous scientific libraries built for Python, catering to areas ranging from deep learning to probabilistic inference to data visualization. TensorFlow is one such library that is well known among the deep learning community, providing many basic and advanced operations that are useful for deep learning. Next, we will use Jupyter Notebook in all our exercises as it provides a rich and interactive environment for coding compared to using Python scripts. We will also use pandas, NumPy and scikit-learn — three popular — two popular libraries for Python—for various miscellaneous purposes such as data preprocessing. Another library we will be using for various text-related operations is NLTK—the Python Natural Language Toolkit. Finally, we will use Matplotlib for data visualization.

Installing Anaconda and Python

Python is hassle-free to install in any of the commonly used operating systems, such as Windows, macOS, or Linux. We will use Anaconda to set up Python, as it does all the laborious work for setting up Python as well as the essential libraries.

To install Anaconda, follow these steps:

1. Download Anaconda from <https://www.continuum.io/downloads>
2. Select the appropriate OS and download Python 3.7
3. Install Anaconda by following the instructions at <https://docs.continuum.io/anaconda/install/>

To check whether Anaconda was properly installed, open a Terminal window (Command Prompt in Windows), and then run the following command:

```
conda --version
```

If installed properly, the version of the current Anaconda distribution should be shown in the Terminal.

Creating a Conda environment

One of the attractive features of Anaconda is that it allows you to create multiple Conda, or virtual, environments. Each Conda environment can have its own environment variables and Python libraries. For example, one Conda environment can be created to run TensorFlow 1.x, whereas another can run TensorFlow 2.x. This is great because it allows you to separate your development environments from any changes taking place in the host's Python installation. Then, you can activate or deactivate Conda environments depending on which environment you want to use.

To create a Conda environment, follow these instructions:

1. Run Conda and create -n packt.nlp.2 python=3.7 in the terminal window using the command `conda create -n packt.nlp.2 python=3.7`.
2. Change directory (`cd`) to the project directory.
3. Activate the new Conda environment by entering `activate packt.nlp.2` in the terminal. If successfully activated, you should see `(packt.nlp.2)` appearing before the user prompt in the terminal.
4. Install the required libraries using one of the following options.
5. If you **have a GPU**, use `pip install -r requirements-base.txt -r requirements-tf-gpu.txt`
6. If you **do not have a GPU**, use `pip install -r requirements-base.txt -r requirements-tf.txt`

Next, we'll discuss some prerequisites for GPU support for TensorFlow.

TensorFlow (GPU) software requirements

If you are using the TensorFlow GPU version, you will need to satisfy certain software requirements such as installing CUDA 11.0. An exhaustive list is available at https://www.tensorflow.org/install/gpu#software_requirements.

Accessing Jupyter Notebook

After running the `pip install` command, you should have Jupyter Notebook available in the Conda environment. To check whether Jupyter Notebook is properly installed and can be accessed, follow these steps:

1. Open a Terminal window.

2. Activate the packt.nlp.2 Conda environment if it is not already by running `activate packt.nlp.2`
3. Run the command: `jupyter notebook`

You should be presented with a new browser window that looks like *Figure 1.6*:



Figure 1.6: Jupyter Notebook installed successfully

Verifying the TensorFlow installation

In this book, we are using TensorFlow 2.7.0. It is important that you install the exact version used in the book as TensorFlow can undergo many changes while migrating from one version to the other. TensorFlow should be installed in the packt.nlp.2 Conda environment if everything went well. If you are having trouble installing TensorFlow, you can find guides and troubleshooting instructions at <https://www.tensorflow.org/install>.

To check whether TensorFlow installed properly, follow these steps:

1. Open Command Prompt in Windows or Terminal in Linux or macOS.
2. Activate the packt.nlp.2 Conda environment.
3. Type `python` to enter the Python prompt. You should now see the Python version right below. Make sure that you are using Python 3.
4. Next, enter the following commands:

```
import tensorflow as tf
print(tf. version )
```

If all went well, you should not have any errors (there might be warnings if your computer does not have a dedicated GPU, but you can ignore them) and TensorFlow version 2.7.0 should be shown.



Many cloud-based computational platforms are also available, where you can set up your own machine with various customization (operating system, GPU card type, number of GPU cards, and so on). Many are migrating to such cloud-based services due to the following benefits:

- More customization options
- Less maintenance effort
- No infrastructure requirements

Several popular cloud-based computational platforms are as follows:

- **Google Colab:** <https://colab.research.google.com/>
- **Google Cloud Platform (GCP):** <https://cloud.google.com/>
- **Amazon Web Services (AWS):** <https://aws.amazon.com/>

Google Colab is a great cloud-based platform that allows you to write TensorFlow code and execute it on CPU/GPU hardware for free.

Summary

In this chapter, we broadly explored NLP to get an impression of the kind of tasks involved in building a good NLP-based system. First, we explained why we need NLP and then discussed various tasks of NLP to generally understand the objective of each task and how difficult it is to succeed at them.

After that, we looked at the classical approach of solving NLP and went into the details of the workflow using an example of generating sport summaries for football games. We saw that the traditional approach usually involves cumbersome and tedious feature engineering. For example, in order to check the correctness of a generated phrase, we might need to generate a parse tree for that phrase. Then, we discussed the paradigm shift that transpired with deep learning and saw how deep learning made the feature engineering step obsolete. We started with a bit of time-traveling to go back to the inception of deep learning and artificial neural networks and worked our way through to the massive modern networks with hundreds of hidden layers. Afterward, we walked through a simple example illustrating a deep model—a multilayer perceptron model—to understand the mathematical wizardry taking place in such a model (on the surface of course!).

With a foundation in both the traditional and modern ways of approaching NLP, we then discussed the roadmap to understand the topics we will be covering in the book, from learning word embeddings to mighty LSTMs, and to state-of-the-art Transformers! Finally, we set up our virtual Conda environment by installing Python, scikit-learn, Jupyter Notebook, and TensorFlow.

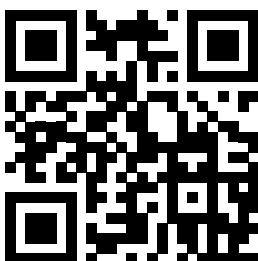
In the next chapter, you will learn the basics of TensorFlow. By the end of the chapter, you should be comfortable with writing a simple algorithm that can take some input, transform the input through a defined function and output the result.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside

more than 1000 members at: <https://packt.link/nlp>



2

Understanding TensorFlow 2

In this chapter, you will get an in-depth understanding of TensorFlow. This is an open source distributed numerical computation framework, and it will be the main platform on which we will be implementing all our exercises. This chapter covers the following topics:

- What is TensorFlow?
- The building blocks of TensorFlow (for example, variables and operations)
- Using Keras for building models
- Implementing our first neural network

We will get started with TensorFlow by defining a simple calculation and trying to compute it using TensorFlow. After we complete this, we will investigate how TensorFlow executes this computation. This will help us to understand how the framework creates a computational graph to compute the outputs and execute this graph to obtain the desired outputs. Then we will dive into the details of how TensorFlow architecture operates by looking at how TensorFlow executes things, with the help of an analogy of how a fancy café works. We will then see how TensorFlow 1 used to work so that we can better appreciate the amazing features TensorFlow 2 offers. Note that when we use the word “TensorFlow” by itself, we are referring to TensorFlow 2. We will specifically mention TensorFlow 1 if we are referring to TensorFlow 1.

Having gained a good conceptual and technical understanding of how TensorFlow operates, we will look at some of the important computations the framework offers. First, we will look at defining various data structures in TensorFlow, such as variables and tensors, and we'll also see how to read inputs through data pipelines. Then we will work through some neural network-related operations (for example, convolution operation, defining losses, and optimization).

Finally, we will apply this knowledge in an exciting exercise, where we will implement a neural network that can recognize images of handwritten digits. You will also see that you can implement or prototype neural networks very quickly and easily by using a high-level submodule such as Keras.

What is TensorFlow?

In *Chapter 1, Introduction to Natural Language Processing*, we briefly discussed what TensorFlow is. Now let's take a closer look at it. TensorFlow is an open source, distributed numerical computation framework released by Google that is mainly intended to alleviate the painful details of implementing a neural network (for example, computing derivatives of the weights of the neural network). TensorFlow takes this a step further by providing efficient implementations of such numerical computations using **Compute Unified Device Architecture (CUDA)**, which is a parallel computational platform introduced by NVIDIA (for more information on CUDA, visit <https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/>). The **Application Programming Interface (API)** of TensorFlow at https://www.tensorflow.org/api_docs/python/tf/all_symbols shows that TensorFlow provides thousands of operations that make our lives easier.

TensorFlow was not developed overnight. This is a result of the persistence of talented, good-hearted developers and scientists who wanted to make a difference by bringing deep learning to a wider audience. If you are interested, you can take a look at the TensorFlow code at <https://github.com/tensorflow/tensorflow>. Currently, TensorFlow has around 3,000 contributors, and it sits on top of more than 115,000 commits, evolving to be better and better every day.

Getting started with TensorFlow 2

Now let's learn about a few essential components in the TensorFlow framework by working through a code example. Let's write an example to perform the following computation, which is very common for neural networks:

$$h = \text{sigmoid}(W \cdot x + b)$$

This computation encompasses what happens in a single layer of a fully connected neural network. Here W and x are matrices and b is a vector. Then, “ \cdot ” denotes the dot product. sigmoid is a non-linear transformation given by the following equation:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

We will discuss how to do this computation through TensorFlow step by step.

First, we will need to import TensorFlow and NumPy. NumPy is another scientific computation framework that provides various mathematical and other operations to manipulate data. Importing them is essential before you run any type of TensorFlow or NumPy-related operation in Python:

```
import tensorflow as tf  
import numpy as np
```

First, we will write a function that can take the inputs x , W , and b and perform this computation for us:

```
def layer(x, W, b):  
    # Building the graph  
    h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to perform  
    return h
```

Next, we add a Python decorator called `tf.function` as follows:

```
@tf.function  
def layer(x, W, b):  
    # Building the graph  
    h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to perform  
    return h
```

Put simply, a Python decorator is just another function. A Python decorator provides a clean way to call another function whenever you call the decorated function. In other words, every time the `layer()` function is called, `tf.function()` is called. This can be used for various purposes, such as:

- Logging the content and operations in a function
- Validating the inputs and outputs of another function

When the `layer()` function is passing through `tf.function()`, TensorFlow will trace the content (in other words, the operations and data) in the function and build a computational graph automatically.

The computational graph (also known as the dataflow graph) builds a DAG (a directed acyclic graph) that shows what kind of inputs are required, and what sort of computations need to be done in the program.

In our example, the `layer()` function produces `h` by using inputs `x`, `W`, and `b`, and some transformations or operations such as `+` and `tf.matmul()`:

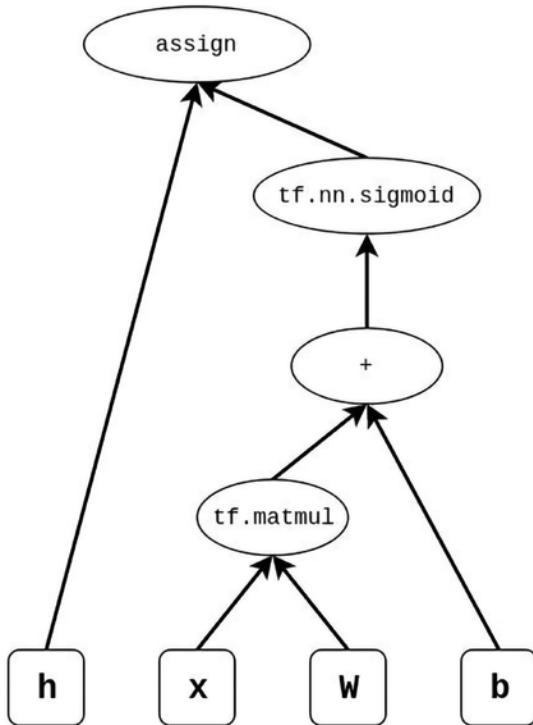


Figure 2.1: A computational graph of the client

If we look at an analogy for a DAG, if you think of the output as a *cake*, then the *graph* would be the recipe to make that cake using *ingredients* (that is, inputs).

The feature that builds this computational graph automatically in TensorFlow is known as **AutoGraph**. AutoGraph is not just looking at the operations in the passed function; it also scrutinizes the flow of operations. This means that you can have `if` statements, or `for/while` loops in your function, and AutoGraph will take care of those when building the graph. You will see more on AutoGraph in the next section.



In TensorFlow 1.x, the user needed to implement the computational graph explicitly. This meant the user could not write typical Python code using `if-else` statements or `for` loops, but had to explicitly control the flow of operations using special bespoke TensorFlow operations such as `tf.cond()` and `tf.control_dependencies()`. This is because, unlike TensorFlow 2.x, TensorFlow 1.x did not immediately execute operations when you called them. Rather, after they were defined, they needed to be executed explicitly using the context of a TensorFlow Session. For example, when you run the following in TensorFlow 1,

```
h = tf.nn.sigmoid(tf.matmul(x,W) + b)
```

`h` will not have any value until `h` is executed in the context of a `Session`. Therefore, `h` could not be treated like any other Python variable. Don't worry if you don't understand how the `Session` works. It will be discussed in the coming sections.

Next, you can use this function right away, as follows:

```
x = np.array([[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]],  
dtype=np.float32)
```

Here, `x` is a simple NumPy array:

```
init_w = tf.initializers.RandomUniform(minval=-0.1, maxval=0.1)  
(shape=[10,5])  
W = tf.Variable(init_w, dtype=tf.float32, name='W')  
  
init_b = tf.initializers.RandomUniform()(shape=[5])  
b = tf.Variable(init_b, dtype=tf.float32, name='b')
```

`W` and `b` are TensorFlow variables defined using the `tf.Variable` object. `W` and `b` hold tensors. A tensor is essentially an n -dimensional array. For example, a one-dimensional vector or a two-dimensional matrix are called **tensors**. A `tf.Variable` is a mutable structure, which means the values in the tensor stored in that variable can change over time. For example, variables are used to store neural network weights, which change during the model optimization.

Also, note that for `W` and `b`, we provide some important arguments, such as the following:

```
init_w = tf.initializers.RandomUniform(minval=-0.1, maxval=0.1)
      (shape=[10,5])
init_b = tf.initializers.RandomUniform()(shape=[5])
```

These are called variable initializers and are the tensors that will be assigned to the `W` and `b` variables initially. A variable must have an initial value provided. Here, `tf.initializers.RandomUniform` means that we uniformly sample values between `minval` (-0.1) and `maxval` (0.1) to assign values to the tensors. There are many different initializers provided in TensorFlow (https://www.tensorflow.org/api_docs/python/tf/keras/initializers). It is also very important to define the `shape` of your initializer when you are defining the initializer itself. The `shape` property defines the size of each dimension of the output tensor. For example, if `shape` is [10, 5], this means that it will be a two-dimensional structure and will have 10 elements on axis 0 (rows) and 5 elements on axis 1 (columns):

```
h = layer(x,W,b)
```

Finally, `h` is called a TensorFlow tensor in general. A TensorFlow tensor is an immutable structure. Once a value is assigned to a TensorFlow tensor, it cannot be changed.

As you can see, the term “tensor” is used in two ways:

- To refer to an n -dimensional array
- To refer to an immutable data structure in TensorFlow

For both, the underlying concept is the same as they hold an n -dimensional data structure, only differing in the context they are used. The term will be used interchangeably to refer to these structures in our discussion.

Finally, you can immediately see the value of `h` using,

```
print(f" h = {h.numpy()}" )
```

which will give,

```
h = [[0.7027744 0.687556 0.635395 0.6193934 0.6113584]]
```

The `numpy()` function retrieves the NumPy array from the TensorFlow Tensor object. The full code is as below. All the code examples in this chapter will be available in the `tensorflow_introduction.ipynb` file in the `ch2` folder:

```
@tf.function
def layer(x, W, b):
    # Building the graph
    h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to be performed
    return h

x = np.array([[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]], 
             dtype=np.float32)

# Variable
init_w = tf.initializers.RandomUniform(minval=-0.1, maxval=0.1)
(shape=[10,5])
W = tf.Variable(init_w, dtype=tf.float32, name='W')
# Variable
init_b = tf.initializers.RandomUniform()(shape=[5])
b = tf.Variable(init_b, dtype=tf.float32, name='b')

h = layer(x,W,b)
print(f"h = {h.numpy()}")
```

For future reference, let's call our example *the sigmoid example*.

As you can already see, defining a TensorFlow computational graph and executing that is very “Pythonic”. This is because TensorFlow executes its operations “eagerly”, or immediately after the `layer()` function is called. This is a special mode in TensorFlow known as *eager execution* mode. This was an optional mode for TensorFlow 1, but has been made the default in TensorFlow 2.

Also note that the next two sections will be somewhat complex and technical. However, don't worry if you don't understand everything completely because the explanation will be supplemented with a more digestible, and thorough, real-world example that explains how an order is fulfilled in our new-and-improved restaurant, *Café Le TensorFlow 2*.

TensorFlow 2 architecture – What happens during graph build?

Let's now understand what TensorFlow does when you execute TensorFlow operations.

When you call a function decorated by `tf.function()`, such as the `layer()` function, there is quite a bit happening in the background. First, TensorFlow will trace all the TensorFlow operations taking place in the function and build the computational graph automatically.

In fact, `tf.function()` will return a function that executes the built dataflow graph when invoked. Therefore, `tf.function()` is a multi-stage process, where it first builds the dataflow graph and then executes it. Additionally, since TensorFlow traces each line in the function, if something goes wrong, TensorFlow can point to the exact line that is causing the issue.

In our sigmoid example, the computational, or dataflow, graph would look like *Figure 2.2*. A single element or vertex of the graph is called a **node**. There are two main types of objects in this graph: *operations* and *tensors*. In the preceding example, `tf.nn.sigmoid` is an operation and `h` is a tensor:

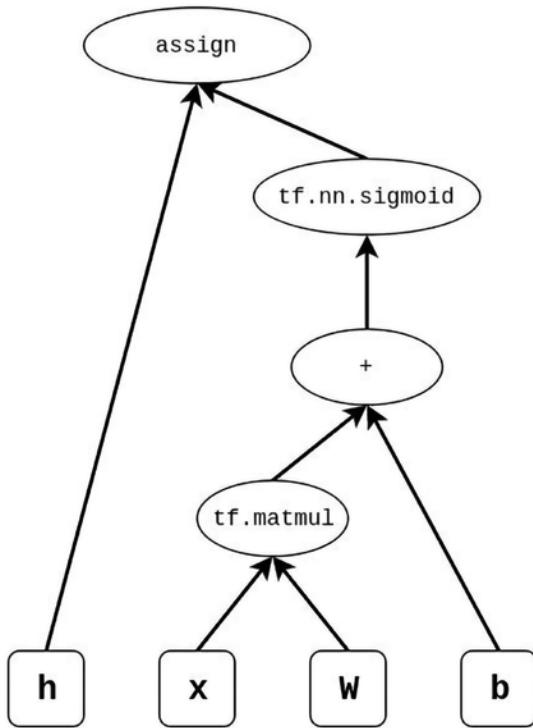


Figure 2.2: A computational graph of the client

The preceding graph shows the order of operations as well as how inputs flow through them.



Keep in mind that `tf.function()` or AutoGraph is not a silver bullet that turns any arbitrary Python function using TensorFlow operations into a computational graph; it has its limitations. For example, the current version cannot handle recursive calls. To see a full list of the eager mode capabilities, refer to the following link: <https://github.com/sourcecode369/tensorflow-1/blob/master/tensorflow/python/autograph/g3doc/reference/limitations.md>.

Now we know that TensorFlow is skilled at creating a nice computational graph, with all the dependencies and operations so that it knows exactly how, when, and where the data flows. However, we did not quite answer how this graph is executed. In fact, TensorFlow does quite a bit behind the scenes. For example, the graph might be divided into subgraphs, and subsequently into even finer pieces, to achieve parallelization. These subgraphs or pieces will then be assigned to workers that will perform the assigned task.

TensorFlow architecture – what happens when you execute the graph?

The computational graph uses the `tf.GraphDef` protocol to canonicalize the dataflow graph and send it to the distributed master. The distributed master would perform the actual operation execution and parameter updates in a single-process setting. In a distributed setting, the master would delegate these tasks to worker processes/devices and manage these worker processes. `tf.GraphDef` is a standardized representation of the graph specific to TensorFlow. The distributed master sees all computations in the graph and divides the computations into different devices (for example, different GPUs and CPUs). TensorFlow operations have multiple kernels. A kernel is a device-specific implementation of a certain operation. For example, the `tf.matmul()` function will be implemented differently to run on the CPU or GPU since, on a GPU, you can achieve much better performance due to more parallelization.

Next, the computational graph will be broken into subgraphs and pruned by the distributed master. Although decomposing the computational graph in *Figure 2.2* appears too trivial in our example, the computational graph can exponentially grow in real-world solutions with many hidden layers. Additionally, it becomes important to break the computational graph into multiple pieces and shave off any redundant computations in order to get results faster (for example, in a multi-device setting).

Executing the graph or a subgraph (if the graph is divided into subgraphs) is called a single *task*, where each task is allocated to a single worker (which could be a single process or an entire device). These workers can run as a single process in a multi-process device (for example, a multi-processing CPU), or run on different devices (for example, CPUs and GPUs). In a distributed setting, we would have multiple workers executing tasks (for example, multiple workers training the model on different batches of data). On the contrary, we have only one set of parameters. So how do multiple workers manage to update the same set of parameters?

To solve this, there is one worker that is considered the parameter server and will hold the main copy of the parameters. The workers will copy the parameters over, update them, and send them back to the parameter server. Typically, the parameter server will define some resolution strategy to resolve multiple updates coming from multiple workers (for example, taking the mean). These details were provided so you can understand the complexity that has gone into TensorFlow. However, our book will be based on using TensorFlow in a single-process/worker setting. In this setting, the organization of the distributed master, workers, and the parameter server is much more straightforward and is absorbed mostly by a special session implementation used by TensorFlow. This general workflow of a TensorFlow client is depicted in *Figure 2.3*:

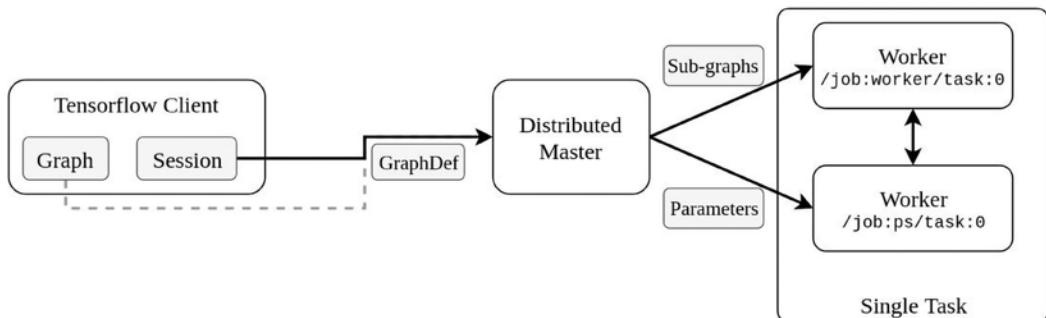


Figure 2.3: The generic execution of a TensorFlow client. A TensorFlow client starts with a graph that gets sent to the distributed master. The master spins up worker processes to perform actual tasks and parameter updates

Once the calculation is done, the session brings back the updated data to the client from the parameter server. The architecture of TensorFlow is shown in *Figure 2.4*:

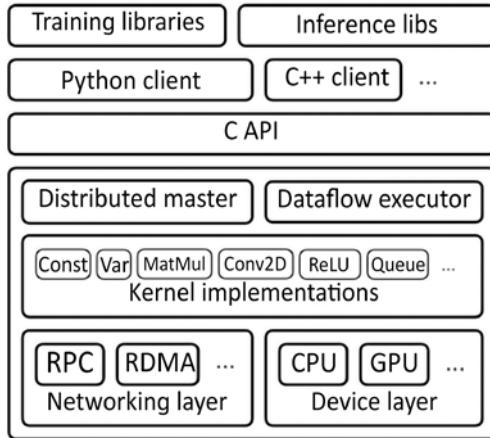


Figure 2.4: TensorFlow framework architecture. This explanation is based on the official TensorFlow documentation found at: <https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/architecture.md>

Most of the changes introduced in TensorFlow 2 can be attributed to front-end changes. That is, how the dataflow graph is built and when the graph is executed. The way the graph is executed remains more or less the same in TensorFlow 1 and 2.

Now we know what happens end-to-end from the moment you execute `tf.function()`, but this was a very technical explanation, and nothing explains something better than a good analogy. Therefore, we will try to understand TensorFlow 2 with an analogy to our new and improved Café Le TensorFlow 2.

Café Le TensorFlow 2 – understanding TensorFlow 2 with an analogy

Let's say the owners renovated our previous Café Le TensorFlow (this is an analogy from the first edition) and reopened it as Café Le TensorFlow 2. The word around the town is that it's much more opulent than it used to be. Remembering the great experience you had before, you book a table instantly and go there to grab a seat.

You want to order a *chicken burger with extra cheese and no tomatoes*. And you realize the café is indeed fancy. There're no waiters here, but a voice-enabled tablet for each table into which you say what you want. This will get converted to a standard format that the chefs will understand (for example, table number, menu item ID, quantity, and special requirements).

Here, you represent the TensorFlow 2 program. The ability of the voice-enabled tablet that converts your voice (or TensorFlow operations) to the standard format (or GraphDef format) is analogous to the AutoGraph feature.

Now comes the best part. As soon as you start speaking, a manager will be looking at your order and assigning various tasks to chefs. The manager is responsible for making sure things happen as quickly as possible. The kitchen manager makes decisions, such as how many chefs are required to make the dish and which chefs are the best candidates for the job. The kitchen manager represents the distributed master.

Each chef has a cook whose responsibility it is to provide the chef with the right ingredients, equipment, and so forth. So, the kitchen manager takes the order to a single chef and a cook (a burger is not that hard to prepare) and asks them to prepare the dish. The chef looks at the order and tells the cook what is needed. So, the cook first finds the things that will be required (for example, buns, patties, and onions) and keeps them close to fulfill the chef's requests as soon as possible. Moreover, the chef might also ask to keep the intermediate results (for example, cut vegetables) of the dish temporarily until the chef needs it back again. In our example, the chef is the operation executor, and the cook is the parameter server.

This café is full of surprises. As you are speaking out your order (that is, invoking Python functions that have TensorFlow operations), you see it getting prepared in real time through the tablet on your table (that is, eager execution).

The best thing about this video feed is that, if you see that the chef did not put enough cheese, you know exactly why the burger wasn't as good as expected. So, you can either order another one or provide specific feedback. This is a great improvement over how TensorFlow 1 did things, where they would take your order and you would not see anything until the full burger had been prepared. This process is shown in *Figure 2.5*:

Cafe Le TensorFlow 2

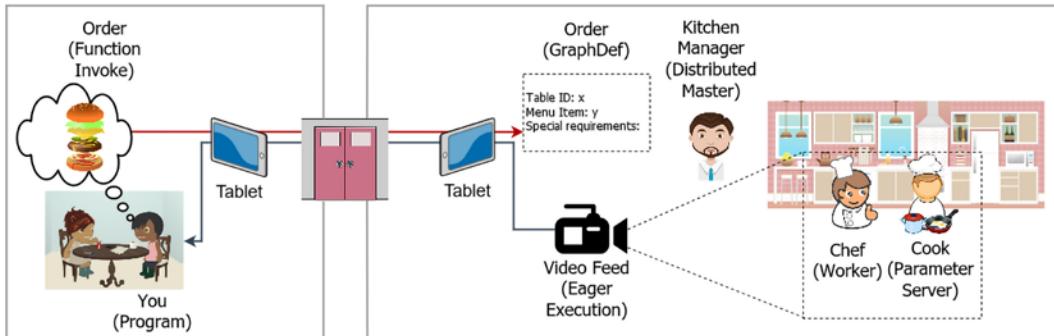


Figure 2.5: The restaurant analogy illustrated

Let's now have a look back at how TensorFlow 1 used to work.

Flashback: TensorFlow 1

We said numerous times that TensorFlow 2 is very different from TensorFlow 1. But we still don't know what it used to be like. Therefore, let's now do a bit of time traveling to see how the same sigmoid computation could have been implemented in TensorFlow 1.



Warning

You will not be able to execute the following code in TensorFlow 2.x as it stands.

First, we'll define a `graph` object, which we will populate with operations and variables later:

```
graph = tf.Graph() # Creates a graph
session = tf.InteractiveSession(graph=graph) # Creates a session
```

The `graph` object contains the computational graph that connects the various inputs and outputs we define in our program to get the final desired output. This is the same graph we discussed earlier. Also, we'll define a `session` object that takes the defined graph as the input, which executes the graph. In other words, compared to TensorFlow 2, the `graph` object and the `session` object do what happens when you invoke them decorated by `tf.function()`.

Now we'll define a few tensors, namely `x`, `W`, `b`, and `h`. There are several different ways that you can define tensors in TensorFlow 1. Here, we will look at three such different approaches:

- First, `x` is a placeholder. Placeholders, as the name suggests, are not initialized with any value. Rather, we will provide the value on the fly at the time of the graph execution. If you remember from the TensorFlow 2 sigmoid exercise, we fed `x` (which was a NumPy array) directly to the function `layer(x, w, b)`. Unlike in TensorFlow 2, you cannot feed NumPy arrays directly to TensorFlow 1 graphs or operations.
- Next, we have the variables `W` and `b`. Variables are defined similarly to TensorFlow 2 with some minor changes in the syntax.
- Finally, we have `h`, which is an immutable tensor produced by performing some operations on `x`, `W`, and `b`. Note that you will not see the value of `h` immediately as you needed to manually execute the graph in TensorFlow 1.

These tensors are defined as follows:

```
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1,
dtype=tf.float32),name='W')
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b') h = tf.nn.
sigmoid(tf.matmul(x,W) + b)
```



The lifetime of variables in TensorFlow 1 was managed by the session object, meaning that variables lived in memory for as long as the session lived (even after losing references to them in the code). However, in TensorFlow 2, variables are removed soon after the variables are not referenced in the code, just like in Python.

Next, we'll run an initialization operation that initializes the variables in the graph, `W` and `b`:

```
tf.global_variables_initializer().run()
```

Now, we will execute the graph to obtain the final output we need, `h`. This is done by running `session.run(...)`, where we provide the value to the placeholder as an argument of the `session.run()` command:

```
h_eval = session.run(h,feed_dict={x: np.random.rand(1,10)})
```

Finally, we close the session, releasing any resources held by the `session` object:

```
    session.close()
```

Here is the full code of this TensorFlow 1 example:

```
import tensorflow as tf
import numpy as np

# Defining the graph and session
graph = tf.Graph() # Creates a graph
session = tf.InteractiveSession(graph=graph) # Creates a session

# Building the graph
# A placeholder is an symbolic input
x = tf.placeholder(shape=[1,10], dtype=tf.float32, name='x')

# Variable
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1,
                                    dtype=tf.float32), name='W')
b = tf.Variable(tf.zeros(shape=[5]), dtype=tf.float32, name='b')

h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to be performed

# Executing operations and evaluating nodes in the graph
tf.global_variables_initializer().run() # Initialize the variables

# Run the operation by providing a value to the symbolic input x
h_eval = session.run(h, feed_dict={x: np.random.rand(1,10)})

# Closes the session to free any held resources by the session
session.close()
```

As you can see, before TensorFlow 2 the user had to:

- Define the computational graph using various TensorFlow data structures (for example, `tf.placeholder`) and operations (for example, `tf.matmul()`)
- Execute the required part of the graph using `session.run()` to fetch the results by feeding the correct data into the session

In conclusion, TensorFlow 1.x had several limitations:

- Coding with TensorFlow 1 did not provide the same intuitive “Pythonic” feeling as you needed to define the computational graph first and then invoke the execution of it. This is known as declarative programming.
- The design in TensorFlow 1 made it very hard to break the code down into manageable functions as the user needed to define the graph fully, before doing any computations. This resulted in very large functions or pieces of code containing very large computational graphs.
- It was very difficult to do real-time debugging of the code as TensorFlow had its own runtime that used `session.run()`.
- But also, it was not without some advantages, such as the efficiency brought about by declaring the full computational graph upfront. Knowing all the computations in advance meant TensorFlow 1 could perform all sorts of optimizations (for example, graph pruning) to run the graph efficiently.

In this part of the chapter, we discussed our first example in TensorFlow2 and the architecture of TensorFlow. Finally, we compared and contrasted TensorFlow 1 and 2. Next, we will discuss the various building blocks of TensorFlow 2.

Inputs, variables, outputs, and operations

Now we are returning from our journey into TensorFlow 1 and stepping back to TensorFlow 2. Let’s proceed to the most common elements that comprise a TensorFlow 2 program. If you read any of the millions of TensorFlow clients available on the internet, the TensorFlow-related code all falls into one of these buckets:

- **Inputs:** Data used to train and test our algorithms
- **Variables:** Mutable tensors, mostly defining the parameters of our algorithms
- **Outputs:** Immutable tensors storing both terminal and intermediate outputs
- **Operations:** Various transformations for inputs to produce the desired outputs

In our earlier sigmoid example, we can find instances of all these categories. We list the respective TensorFlow elements and the notation used in the sigmoid example in *Table 2.1*:

TensorFlow element	Value from example client
Inputs	x
Variables	W and b
Outputs	h
Operations	<code>tf.matmul(...), tf.nn.sigmoid(...)</code>

Table 2.1: The different types of TensorFlow primitives we have encountered so far

The following subsections explain each of these TensorFlow elements listed in the table in more detail.

Defining inputs in TensorFlow

There are three different ways you can feed data to a TensorFlow program:

- Feeding data as NumPy arrays
- Feeding data as TensorFlow tensors
- Using the `tf.data` API to create an input pipeline

Next, we will discuss a few different ways you can feed data to TensorFlow operations.

Feeding data as NumPy arrays

This is the simplest way to feed data into a TensorFlow program. Here, you pass a NumPy array as an input to the TensorFlow operation and the result is executed immediately. This is exactly what we did in the sigmoid example. If you look at `x`, it is a NumPy array.

Feeding data as tensors

The second method is like the first one, but the type of data is different. Here, we are defining `x` as a TensorFlow tensor.

To see this in action, let's modify our sigmoid example. Remember that we defined `x` as:

```
x = np.array([[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]]),
dtype=np.float32)
```

Instead, let's define this as a tensor that contains specific values:

```
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]],
dtype=tf.float32,name='x')
```

Also, the full code would become as follows:

```
import tensorflow as tf

@tf.function
def layer(x, W, b):
    # Building the graph
    h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to be performed
    return h

# A pre-loaded input
x = tf.constant(value=[[0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]],
                dtype=tf.float32, name='x')

# Variable
init_w = tf.initializers.RandomUniform(minval=-0.1, maxval=0.1)
(shape=[10,5])
W = tf.Variable(init_w, dtype=tf.float32, name='W')
# Variable
init_b = tf.initializers.RandomUniform()(shape=[5])
b = tf.Variable(init_b, dtype=tf.float32, name='b')

h = layer(x,W,b)
print(f"h = {h}")
print(f"h is of type {type(h)}")
```

Let's now discuss how we can define data pipelines in TensorFlow.

Building a data pipeline using the `tf.data` API

`tf.data` provides you with a convenient way to build data pipelines in TensorFlow. Input pipelines are designed for more heavy-duty programs that need to process a lot of data. For example, if you have a small dataset (for example, the MNIST dataset) that fits into the memory, input pipelines would be excessive. However, when working with complex data or problems, where you might need to work with large datasets that do not fit in memory, augment the data (for example, for adjusting image contrast/brightness), numerically transform it (for example, standardize), and so on. The `tf.data` API provides convenient functions that can be used to easily load and transform your data. Furthermore, it streamlines your data ingestion code with the model training.

Additionally, the `tf.data` API offers various options to enhance the performance of your data pipeline, such as multi-processing and pre-fetching data. Pre-fetching refers to bringing data into the memory before it's required and keeping it ready. We will discuss these methods in more detail as they are used in the upcoming chapters.

When creating an input pipeline, we intend to perform the following:

- Source the data from a data source (for example, an in-memory NumPy array, CSV file on disk, or individual files such as images).
- Apply various transformations to the data (for example, cropping/resizing image data).
- Iterate the resulting dataset element/batch-wise. Batching is required as deep learning models are trained on randomly sampled batches of data. As the datasets these models are trained on are large, they typically do not fit in memory.

Let's write an input pipeline using TensorFlow's `tf.data` API. In this example, we have three text files (`iris.data.1`, `iris.data.2`, and `iris.data.3`) in CSV format, each file having 50 lines and each line having 4 floating-point numbers (in other words, various lengths associated with a flower) and a string label separated by commas (an example line would be `5.6,2.9,3.6,1.3,Iris-versicolor`). We will now use the `tf.data` API to read data from these files. We also know that some of this data is corrupted (as with any real-life machine learning project). In our case, some data points have negative lengths. So, let's first write a pipeline to go through the data row by row and print the corrupted outputs.



For more information, refer to the official TensorFlow page on importing data at <https://www.tensorflow.org/guide/data>.

First, let's import a few important libraries as before:

```
import tensorflow as tf  
import numpy as np
```

Next, we will define a list containing the filenames:

```
filenames = [f"./iris.data.{i}" for i in range(1,4)]
```

Now we will use one of the dataset readers provided in TensorFlow. The dataset reader takes in a list of filenames and another list that specifies the data types of each column in the dataset. As we saw previously, we have four floating numbers and one string:

```
dataset = tf.data.experimental.CsvDataset(filenames, [tf.float32,  
tf.float32, tf.float32, tf.float32, tf.string])
```

Now we will organize our data into inputs and labels as follows:

```
dataset = dataset.map(lambda x1,x2,x3,x4,y: (tf.stack([x1,x2,x3,x4]), y))
```

We are using lambda functions to separate out x_1, x_2, x_3, x_4 into one dataset and y to another dataset, along with the `dataset.map()` function.



Lambda functions are a special type of function that allow you to define some computations succinctly. With lambda functions, you don't need to name your function, which can be quite handy if you are using a certain function only once in your code. The format of the lambda function looks like:

```
lambda <arguments>: <result returned after the computation>
```

For example, if you need to write a function that adds two numbers, simply write:

```
lambda x, y: x+y
```

Here, `tf.stack()` stacks individual tensors (here, the individual feature) to a single tensor. When using the `map` function, you first need to visualize what needs to be done to a single item in the dataset (a single item in our case is a single row from the dataset), and write the transformation.



The map function is very simple but powerful. All it does is transform a set of given inputs into a new set of values. For example, if you have a list, xx , that contains a list of numbers and want to convert them to power 2 element-wise, you can write something like $xx_pow = \text{map}(\text{lambda } x: x^{**2}, xx)$. And this can be very easily parallelized as there's no dependency between items.

Next, you can iterate through this dataset, examining individual data points, as you would iterate through a normal Python list. Here, we are printing out all the corrupted items:

```
for next_element in dataset:  
    x, y = next_element[0].numpy(), next_element[1].numpy().  
    decode('ascii')  
    if np.min(x)<0.0:  
        print(f"(corrupted) X => {x}\tY => {y}")
```

Since you don't want those corrupted inputs in your dataset, you can use the `dataset.filter()` function to filter out those corrupted entries as follows:

```
dataset = dataset.filter(lambda x,y: tf.reduce_min(x)>0)
```

Here we are checking whether the minimum element in `x` is greater than zero; if not, those elements will be filtered out of the dataset.

Another useful function is `dataset.batch()`. When training deep neural networks, we often traverse the dataset in batches, not individual items. `dataset.batch()` provides a convenient way to do that:

```
batch_size = 5  
dataset = dataset.batch(batch_size=batch_size)
```

Now, if you print the shape of a single element in your dataset, you should get the following:

```
x.shape = (5, 4), y.shape = (5,)
```

Now that we have examined the three different methods you can use to define inputs in TensorFlow, let's see how we can define variables in TensorFlow.

Defining variables in TensorFlow

Variables play an important role in TensorFlow. A variable is essentially a tensor with a specific shape defining how many dimensions the variable will have and the size of each dimension. However, unlike a regular TensorFlow tensor, variables are *mutable*; meaning that the value of the variables can change after they are defined. This is an ideal property to have to implement the parameters of a learning model (for example, neural network weights), where the weights change slightly after each step of learning. For example, if you define a variable with `x = tf.Variable(0,dtype=tf.int32)`, you can change the value of that variable using a TensorFlow operation such as `tf.assign(x,x+1)`. However, if you define a tensor such as `x = tf.constant(0,dtype=tf.int32)`, you cannot change the value of the tensor, as you could for a variable. It should stay 0 until the end of the program execution.

Variable creation is quite simple. In our sigmoid example, we already created two variables, `W` and `b`. When creating a variable, a few things are extremely important. We will list them here and discuss each in detail in the following paragraphs:

- Variable shape
- Initial values

- Data type
- Name (optional)

The variable shape is a list of the `[x, y, z, ...]` format. Each value in the list indicates how large the corresponding dimension or axis is. For instance, if you require a 2D tensor with 50 rows and 10 columns as the variable, the shape would be equal to `[50, 10]`.

The dimensionality of the variable (that is, the length of the shape vector) is recognized as the rank of the tensor in TensorFlow. Do not confuse this with the rank of a matrix.



Tensor rank in TensorFlow indicates the dimensionality of the tensor; for a two-dimensional matrix, `rank = 2`.

Next, a variable requires an *initial* value to be initialized with. TensorFlow provides several different initializers for our convenience, including constant initializers and normal distribution initializers. Here are a few popular TensorFlow initializers you can use to initialize variables:

- `tf.initializers.Zeros`
- `tf.initializers.Constant`
- `tf.initializers.RandomNormal`
- `tf.initializers.GlorotUniform`

The shape of the variable can be provided as a part of the initializer as follows:

```
tf.initializers.RandomUniform(minval=-0.1, maxval=0.1)(shape=[10, 5])
```

The data type plays an important role in determining the size of a variable. There are many different data types, including the commonly used `tf.bool`, `tf.uint8`, `tf.float32`, and `tf.int32`. Each data type has a number of bits required to represent a single value with that type. For example, `tf.uint8` requires 8 bits, whereas `tf.float32` requires 32 bits. It is common practice to use the same data types for computations, as doing otherwise can lead to data type mismatches. So, if you have two different data types for two tensors that you need to transform, you have to explicitly convert one tensor to the other tensor's type using the `tf.cast(...)` operation.

The `tf.cast(...)` operation is designed to cope with such situations. For example, if you have an `x` variable with the `tf.int32` type, which needs to be converted to `tf.float32`, employ `tf.cast(x, dtype=tf.float32)` to convert `x` to `tf.float32`.

Finally, the *name* of the variable will be used as an ID to identify that variable in the graph. If you ever visualize the computational graph, the variable will appear by the argument passed to the `name` keyword. If you do not specify a name, TensorFlow will use the default naming scheme.



Note that the Python variable `tf.Variable` is assigned to is not known by the computational graph, and is not a part of TensorFlow variable naming. Consider this example where you specify a TensorFlow variable as follows:

```
a = tf.Variable(tf.zeros([5]), name='b')
```

Here, the TensorFlow graph will know this variable by the name `b` and not `a`.

Moving on, let's talk about how to define TensorFlow outputs.

Defining outputs in TensorFlow

TensorFlow outputs are usually tensors, and the result of a transformation to either an input, or a variable, or both. In our example, `h` is an output, where `h = tf.nn.sigmoid(tf.matmul(x,W) + b)`. It is also possible to give such outputs to other operations, forming a chained set of operations. Furthermore, they do not necessarily have to be TensorFlow operations. You also can use standard Python arithmetic with TensorFlow. Here is an example:

```
x = tf.matmul(w,A)
y = x + B
```

Below, we explain various operations available in TensorFlow and how to use them.

Defining operations in TensorFlow

An operation in TensorFlow takes one or more inputs and produces one or more outputs. If you take a look at the TensorFlow API at https://www.tensorflow.org/api_docs/python/tf, you will see that TensorFlow has a massive collection of operations available. Here, we will take a look at a selected few of the myriad TensorFlow operations.

Comparison operations

Comparison operations are useful for comparing two tensors. The following code example includes a few useful comparison operations.

To understand the working of these operations, let's consider two example tensors, x and y :

```
# Let's assume the following values for x and y
# x (2-D tensor) => [[1,2],[3,4]]
# y (2-D tensor) => [[4,3],[3,2]]
x = tf.constant([[1,2],[3,4]], dtype=tf.int32)
y = tf.constant([[4,3],[3,2]], dtype=tf.int32)

# Checks if two tensors are equal element-wise and returns a boolean
# tensor
# x_equal_y => [[False,False],[True,False]]
x_equal_y = tf.equal(x, y, name=None)

# Checks if x is less than y element-wise and returns a boolean tensor
# x_less_y => [[True,True],[False,False]]
x_less_y = tf.less(x, y, name=None)

# Checks if x is greater or equal than y element-wise and returns a
# boolean tensor
# x_great_equal_y => [[False,False],[True,True]]
x_great_equal_y = tf.greater_equal(x, y, name=None)

# Selects elements from x and y depending on whether, # the condition
# is satisfied (select elements from x) # or the condition failed (select
# elements from y)
condition = tf.constant([[True,False],[True,False]],dtype=tf.bool)
# x_cond_y => [[1,3],[3,2]]
x_cond_y = tf.where(condition, x, y, name=None)
```

Next, let's look at some mathematical operations.

Mathematical operations

TensorFlow allows you to perform math operations on tensors that range from the simple to the complex. We will discuss a few of the mathematical operations made available in TensorFlow. The complete set of operations is available at https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/math:

```
# Let's assume the following values for x and y
# x (2-D tensor) => [[1,2],[3,4]]
```

```
# y (2-D tensor) => [[4,3],[3,2]]
x = tf.constant([[1,2],[3,4]], dtype=tf.float32)
y = tf.constant([[4,3],[3,2]], dtype=tf.float32)

# Add two tensors x and y in an element-wise fashion
# x_add_y => [[5,5],[6,6]]
x_add_y = tf.add(x, y)

# Performs matrix multiplication (not element-wise)
# x_mul_y => [[10,7],[24,17]]
x_mul_y = tf.matmul(x, y)

# Compute natural logarithm of x element-wise # equivalent to computing
ln(x)
# log_x => [[0,0.6931],[1.0986,1.3863]]
log_x = tf.log(x)

# Performs reduction operation across the specified axis
# x_sum_1 => [3,7]
x_sum_1 = tf.reduce_sum(x, axis=[1], keepdims=False)

# x_sum_2 => [[4,6]]
x_sum_2 = tf.reduce_sum(x, axis=[0], keepdims=True)

# Segments the tensor according to segment_ids (items with same id in
# the same segment) and computes a segmented sum of the data
data = tf.constant([1,2,3,4,5,6,7,8,9,10], dtype=tf.float32)
segment_ids = tf.constant([0,0,0,1,1,2,2,2,2,2 ], dtype=tf.int32)
# x_seg_sum => [6,9,40]
x_seg_sum = tf.segment_sum(data, segment_ids)
```

Now, we will look at the scatter operation.

Updating (scattering) values in tensors

A scatter operation, which refers to changing the values at certain indices of a tensor, is very common in scientific computing problems. This functionality was originally provided through an intimidating `tf.scatter_nd()` function, which can be difficult to understand.

However, in recent TensorFlow versions, you can perform scatter operations via array indexing and slicing using NumPy-like syntax. Let's see a few examples. Say you have the TensorFlow variable `v`, which is a [3,2] matrix:

```
v = tf.Variable(tf.constant([[1,9],[3,10],[5,11]]),  
    dtype=tf.float32), name='ref')
```

You can change the 0th row of this tensor with:

```
v[0].assign([-1, -9])
```

which results in:

```
<tf.Variable 'ref:0' shape=(3, 2) dtype=float32, numpy=  
array([[-1., -9.],  
      [ 3., 10.],  
      [ 5., 11.]], dtype=float32)>
```

You can change the value at index [1,1] with:

```
v[1,1].assign(-10)
```

which results in:

```
<tf.Variable 'ref:0' shape=(3, 2) dtype=float32, numpy=  
array([[ 1.,   9.],  
      [ 3., -10.],  
      [ 5.,  11.]], dtype=float32)>
```

You can perform row slicing with:

```
v[1:,0].assign([-3,-5])
```

which results in:

```
<tf.Variable 'ref:0' shape=(3, 2) dtype=float32, numpy=  
array([[ 1.,   9.],  
      [-3., 10.],  
      [-5., 11.]], dtype=float32)>
```



It is important to remember that the scatter operation (performed via the `assign()` operation) can only be performed on `tf.Variables`, which are mutable structures. Remember that `tf.Tensor`/`tf.EagerTensor` are immutable objects.

Collecting (gathering) values from a tensor

A gather operation is very similar to a scatter operation. Remember that scattering is about assigning values to tensors, whereas gathering retrieves the values of a tensor. Let's understand this through an example. Say you have a TensorFlow tensor, t:

```
t = tf.constant([[1,9],[3,10],[5,11]],dtype=tf.float32)
```

You can obtain the 0th row of t with:

```
t[0].numpy()
```

which will return:

```
[1. 9.]
```

You can also perform row-slicing with:

```
t[1:,:0].numpy()
```

which will return:

```
[3. 5.]
```

Unlike the scatter operation, the gather operation works both on `tf.Variable` and `tf.Tensor` structures.

Neural network-related operations

Now, let's look at several useful neural network-related operations that we will use heavily in the following chapters. The operations we will discuss here range from simple element-wise transformations (that is, activations) to computing partial derivatives of a set of parameters with respect to another value. We will also implement a simple neural network as an exercise.

Nonlinear activations used by neural networks

Nonlinear activations enable neural networks to perform well at numerous tasks. Typically, there is a nonlinear activation transformation (that is, activation layer) after each layer output in a neural network (except for the last layer). A nonlinear transformation helps a neural network to learn various nonlinear patterns that are present in data. This is very useful for complex real-world problems, where data often has more complex nonlinear patterns, in contrast to linear patterns. If not for the nonlinear activations between layers, a deep neural network would be a bunch of linear layers stacked on top of each other. Also, a set of linear layers can essentially be compressed to a single bigger linear layer.

In conclusion, if not for the nonlinear activations, we cannot create a neural network with more than one layer.

Let's observe the importance of nonlinear activation through an example. First, recall the computation for the neural networks we saw in the sigmoid example. If we disregard b , it will be this:

$$h = \text{sigmoid}(W*x)$$



Assume a three-layer neural network (having $W1$, $W2$, and $W3$ as layer weights) where each layer does the preceding computation; we can summarize the full computation as follows:

$$h = \text{sigmoid}(W3 * \text{sigmoid}(W2 * \text{sigmoid}(W1 * x)))$$

However, if we remove the nonlinear activation (that is, sigmoid), we get this:

$$h = (W3 * (W2 * (W1 * x))) = (W3 * W2 * W1) * x$$

So, without the nonlinear activations, the three layers can be brought down to a single linear layer.

Now we'll list two commonly used nonlinear activations in neural networks (in other words, sigmoid and ReLU) and how they can be implemented in TensorFlow:

```
# Sigmoid activation of x is given by 1 / (1 + exp(-x))
tf.nn.sigmoid(x, name=None)
```

```
# ReLU activation of x is given by max(0, x)
tf.nn.relu(x, name=None)
```

The functional form of these computations is visualized in *Figure 2.6*:

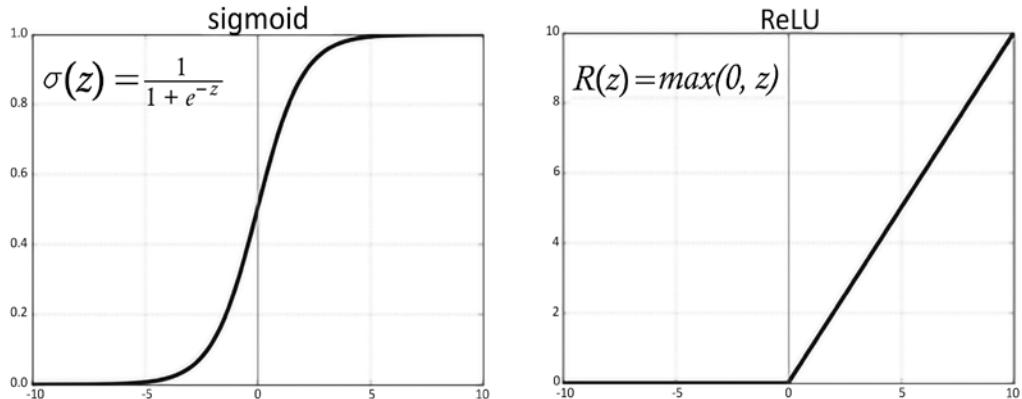


Figure 2.6: The functional forms of sigmoid (left) and ReLU (right) activations

Next, we will discuss the convolution operation.

The convolution operation

A convolution operation is a widely used signal-processing technique. For images, convolution is used to produce different effects (such as blurring), or extract features (such as edges) from an image. An example of edge detection using convolution is shown in *Figure 2.7*. This is achieved by shifting a convolution filter on top of an image to produce a different output at each location (see *Figure 2.8* later in this section). Specifically, at each location, we do element-wise multiplication of the elements in the convolution filter with the image patch (the same size as the convolution filter) that overlaps with the convolution filter and takes the sum of the multiplication:

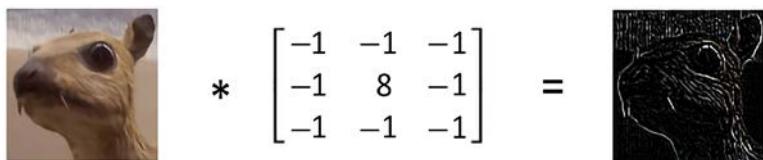


Figure 2.7: Using the convolution operation for edge detection in an image (Source: [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)))

The following is the implementation of the convolution operation:

```
x = tf.constant(  
    [[  
        [[1],[2],[3],[4]],  
        [[4],[3],[2],[1]],  
        [[5],[6],[7],[8]],  
        [[8],[7],[6],[5]]  
    ]],  
    dtype=tf.float32)  
  
x_filter = tf.constant(  
    [ [ [[0.5]],[[1]] ],  
      [ [[0.5]],[[1]] ]  
    ],  
    dtype=tf.float32)  
  
x_stride = [1,1,1,1]  
x_padding = 'VALID'  
  
x_conv = tf.nn.conv2d(  
    input=x, filters=x_filter, strides=x_stride, padding=x_padding  
)
```

Here, the apparently excessive number of square brackets used might make you think that the example can be made easy to follow by getting rid of these redundant brackets. Unfortunately, that is not the case. For the `tf.nn.conv2d(...)` operation, TensorFlow requires `input`, `filters`, and `strides` to be of an exact format. We will now go through each argument in `tf.conv2d(input, filters, strides, padding)` in more detail:

- **input:** This is typically a 4D tensor where the dimensions should be ordered as [`batch_size, height, width, channels`]:
 - **batch_size:** This is the amount of data (for example, inputs such as images, and words) in a single batch of data. We normally process data in batches as large datasets are used for learning. At a given training step, we randomly sample a small batch of data that approximately represents the full dataset. And doing this for many steps allows us to approximate the full dataset quite well. This `batch_size` parameter is the same as the one we discussed in the TensorFlow input pipeline example.
 - **height and width:** This is the height and the width of the input.
 - **channels:** This is the depth of an input (for example, for an RGB image, the number of channels will be 3—a channel for each color).
- **filters:** This is a 4D tensor that represents the convolution window of the convolution operation. The filter dimensions should be [`height, width, in_channels, out_channels`]:
 - **height and width:** This is the height and the width of the filter (often smaller than that of the input)
 - **in_channels:** This is the number of channels of the input to the layer
 - **out_channels:** This is the number of channels to be produced in the output of the layer
- **strides:** This is a list with four elements, where the elements are [`batch_stride, height_stride, width_stride, channels_stride`]. The `strides` argument denotes how many elements to skip during a single shift of the convolution window on the input. Usually, you don't have to worry about `batch_stride` and `channels_stride`. If you do not completely understand what `strides` is, you can use the default value of 1.
- **padding:** This can be one of `['SAME', 'VALID']`. It decides how to handle the convolution operation near the boundaries of the input. The `VALID` operation performs the convolution without padding. If we were to convolve an input of n length with a convolution window of size h , this will result in an output of size $(n-h+1 < n)$. The diminishing of the output size can severely limit the depth of neural networks. `SAME` pads zeros to the boundary such that the output will have the same height and width as the input.

To gain a better understanding of what filter size, stride, and padding are, refer to *Figure 2.8*:

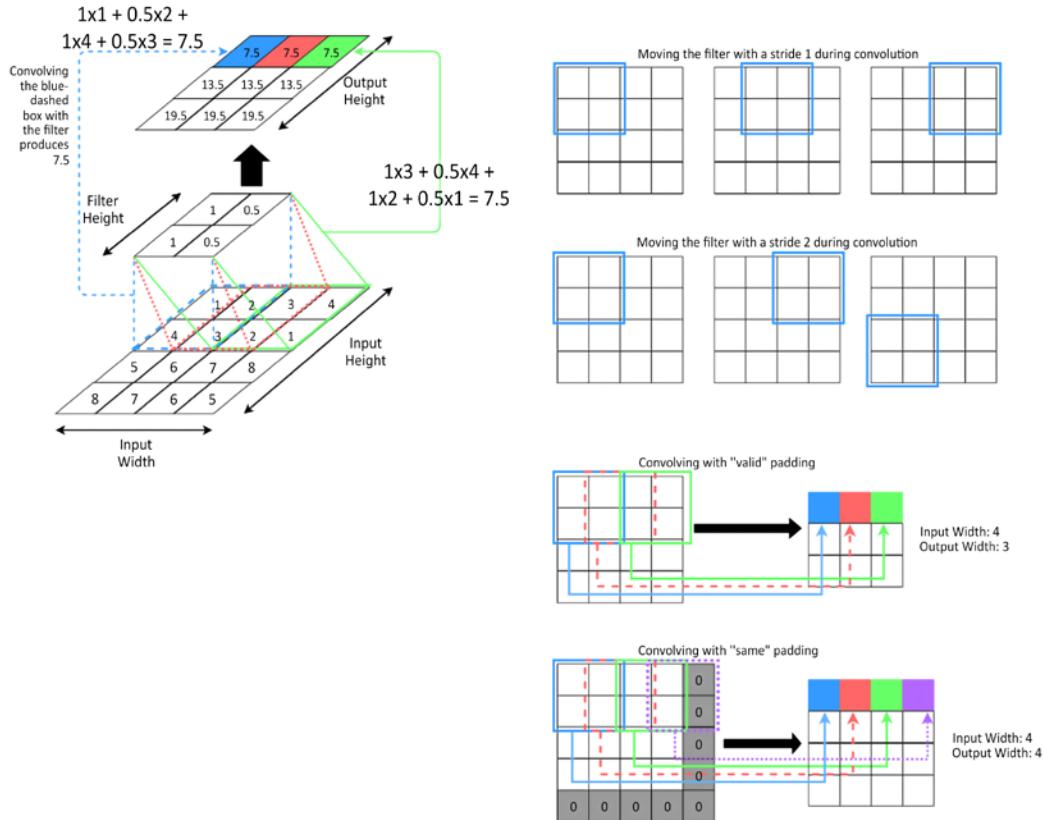


Figure 2.8: The convolution operation. Note how the kernel is moved over the input to compute values at each position

Next, we will discuss the pooling operation.

The pooling operation

A pooling operation behaves similarly to the convolution operation, but the final output is different. Instead of outputting the sum of the element-wise multiplication of the filter and the image patch, we now take the maximum element of the image patch for that location (see *Figure 2.9*):

```
x = tf.constant(
    [
        [[1],[2],[3],[4]],
        [[4],[3],[2],[1]],
        [[5],[6],[7],[8]],
    ])
```

```

[[[8],[7],[6],[5]]]
]],
dtype=tf.float32)

x_ksize = [1,2,2,1]
x_stride = [1,2,2,1]
x_padding = 'VALID'

x_pool = tf.nn.max_pool2d(
    input=x, ksize=x_ksize,
    strides=x_stride, padding=x_padding
)
# Returns (out) => [[[4.],[4.],[8.],[8.]]]

```

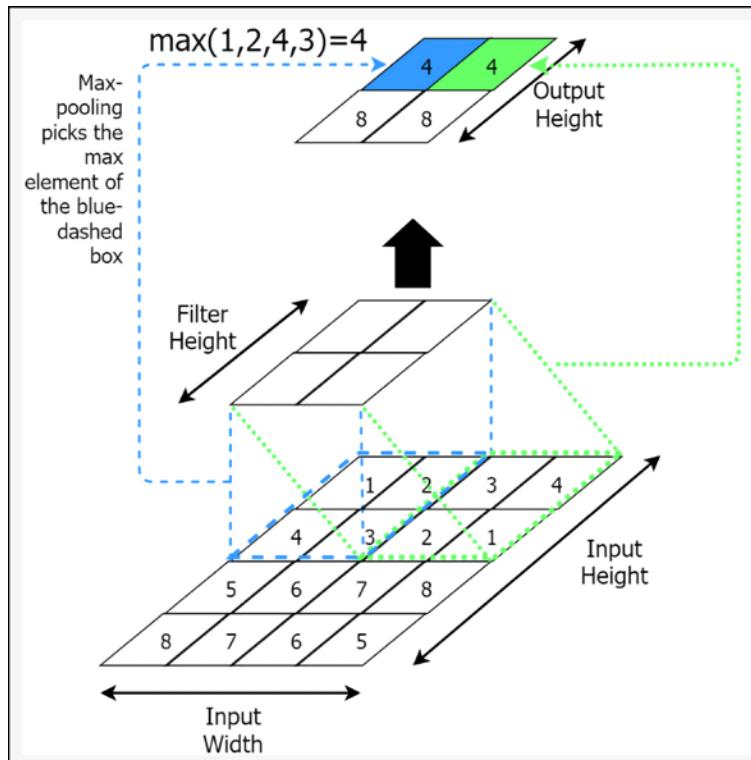


Figure 2.9: The max-pooling operation

Defining loss

We know that, for a neural network to learn something useful, a loss needs to be defined. The loss represents how close or far away the predictions are from actual targets. There are several functions for automatically calculating the loss in TensorFlow, two of which are shown in the following code. The `tf.nn.l2_loss` function is the mean squared error loss, and `tf.nn.softmax_cross_entropy_with_logits` is another type of loss that actually gives better performance in classification tasks. And by logits here, we mean the unnormalized output of the neural network (that is, the linear output of the last layer of the neural network):

```
# Returns half of L2 norm of t given by sum(t**2)/2
x = tf.constant([[2,4],[6,8]],dtype=tf.float32)
x_hat = tf.constant([[1,2],[3,4]],dtype=tf.float32)
# MSE = (1**2 + 2**2 + 3**2 + 4**2)/2 = 15
MSE = tf.nn.l2_loss(x-x_hat)

# A common Loss function used in neural networks to optimize the network
# Calculating the cross_entropy with logits (unnormalized outputs of the
# last layer)
# instead of probabilistic outputs leads to better numerical stabilities

y = tf.constant([[1,0],[0,1]],dtype=tf.float32)
y_hat = tf.constant([[3,1],[2,5]],dtype=tf.float32)
# This function alone doesn't average the cross entropy losses of all data
# points,
# You need to do that manually using reduce_mean function
CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_
hat,labels=y))
```

Here, we discussed several important operations intertwined with neural networks, such as the convolution operation and the pooling operation. We will now discuss how a sub-library in TensorFlow known as Keras can be used to build models.

Keras: The model building API of TensorFlow

Keras was developed as a separate library that provides high-level building blocks to build models conveniently. It was initially platform-agnostic and supported many softwares (for example, TensorFlow and Theano).

However, TensorFlow acquired Keras and now is an integral part of TensorFlow for building models effortlessly.

Keras's primary focus is model building. For that, Keras provides several different APIs with varying degrees of flexibility and complexity. Choosing the right API for the job will require sound knowledge of the limitations of each API as well as experience. The APIs provided by Keras are:

- Sequential API – The most easy-to-use API. In this API, you simply stack layers on top of each other to create a model.
- Functional API – The functional API provides more flexibility by allowing you to define custom models that can have multiple input layers/multiple output layers.
- Sub-classing API – The sub-classing API enables you to define custom reusable layers/models as Python classes. This is the most flexible API, but it requires strong familiarity with the API and raw TensorFlow operations to use it correctly.



Do not confuse the Keras TensorFlow sub-module (https://www.tensorflow.org/api_docs/python/tf/keras) with the external Keras library (<https://keras.io/>). They share roots in terms of where they've come from, but they are not the same. You will run into strange issues if you treat them as the same during your development. In this book, we exclusively use `tf.keras`.

One of the most innate concepts in Keras is that a model is composed of one or more layers connected in a specific way. Here, we will briefly go through what the code looks like, using different APIs to develop models. You are not expected to fully understand the code below. Rather, focus on the code style to spot any differences between the three methods.

Sequential API

When using the Sequential API, you simply define your model as a list of layers. Here, the first element in the list is the closest to the input, where the last is the output layer:

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(500, activation='relu', shape=(784, )),
    tf.keras.layers.Dense(250, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

In the preceding code, we have three layers. The first layer has 500 output nodes and takes in a vector of 784 elements as the input. The second layer is automatically connected to the first one, whereas the last layer is connected to the second layer. All of these layers are fully-connected layers, where all input nodes are connected to all output nodes.

Functional API

In the Functional API, we do things differently. We first define one or more input layers, and other layers that carry computations. Then we connect the inputs to outputs ourselves, as shown in the following code:

```
inp = tf.keras.layers.Input(shape=(784,))
out_1 = tf.keras.layers.Dense(500, activation='relu')(inp)
out_2 = tf.keras.layers.Dense(250, activation='relu')(out_1)
out = tf.keras.layers.Dense(10, activation='softmax')(out_2)

model = tf.keras.models.Model(inputs=inp, outputs=out)
```

In the code, we start with an input layer that accepts a 784 element-long vector. The input is passed to a Dense layer that has 500 nodes. The output of that layer is assigned to `out_1`. Then `out_1` is passed to another Dense layer, which outputs `out_2`. Next, a Dense layer with 10 nodes outputs the final output. Finally, the model is defined as a `tf.keras.models.Model` object that takes two arguments:

- `inputs` – One or more input layers
- `outputs` – One or more outputs produced by any `tf.keras.layers` type object

The model is identical to what was defined in the previous section. One of the benefits of the Functional API is that you can create far more complex models as you're not bounded to have layers as a list. Because of this freedom, you can have multiple inputs connecting to many layers in many different ways and potentially produce many outputs as well.

Sub-classing API

Finally, we will use the sub-classing API to define a model. With sub-classing, you define your model as a Python object that inherits from the base object, `tf.keras.Model`. When using sub-classing, you need to define two important functions: `__init__()`, which will specify any special parameters, layers, and so on required to successfully perform the computations, and `call()`, which defines the computations that need to happen in the model:

```
class MyModel(tf.keras.Model):

    def __init__(self, num_classes):
        super().__init__()
        self.hidden1_layer = tf.keras.layers.Dense(500, activation='relu')
        self.hidden2_layer = tf.keras.layers.Dense(250, activation='relu')
        self.final_layer = tf.keras.layers.Dense(num_classes,
                                                activation='softmax')

    def call(self, inputs):
        h = self.hidden1_layer(inputs)
        h = self.hidden2_layer(h)
        y = self.final_layer(h)
        return y

model = MyModel(num_classes=10)
```

Here, you can see that our model has three layers, just like all the previous models we defined. Next, the `call` function defines how these layers connect to produce the final output. The sub-classing API is considered the most difficult to master, mainly due to the freedom allowed by the method. However, the rewards are immense once you learn the API as it enables you to define very complex models/layers as unit computations that can be reused later. Now that you understand how each API works, let's implement a neural network using Keras and train it on a dataset.

Implementing our first neural network

Great! Now that you've learned the architecture and foundations of TensorFlow, it's high time that we move on and implement something slightly more complex. Let's implement a neural network. Specifically, we will implement a fully connected neural network model (FCNN), which we discussed in *Chapter 1, Introduction to Natural Language Processing*.

One of the stepping stones to the introduction of neural networks is to implement a neural network that is able to classify digits. For this task, we will be using the famous MNIST dataset made available at <http://yann.lecun.com/exdb/mnist/>.

You might feel a bit skeptical regarding our using a computer vision task rather than an NLP task. However, vision tasks can be implemented with less preprocessing and are easy to understand.

As this is our first encounter with neural networks, we will see how to implement this model using Keras. Keras is the high-level submodule that provides a layer of abstraction over TensorFlow. Therefore, you can implement neural networks with much less effort with Keras than using TensorFlow's raw operations. To run the examples end to end, you can find the full exercise in the `tensorflow_introduction.ipynb` file in the Ch02-Understanding-TensorFlow folder. The next step is to prepare the data.

Preparing the data

First, we need to download the dataset. TensorFlow out of the box provides convenient functions to download data and MNIST is one of those supported datasets. We will be performing four important steps during the data preparation:

- Downloading the data and storing it as `numpy.ndarray` objects. We will create a folder named `data` within our `ch2` directory and store the data there.
- Reshaping the images so that 2D grayscale images in the dataset will be converted to 1D vectors.
- Standardizing the images to have a zero-mean and unit-variance (also known as **whitening**).
- One-hot encoding the integer class labels. One-hot encoding refers to the process of representing integer class labels as a vector. For example, if you have 10 classes and a class label of 3 (where labels range from 0-9), your one-hot encoded vector will be `[0, 0, 0, 1, 0, 0, 0, 0, 0]`.

The following code performs these functions for us:

```
os.makedirs('data', exist_ok=True)

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data(
    path=os.path.join(os.getcwd(), 'data', 'mnist.npz')
)

# Reshaping x_train and x_test tensors so that each image is represented
# as a 1D vector
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)

# Standardizing x_train and x_test tensors
x_train = (
```

```
x_train - np.mean(x_train, axis=1, keepdims=True)
)/np.std(x_train, axis=1, keepdims=True)
x_test = (
    x_test - np.mean(x_test, axis=1, keepdims=True)
)/np.std(x_test, axis=1, keepdims=True)

# One hot encoding y_train and y_test
y_onehot_train = np.zeros((y_train.shape[0], num_labels),
dtype=np.float32)
y_onehot_train[np.arange(y_train.shape[0]), y_train] = 1.0

y_onehot_test = np.zeros((y_test.shape[0], num_labels), dtype=np.float32)
y_onehot_test[np.arange(y_test.shape[0]), y_test] = 1.0
```

You can see that we are using the `tf.keras.datasets.mnist.load_data()` function provided by TensorFlow to download the training and testing data. It will be downloaded to a folder named `data` within the `Ch02-Understanding-TensorFlow` folder. This will provide four output tensors:

- `x_train` – A 60000 x 28 x 28 sized tensor where each image is 28 x 28
- `y_train` – A 60000 sized vector, where each element is a class label between 0-9
- `x_test` – A 10000 x 28 x 28 sized tensor
- `y_test` – A 10000 sized vector

Once the data is downloaded, we reshape the 28 x 28 sized images into a 1D vector. This is because we will be implementing a fully connected neural network. Fully connected neural networks take a 1D vector as the input. Therefore, all the pixels in the image will be arranged as a sequence of pixels in order to feed into the model. Finally, if you look at the range of values present in the `x_train` and `x_test` tensors, they will be in the range of 0-255 (typical grayscale range). We would bring these values to a zero mean unit-variance range by subtracting the mean of each image and dividing by the standard deviation.

Implementing the neural network with Keras

Let's now examine how to implement the type of neural network we discussed in *Chapter 1, Introduction to Natural Language Processing*, with Keras. The network is a fully connected neural network with 3 layers having 500, 250, and 10 nodes, respectively. The first two layers will use ReLU activation, whereas the last layer uses softmax. To implement this, we are going to use the simplest of the Keras APIs available to us – the Sequential API.

You can find the full exercise in the `tensorflow_introduction.ipynb` file in the Ch02-Understanding-TensorFlow folder:

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(500, activation='relu'),
    tf.keras.layers.Dense(250, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

You can see that all it takes is a single line in the Keras Sequential API to define the model we just defined. Keras provides various types of layers. You can see the full list of layers available to you at https://www.tensorflow.org/api_docs/python/tf/keras/layers. For a fully connected network, we only need Dense layers that mimic the computations of a hidden layer in a fully connected network. With the model defined, you need to compile this model with an appropriate loss function, an optimizer, and, optionally, performance metrics:

```
optimizer = tf.keras.optimizers.RMSprop()
loss_fn = tf.keras.losses.CategoricalCrossentropy()
model.compile(optimizer=optimizer, loss=loss_fn, metrics=['acc'])
```

With the model defined and compiled, we can now train our model on the prepared data.

Training the model

Training a model could not be easier in Keras. Once the data is prepared, all you need to do is call the `model.fit()` function with the required arguments:

```
batch_size = 100
num_epochs = 10

train_history = model.fit(
    x=x_train,
    y=y_onehot_train,
    batch_size=batch_size,
    epochs= num_epochs,
    validation_split=0.2
)
```

`model.fit()` accepts several important arguments. We will go through them in more detail here:

- `x` – An input tensor. In our case, this is a 60000 x 784 sized tensor.

- y – The one-hot encoded label tensor. In our case, this is a 60000×10 sized tensor.
- `batch_size` – Deep learning models are trained with batches of data (in other words, stochastically) as opposed to feeding the full dataset at once. The batch size defines how many examples are included in a single batch. The larger the batch size, the better the accuracy of your model would be generally.
- `epochs` – Deep learning models iterate through the dataset in batches several times. The number of times iterated through the dataset is known as the number of epochs. In our example, this is set to 10.
- `validation_split` – When training deep learning models, a validation set is used to monitor performance, where the validation set acts as a proxy for real-world performance. `validation_split` defines how much of the full dataset is to be used as the validation subset. In our example, this is set to 20% of the total dataset size.

Here's what the training loss and validation accuracy look like over the number of epochs we trained the model (*Figure 2.10*):

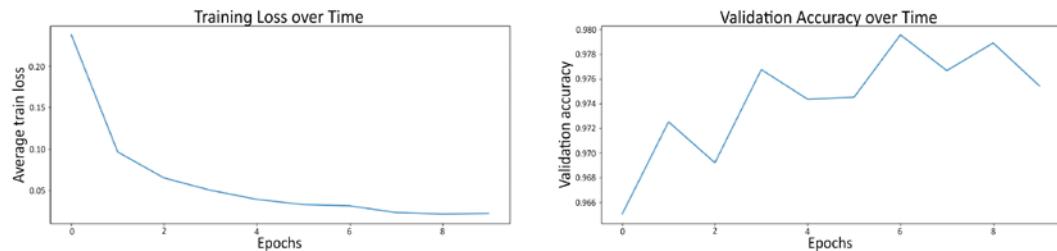


Figure 2.10: Training loss and validation accuracy over 10 epochs as the model is trained

Next up is testing our model on some unseen data.

Testing the model

Testing the model is also straightforward. During testing, we measure the loss and the accuracy of the model on the test dataset. In order to evaluate the model on a dataset, Keras models provide a convenient function called `evaluate()`:

```
test_res = model.evaluate(
    x=x_test,
    y=y_onehot_test,
    batch_size=batch_size
)
```

The arguments expected by the `model.evaluate()` function are already covered during our discussion of `model.fit()`:

- `x` – An input tensor. In our case, this is a 10000×784 sized tensor.
- `y` – The one-hot encoded label tensor. In our case, this is a 10000×10 sized tensor.
- `batch_size` – Batch size defines how many examples are included in a single batch. The larger the batch size, the better the accuracy of your model would be generally.

You will get a loss of 0.138 and an accuracy of 98%. You will not get the exact same values due to various randomness present in the model, as well as during training.

In this section, we went through an end-to-end example of training a neural network. We prepared the data, trained the model on that data, and finally tested it on some unseen data.

Summary

In this chapter, you took your first steps to solving NLP tasks by understanding the primary underlying platform (TensorFlow) on which we will be implementing our algorithms. First, we discussed the underlying details of TensorFlow architecture. Next, we discussed the essential ingredients of a meaningful TensorFlow program. We got to know some new features in TensorFlow 2, such as the AutoGraph feature, in depth. We then discussed more exciting elements in TensorFlow such as data pipelines and various TensorFlow operations.

Specifically, we discussed the TensorFlow architecture by lining up the explanation with an example TensorFlow program; the sigmoid example. In this TensorFlow program, we used the AutoGraph feature to generate a TensorFlow graph; that is, using the `tf.function()` decorator over the function that performs the TensorFlow operations. Then, a `GraphDef` object was created representing the graph and sent to the distributed master. The distributed master looked at the graph, decided which components to use for the relevant computation, and divided it into several subgraphs to make the computations faster. Finally, workers executed subgraphs and returned the result immediately.

Next, we discussed the various elements that comprise a typical TensorFlow client: inputs, variables, outputs, and operations. Inputs are the data we feed to the algorithm for training and testing purposes. We discussed three different ways of feeding inputs: using NumPy arrays, preloading data as TensorFlow tensors, and using `tf.data` to define an input pipeline. Then we discussed TensorFlow variables, how they differ from other tensors, and how to create and initialize them. Following this, we discussed how variables can be used to create intermediate and terminal outputs.

Finally, we discussed several available TensorFlow operations, including mathematical operations, matrix operations, and neural network-related operations that will be used later in the book.

Later, we discussed Keras, a sub-module in TensorFlow that supports building models. We learned that there are three different APIs for building models: the Sequential API, the Functional API, and the Sub-classing API. We learned that the Sequential API is the easiest to use, whereas the Sub-classing API takes much more effort. However, the Sequential API is very restrictive in terms of the type of models that can be implemented with it.

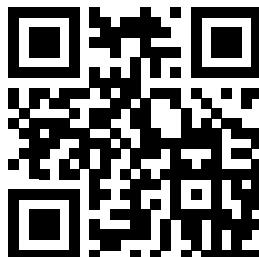
Finally, we implemented a neural network using all the concepts learned previously. We used a three-layer neural network to classify a MNIST digit dataset, and we used Keras (a high-level sub-module in TensorFlow) to implement this model.

In the next chapter, we will see how to use the fully connected neural network we implemented in this chapter for learning the semantic, numerical word representation of words.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside
more than 1000 members at: <https://packt.link/nlp>



3

Word2vec – Learning Word Embeddings

In this chapter, we will discuss a topic of paramount importance in NLP—Word2vec, a data-driven technique for learning powerful numerical representations (that is, vectors) of words or tokens in a language. Languages are complex. This warrants sound language understanding capabilities in the models we build to solve NLP problems. When transforming words to a numerical representation, a lot of methods aren't able to sufficiently capture the semantics and contextual information that word carries. For example, the feature representation of the word *forest* should be very different from *oven* as these words are rarely used in similar contexts, whereas the representations of *forest* and *jungle* should be very similar. Not being able to capture this information leads to underperforming models.

Word2vec tries to overcome this problem by learning word representations by consuming large amounts of text.



Word2vec is called a *distributed representation*, as the semantics of the word are captured by the activation pattern of the full representation vector, in contrast to a single element of the representation vector (for example, setting a single element in the vector to 1 and rest to 0 for a single word).

In this chapter, we will learn the mechanics of several Word2vec algorithms. But first, we will discuss the classical approaches to solving this problem and their limitations. This then motivates us to look at learning neural-network-based Word2vec algorithms that deliver state-of-the-art performance when finding good word representations.

We will train a model on a dataset and analyze the representations learned by the model. We visualize (using t-SNE, a visualization technique for high-dimensional data) these learned word embeddings for a set of words on a 2D canvas in *Figure 3.1*. If you take a closer look, you will see that similar things are placed close to each other (for example, numbers in the cluster in the middle):

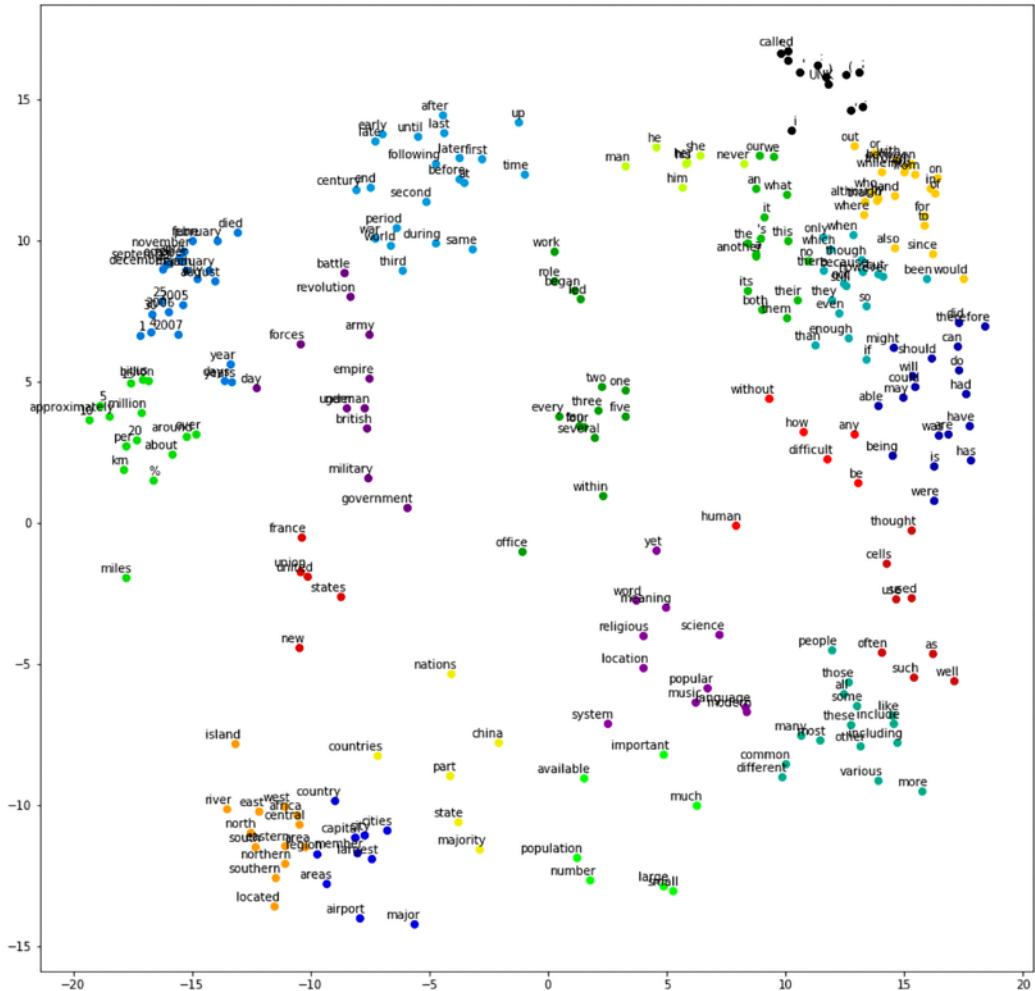


Figure 3. 1: An example visualization of learned word embeddings using t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE)



This is a dimensionality reduction technique that projects high-dimensional data to a two-dimensional space. This allows us to imagine how high-dimensional data is distributed in space, because humans are generally not so good at intuitively understanding data in more than three dimensions. You will learn about t-SNE in more detail in the next chapter.

This chapter covers this information through the following main topics:

- What is a word representation or meaning?
- Classical approaches to learning word representations
- Word2vec – a neural network-based approach to learning word representation
- The skip-gram algorithm
- The Continuous Bag-of-Words algorithm

By the end of this chapter, you will have gained a thorough understanding of how the history of word representations has led to Word2vec, how to utilize two different Word2vec algorithms, and the vital importance of Word2vec for NLP.

What is a word representation or meaning?

What is meant by the word *meaning*? This is more of a philosophical question than a technical one. So, we will not try to discern the best answer for this question, but accept a more modest answer, that is, *meaning* is the idea conveyed by or some representation associated with a word. For example, when you hear the word “cat” you conjure up a mental picture of something that meows, has four legs, has a tail, and so on; then, if you hear the word “dog,” you again formulate a mental image of something that barks, has a bigger body than a cat, has four legs, has a tail, and so on. In this new space (that is, the mental pictures), it is easier for you to understand that cats and dogs are similar than by just looking at the words. Since the primary objective of NLP is to achieve human-like performance in linguistic tasks, it is sensible to explore principled ways of representing words for machines. To achieve this, we will use algorithms that can analyze a given text corpus and come up with good numerical representations of words (that is, word embeddings) such that words that fall within similar contexts (for example, *one* and *two*, *I* and *we*) will have similar numerical representations compared to words that are unrelated (for example, *cat* and *volcano*).

First, we will discuss some classical approaches to achieve this and then move on to understanding recent, more sophisticated methods that use neural networks to learn feature representations and deliver state-of-the-art performance.

Classical approaches to learning word representation

In this section, we will discuss some of the classical approaches used for numerically representing words. It is important to have an understanding of the alternatives to word vectors, as these methods are still used in the real world, especially when limited data is available.

More specifically, we will discuss common representations, such as **one-hot encoding** and **Term Frequency-Inverse Document Frequency (TF-IDF)**.

One-hot encoded representation

One of the simpler ways of representing words is to use the one-hot encoded representation. This means that if we have a vocabulary of size V , for each i^{th} word w_i , we will represent the word w_i with a V -length vector $[0, 0, 0, \dots, 0, 1, 0, \dots, 0, 0, 0]$ where the i^{th} element is 1 and other elements are 0. As an example, consider this sentence:

Bob and Mary are good friends.

The one-hot encoded representation of each word might look like this:

Bob: [1,0,0,0,0,0]

and: [0,1,0,0,0,0]

Mary: [0,0,1,0,0,0]

are: [0,0,0,1,0,0]

good: [0,0,0,0,1,0]

friends: [0,0,0,0,0,1]

However, as you might have already figured out, this representation has many drawbacks.

This representation does not encode the similarity between words in any way and completely ignores the context in which the words are used. Let's consider the dot product between the word vectors as the similarity measure. The more similar two vectors are, the higher the dot product is for those two vectors. For example, the representation of the words *car* and *automobile* will have a similarity distance of 0, while *car* and *pencil* will also have the same value.

This method becomes extremely ineffective for large vocabularies. Also, for a typical NLP task, the vocabulary easily can exceed 50,000 words. Therefore, the word representation matrix for 50,000 words will result in a very sparse $50,000 \times 50,000$ matrix.

However, one-hot encoding plays an important role even in state-of-the-art word embedding learning algorithms. We use one-hot encoding to represent words numerically and feed them into neural networks so that the neural networks can learn better and smaller numerical feature representations of the words.



One-hot encoding is also known as a localist representation (the opposite to the distributed representation), as the feature representation is decided by the activation of a single element in the vector.

We will now discuss another technique for representing words, known as the TF-IDF method.

The TF-IDF method

TF-IDF is a frequency-based method that takes into account the frequency with which a word appears in a corpus. This is a word representation in the sense that it represents the importance of a specific word in a given document. Intuitively, the higher the frequency of the word, the more important that word is in the document. For example, in a document about cats, the word *cats* will appear more often than in a document that isn't about cats. However, just calculating the frequency would not work because words such as *this* and *is* are very frequent in documents but do not contribute much information. TF-IDF takes this into consideration and gives values of near-zero for such common words.

Again, *TF* stands for term frequency and *IDF* stands for inverse document frequency:

$$TF(w_i) = \text{number of times } w_i \text{ appear} / \text{total number of words}$$

$$IDF(w_i) = \log(\text{total number of documents} / \text{number of documents with } w_i \text{ in it})$$

$$TF-IDF(w_i) = TF(w_i) \times IDF(w_i)$$

Let's do a quick exercise. Consider two documents:

- Document 1: *This is about cats. Cats are great companions.*
- Document 2: *This is about dogs. Dogs are very loyal.*

Now let's crunch some numbers:

$$\text{TF-IDF}(\text{cats}, \text{doc1}) = (2/8) * \log(2/1) = 0.075$$

$$\text{TF-IDF}(\text{this}, \text{doc2}) = (1/8) * \log(2/2) = 0.0$$

Therefore, the word *cats* is informative, while *this* is not. This is the desired behavior we needed in terms of measuring the importance of words.

Co-occurrence matrix

Co-occurrence matrices, unlike one-hot-encoded representations, encode the context information of words, but require maintaining a $V \times V$ matrix. To understand the co-occurrence matrix, let's take two example sentences:

- *Jerry and Mary are friends.*
- *Jerry buys flowers for Mary.*

The co-occurrence matrix will look like the following matrix. We only show one half of the matrix, as it is symmetrical:

	Jerry	and	Mary	are	friends	buys	flowers	for
Jerry	0	1	0	0	0	1	0	0
and		0	1	0	0	0	0	0
Mary			0	1	0	0	0	1
are				0	1	0	0	0
friends					0	0	0	0
buys						0	1	0
flowers							0	1
for								0

However, it is not hard to see that maintaining such a co-occurrence matrix comes at a cost as the size of the matrix grows polynomially with the size of the vocabulary. Furthermore, it is not straightforward to incorporate a context window size larger than 1. One option is to have a weighted count, where the weight for a word in the context deteriorates with the distance from the word of interest.

As you can see, these methods are very limited in their representational power.

For example, in the one-hot encoded method, all words will have the same vector distance to each other. The TF-IDF method represents a word with a single number and is unable to capture the semantics of words. Finally calculating the co-occurrence matrix is very expensive and provides limited information about a word's context.

We end our discussion about simple representations of words here. In the following section, we will first develop an intuitive understanding of word embeddings by working through an example. Then we will define a loss function so that we can use machine learning to learn word embeddings. Also, we will discuss two Word2vec algorithms, namely, the **skip-gram** and **Continuous Bag-of-Words (CBOW)** algorithms.

An intuitive understanding of Word2vec – an approach to learning word representation



“You shall know a word by the company it keeps.”

—J.R. Firth

This statement, uttered by J. R. Firth in 1957, lies at the very foundation of Word2vec, as Word2vec techniques use the context of a given word to learn its semantics.

Word2vec is a groundbreaking approach that allows computers to learn the meaning of words without any human intervention. Also, Word2vec learns numerical representations of words by looking at the words surrounding a given word.

We can test the correctness of the preceding quote by imagining a real-world scenario. Imagine you are sitting an exam and you find this sentence in your first question: “Mary is a very stubborn child. Her *pervicacious* nature always gets her in trouble.” Now, unless you are very clever, you might not know what *pervicacious* means. In such a situation, you automatically will be compelled to look at the phrases surrounding the word of interest. In our example, *pervicacious* is surrounded by *stubborn*, *nature*, and *trouble*. Looking at these three words is enough to determine that *pervicacious* in fact means the state of being stubborn. I think this is adequate evidence to observe the importance of context for a word's meaning.

Now let's discuss the basics of Word2vec. As already mentioned, Word2vec learns the meaning of a given word by looking at its context and representing it numerically.

By *context*, we refer to a fixed number of words in front of and behind the word of interest. Let's take a hypothetical corpus with N words. Mathematically, this can be represented by a sequence of words denoted by w_0, w_1, \dots, w_i , and w_N , where w_i is the i^{th} word in the corpus.

Next, if we want to find a good algorithm that is capable of learning word meanings, given a word, our algorithm should be able to predict the context words correctly.

This means that the following probability should be high for any given word w_i :

$$P(w_{i-m}, \dots, w_{i-1}, w_{i+1}, \dots, w_{i+m} | w_i) = \prod_{j \neq i \wedge j=i-m}^{i+m} P(w_j | w_i)$$

To arrive at the right-hand side of the equation, we need to assume that given the target word (w_i), the context words are independent of each other (for example, w_{i-2} and w_{i-1} are independent). Though not entirely true, this approximation makes the learning problem practical and works well in practice. Let's go through an example to understand the computations.

Exercise: does queen = king – he + she?

Before proceeding further, let's do a small exercise to understand how maximizing the previously mentioned probability leads to finding good meaning (or representations) of words. Consider the following very small corpus:

There was a very rich king. He had a beautiful queen. She was very kind.

To keep the exercise simple, let's do some manual preprocessing and remove the punctuation and the uninformative words:

was rich king he had beautiful queen she was kind

Now let's form a set of tuples for each word with their context words in the format (*target word* --> *context word 1, context word 2*). We will assume a context window size of 1 on either side:

was --> *rich*

rich --> *was, king*

king --> *rich, he*

he --> *king, had*

had --> *he, beautiful*

beautiful --> *had, queen*

queen --> *beautiful*, *she*

she --> *queen*, *was*

was --> *she*, *kind*

kind --> *was*

Remember, our goal is to be able to predict the words on the right given the word on the left. To do this, for a given word, the words on the right-side context should share a high numerical or geometrical similarity with the words on the left-side context. In other words, the word of interest should be conveyed by the surrounding words. Now let's consider actual numerical vectors to understand how this works. For simplicity, let's only consider the tuples highlighted in bold.

Let's begin by assuming the following for the word *rich*:

rich --> [0,0]

To be able to correctly predict *was* and *king* from *rich*, *was* and *king* should have high similarity with the word *rich*. The Euclidean distance will be used to measure the distance between words.

Let's try the following values for the words *king* and *rich*:

king --> [0,1]

was --> [-1,0]

This works out fine as the following:

$Dist(rich, king) = 1.0$

$Dist(rich, was) = 1.0$

Here, *Dist* is the Euclidean distance between two words. This is illustrated in *Figure 3.3*:

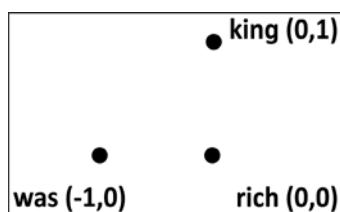


Figure 3.2: The positioning of word vectors for the words “rich”, “was” and “king”

Now let's consider the following tuple:

king --> *rich*, *he*

We have established the relationship between *king* and *rich* already. However, it is not done yet; the more we see a relationship, the closer these two words should be. So, let's first adjust the vector of *king* so that it is a bit closer to *rich*:

king --> [0,0.8]

Next, we will need to add the word *he* to the picture. The word *he* should be closer to *king*. This is all the information that we have right now about the word *he*: *he* --> [0.5,0.8].

At this moment, the graph with the words looks like *Figure 3.4*:

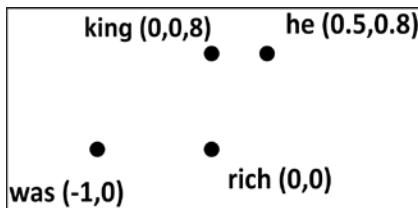


Figure 3.3: The positioning of word vectors for the words “rich”, “was”, “king,” and “he”

Now let's proceed with the next two tuples: *queen* --> *beautiful*, *she* and *she* --> *queen*, *was*. Note that I have swapped the order of the tuples as this makes it easier for us to understand the example:

she --> *queen*, *was*

Now, we will have to use our prior knowledge of English to proceed further.

It is a reasonable decision to place the word *she* the same distance from *was* that *he* is from *was*, because their usage in the context of the word *was* is equivalent. Therefore, let's use this:

she --> [0.5,0.6]

Next, we will use the word *queen* close to the word *she*: *queen* --> [0.0,0.6].

This is illustrated in *Figure 3.5*:

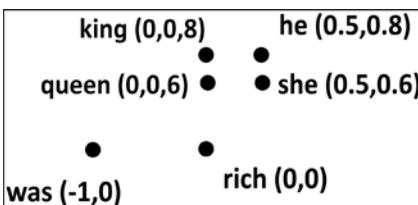


Figure 3.4: The positioning of word vectors for the words “rich,” “was,” “king,” “he,” “she,” and “queen”

Next, we only have the following tuple:

queen --> *beautiful, she*

Here, the word *beautiful* is found. It should be approximately the same distance from the words *queen* and *she*. Let's use the following:

beautiful --> [0.25, 0]

Now we have the following graph depicting the relationships between words. When we observe *Figure 3.6*, it seems to be a very intuitive representation of the meanings of words:

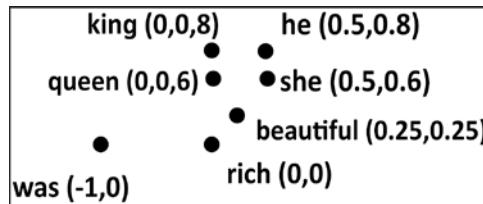


Figure 3.5: The positioning of word vectors for the words “rich,” “was,” “king,” “he,” “she,” “queen,” and “beautiful”

Now, let's look at the question that has been lurking in our minds since the beginning of this exercise. Are the quantities in this equation equivalent: *queen* = *king* - *he* + *she*? Well, we've got all the resources that we'll need to solve this mystery now. Let's try the right-hand side of the equation first:

$$= \text{king} - \text{he} + \text{she}$$

$$= [0, 0.8] - [0.5, 0.8] + [0.5, 0.6]$$

$$= [0, 0.6]$$

It all works out in the end. If you look at the word vector we obtained for the word *queen*, you see that this is exactly the same as the answer we deduced earlier.

Note that this is a crude way to show how word embeddings are learned, and this might differ from the exact positions of word embeddings learned using an algorithm.

Also keep in mind that this is an unrealistically scaled-down exercise with regard to what a real-world corpus might look like. So, you will not be able to work out these values by hand just by crunching a dozen numbers. Sophisticated function approximators such as neural networks do this job for us. But, to use neural networks, we need to formulate our problem in a mathematically assertive way. However, this is a good exercise to show the power of word vectors.

Now that we have a good understanding of how Word2vec enables us to learn word representations, let's look at the actual algorithms Word2vec utilizes in the next two sections.

The skip-gram algorithm

The first algorithm we will talk about is known as the **skip-gram algorithm**: a type of Word2vec algorithm. As we have discussed in numerous places, the meaning of a word can be elicited from the contextual words surrounding it. However, it is not entirely straightforward to develop a model that exploits this way of learning word meanings. The skip-gram algorithm, introduced by Mikolov et al. in 2013, is an algorithm that does exploit the context of the words in a written text to learn good word embeddings.

Let's go through the skip-gram algorithm step by step. First, we will discuss the data preparation process. Understanding the format of the data puts us in a great position to understand the algorithm. We will then discuss the algorithm itself. Finally, we will implement the algorithm using TensorFlow.

From raw text to semi-structured text

First, we need to design a mechanism to extract a dataset that can be fed to our learning model. Such a dataset should be a set of tuples of the format (target, context). Moreover, this needs to be created in an unsupervised manner. That is, a human should not have to manually engineer the labels for the data. In summary, the data preparation process should do the following:

- Capture the surrounding words of a given word (that is, the context)
- Run in an unsupervised manner

The skip-gram model uses the following approach to design a dataset:

- For a given word w_i , a context window size of m is assumed. By *context window size*, we mean the number of words considered as context on a single side. Therefore, for w_i , the context window (including the target word w_i) will be of size $2m+1$ and will look like this: $[w_{i-m}, \dots, w_{i-1}, w_i, w_{i+1}, \dots, w_{i+m}]$.
- Next, (target, context) tuples are formed as $[..., (w_i, w_{i-m}), \dots, (w_i, w_{i-1}), (w_i, w_{i+1}), \dots, (w_i, w_{i+m}), ...]$; here, $m + 1 \leq i \leq N - m$ and N is the number of words in the text. Let's use the following sentence and a context window size (m) of 1:

The dog barked at the mailman.

For this example, the dataset would be as follows:

$[(\text{dog}, \text{The}), (\text{dog}, \text{barked}), (\text{barked}, \text{dog}), (\text{barked}, \text{at}), \dots, (\text{the}, \text{at}), (\text{the}, \text{mailman})]$

Once the data is in the $(\text{target}, \text{context})$ format, we can use a neural network to learn the word embeddings.

Understanding the skip-gram algorithm

First, let's identify the variables and notation we need to learn the word embeddings. To store the word embeddings, we need two $V \times D$ matrices, where V is the vocabulary size and D is the dimensionality of the word embeddings (that is, the number of elements in the vector that represent a single word). D is a user-defined hyperparameter. The higher D is, the more expressive the word embeddings learned will be. We need two matrices, one to represent the context words and one to represent the target words. These matrices will be referred to as the *context embedding space* (or *context embedding layer*) and the *target embedding space* (or *target embedding layer*), or in general as the embedding space (or the embedding layer).

Each word will be represented with a unique ID in the range $[1, V+1]$. These IDs are passed to the embedding layer to look up corresponding vectors. To generate these IDs, we will use a special object called a Tokenizer that's available in TensorFlow. Let's refer to an example target-context tuple (w_i, w_j) , where the target word ID is w_i , and one of the context words is w_j . The corresponding target embedding of w_i is t_i , and the corresponding context embedding of w_j is c_j . Each target-context tuple is accompanied by a label (0 or 1), denoted by y_i , where true target-context pairs will get a label of 1, and negative (or false) target-context candidates will get a label of 0. It is easy to generate negative target-context candidates by sampling a word that does not appear in the context of a given target as the context word. We will talk about this in more detail later.

At this point, we have defined the necessary variables. Next, for each input w_i , we will look up the embedding vectors from the context embedding layer corresponding to the input. This operation provides us with c_i , which is a D -sized vector (that is, a D -long embedding vector). We do the same for the input w_j , using the context embedding space to retrieve c_j . Afterward, we calculate the prediction output for (w_i, w_j) using the following transformation:

$$\text{logit}(w_i, w_j) = c_i \cdot t_j$$

$$\hat{y}_{ij} = \text{sigmoid}(\text{logit}(w_i, w_j))$$

Here, $\text{logit}(w_i, w_j)$ represents the unnormalized scores (that is, logits), \hat{y}_{ij} is a single-valued predicted output (representing the probability of context word belonging in the context of the target word).

We will visualize both the conceptual (*Figure 3.7*) and implementation (*Figure 3.8*) views of the skip-gram model. Here is a summary of the notation:

- V : This is the size of the vocabulary
- D : This is the dimensionality of the embedding layer
- w_i : Target word
- w_j : Context word
- t_i : Target embedding of the word w_i
- c_j : Context embedding of the word w_j
- y_i : This is the one-hot-encoded output word corresponding to x_i
- \hat{y}_i : This is the predicted output for x_i
- $\text{logit}(w_i, w_j)$: This is the unnormalized score for the input x_i

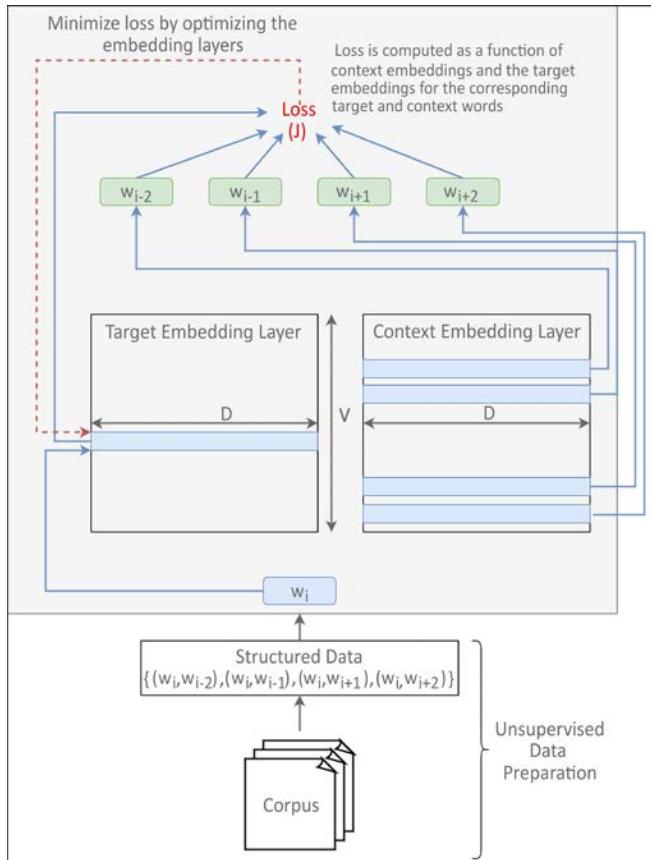


Figure 3.6: The conceptual skip-gram model

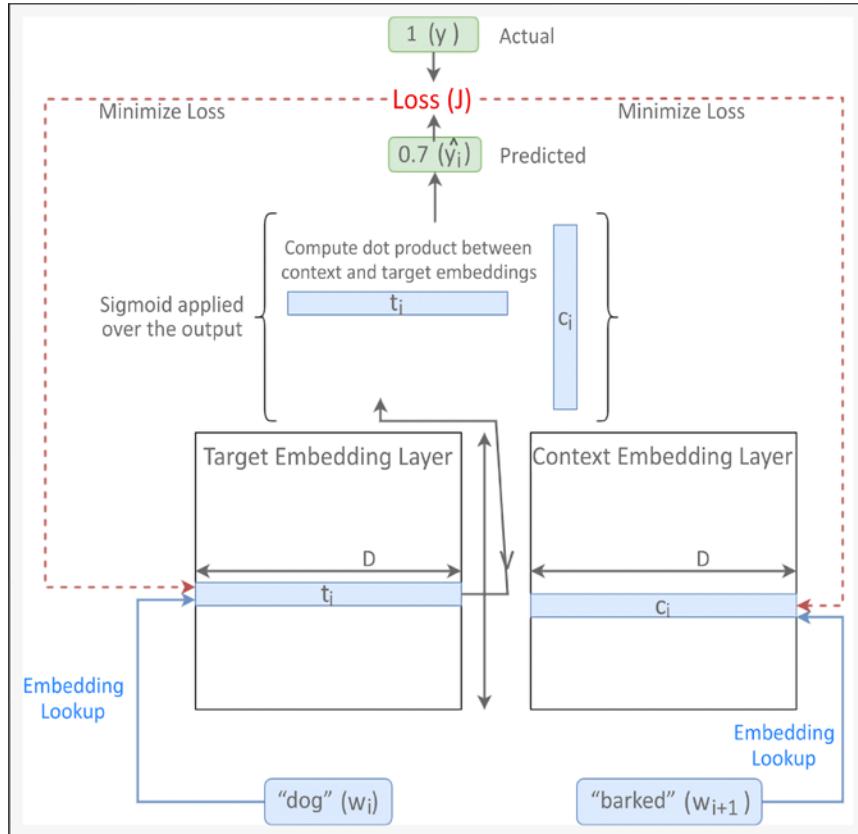


Figure 3.7: The implementation of the skip-gram model

Using both the existing and derived entities, we can now use the cross-entropy loss function to calculate the loss for a given data point $[(w_i, w_j), y_i]$.

For binary labels, the cross-entropy loss for a single sample (x_i, y_i) is computed as:

$$CE(\tilde{y}_i, y_i) = -[y_i \log(\tilde{y}_i) + (1 - y_i) \log(1 - \tilde{y}_i)]$$

Where \tilde{y}_i is the predicted label for x_i . For multi-class classification problems, we generalize the loss by computing the term $y_i \log(\tilde{y}_i)$ for each class:

$$CE(\tilde{y}_i, y_i) = - \sum_{c=0}^C y_{i,c} \log(\tilde{y}_{i,c})$$

Where $y_{i,c}$ represents the value of the c^{th} index of the y_i , where y_i is a one hot encoded vector representing the label of the data point.

Typically, when training neural networks, this loss is computed for each sample in a given batch, then averaged to compute the loss of the batch. Finally, the batch losses are averaged over all the batches in the dataset to compute the final loss.

Why does the original word embeddings paper use two embedding layers?

The original paper (by Mikolov et al., 2013) uses two distinct $V \times D$ embedding spaces to denote words in the target space (words when used as the target) and words in the contextual space (words used as context words). One motivation to do this is that a word does not occur in its own context often. So, we want to minimize the probability of such things happening.

For example, for the target word *dog*, it is highly unlikely that the word *dog* is also found in its context ($P(\text{dog}|\text{dog}) \sim 0$). Intuitively, if we feed the ($w_i=\text{dog}$ and $w_j=\text{dog}$) data point to the neural network, we are asking the neural network to give a higher loss if the neural network predicts *dog* as a context word of *dog*.



In other words, we are asking the word embedding of the word *dog* to have a very high distance to the word embedding of the word *dog*. This creates a strong contradiction as the distance between the embeddings of the same word will be 0. Therefore, we cannot achieve this if we only have a single embedding space.

However, having two separate embedding spaces for target words and contextual words allows us to have this property because this way we have two separate embedding vectors for the same word. In practice, as long as you avoid feeding input-output tuples, having the same word as input and output allows us to work with a single embedding space and eliminates the need for two distinct embedding layers.

Let's now implement the data generation process with TensorFlow.

Implementing and running the skip-gram algorithm with TensorFlow

We are now going to get our hands dirty with TensorFlow and implement the algorithm from end to end. First, we will discuss the data we're going to use and how TensorFlow can help us to get that data in the format the model accepts. We will implement the skip-gram algorithm with TensorFlow and finally train the model and evaluate it on data that was prepared.

Implementing the data generators with TensorFlow

First, we will investigate how data can be generated in the correct format for the model. For this exercise, we are going to use the BBC news articles dataset available at <http://mlg.ucd.ie/datasets/bbc.html>. It contains 2,225 news articles belonging to 5 topics, business, entertainment, politics, sport, and tech, which were published on the BBC website between 2004-2005.

We write the function `download_data()` below to download the data to a given folder and extract it from its compressed format:

```
def download_data(url, data_dir):
    """Download a file if not present, and make sure it's the right
    size."""

    os.makedirs(data_dir, exist_ok=True)

    file_path = os.path.join(data_dir, 'bbc-fulltext.zip')

    if not os.path.exists(file_path):
        print('Downloading file...')
        filename, _ = urlretrieve(url, file_path)
    else:
        print("File already exists")

    extract_path = os.path.join(data_dir, 'bbc')
    if not os.path.exists(extract_path):

        with zipfile.ZipFile(
            os.path.join(data_dir, 'bbc-fulltext.zip'),
            'r'
        ) as zipf:
            zipf.extractall(data_dir)

    else:
        print("bbc-fulltext.zip has already been extracted")
```

The function first creates the `data_dir` if it doesn't exist. Next, if the `bbc-fulltext.zip` file does not exist, it will be downloaded from the provided URL. If `bbc-fulltext.zip` has not been extracted yet, it will be extracted to `data_dir`.

We can call this function as follows:

```
url = 'http://mlg.ucd.ie/files/datasets/bbc-fulltext.zip'
download_data(url, 'data')
```

With that, we are going to focus on reading the data contained in the news articles (in .txt format) into the memory. To do that, we will define the `read_data()` function, which takes a data directory path (`data_dir`), and reads the .txt files (except for the README file) found in the data directory:

```
def read_data(data_dir):
    news_stories = []
    print("Reading files")
    for root, dirs, files in os.walk(data_dir):
        for fi, f in enumerate(files):
            if 'README' in f:
                continue
            print("." * fi, f, end='\r')
            with open(os.path.join(root, f), encoding='latin-1') as f:
                story = []
                for row in f:
                    story.append(row.strip())
                story = ' '.join(story)
                news_stories.append(story)
    print(f"\nDetected {len(news_stories)} stories")
    return news_stories
```

With the `read_data()` function defined, let's use it to read in the data and print some samples as well as some statistics:

```
news_stories = read_data(os.path.join('data', 'bbc'))

print(f'{sum([len(story.split(' ')) for story in news_stories])} words
found in the total news set')
print('Example words (start): ',news_stories[0][:50])
print('Example words (end): ',news_stories[-1][-50:])
```

This will print the following:

```
Reading files
..... 361.txt
Detected 2225 stories
```

```
865163 words found in the total news set
Example words (start): Windows worm travels with Tetris Users are being
Example words (end): is years at Stradey as "the best time of my life."
```

As we said at the beginning of this section, there are 2,225 stories with close to a million words. In the next step, we need to tokenize each story (in the form of a long string) to a list of tokens (or words). Along with that, we will perform some preprocessing on the text:

- Lowercase all the characters
- Remove punctuation

All of these can be achieved with the `tensorflow.keras.preprocessing.text.Tokenizer` object. We can define a Tokenizer as follows:

```
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(
    num_words=None,
    filters='!"#$%&()*+,-./:;=>?@[\\]^_{}|~\t\n',
    lower=True,
    split=' '
)
```

Here, you can see some of the most popular keyword arguments and their default values used when defining a Tokenizer:

- `num_words` – Defines the size of the vocabulary. Defaults to `None`, meaning it will consider all the words appearing in the text corpus. If set to the integer `n`, it will only consider the `n` most common words appearing in the corpus.
- `filters` – Defines any characters that need to be omitted during preprocessing. By default, it defines a string containing most of the common punctuation marks and symbols.
- `lower` – Defines whether the text needs to be converted to lowercase.
- `split` – Defines the character that the words will be tokenized on.

Once the Tokenizer is defined, you can call its `fit_on_texts()` method with a list of strings (where each string is a news article) so that the Tokenizer will learn the vocabulary and map the words to unique IDs:

```
tokenizer.fit_on_texts(news_stories)
```

Let's take a moment to analyze what the Tokenizer has produced after it has been fitted on the text. Once it has been fitted, the Tokenizer will have two important attributes populated: `word_index` and `index_word`. Here `word_index` is a dictionary that maps each word to a unique ID. The `index_word` attribute is the opposite of `word_index`, that is, a dictionary that maps each unique word ID to the corresponding word:

```
n_vocab = len(tokenizer.word_index.items())+1
print(f"Vocabulary size: {n_vocab}")

print("\nWords at the top")
print('\t', dict(list(tokenizer.word_index.items())[:10]))
print("\nWords at the bottom")
print('\t', dict(list(tokenizer.word_index.items())[-10:]))

Note how we are using the length of the word_index dictionary to derive the vocabulary size. We need an additional 1 as the ID 0 is a reserved ID and will not be used for any word. This will output the following:
```

```
Vocabulary size: 32361

Words at the top
{'the': 1, 'to': 2, 'of': 3, 'and': 4, 'a': 5, 'in': 6, 'for': 7,
'is': 8, 'that': 9, 'on': 10}

Words at the bottom
{'counsellor': 32351, "'frag)": 32352, 'relasing': 32353, "'real)": 32354, 'hrs': 32355, 'enviroment': 32356, 'trifling': 32357, '24hours': 32358, 'ahhhh': 32359, 'lol': 32360}
```

The more frequent a word is in the corpus, the lower the ID will be. Words such as “the”, “to” and “of” which tend to be common (and are called stop words) are in fact the most common words. As the next step, we are going to refine our Tokenizer object to have a limited-sized vocabulary. Because we are working with a relatively small corpus, we have to make sure the vocabulary is not too large, as it can lead to poorly learned word vectors due to the lack of data:

```
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(
    num_words=15000,
```

```
filters='! "#$%&()*+,-./:;<=>?@[\\]^_{}|}~\\t\\n',
lower=True, split=' ', oov_token='',
)

tokenizer.fit_on_texts(news_stories)
```

Since we have a total vocabulary of more than 30,000 words, we'll restrict the size of the vocabulary to 15,000. This means the Tokenizer will only keep the most common 15,000 words as the vocabulary. When we restrict a vocabulary this way, a new problem arises. As the Tokenizer's vocabulary does not encompass all possible words in the true vocabulary, out-of-vocabulary words (or OOV words) can rear their heads. Some solutions are to replace OOV words with a special token (such as <UNK>) or remove them from the corpus. This is possible by passing the string you want to replace OOV tokens with to the `oov_token` argument in the Tokenizer. In this case, we will remove OOV words. If we are careful when setting the size of the vocabulary, omitting some of the rare words would not harm learning the context of words accurately.

We can have a look at the transformation done on the text by the Tokenizer as follows. Let's convert a string of the first 100 characters of the first story in our corpus (stored in the `news_stories` variable):

```
print(f"Original: {news_stories[0][:100]}")
```

Then we can call the Tokenizer's `texts_to_sequences()` method to convert a list of documents (where each document is a string) to a list of list of word IDs (that is, each document is converted to a list of word IDs).

```
print(f"Sequence IDs: {tokenizer.texts_to_sequences([news_stories[0]
[:100]])[0]}")
```

This will print out:

```
Original: Ad sales boost Time Warner profit Quarterly profits at US media
giant TimeWarner jumped 76% to $1.1
Sequence IDs: [4223, 187, 716, 66, 3596, 1050, 3938, 626, 21, 49, 303,
717, 8263, 2972, 5321, 3, 108, 108]
```

We now have our Tokenizer sorted. There's nothing left to do but to convert all of our news articles to sequences of word IDs with a single line of code:

```
news_sequences = tokenizer.texts_to_sequences(news_stories)
```

Let's move on to generating skip-grams using the `tf.keras.preprocessing.sequence.skipgrams()` function, provided by TensorFlow. We call the function on a sample phrase representing the first 5 words extracted from the first article in the dataset:

```
sample_word_ids = news_sequences[0][:5]
sample_phrase = ' '.join([tokenizer.index_word[wid] for wid in sample_
word_ids])
print(f"Sample phrase: {sample_phrase}")
print(f"Sample word IDs: {sample_word_ids }\n")
```

This will output:

```
Sample phrase: ad sales boost time warner
Sample word IDs: [4223, 187, 716, 66, 3596]
```

Let's consider a window size of 1. This means, for a given target word, we define the context as one word from each side of the target word.

```
window_size = 1 # How many words to consider left and right.
```

We have all the ingredients to define extract skip-grams from the sample phrase we chose as follows. When run, this function will output data in the exact format we need the data in, that is, (target-context) tuples as inputs and corresponding labels (0 or 1) as outputs:

```
inputs, labels = tf.keras.preprocessing.sequence.skipgrams(
    sequence=sample_word_ids,
    vocabulary_size=n_vocab,
    window_size=window_size,
    negative_samples=1.0,
    shuffle=False,
    categorical=False,
    sampling_table=None,
    seed=None
)
```

Let's take a moment to reflect on some of the important arguments that have been used:

- `sequence` (`list[str]` or `list[int]`) – A list of words or word IDs.
- `vocabulary_size` (`int`) – Size of the vocabulary.
- `window_size` (`int`) – Size of the window to be considered for the context. `window_size` defines the length on each side.

- `negative_samples` (int) – Fraction of negative candidates to generate. For example, a value of 1 means there will be an equal number of positive and negative skipgram candidates. A value of 0 means there will not be any negative candidates.
- `shuffle` (bool) – Whether to shuffle the generated inputs or not.
- `categorical` (bool) – Whether to produce labels as categorical (that is, one-hot encoded) or integers.
- `sampling_table` (`np.ndarray`) – An array of the same size as the vocabulary. An element in a given position in the array represents the probability of sampling the word indexed by that position in the Tokenizer’s word ID to word mapping. As we will see soon, this is a handy way to avoid common uninformative words being over-sampled much.
- `seed` (int) – If shuffling is enabled, this is the random seed to be used for shuffling.

With the inputs and labels generated, let’s print some data:

```
print("Sample skip-grams")

for inp, lbl in zip(inputs, labels):
    print(f"\tInput: {inp} ({[tokenizer.index_word[wi] for wi in inp]}) /"
          f"\n\tLabel: {lbl}")
```

This will produce:

```
Sample skip-grams
Input: [4223, 187] (['ad', 'sales']) / Label: 1
Input: [187, 4223] (['sales', 'ad']) / Label: 1
Input: [187, 716] (['sales', 'boost']) / Label: 1
Input: [716, 187] (['boost', 'sales']) / Label: 1
Input: [716, 66] (['boost', 'time']) / Label: 1
Input: [66, 716] (['time', 'boost']) / Label: 1
Input: [66, 3596] (['time', 'warner']) / Label: 1
Input: [3596, 66] (['warner', 'time']) / Label: 1
Input: [716, 9685] (['boost', "kenya's"]) / Label: 0
Input: [3596, 12251] (['warner', 'rear']) / Label: 0
Input: [4223, 3325] (['ad', 'racing']) / Label: 0
Input: [66, 7978] (['time', 'certificate']) / Label: 0
Input: [716, 12756] (['boost', 'crushing']) / Label: 0
Input: [66, 14543] (['time', 'touchy']) / Label: 0
Input: [187, 3786] (['sales', '9m']) / Label: 0
```

```
Input: [187, 3917] (['sales', 'doherty']) / Label: 0
```

For example, since the word “sales” appears in the context of the word “ad”, it is considered a positive candidate. On the other hand, since the word “racing” (randomly sampled from the vocabulary) does not appear in the context of the word “ad”, it is added as a negative candidate.

When selecting negative candidates, the `skipgrams()` function selects them randomly, giving uniform weights to all the words in the vocabulary. However, the original paper explains that this can lead to poor performance. A better strategy is to use the unigram distribution as a prior for selecting negative context words.

You might be wondering what a unigram distribution is. It represents the frequency counts of unigrams (or tokens) found in the text. Then the frequency counts are easily converted to probabilities (or normalized frequencies) by dividing them by the sum of all frequencies. The most amazing thing is that you don’t have to compute this by hand for every corpus of text! It turns out that if you take any sufficiently large corpus of text, compute the normalized frequencies of unigrams, and order them from high to low, you’ll see that the corpus approximately follows a certain constant distribution. For the word with rank *math* in a corpus of *math* unigrams, the normalized frequency f_k is given by:

$$f_k = \frac{k^{-s}}{\sum_{i=1}^N n^{-s}}$$

Here, *math* is a hyperparameter that can be tuned to match the true distribution more closely. This is known as *Zipf’s law*. In other words, if you have a vocabulary where words are ranked (ID-ed) from most common to least common, you can approximate the normalized frequency of each word using Zipf’s law. We will be sampling words according to the probabilities output through Zipf’s law instead of giving equal probabilities to the words. This means words are sampled according to their presence (that is, the more frequent, the higher the chance of being sampled) in the corpus.

To do that, we can use the `tf.random.log_uniform_candidate_sampler()` function. This function takes a batch of positive context candidates of shape `[b, num_true]`, where `b` is the batch size and `num_true` is the number of true candidates per example (1 for the skip-gram model), and it outputs a `[num_sampled]` sized array, where `num_sampled` is the number of negative samples we need. We will discuss the nitty-gritty of this function soon, while going through an exercise. But let’s first generate some positive candidates using the `tf.keras.preprocessing.sequence.skipgrams()` function:

```
inputs, labels = tf.keras.preprocessing.sequence.skipgrams(
```

```
    sample_phrase_word_ids,  
    vocabulary_size=len(tokenizer.word_index.items())+1,  
    window_size=window_size,  
    negative_samples=0,  
    shuffle=False  
)  
  
inputs, labels = np.array(inputs), np.array(labels)
```

Note that we're specifying `negative_samples=0`, as we will be generating negative samples with the candidate sampler. Let's now discuss how we can use the `tf.random.log_uniform_candidate_sampler()` function to generate negative candidates. Here we will first use this function to generate negative candidates for a single word:

```
negative_sampling_candidates, true_expected_count, sampled_expected_count  
= tf.random.log_uniform_candidate_sampler(  
    true_classes=inputs[:, 1:], # [b, 1] sized tensor  
    num_true=1, # number of true words per example  
    num_sampled=10,  
    unique=True,  
    range_max=n_vocab,  
    name="negative_sampling")
```

This function takes the following arguments:

- `true_classes` (`np.ndarray` or `tf.Tensor`) – A tensor containing true target words. This needs to be a `[b, num_true]` sized array, where `num_true` denotes the number of true context candidates per example. Since we have one context word per example, this is 1.
- `num_true` (`int`) – The number of true context terms per example.
- `num_sampled` (`int`) – The number of negative samples to generate.
- `unique` (`bool`) – Whether to generate unique samples or with replacement.
- `range_max` (`int`) – The size of the vocabulary.

It returns:

- `sampled_candidates` (`tf.Tensor`) – A tensor of size `[num_sampled]` containing negative candidates

- `true_expected_count` (`tf.Tensor`) – A tensor of size `[b, num_true]`; the probability of each true candidate being sampled (according to Zipf's law)
- `sampled_expected_count` (`tf.Tensor`) – A tensor of size `[num_sampled]`; the probabilities of each negative sample occurring along with true candidates, if sampled from the corpus

We will not worry too much about the latter two entities. The most important to us is `sampled_candidates`. When calling the function, we have to make sure `true_classes` has the shape `[b, num_true]`. In our case, we will run this in a single input word ID, which will be in the shape `[1, 1]`. It returns the following:

```
Positive sample: [[187]]
Negative samples: [ 1 10 9744 3062 139 5 14 78 1402 115]

true_expected_count: [[0.00660027]]

sampled_expected_count: [4.0367463e-01 1.0333969e-01 1.2804421e-04
4.0727769e-04 8.8460185e-03
1.7628242e-01 7.7631921e-02 1.5584969e-02 8.8879210e-04 1.0659459e-02]
```

Now, putting everything together, let's write a data generator function that generates batches of data for the model. This function, named `skip_gram_data_generator()`, takes the following arguments:

- `sequences` (`List[List[int]]`) – A list of list of word IDs. This is the output generated by the Tokenizer's `texts_to_sequences()` function.
- `window_size` (`int`) – The window size for the context.
- `batch_size` (`int`) – The batch size.
- `negative_samples` (`int`) – The number of negative samples per example to generate.
- `vocabulary_size` (`int`) – The vocabulary size.
- `seed` – The random seed.

It will return a batch of data containing:

- A batch of target word IDs
- A batch of corresponding context word IDs (both positive and negative)
- A batch of labels (0 and 1)

The function signature looks as follows:

```
def skip_gram_data_generator(sequences, window_size, batch_size, negative_
samples, vocab_size, seed=None):
```

First, we are going to shuffle the news articles so that every time we generate data, they are fetched in a different order. This helps the model to generalize better:

```
rand_sequence_ids = np.arange(len(sequences))
np.random.shuffle(rand_sequence_ids)
```

Next, for each text sequence in the corpus we generate positive skip grams. `positive_skip_grams` contains tuples of (target, context) word pairs in that order:

```
for si in rand_sequence_ids:

    positive_skip_grams, _ =
        tf.keras.preprocessing.sequence.skipgrams(
            sequences[si],
            vocabulary_size=vocab_size,
            window_size=window_size,
            negative_samples=0.0,
            shuffle=False,
            sampling_table=sampling_table,
            seed=seed
        )
```

Note that we are passing a `sampling_table` argument. This is another strategy to enhance the performance of Word2vec models. `sampling_table` is simply an array that is the same size as your vocabulary and specifies a probability at each index of the array with which the word indexed by that index will be sampled during skip gram generation. This technique is known as subsampling. Each word w_i is sampled with the probability given by the following equation:

$$p(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Here, t is a tunable parameter. It defaults to 0.00001 for a large enough corpus. In TensorFlow, you can generate this table easily as follows.

You don't need the exact frequencies to compute the sampling table, as we can leverage Zipf's law to approximate those frequencies:

```
sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(  
    n_vocab, sampling_factor=1e-05  
)
```

For each tuple contained in `positive_skip_grams`, we generate `negative_samples` number of negative candidates. We then populate targets, contexts, and label lists with both positive and negative candidates:

```
targets, contexts, labels = [], [], []  
  
for target_word, context_word in positive_skip_grams:  
    context_class = tf.expand_dims(tf.constant([context_word],  
        dtype="int64"), 1)  
  
    negative_sampling_candidates, _, _ =  
        tf.random.log_uniform_candidate_sampler(  
            true_classes=context_class,  
            num_true=1,  
            num_sampled=negative_samples,  
            unique=True,  
            range_max=vocab_size,  
            name="negative_sampling")  
  
    # Build context and label vectors (for one target word)  
    context = tf.concat(  
        [tf.constant([context_word], dtype='int64'),  
         negative_sampling_candidates],  
        axis=0  
    )  
    label = tf.constant([1] + [0]*negative_samples,  
        dtype="int64")  
  
    # Append each element from the training example to global  
    # lists.  
    targets.append([target_word]*(negative_samples+1))
```

```
    contexts.append(context)
    labels.append(label)
```

We will then convert these to arrays as follows and randomly shuffle the data. When shuffling, you have to make sure all the arrays are consistently shuffled. Otherwise, you will corrupt the labels associated with the inputs:

```
contexts, targets, labels = np.concatenate(contexts),
np.array(targets), np.concatenate(labels)

# If seed is not provided generate a random one
if not seed:
    seed = random.randint(0, 10e6)

np.random.seed(seed)
np.random.shuffle(contexts)
np.random.seed(seed)
np.random.shuffle(targets)
np.random.seed(seed)
np.random.shuffle(labels)
```

Finally, batches of data are generated as follows:

```
for eg_id_start in range(0, contexts.shape[0], batch_size):
    yield (
        targets[eg_id_start: min(eg_id_start+batch_size,
                                inputs.shape[0])],
        contexts[eg_id_start: min(eg_id_start+batch_size,
                                inputs.shape[0])]
    ), labels[eg_id_start: min(eg_id_start+batch_size,
                                inputs.shape[0])]
```

Next, we will look at the specifics of the model we're going to use.

Implementing the skip-gram architecture with TensorFlow

We will now walk through an implementation of the skip-gram algorithm that uses the TensorFlow library. The full exercise is available in `ch3_word2vec.ipynb` in the `Ch03-Word-Vectors` exercise directory.

First, let's define the hyperparameters of the model. You are free to change these hyperparameters to see how they affect final performance (for example, `batch_size = 1024` or `batch_size = 2048`). However, since this is a simpler problem than the more complex real-world problems, you might not see any significant differences (unless you change them to extremes, for example, `batch_size = 1` or `num_sampled = 1`):

```
batch_size = 4096 # Data points in a single batch

embedding_size = 128 # Dimension of the embedding vector.

window_size=1 # We use a window size of 1 on either side of target word
negative_samples = 4 # Number of negative samples generated per example

epochs = 5 # Number of epochs to train for

# We pick a random validation set to sample nearest neighbors
valid_size = 16 # Random set of words to evaluate similarity on.
# We sample valid datapoints randomly from a large window without always
# being deterministic
valid_window = 250

# When selecting valid examples, we select some of the most frequent words
# as well as some moderately rare words as well
np.random.seed(54321)
random.seed(54321)

valid_term_ids = np.array(random.sample(range(valid_window), valid_size))
valid_term_ids = np.append(
    valid_term_ids, random.sample(range(1000, 1000+valid_window),
    valid_size),
    axis=0
)
```

Next, we define the model. To do this, we will be relying on the Functional API of Keras. We need to go beyond the simplest API, that is, the Sequential API, as this model requires two input streams (one for the context and one for the target).

We will start off with an import. Then we will clear any current running sessions, to make sure there aren't any other models occupying the hardware:

```
import tensorflow.keras.backend as K  
K.clear_session()
```

We will define two input layers:

```
# Inputs - skipgrams() function outputs target, context in that order  
input_1 = tf.keras.layers.Input(shape=(), name='target')  
input_2 = tf.keras.layers.Input(shape=(), name='context')
```

Note how the shape is defined as `()`. When defining the `shape` argument, the actual output shape will have a new undefined dimension (i.e. `None` sized) added. In other words, the final output shape will be `[None]`.

Next, we define two embedding layers: a target embedding layer and a context embedding layer. These layers will be used to look up the embeddings for target and context word IDs that will be generated by the input generation function.

```
# Two embeddings layers are used one for the context and one for the  
# target  
target_embedding_layer = tf.keras.layers.Embedding(  
    input_dim=n_vocab, output_dim=embedding_size,  
    name='target_embedding'  
)  
context_embedding_layer = tf.keras.layers.Embedding(  
    input_dim=n_vocab, output_dim=embedding_size,  
    name='context_embedding'  
)
```

With the embedding layers defined, let's look up the embeddings for the word IDs that will be fed to the input layers:

```
# Lookup outputs of the embedding layers  
target_out = target_embedding_layer(input_1)  
context_out = context_embedding_layer(input_2)
```

We now need to compute the dot product of `target_out` and `context_out`.

To do that, we are going to use the `tf.keras.layers.Dot` layer:

```
# Computing the dot product between the two
out = tf.keras.layers.Dot(axes=-1)([context_out, target_out])
```

Finally, we define our model as a `tf.keras.models.Model` object, where we specify `inputs` and `outputs` arguments. `inputs` need to be one or more input layers, and `outputs` can be one or more outputs produced by a series of `tf.keras.layers` objects:

```
# Defining the model
skip_gram_model = tf.keras.models.Model(inputs=[input_1, input_2],
                                         outputs=out, name='skip_gram_model')
```

We compile the model using a loss function and an optimizer:

```
# Compiling the model
skip_gram_model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_
logits=True), optimizer='adam', metrics=['accuracy'])
```

Let's see a summary of our model by calling the following:

```
skip_gram_model.summary()
```

This will output:

```
Model: "skip_gram_model"

-----  

Layer (type)          Output Shape       Param #  Connected to  

-----  

context (InputLayer)    [(None, )]        0  

-----  

target (InputLayer)     [(None, )]        0  

-----  

context_embedding (Embedding) (None, 128)   1920128   context[0][0]  

-----  

target_embedding (Embedding) (None, 128)   1920128   target[0][0]  

-----  

dot (Dot)              (None, 1)         0   context_embedding[0][0]  

                                         target_embedding[0][0]  

-----  

Total params: 3,840,256
```

```
Trainable params: 3,840,256  
Non-trainable params: 0
```

Training and evaluating the model will be the next item on our agenda.

Training and evaluating the model

Our training process is going to be very simple as we have defined a function to generate batches of data in the exact format the model needs them in. But before we go ahead with training the model, we need to think about how we evaluate word vector models. The idea of word vectors is that words sharing semantic similarity will have a smaller distance between them, whereas words with no similarity will be far apart. To compute the similarities between words, we can use the cosine distance. We picked a set of random word IDs and stored them in `valid_term_ids` during our hyperparameter discussion. We will implement a way to compute the closest `k` words to each of those terms at the end of every epoch.

For this, we utilize Keras callbacks. Keras callbacks give you a way to execute some important operation(s) at the end of every training iteration, epoch, prediction step, and so on. You can see a full list of the available callbacks at https://www.tensorflow.org/api_docs/python/tf/keras/callbacks. Since we need a bespoke evaluation mechanism designed for word vectors, we will need to implement our own callback. Our callback will take a list of word IDs intended as the validation words, a model containing the embedding matrix, and a Tokenizer to decode word IDs:

```
class ValidationCallback(tf.keras.callbacks.Callback):  
  
    def __init__(self, valid_term_ids, model_with_embeddings, tokenizer):  
  
        self.valid_term_ids = valid_term_ids  
        self.model_with_embeddings = model_with_embeddings  
        self.tokenizer = tokenizer  
  
        super().__init__()  
  
    def on_epoch_end(self, epoch, logs=None):  
        """ Validation logic """  
  
        # We will use context embeddings to get the most similar words
```

```

# Other strategies include: using target embeddings, mean
# embeddings after averaging context/target
embedding_weights =
    self.model_with_embeddings.get_layer(
        "context_embedding"
    ).get_weights()[0]
normalized_embeddings = embedding_weights /
    np.sqrt(np.sum(embedding_weights**2, axis=1, keepdims=True))

# Get the embeddings corresponding to valid_term_ids
valid_embeddings = normalized_embeddings[self.valid_term_ids,
    :]

# Compute the similarity between valid_term_ids and all the
# embeddings
# V x d (d x D) => V x D
top_k = 5 # Top k items will be displayed
similarity = np.dot(valid_embeddings, normalized_embeddings.T)

# Invert similarity matrix to negative
# Ignore the first one because that would be the same word as the
# probe word
similarity_top_k = np.argsort(-similarity, axis=1)[:, 1:
    top_k+1]

# Print the output
for i, term_id in enumerate(valid_term_ids):

    similar_word_str = ', '.join([self.tokenizer.index_word[j]
        for j in similarity_top_k[i, :] if j > 1])
    print(f'{self.tokenizer.index_word[term_id]}:
{similar_word_str }')

print('\n')

```

The evaluation will be done at the end of a training epoch, therefore we will override the `on_epoch_end()` function. The function extracts the embeddings from the context embedding layer.

Then the embeddings are normalized to have a unit length. Afterward, embeddings corresponding to validation words are extracted to a separate matrix called `valid_embeddings`. Then the cosine distance is computed between the validation embeddings and all word embeddings, which results in a `[valid_size, vocabulary_size]` sized matrix. From this, we extract the top k similar words and display them through `print` statements.

Finally, the model can be trained as follows:

```
skipgram_validation_callback = ValidationCallback(valid_term_ids, skip_gram_model, tokenizer)

for ei in range(epochs):

    print(f"Epoch: {ei+1}/{epochs} started")

    news_skip_gram_gen = skip_gram_data_generator(
        news_sequences, window_size, batch_size, negative_samples,
        n_vocab
    )

    skip_gram_model.fit(
        news_skip_gram_gen, epochs=1,
        callbacks=skipgram_validation_callback,
    )
```

We are simply defining an instance of the callback first. Next, we train the model for several epochs. In each, we generate skip gram data (while shuffling the order of the articles) and call `skip_gram_model.fit()` on the data. Here's the result after five epochs of training:

```
Epoch: 5/5 ended
2233/2233 [=====] - 146s 65ms/step - loss: 0.4842
- accuracy: 0.8056
months: days, weeks, years, detained, meaning
were: are, was, now, davidson, widened
mr: resignation, scott, tony, stead, article
champions: premier, pottage, kampala, danielli, dominique
businesses: medium, port, 2002's, tackling, doug
positive: electorate, proposal, bolz, visitors', strengthen
pop: 'me', style, lacks, tourism, tuesdays
```

Here, we denote some of the most sensible word vectors learned. For example, we can see that two of the most similar words to the word “months” are “days” and “weeks”. The title “mr” is accompanied by male names such as “scott” and “tony”. The word “premier” appears as a similar word to “champion”. You can further experiment with:

- Different negative candidate sampling methods available at https://www.tensorflow.org/api_docs/python/tf/random
- Different hyperparameter choices (such as the embedding size and the number of negative samples)

In this section, we discussed the skip-gram algorithm from end to end. We saw how we can use functions in TensorFlow to transform data. Then we implemented the skip-gram architecture using layers in Keras and the Functional API. Finally, we trained the model and visually inspected its performance on some test data. We will now discuss another popular Word2vec algorithm known as the **Continuous Bag-of-Words (CBOW)** model.

The Continuous Bag-of-Words algorithm

The CBOW model works in a similar way to the skip-gram algorithm, with one significant change in the problem formulation. In the skip-gram model, we predict the context words from the target word. However, in the CBOW model, we predict the target word from contextual words. Let’s compare what data looks like for the skip-gram algorithm and the CBOW model by taking the previous example sentence:

The dog barked at the mailman.

For the skip-gram algorithm, the data tuples—(*input word, output word*)—might look like this:

(*dog, the*), (*dog, barked*), (*barked, dog*), and so on

For CBOW, the data tuples would look like the following:

([*the, barked*], *dog*), ([*dog, at*], *barked*), and so on

Consequently, the input of the CBOW has a dimensionality of $2 \times m \times D$, where m is the context window size and D is the dimensionality of the embeddings. The conceptual model of CBOW is shown in *Figure 3.13*:



Figure 3.8: The CBOW model

We will not go into great detail about the intricacies of CBOW as it is quite similar to skip-gram. For example, once the embeddings are aggregated (that is, concatenated or summed), they flow through a softmax layer to finally compute the same loss as we did with the skip-gram algorithm. However, we will discuss the algorithm's implementation (though not in depth) to get a clear understanding of how to properly implement CBOW. The full implementation of CBOW is available at `ch3_word2vec.ipynb` in the `Ch03-Word-Vectors` exercise folder.

Generating data for the CBOW algorithm

Unfortunately, unlike for the skip-gram algorithm, we do not have a handy function to generate data for the CBOW algorithm at our disposal. Therefore, we will need to implement this function ourselves.

You can find the implementation of this function (named `cbow_grams()`) in `ch3_word2vec.ipynb` in the Ch03-Word-Vectors folder. The procedure will be quite similar to the one we used for skip-grams. However, the format of the data will be slightly different. Therefore, we will discuss the format of the data returned by this function.

The function takes the same arguments as the `skip_gram_data_generator()` function we discussed earlier:

- `sequences (List[List[int]])` – A list of list of word IDs. This is the output generated by Tokenizer's `texts_to_sequences()` function.
- `window_size (int)` – The window size for the context.
- `batch_size (int)` – The batch size.
- `negative_samples (int)` – The number of negative samples per example to generate.
- `vocabulary_size (int)` – The vocabulary size.
- `seed` – The random seed.

The data returned also has a slightly different format. It will return a batch of data containing:

- A batch of target word IDs, these target words are both positive and negative.
- A batch of corresponding context word IDs. Unlike skip-grams, for CBOW, we need all the words in the context, not just one. For example, if we define a batch size of b and window size of w , this will be a $[b, 2w]$ sized tensor.
- A batch of labels (0 and 1).

We will now learn about the specifics of the algorithm.

Implementing CBOW in TensorFlow

We will use the same hyperparameters as before:

```
batch_size = 4096 # Data points in a single batch

embedding_size = 128 # Dimension of the embedding vector.

window_size=1 # We use a window size of 1 on either side of target word
epochs = 5 # Number of epochs to train for
negative_samples = 4 # Number of negative samples generated per example

# We pick a random validation set to sample nearest neighbors
```

```
valid_size = 16 # Random set of words to evaluate similarity on.  
# We sample valid datapoints randomly from a large window without always  
# being deterministic  
valid_window = 250  
  
# When selecting valid examples, we select some of the most frequent words  
# as well as some moderately rare words as well  
np.random.seed(54321)  
random.seed(54321)  
  
valid_term_ids = np.array(random.sample(range(valid_window), valid_size))  
valid_term_ids = np.append(  
    valid_term_ids, random.sample(range(1000, 1000+valid_window),  
    valid_size),  
    axis=0  
)
```

Just as before, let's first clear out any remaining sessions, if there are any:

```
import tensorflow.keras.backend as K  
K.clear_session()
```

We define two input layers. Note how the second input layer is defined to have $2 \times \text{window_size}$ dimensions. This means the final shape of that layer will be $[\text{None}, 2 \times \text{window_size}]$:

```
# Inputs  
input_1 = tf.keras.layers.Input(shape=())  
input_2 = tf.keras.layers.Input(shape=(window_size*2,))
```

Let's now define two embedding layers: one for the context words and one for the target words. We will feed the inputs from the input layers and produce `context_out` and `target_out`:

```
context_embedding_layer = tf.keras.layers.Embedding(  
    input_dim=n_vocab+1, output_dim=embedding_size,  
    name='context_embedding'  
)  
  
target_embedding_layer = tf.keras.layers.Embedding(  
    input_dim=n_vocab+1, output_dim=embedding_size,
```

```

        name='target_embedding'
    )

context_out = context_embedding_layer(input_2)
target_out = target_embedding_layer(input_1)

```

If you look at the shape of `context_out`, you will see that it has the shape [None, 2, 128], where 2 is `2 * window_size`, due to taking the whole context around a word. This needs to be reduced to [None, 128] by taking the average of all the context words. This is done by using a Lambda layer:

```

mean_context_out = tf.keras.layers.Lambda(lambda x: tf.reduce_mean(x,
axis=1))(context_out)

```

We pass a Lambda function to the `tf.keras.layers.Lambda` layer to reduce the `context_out` tensor on the second dimension to produce a [None, 128] sized tensor. With both the `target_out` and `mean_context_out` tensors having the shape [None, 128], we can compute the dot product of the two to produce an output tensor [None, 1]:

```

out = tf.keras.layers.Dot(axes=-1)([context_out, target_out])

```

With that, we can define the final model as follows:

```

cbow_model = tf.keras.models.Model(inputs=[input_1, input_2], outputs=out,
name='cbow_model')

```

Similar to `skip_gram_model`, we will compile `cbow_model` as follows:

```

cbow_model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    optimizer='adam',
    metrics=['accuracy']
)

```

Again, if you would like to see the summary of the model, you can run `cbow_model.summary()`.

Training and evaluating the model

The model training is identical to how we trained the skip-gram model. First, let's define a callback to find the top k words similar to the words defined in the `valid_term_ids` set:

```

cbow_validation_callback = ValidationCallback(valid_term_ids, cbow_model,
tokenizer)

```

Next, we train cbow_model for several epochs:

```
for ei in range(epochs):
    print(f"Epoch: {ei+1}/{epochs} started")
    news_cbow_gen = cbow_data_generator(
        news_sequences,
        window_size,
        batch_size,
        negative_samples
    )
    cbow_model.fit(
        news_cbow_gen,
        epochs=1,
        callbacks=cbow_validation_callback,
    )
```

The output should look like the following. We have cherry-picked some of the most sensible word vectors learned:

```
months: years, days, weeks, minutes, seasons
you: we, they, i, don't, we'll
were: are, aren't, have, because, need
music: terrestrial, cameras, casual, divide, camera
also: already, previously, recently, rarely, reportedly
best: supporting, actress, category, fiction, contenders
him: them, me, themselves, won't, censors
mr: tony, gordon, resignation, cherie, jack
5bn: 5m, 7bn, 4bn, 8bn, 8m
champions: premier, rugby, appearances, irish, midfielder
deutsche: austria, austria's, butcher, violence, 1989
files: movies, collections, vast, habit, ballad
pop: fiction, veteran, scrubs, wars, commonwealth
```

From visual inspection, it seems CBOW has learned some effective word vectors. Similar to the skip-gram model, it has picked words like “years” and “days” as similar to “months”. Numerical values such as “5bn” have “5m” and “7bn” around them. But it’s important to remember that visual inspection is just a quick and dirty way to evaluate word vectors.

Typically, word vectors are evaluated on some downstream tasks. One of the popular tasks is the word analogical reasoning task. It focuses on answering questions like:

Athens is to Greece as Baghdad to ____

The answer is Iraq. How is the answer computed? If the word vectors are sensible, then:

$$\text{Word2vec(Athens)} - \text{Word2vec(Greece)} = \text{Word2vec(Baghdad)} - \text{Word2vec(Iraq)}$$

or

$$\text{Word2vec(Iraq)} = \text{Word2vec(Baghdad)} - \text{Word2vec(Athens)} + \text{Word2vec(Greece)}$$

The answer is computed as the vector given by $\text{Word2vec(Baghdad)} - \text{Word2vec(Athens)} + \text{Word2vec(Greece)}$. The next step for this analogy task would be to see if the most similar vector to the resulting vector is given by the word Iraq. This way, accuracy can be computed for an analogy reasoning task. However, we will not utilize this task in this chapter, as our dataset is not big enough to perform well in this task.

Here, we conclude our discussion on the CBOW algorithm. Though CBOW shares similarities with the skip-gram algorithm, it had some architectural differences as well as differences in data.

Summary

Word embeddings have become an integral part of many NLP tasks and are widely used for tasks such as machine translation, chatbots, image caption generation, and language modeling. Not only do word embeddings act as a dimensionality reduction technique (compared to one-hot encoding), they also give a richer feature representation than other techniques. In this chapter, we discussed two popular neural-network-based methods for learning word representations, namely the skip-gram model and the CBOW model.

First, we discussed the classical approaches to this problem to develop an understanding of how word representations were learned in the past. We discussed various methods, such as using WordNet, building a co-occurrence matrix of the words, and calculating TF-IDF.

Next, we explored neural-network-based word representation learning methods. First, we worked out an example by hand to understand how word embeddings or word vectors can be calculated to help us understand the computations involved.

Next, we discussed the first word-embedding learning algorithm—the skip-gram model. We then learned how to prepare the data to be used for learning. Later, we examined how to design a loss function that allows us to use word embeddings using the context words of a given word. Finally, we discussed how to implement the skip-gram algorithm using TensorFlow.

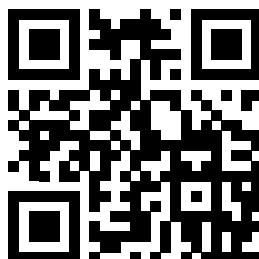
Then we reviewed the next choice for learning word embeddings—the CBOW model. We also discussed how CBOW differs from the skip-gram model. Finally, we discussed a TensorFlow implementation of CBOW as well.

In the next chapter, we will learn several other word embedding learning techniques known as Global Vectors, or GloVe, and Embeddings from Language Models, or ELMo.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside
more than 1000 members at: <https://packt.link/nlp>



4

Advanced Word Vector Algorithms

In *Chapter 3, Word2vec – Learning Word Embeddings*, we introduced you to Word2vec, the basics of learning word embeddings, and the two common Word2vec algorithms: skip-gram and CBOW. In this chapter, we will discuss several other word vector algorithms:

- GloVe – Global Vectors
- ELMo – Embeddings from Language Models
- Document classification with ELMo

First, you will learn a word embedding learning technique known as **Global Vectors (GloVe)** and the specific advantages that GloVe has over skip-gram and CBOW.

You will also look at a recent approach for representing language called **Embeddings from Language Models (ELMo)**. ELMo has an edge over other algorithms as it is able to disambiguate words, as well as capture semantics. Specifically, ELMo generates “contextualized” word representations, by using a given word along with its surrounding words, as opposed to treating word representations independently, as in skip-gram or CBOW.

Finally, we will solve an exciting use-case of document classification using our newly founded ELMo vectors.

GloVe – Global Vectors representation

One of the main limitations of skip-gram and CBOW algorithms is that they can only capture local contextual information, as they only look at a fixed-length window around a word. There's an important part of the puzzle missing here as these algorithms do not look at global statistics (by global statistics we mean a way for us to see all the occurrences of words in the context of another word in a text corpus).

However, we have already studied a structure that could contain this information in *Chapter 3, Word2vec – Learning Word Embeddings*: the co-occurrence matrix. Let's refresh our memory on the co-occurrence matrix, as GloVe uses the statistics captured in the co-occurrence matrix to compute vectors.

Co-occurrence matrices encode the context information of words, but they require maintaining a $V \times V$ matrix, where V is the size of the vocabulary. To understand the co-occurrence matrix, let's take two example sentences:

- *Jerry and Mary are friends.*
- *Jerry buys flowers for Mary.*

If we assume a context window of size 1, on each side of a chosen word, the co-occurrence matrix will look like the following (we only show the upper triangle of the matrix, as the matrix is symmetric):

	Jerry	and	Mary	are	friends	buys	flowers	for
Jerry	0	1	0	0	0	1	0	0
and		0	1	0	0	0	0	0
Mary			0	1	0	0	0	1
are				0	1	0	0	0
friends					0	0	0	0
buys						0	1	0
flowers							0	1
for								0

We can see that this matrix shows us how a word in a corpus is related to any other word, hence it contains global statistics about the corpus. That said, what are some of the advantages of having a co-occurrence matrix, as opposed to seeing just the local context?

- It provides you with additional information about the characteristics of the words. For example, if you consider the sentence “the cat sat on the mat,” it is difficult to say if “the” is a special word that appears in the context of words such as “cat” or “mat.” However, if you have a large-enough corpus and a co-occurrence matrix, it’s very easy to see that “the” is a frequently occurring stop word.
- The co-occurrence matrix recognizes the repeating usages of contexts or phrases, whereas in the local context this information is ignored. For example, in a large enough corpus, “New York” will be a clear winner, showing that the two words appear in the same context many times.

It is important to keep in mind that Word2vec algorithms use various techniques to approximately inject some word co-occurrence patterns, while learning word vectors. For example, the sub-sampling technique we used in the previous chapter (i.e. sampling lower-frequency words more) helps to detect and avoid stop words. But they introduce additional hyperparameters and are not as informative as the co-occurrence matrix.



Using global statistics to come up with word representations is not a new concept. An algorithm known as **Latent Semantic Analysis (LSA)** has been using global statistics in its approach.

LSA is used as a document analysis technique that maps words in the documents to something known as a **concept**, a common pattern of words that appears in a document. Global matrix factorization-based methods efficiently exploit the global statistics of a corpus (for example, co-occurrence of words in a global scope), but have been shown to perform poorly at word analogy tasks. On the other hand, context window-based methods have been shown to perform well at word analogy tasks, but do not utilize global statistics of the corpus, leaving space for improvement. GloVe attempts to get the best of both worlds—an approach that efficiently leverages global corpus statistics while optimizing the learning model in a context window-based manner similar to skip-gram or CBOW.

GloVe, a new technique for learning word embeddings was introduced in the paper “GloVe: Global Vectors for Word Representation” by Pennington et al. (<https://nlp.stanford.edu/pubs/glove.pdf>). GloVe attempts to bridge the gap of missing global co-occurrence information in Word2vec algorithms. The main contribution of GloVe is a new cost function (or an objective function) that uses the valuable statistics available in the co-occurrence matrix. Let’s first understand the motivation behind the GloVe method.

Understanding GloVe

Before looking at the implementation details of GloVe, let's take time to understand the concepts governing the computations in GloVe. To do so, let's consider an example:

- Consider word $i=Ice$ and $j=Steam$
- Define an arbitrary probe word k
- Define P_{ik} to be the probability of words i and k occurring close to each other, and P_{jk} to be the words j and k occurring together

Now let's look at how the P_{ik}/P_{jk} entity behaves with different values for k .

For $k = \text{"Solid"}$, it is highly likely to appear with i , thus, P_{ik} will be high. However, k would not often appear along with j causing a low P_{jk} . Therefore, we get the following expression:

$$P_{ik}/P_{jk} \gg 1$$

Next, for $k = \text{"gas"}$, it is unlikely to appear in the close proximity of i and therefore will have a low P_{ik} ; however, since k highly correlates with j , the value of P_{jk} will be high. This leads to the following:

$$P_{ik}/P_{jk} \approx 0$$

Now, for words such as $k = \text{"water"}$, which has a strong relationship with both i and j , or $k = \text{"Fashion"}$, which i and j both have minimal relevance to, we get this:

$$P_{ik}/P_{jk} \approx 1$$

If you assume we have learned sensible word embeddings for these words, these relationships can be visualized in a vectors space to understand why the ratio P_{ik}/P_{jk} behaves this way (see *Figure 4.1*). In the figure below, the solid arrow shows the distance between the words (i, j) , whereas the dashed lines express the distance between the words, (i, k) and (j, k) . These distances can then be associated with the probability values we discussed. For example, when $i = \text{"ice"}$ and $k = \text{"solid"}$, we expect their vectors to have a shorter distance between them (i.e. more frequently co-occurring). Therefore, we can associate distance between (i, k) as the inverse of P_{ik} (i.e. $1/P_{ik}$) due to the definition of P_{ik} . This diagram shows how these distances vary as the probe word k changes:

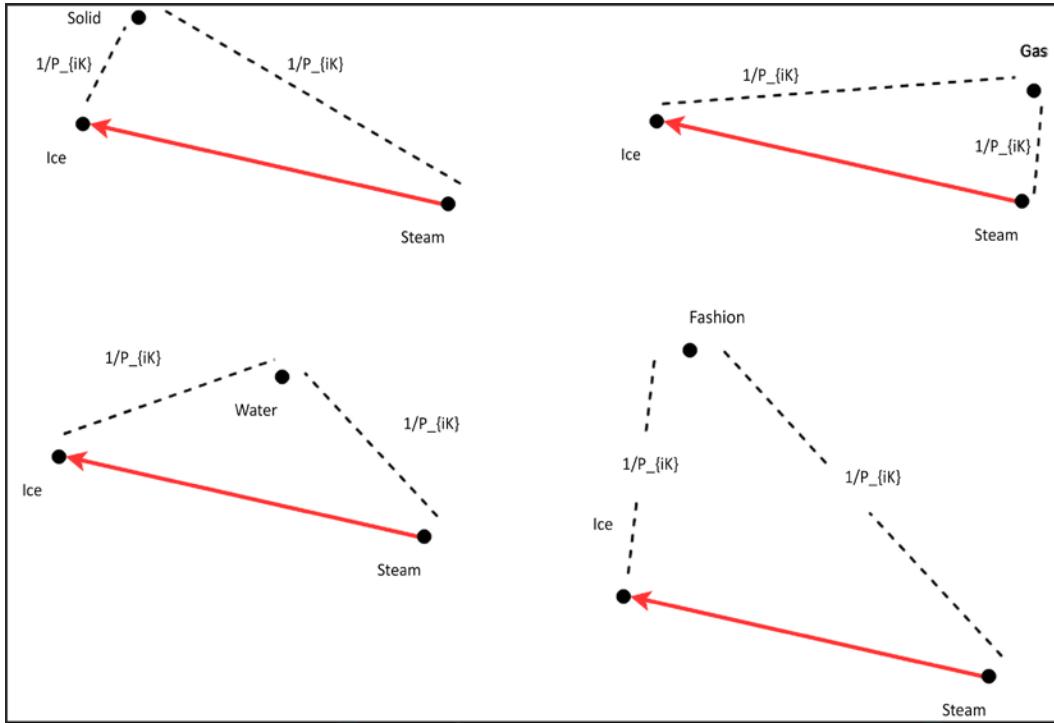


Figure 4.1: How the entities P_{ik} and P_{jk} behave as the probe word changes in proximity to the words i and j

It can be seen that the P_{ik}/P_{jk} entity, which is calculated by measuring the frequency of two words appearing close to each other, behaves in different ways as the relationship between the three words changes. As a result, it becomes a good candidate for learning word vectors. Therefore, a good starting point for defining the loss function will be as shown here:

$$F(w_i, w_j, \tilde{w}_k) = P_{ik}/P_{jk}$$

Here, F is some function and w and \tilde{w} are two different embedding spaces we'll be using. In other words, the words i and j are looked up from one embedding space, whereas the probe word k is looked up from another. From this point, the original paper goes through the derivation meticulously to reach the following loss function:

$$J = \sum_{i,j=1}^V f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij}))^2$$

We will not go through the derivation here, as that's out of scope for this book. Rather we will use the derived loss function and implement the algorithm with TensorFlow. If you need a less mathematically dense explanation of how we can derive this cost function, please refer to the author-written article at <https://towardsdatascience.com/light-on-math-ml-intuitive-guide-to-understanding-glove-embeddings-b13b4f19c010>.

Here, $f(x)$ is defined as $f(x) = (x/x_{max})^{(3/4)}$, if $x < x_{max}$, else 1, where X_{ij} is the frequency with which the word j appeared in the context of the word i . x_{max} is a hyperparameter we set. Remember that we defined two embedding spaces w and \tilde{w} in our loss function. w_i and b_i represent the word embedding and the bias embedding for the word i obtained from embedding space w , respectively. And, \tilde{w}_j and \tilde{b}_j represent the word embedding and bias embedding for word j obtained from embedding space \tilde{w} , respectively. Both these embeddings behave similarly except for the randomization at the initialization. At the evaluation phase, these two embeddings are added together, leading to improved performance.

Implementing GloVe

In this subsection, we will discuss the steps for implementing GloVe. The full code is available in the `ch4_glove.ipynb` exercise file located in the `ch4` folder.

First, we'll define the hyperparameters as we did in the previous chapter:

```
batch_size = 4096 # Data points in a single batch

embedding_size = 128 # Dimension of the embedding vector.

window_size=1 # We use a window size of 1 on either side of target word

epochs = 5 # Number of epochs to train for

# We pick a random validation set to sample nearest neighbors
valid_size = 16 # Random set of words to evaluate similarity on.
# We sample valid datapoints randomly from a large window without always
# being deterministic
valid_window = 250

# When selecting valid examples, we select some of the most frequent words
# as well as some moderately rare words as well
```

```
np.random.seed(54321)
random.seed(54321)

valid_term_ids = np.array(random.sample(range(valid_window), valid_size))
valid_term_ids = np.append(
    valid_term_ids, random.sample(range(1000, 1000+valid_window), valid_
    size),
    axis=0
)
```

The hyperparameters you define here are the same hyperparameters we defined in the previous chapter. We have a batch size, embedding size, window size, the number of epochs, and, finally, a set of held-out validation word IDs that we will print the most similar words to.

We will then define the model. First, we will import a few things we will need down the line:

```
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Input, Embedding, Dot, Add
from tensorflow.keras.models import Model

K.clear_session()
```

The model is going to have two input layers: `word_i` and `word_j`. They represent a batch of context words and a batch of target words (or a batch of positive skip-grams):

```
# Define two input layers for context and target words
word_i = Input(shape=())
word_j = Input(shape=())
```

Note how the shape is defined. The shape is defined as an empty tuple. This means the final shape of `word_i` and `word_j` would be `[None]`, meaning it will take a vector of an arbitrary number of elements as the input.

Next, we are going to define the embedding layers. There will be four embedding layers:

- `embeddings_i` – The context embedding layer
- `embeddings_j` – The target embedding layer
- `b_i` – The context embedding bias
- `b_j` – The target embedding bias

The following code defines these:

```
# Each context and target has their own embeddings (weights and biases)
# Embedding weights
embeddings_i = Embedding(n_vocab, embedding_size, name='target_embedding')(word_i)
embeddings_j = Embedding(n_vocab, embedding_size,
name='context_embedding')(word_j)

# Embedding biases
b_i = Embedding(n_vocab, 1, name='target_embedding_bias')(word_i)
b_j = Embedding(n_vocab, 1, name='context_embedding_bias')(word_j)
```

Next, we are going to compute the output. The output of this model will be:

$$w_i^T \tilde{w}_j + b_i + \tilde{b}_j$$

As you can see, that's a portion of our final loss function. We have all the right ingredients to compute this result:

```
# Compute the dot product between embedding vectors (i.e. w_i.w_j)
ij_dot = Dot(axes=-1)([embeddings_i,embeddings_j])

# Add the biases (i.e. w_i.w_j + b_i + b_j )
pred = Add()([ij_dot, b_i, b_j])
```

First we will use the `tensorflow.keras.layers.Dot` layer to compute the dot product batch-wise between the context embedding lookup (`embeddings_i`) and the target embedding lookup (`embeddings_j`). For example, the two inputs to the Dot layer will be of size [batch size, embedding size]. After the dot product, the output `ij_dot` will be [batch size, 1], where `ij_dot[k]` will be the dot product between `embeddings_i[k, :]` and `embeddings_j[k, :]`. Then we simply add `b_i` and `b_j` (which has shape [None, 1]) element-wise to `ij_dot`.

Finally, the model is defined as taking `word_i` and `word_j` as inputs and outputting `pred`:

```
# The final model
glove_model = Model(
    inputs=[word_i, word_j],outputs=pred,
name='glove_model')
```

Next, we are going to do something quite important.

We have to devise a way to compute the complex loss function defined above, using various components/functionality available in a model. First let's revisit the loss function.

$$J = \sum_{i,j=1}^V f(X_{ij}) (w_i^T \tilde{w}_j + b_i + b_j - \log(X_{ij}))^2$$

where,

$$f(x) = (x/x_{max})^{(3/4)}, \text{ if } x < x_{max}, \text{ else } 1.$$

Although it looks complex, we can use already existing loss functions and other functionality to implement the GloVe loss. You can abstract this loss function into three components as shown in the image below:

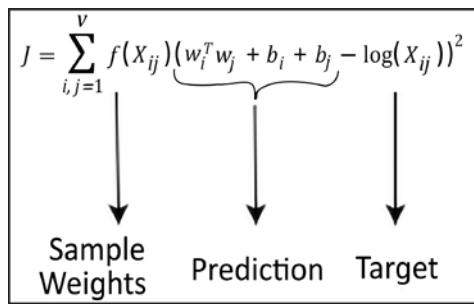


Figure 4.2: The breakdown of the GloVe loss function showing how predictions, targets, and weights interact with each other to compute the final loss

Therefore, if sample weights are denoted by W , predictions are denoted by \hat{Y} , and true targets are denoted by Y , then we can write the loss as:

$$J = W(\hat{Y} - Y)^2$$

This is simply a weighted mean squared loss. Therefore, we will use "`mse`" as the loss for our model:

```
# Glove has a specific Loss function with a sound mathematical
# underpinning
# It is a form of mean squared error
glove_model.compile(loss="mse", optimizer = 'adam')
```

We will later see how we can feed in sample weights to the model to complete the loss function. So far, we have defined different components of the GloVe algorithm and compiled the model. Next, we are going to have a look at how data can be generated to train the GloVe model.

Generating data for GloVe

The dataset we will be using is the same as the dataset from the previous chapter. To recap, we will be using the BBC news articles dataset available at <http://mlg.ucd.ie/datasets/bbc.html>. It contains 2225 news articles belonging to 5 topics, business, entertainment, politics, sport, and tech, which were published on the BBC website between 2004 and 2005.

Let's now generate the data. We will be encapsulating the data generation in a function called `glove_data_generator()`. As the first step, let us write a function signature:

```
def glove_data_generator(
    sequences, window_size, batch_size, vocab_size, cooccurrence_matrix,
    x_max=100.0, alpha=0.75, seed=None
):
```

The function takes several arguments:

- `sequences` (`List[List[int]]`) – a list of a list of word IDs. This is the output generated by tokenizer's `texts_to_sequences()` function.
- `window_size` (`int`) – Window size for the context.
- `batch_size` (`int`) – Batch size.
- `vocab_size` (`int`) – Vocabulary size.
- `cooccurrence_matrix` (`scipy.sparse.lil_matrix`) – A sparse matrix containing co-occurrences of words.
- `x_max` (`int`) – Hyperparameter used by GloVe to compute sample weights.
- `alpha` (`float`) – Hyperparameter used by GloVe to compute sample weights.
- `seed` – The random seed.

It also has several outputs:

- A batch of (target, context) word ID tuples
- The corresponding $\log(X_{ij})$ values for the (target, context) tuples
- Sample weights (i.e. $f(X_{ij})$) values for the (target, context) tuples

First we will shuffle the order of news articles:

```
# Shuffle the data so that, every epoch, the order of data is
# different
rand_sequence_ids = np.arange(len(sequences))
np.random.shuffle(rand_sequence_ids)
```

Next, we will create the sampling table, so that we can use sub-sampling to avoid over-sampling common words (e.g. stop words):

```
sampling_table =  
    tf.keras.preprocessing.sequence.make_sampling_table(vocab_size)
```

With that, for every sequence (i.e. list of word IDs) representing an article, we generate positive skip-grams. Note how we are keeping `negative_samples=0.0` as, unlike skip-gram or CBOW algorithms, GloVe does not rely on negative candidates:

```
# For each story/article  
for si in rand_sequence_ids:  
  
    # Generate positive skip-grams while using sub-sampling  
    positive_skip_grams, _ = tf.keras.preprocessing.sequence.  
        skipgrams(  
            sequences[si],  
            vocabulary_size=vocab_size,  
            window_size>window_size,  
            negative_samples=0.0,  
            shuffle=False,  
            sampling_table=sampling_table,  
            seed=seed  
        )
```

With that, we first break down the skip-gram tuples into two lists, one containing targets and the other containing context words, and convert them to NumPy arrays subsequently:

```
# Take targets and context words separately  
targets, context = zip(*positive_skip_grams)  
targets, context = np.array(targets).ravel(),  
np.array(context).ravel()
```

We then index the positions given by the (target, context) word pairs, from the co-occurrence matrix to retrieve the corresponding X_{ij} values, where (i,j) represents a (target, context) pair:

```
x_ij = np.array(cooccurrence_matrix[targets,  
    context].toarray()).ravel()
```

Then we compute a corresponding $\log(X_{ij})$ (denoted by `log_x_ij`) and $f(X_{ij})$ (denoted by `sample_weights`):

```
# Compute Log - Introducing an additive shift to make sure we
# don't compute Log(0)
log_x_ij = np.log(x_ij + 1)

# Sample weights
# if x < x_max => (x/x_max)**alpha / else => 1
sample_weights = np.where(x_ij < x_max, (x_ij/x_max)**alpha, 1)
```

If a code is not chosen, a random seed is set. Afterward, all of `context`, `targets`, `log_x_ij`, and `sample_weights` are shuffled while maintaining the correspondence of elements between the arrays:

```
# If seed is not provided generate a random one
if not seed:
    seed = random.randint(0, 10e6)

# Shuffle data
np.random.seed(seed)
np.random.shuffle(context)
np.random.seed(seed)
np.random.shuffle(targets)
np.random.seed(seed)
np.random.shuffle(log_x_ij)
np.random.seed(seed)
np.random.shuffle(sample_weights)
```

Finally, we iterate through batches of the data we created above. Each batch will consist of

- A batch of (target, context) word ID tuples
- The corresponding $\log(X_{ij})$ values for the (target, context) tuples
- Sample weights (i.e. $f(X_{ij})$) values for the (target, context) tuples

in that order.

```
# Generate a batch or data in the format
# ((target words, context words), log(X_ij) <- true targets,
# f(X_ij) <- sample weights)
```

```
for eg_id_start in range(0, context.shape[0], batch_size):
    yield (
        targets[eg_id_start: min(eg_id_start+batch_size,
        targets.shape[0])],
        context[eg_id_start: min(eg_id_start+batch_size,
        context.shape[0])],
        log_x_ij[eg_id_start: min(eg_id_start+batch_size,
        log_x_ij.shape[0])],
        sample_weights[eg_id_start: min(eg_id_start+batch_size,
        sample_weights.shape[0])]
```

Now that the data is ready to be pumped in, let's discuss the final piece of the puzzle: training the model.

Training and evaluating GloVe

Training the model is effortless, as we have all the components to train the model. As the first step, we will reuse the `ValidationCallback` we created in *Chapter 3, Word2vec – Learning Word Embeddings*. To recap, `ValidationCallback` is a Keras callback. Keras callbacks give you a way to execute some important operation(s) at the end of every training iteration, epoch, prediction step, etc. Here we are using the callback to perform a validation step at the end of every epoch. Our callback would take a list of word IDs intended as the validation words (held out in `valid_term_ids`), the model containing the embedding matrix, and a tokenizer to decode word IDs. Then it will compute the most similar top-k words for every word in the validation word set and print that as the output:

```
glove_validation_callback = ValidationCallback(valid_term_ids, glove_
model, tokenizer)

# Train the model for several epochs
for ei in range(epochs):

    print("Epoch: {} / {} started".format(ei+1, epochs))

    news_glove_data_gen = glove_data_generator(
        news_sequences, window_size, batch_size, n_vocab
    )
```

```

glove_model.fit(
    news_glove_data_gen, epochs=1,
    callbacks=glove_validation_callback,
)

```

You should get a sensible-looking output once the model has finished training. Here are some of the cherry-picked results:

```

election: attorney, posters, forthcoming, november's, month's
months: weeks, years, nations, rbs, thirds
you: afford, we, they, goodness, asked
music: cameras, mp3, hp's, refuseniks, divide
best: supporting, category, asante, counterparts, actor
mr: ron, tony, bernie, jack, 63
leave: pay, need, unsubstantiated, suited, return
5bn: 8bn, 2bn, 1bn, 3bn, 7bn
debut: solo, speakerboxxx, youngster, nasty, toshack
images: 117, pattern, recorder, lennon, unexpectedly
champions: premier, celtic, football, representatives, neighbour
individual: extra, attempt, average, improvement, survived
businesses: medium, sell, redder, abusive, handedly
deutsche: central, austria's, donald, ecb, austria
machine: unforced, wireless, rapid, vehicle, workplace

```

You can see that words like “months,” “weeks,” and “years” are grouped together. Numbers like “5bn,” “8bn,” and “2bn” are grouped together as well. “Deutsche” is surrounded by “Austria’s” and “Austria.” Finally, we will save the embeddings to the disk. We will combine weights and the bias of each context and target vector space to a single array, where the last column of the array will represent the bias and save it to the disk:

```

def save_embeddings(model, tokenizer, vocab_size, save_dir):

    os.makedirs(save_dir, exist_ok=True)

    _, words_sorted = zip(*sorted(list(tokenizer.index_word.items()),
        key=lambda x: x[0])[:vocab_size-1])

    words_sorted = [None] + list(words_sorted)

```

```
context_embedding_weights = model.get_layer("context_embedding").get_
weights()[0]
context_embedding_bias = model.get_layer("context_embedding_bias").
get_weights()[0]
context_embedding = np.concatenate([context_embedding_weights,
context_embedding_bias], axis=1)

target_embedding_weights = model.get_layer("target_embedding").get_
weights()[0]
target_embedding_bias = model.get_layer("target_embedding_bias").get_
weights()[0]
target_embedding = np.concatenate([target_embedding_weights, target_
embedding_bias], axis=1)

pd.DataFrame(
    context_embedding,
    index = words_sorted
).to_pickle(os.path.join(save_dir, "context_embedding_and_bias.pkl"))

pd.DataFrame(
    target_embedding,
    index = words_sorted
).to_pickle(os.path.join(save_dir, "target_embedding_and_bias.pkl"))

save_embeddings(glove_model, tokenizer, n_vocab, save_dir='glove_
embeddings')
```

We will save embeddings as pandas DataFrames. First we get all the words sorted by their IDs. We subtract 1 to discount the reserved word ID 0 as we'll add that manually, in the following line. Note that, word ID 0 will not show up in `tokenizer.index_word`. Next we get the required layers by name (namely, `context_embedding`, `target_embedding`, `context_embedding_bias` and `target_embedding_bias`). Once we have the layers we can use the `get_weights()` function to retrieve weights.

In this section, we looked at GloVe, another word embedding learning technique.

The main advantage of GloVe over the Word2vec techniques discussed in *Chapter 3, Word2vec – Learning Word Embeddings*, is that it pays attention to both global and local statistics of the corpus to learn embeddings. As GloVe is able to capture the global information about words, it tends to give better performance, especially when the corpus size increases. Another advantage is that, unlike in Word2vec techniques, GloVe does not approximate the cost function (for example, Word2vec using negative sampling), but calculates the true cost. This leads to better and easier optimization of the loss.

In the next section, we are going to look at one more word vector algorithm known as **Embeddings from Language Models (ELMo)**.

ELMo – Taking ambiguities out of word vectors

So far, we've looked at word embedding algorithms that can give only a unique representation of the words in the vocabulary. However, they will give a constant representation for a given word, no matter how many times you query. Why would this be a problem? Consider the following two phrases:

I went to the bank to deposit some money

and

I walked along the river bank

Clearly, the word “bank” is used in two totally different contexts. If you use a vanilla word vector algorithm (e.g. skip-gram), you can only have one representation for the word “bank”, and it is probably going to be muddled between the concept of a financial institution and the concept of walkable edges along a river, depending on the references to this word found in the corpus it's trained on. Therefore, it is more sensible to provide embeddings for a word while preserving and leveraging the context around it. This is exactly what ELMo is striving for.

Specifically, ELMo takes in a sequence, as opposed to a single token, and provides contextualized representations for each token in the sequence. *Figure 4.3* depicts various components encompassing the model. The first thing to understand is that ELMo is a complicated beast! There are lots of neural network models orchestrating in ELMo to produce the output. Particularly, the model uses:

- A character embedding layer (an embedding vector for each character).
- A **convolutional neural network (CNN)** – a CNN consists of many convolutional layers followed by an optional fully connected classification layer.

A convolution layer takes in a sequence of inputs (e.g. sequence of characters in a word) and moves a window of weights over the input to generate a latent representation. We will discuss CNNs in detail in the coming chapters.

- Two bi-directional LSTM layers – an LSTM is a type of model that is used to process time-series data. Given a sequence of inputs (e.g. sequence of word vectors), an LSTM goes from one input to the other, on the time dimension, and produces an output at each position. Unlike fully connected networks, LSTMs have memory, meaning the output at the current position will be affected by what the LSTM has seen in the past. We will discuss LSTMs in detail in the coming chapters.

The specifics of these different components are outside the scope of this chapter. They will be discussed in detail in the coming chapters. Therefore, do not worry if you do not understand the exact mechanisms of the sub-components shown here (*Figure 4.3*).

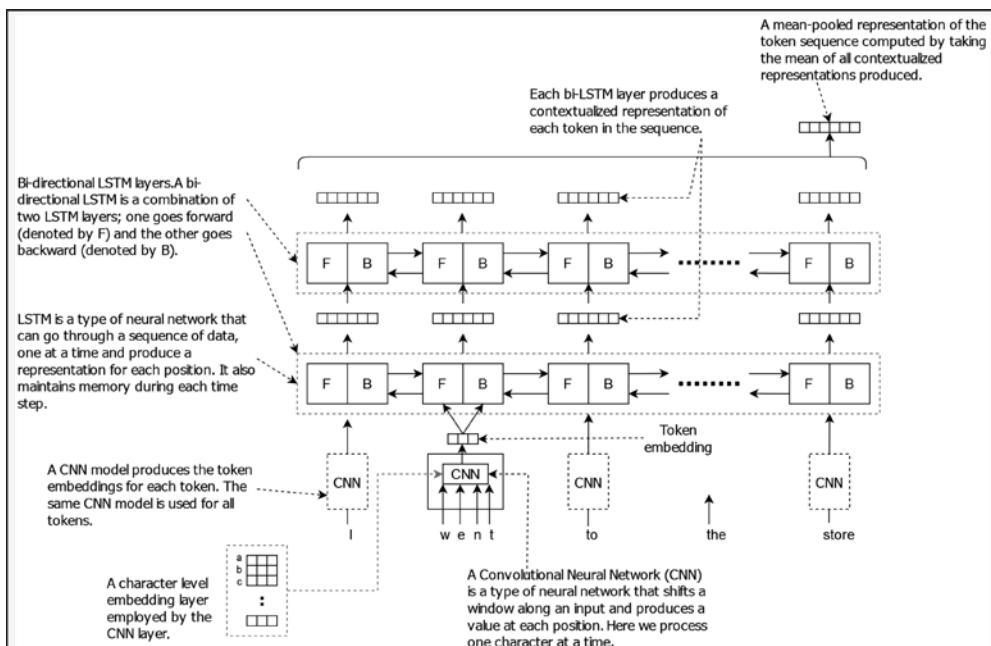


Figure 4.3: Different components of the ELMo model. Token embeddings are generated using a type of neural network known as a CNN. These token embeddings are fed to an LSTM model (that can process time-series data). The output of the first LSTM model is fed to a second LSTM model to generate a latent contextualized representation for each token

We can download a pretrained ELMo model from TensorFlow Hub (<https://tfhub.dev>). TF Hub is a repository for various pretrained models.

It hosts models for tasks such as image classification, text classification, text generation, etc. You can go to the site and browse various available models.

Downloading ELMo from TensorFlow Hub

The ELMo model we will be using is found at <https://tfhub.dev/google/elmo/3>. It has been trained on a very large corpus of text to solve a task known as language modeling. In language modeling, we try to predict the next word given the previous sequence of tokens. We will learn more about language modeling in the coming chapters.

Before downloading the model, let's set the following environment variables:

```
# Not allocating full GPU memory upfront
%env TF_FORCE_GPU_ALLOW_GROWTH=true
# Making sure we cache the models and are not downloaded all the time
%env TFHUB_CACHE_DIR=~/tfhub_modules
```

`TF_FORCE_GPU_ALLOW_GROWTH` allows TensorFlow to allocate GPU memory on-demand as opposed to allocating all GPU memory at once. `TFHUB_CACHE_DIR` sets the directory where the models will be downloaded. We will first import TensorFlow Hub:

```
import tensorflow_hub as hub
```

Next, as usual, we will clear any running TensorFlow sessions by running the following code:

```
import tensorflow as tf
import tensorflow.keras.backend as K
K.clear_session()
```

Finally, we will download the ELMo model. You can employ two ways to download pretrained models from TF Hub and use them in our code:

- `hub.load(<url>, **kwargs)` – Recommended way for downloading and using TensorFlow 2-compatible models
- `hub.KerasLayer(<url>, **kwargs)` – This is a workaround for using TensorFlow 1-based models in TensorFlow 2

Unfortunately, ELMo has not been ported to TensorFlow 2 yet. Therefore, we will use the `hub.KerasLayer()` as the workaround to load ELMo in TensorFlow 2:

```
elmo_layer = hub.KerasLayer(
    "https://tfhub.dev/google/elmo/3",
    signature="tokens",signature_outputs_as_dict=True
)
```

Note that we are providing two arguments, `signature` and `signature_outputs_as_dict`:

- `signature (str)` – Can be default or tokens. The default signature accepts a list of strings, where each string will be converted to a list of tokens internally. The tokens signature takes in inputs as dictionary having two keys. Namely, `tokens` (a list of list of tokens. Each list of tokens is a single phrase/sentence and includes padding tokens to bring them to a fixed length) and “`sequence_len`” (the length of each list of tokens, to determine the padding length).
- `signature_outputs_as_dict (bool)` – When set to `true`, it will return all the outputs defined in the provided signature.

Now that we have understood the components of ELMo and downloaded it from TensorFlow Hub, let's see how we can process input data for ELMo.

Preparing inputs for ELMo

Here we will define a function that will convert a given list of strings to the format ELMo expects the inputs to be in. Remember that we set the signature of ELMo to be `tokens`. An example input to the signature “`tokens`” would look as follows.

```
{  
    'tokens': [  
        ['the', 'cat', 'sat', 'on', 'the', 'mat'],  
        ['the', 'mat', 'sat', '', '', '']  
    ],  
    'sequence_len': [6, 3]  
}
```

Let's take a moment to process what the input comprises. First it has the key `tokens`, which has a list of tokens. Each list of tokens can be thought of as a sentence. Note how padding is added to the end of the short sentence to match the length. This is important as, otherwise, the model will throw an error as it can't convert arbitrary-length sequences to a tensor. Next we have `sequence_len`, which is a list of integers. Each integer specifies the true length of each sequence. Note how the second element says 3, to match the actual tokens present in the second sequence.

Given a list of strings, we can write a function to do this transformation for us. That's what the `format_text_for_elmo()` function will do for us. Let's sink our teeth into the specifics:

```
def format_text_for_elmo(texts, lower=True, split=" ", max_len=None):  
  
    """ Formats a given text for the ELMo model (takes in a list of  
    strings) """
```

```
token_inputs = [] # Maintains individual tokens
token_lengths = [] # Maintains the length of each sequence

max_len_inferred = 0
# We keep a variable to maintain the max length of the input
# Go through each text (string)
for text in texts:

    # Process the text and get a list of tokens
    tokens = tf.keras.preprocessing.text.text_to_word_sequence(text,
        lower=lower, split=split)

    # Add the tokens
    token_inputs.append(tokens)

    # Compute the max length for the collection of sequences
    if len(tokens)>max_len_inferred:
        max_len_inferred = len(tokens)

# It's important to make sure the maximum token length is only as
# Large as the longest input in the sequence
# Here we make sure max_len is only as large as the longest input
if max_len and max_len_inferred < max_len:
    max_len = max_len_inferred
if not max_len:
    max_len = max_len_inferred

# Go through each token sequence and modify sequences to have same
# Length
for i, token_seq in enumerate(token_inputs):

    token_lengths.append(min(len(token_seq), max_len))

    # If the maximum length is less than input length, truncate
    if max_len < len(token_seq):
```

```

        token_seq = token_seq[:max_len]
    # If the maximum length is greater than or equal to input length,
    # add padding as needed
    else:
        token_seq = token_seq + [""]*(max_len-len(token_seq))

    assert len(token_seq)==max_len

    token_inputs[i] = token_seq

# Return the final output
return {
    "tokens": tf.constant(token_inputs),
    "sequence_len": tf.constant(token_lengths)
}

```

We first create two lists, `token_inputs` and `token_lengths`, to contain individual tokens and their respective lengths. Next we go through each string in `texts`, and get the individual tokens using the `tf.keras.preprocessing.text.text_to_word_sequence()` function. While doing so, we will calculate the maximum token length we have observed so far. After iterating through the sequences, we check if the maximum length inferred from the inputs is different to `max_len` (if specified). If so, we will use `max_len_inferred` as the maximum length. This is important, because if you do otherwise, you may unnecessarily lengthen the inputs by defining a large value for `max_len`. Not only that, the model will raise an error like the one below if you do so.

```
#ValueError: Incompatible shapes: [2,6,1] vs. [2,10,1024]
#      [[node mul (defined at .../python3.6/site-packages/tensorflow_
hub/module_v2.py:106) ]] [Op:__inference_pruned_3391]
```

Once the proper maximum length is found, we will go through the sequences and

- If it is longer than `max_len`, truncate the sequence
- If it is shorter than `max_len`, add tokens until it reaches `max_len`

Finally, we will convert them to `tf.Tensor` objects using the `tf.constant` construct. For example, you can call this function with:

```
print(format_text_for_elmo(["the cat sat on the mat", "the mat sat"], max_
len=10))
```

This will output:

```
{'tokens': <tf.Tensor: shape=(2, 6), dtype=string, numpy=
array([[b'the', b'cat', b'sat', b'on', b'the', b'mat'],
       [b'the', b'mat', b'sat', b'', b'', b'']], dtype=object)>,
 'sequence_len': <tf.Tensor: shape=(2,), dtype=int32, numpy=array([6, 3],
 dtype=int32)>}
```

We will now see how ELMo can be used to generate embeddings for the prepared inputs.

Generating embeddings with ELMo

Once the input is prepared, generating embeddings is quite easy. First we will transform the inputs to the stipulated format of the ELMo layer. Here we are using some example titles from the BBC dataset:

```
# Titles of 001.txt - 005.txt in bbc/business
elmo_inputs = format_text_for_elmo([
    "Ad sales boost Time Warner profit",
    "Dollar gains on Greenspan speech",
    "Yukos unit buyer faces loan claim",
    "High fuel prices hit BA's profits",
    "Pernod takeover talk lifts Domecq"
])
```

Next, simply pass the `elmo_inputs` to the `elmo_layer` as the input and get the result:

```
# Get the result from ELMo
elmo_result = elmo_layer(elmo_inputs)
```

Let's now print the results and their shapes with the following line:

```
# Print the result
for k,v in elmo_result.items():
    print("Tensor under key={} is a {} shaped Tensor".format(k, v.shape))
```

This will print out:

```
Tensor under key=sequence_len is a (5,) shaped Tensor
Tensor under key=elmo is a (5, 6, 1024) shaped Tensor
Tensor under key=default is a (5, 1024) shaped Tensor
Tensor under key=lstm_outputs1 is a (5, 6, 1024) shaped Tensor
Tensor under key=lstm_outputs2 is a (5, 6, 1024) shaped Tensor
Tensor under key=word_emb is a (5, 6, 512) shaped Tensor
```

As you can see, the model returns 6 different outputs. Let's go through them one by one:

- `sequence_len` – The same input we provided containing the lengths of the sequences in the input
- `word_emb` – The token embeddings obtained via the CNN layer in the ELMo model. We got a vector of size 512 for all sequence positions (i.e. 6) and for all rows in the batch (i.e. 5).
- `lstm_output1` – The contextualized representations of tokens obtained via the first LSTM layer
- `lstm_output2` – The contextualized representations of tokens obtained via the second LSTM layer
- `default` – The mean embedding vector obtained by averaging all of the `lstm_output1` and `lstm_output2` embeddings
- `elmo` – The weighted sum of all of `word_emb`, `lstm_output1`, and `lstm_output2`, where weights are a set of task-specific trainable parameters that will be jointly trained during the task-specific training

What we are interested in here is the `default` output. That would give us a very good representation of what's contained in the document.

Other word embedding techniques

Apart from the word embedding techniques we discussed here, there are a few notable widely used word embedding techniques. We will discuss a few of those here.

FastText



FastText (<https://fasttext.cc/>), introduced in the paper “Enriching Word Vectors with Subword Information” by Bojanowski et al. (<https://arxiv.org/pdf/1607.04606.pdf>), introduces a technique where word embeddings are computed by considering the sub-components of a word. Specifically, they compute the word embedding as a summation of embeddings of n -grams of the word for several values of n . In the paper, they use $3 \leq n \leq 6$. For example, for the word “banana,” the tri-grams ($n=3$) would be [‘ban’ , ‘ana’ , ‘nan’ , ‘ana’]. This leads to robust embeddings that can withstand common problems of text, such as spelling mistakes.



Swivel embeddings

Swivel embeddings, introduced by the paper “*Swivel: Improving Embeddings by Noticing What’s Missing*” by Shazeer et al. (<https://arxiv.org/pdf/1602.02215.pdf>), tries to blend GloVe and skip-grams with negative sampling. One of the critical limitations of GloVe is that it only uses information about positive contexts. Therefore, the method is not penalized for trying to create similar vectors of words that have not been observed together. But the negative sampling used in skip-grams directly tackles this problem. The biggest innovation of Swivel is a loss function that incorporates unobserved word pairs. As an added benefit, it can also be trained in a distributed environment.

Transformer models

Transformers are a type of model that has reimagined the way we think about NLP problems. The Transformer model was initially introduced in the paper “*Attention is all you need*” by Vaswani (<https://arxiv.org/pdf/1706.03762.pdf>). This model has many different embeddings within it and, like ELMo, can generate an embedding per token by processing a sequence of text. We will talk about Transformer models in detail in later chapters.

We have discussed all the bells and whistles required to confidently use the ELMo model. Next we will classify documents using ELMo, in which ELMo will generate document embeddings as inputs to a classification model.

Document classification with ELMo

Although Word2vec gives a very elegant way of learning numerical representations of words, learning word representations alone is not convincing enough to realize the power of word vectors in real-world applications.

Word embeddings are used as the feature representation of words for many tasks, such as image caption generation and machine translation. However, these tasks involve combining different learning models such as **Convolutional Neural Networks (CNNs)** and **Long Short-Term Memory (LSTM)** models or two LSTM models (the CNN and LSTM models will be discussed in more detail in later chapters). To understand a real-world usage of word embeddings let’s stick to a simpler task—document classification.

Document classification is one of the most popular tasks in NLP. Document classification is extremely useful for anyone who is handling massive collections of data such as those for news websites, publishers, and universities. Therefore, it is interesting to see how learning word vectors can be adapted to a real-world task such as document classification by means of embedding entire documents instead of words.

This exercise is available in the Ch04-Advance-Word-Vectors folder (`ch4_document_classification.ipynb`).

Dataset

For this task, we will use an already-organized set of text files. These are news articles from the BBC. Every document in this collection belongs to one of the following categories: *Business*, *Entertainment*, *Politics*, *Sports*, or *Technology*.

Here are a couple of brief snippets from the actual data:

Business

Japan narrowly escapes recession

Japan's economy teetered on the brink of a technical recession in the three months to September, figures show.

Revised figures indicated growth of just 0.1% - and a similar-sized contraction in the previous quarter. On an annual basis, the data suggests annual growth of just 0.2%,...

First, we will download the data and load the data into memory. We will use the same `download_data()` function to download the data. Then we will slightly modify the `read_data()` function to not only return a list of articles, where each article is a string, but also to return a list of filenames, where each filename corresponds to the file the article was stored in. The filenames will subsequently help us to create the labels for our classification model.

```
def read_data(data_dir):

    # This will contain the full list of stories
    news_stories = []
    filenames = []
    print("Reading files")

    i = 0 # Just used for printing progress
```

```

for root, dirs, files in os.walk(data_dir):

    for fi, f in enumerate(files):

        # We don't read the readme file
        if 'README' in f:
            continue

        # Printing progress
        i += 1
        print("."*i, f, end='\r')

        # Open the file
        with open(os.path.join(root, f), encoding='latin-1') as text_file:
            story = []
            # Read all the lines
            for row in text_file:
                story.append(row.strip())

            # Create a single string with all the rows in the doc
            story = ' '.join(story)
            # Add that to the list
            news_stories.append(story)
            filenames.append(os.path.join(root, f))

        print('', end='\r')

print("\nDetected {} stories".format(len(news_stories)))
return news_stories, filenames

```

news_stories, filenames = read_data(os.path.join('data', 'bbc'))

We will then create and fit a tokenizer on the data, as we have done before.

```

from tensorflow.keras.preprocessing.text import Tokenizer

n_vocab = 15000 + 1

```

```
tokenizer = Tokenizer(  
    num_words=n_vocab - 1,  
    filters='!"#$%&()*+,-./:;=>?@[\\]^_{}|~\\t\\n',  
    lower=True, split=' ', oov_token=''  
)  
tokenizer.fit_on_texts(news_stories)
```

As the next step, we will create labels. Since we are training a classification model, we need both inputs and labels. Our inputs will be document embeddings (we will see how to compute them soon), and the targets will be a label ID between 0 and 4. Each class we mentioned above (e.g. business, tech, etc.) will be assigned to a separate category. Since the filename includes the category as a folder, we can leverage the filename to generate a label ID.

We will use the pandas library to create the labels. First we will convert the list of filenames to a pandas Series object using:

```
labels_ser = pd.Series(filenames, index=filenames)
```

An example entry in this series could look like `data/bbc/tech/127.txt`. Next, we will split each item on the “/” character, which will return a list `['data', 'bbc', 'tech', '127.txt']`. We will also set `expand=True`. `expand=True` will transform our Series object to a DataFrame by turning each item in the list of tokens into a separate column of a DataFrame. In other words, our `pd.Series` object will become an `[N, 4]`-sized `pd.DataFrame` with one token in each column, where `N` is the number of files:

```
labels_ser = labels_ser.str.split(os.path.sep, expand=True)
```

In the resulting data, we only care about the third column, which has the category of a given article (e.g. `tech`). Therefore, we will discard the rest of the data and only keep that column:

```
labels_ser = labels_ser.iloc[:, -2]
```

Finally, we will map the string label to an integer ID using the pandas `map()` function as follows:

```
labels_ser = labels_ser.map({'business': 0, 'entertainment': 1,  
    'politics': 2, 'sport': 3, 'tech': 4})
```

This will result in something like:

```
data/bbc/tech/272.txt      4  
data/bbc/tech/127.txt      4  
data/bbc/tech/370.txt      4
```

```
data/bbc/tech/329.txt    4
data/bbc/tech/240.txt    4
Name: 2, dtype: int64
```

What we did here can be written as just one line by chaining the sequence of commands to a single line:

```
labels_ser = pd.Series(filenames, index=filenames).str.split(os.path.sep,
expand=True).iloc[:, -2].map(
    {'business': 0, 'entertainment': 1, 'politics': 2, 'sport': 3,
     'tech': 4}
)
```

With that, we move on to the next important step, i.e. splitting the data into train/test subsets. When training a supervised model, we generally need three datasets:

- A training set – This is the dataset the model will be trained on.
- A validation set – This will be used during the training to monitor model performance (e.g. signs of overfitting).
- A testing set – This will be not exposed to the model at any time during the model training. It will only be used after the model training to evaluate the model on unseen data.

In this exercise, we will only use the training set and the testing set. This will help us to keep our conversation more focused on embeddings and keep the discussion about the downstream classification model simple. Here we will use 67% of the data as training data and use 33% of data as testing data. Data will be split randomly:

```
from sklearn.model_selection import train_test_split

train_labels, test_labels = train_test_split(labels_ser, test_size=0.33)
```

Now we have a training dataset to train the model and a test dataset to test it on unseen data. We will now see how we can generate document embeddings from token or word embeddings.

Generating document embeddings

Let's first remind ourselves how we stored embeddings for skip-gram, CBOW, and GloVe algorithms. *Figure 4.4* depicts how these look in a `pd.DataFrame` object.

	0	1	2	3	4	5	6	7	8	9	...	118	119	120	
NaN	-1.440415	1.375493	-1.205786	-1.326926	-1.336768	-1.205858	-1.366250	-1.036717	-1.365316	-0.775372	...	1.205486	1.138118	-1.385370	-1.264
	-0.290653	0.574326	0.026581	-0.329045	-0.171542	0.224572	0.071370	0.146927	-0.041791	0.086741	...	0.259992	0.278289	-0.879627	-0.351
the	-0.058531	0.310743	0.232947	-0.338871	-0.334953	0.003350	-0.061267	0.257632	0.031830	-0.587573	...	0.484254	-0.228630	-0.044630	-0.305
to	-0.276640	0.025122	-0.024782	-0.624656	-0.016077	0.046590	-0.094414	-0.848296	-0.327094	-1.684406	...	-0.261233	-0.106222	-0.139904	-0.411
of	-0.281321	-0.034827	-0.068643	-0.210236	0.122180	0.021023	-0.258712	0.238127	-0.408491	-1.221611	...	0.083432	0.030410	-0.390671	-0.030
and	-0.207726	0.023091	0.186237	-0.209033	-0.121335	0.271208	-0.033538	0.390482	0.217749	-1.416989	...	-0.351555	0.300373	-0.019044	-0.298
a	-0.452312	0.305802	-0.962776	-0.884619	0.180468	-0.150204	-0.387859	0.362963	-0.194890	0.303527	...	-0.297423	0.582571	-0.498461	-0.074
in	0.264359	0.130696	0.374273	0.023650	-0.385127	0.085398	0.314853	-1.038579	-0.981065	0.308113	...	-0.258953	0.010786	-0.503573	0.038
for	-0.866129	0.162434	-0.183391	0.079462	-0.439157	-0.600472	-0.600996	-1.004945	-0.321327	-1.096564	...	0.153067	0.338188	-0.180635	0.506
is	-0.004946	0.531260	-0.119664	-0.702005	-0.368837	-0.918419	0.164347	-0.694515	-0.470691	0.242414	...	0.780400	-0.017688	-0.529367	0.020

10 rows × 128 columns

Figure 4.4: A snapshot of the context embeddings of the skip-gram algorithm we saved to the disk. You can see below it says that it has 128 columns (i.e. the embedding size)

ELMo embeddings are an exception to this. Since ELMo generates contextualized representations for all tokens in a sequence, we have stored the mean embedding vectors resulting from averaging all the generated vectors:

	0	1	2	3	4	5	6	7	8	9	...	1014	1015	
data/bbc/tech/272.txt	0.144291	0.015962	-0.151633	0.096555	-0.015913	0.075896	-0.033353	-0.118644	-0.192030	-0.124919	...	-0.418126	0.204618	0.
data/bbc/tech/127.txt	0.022870	-0.142899	-0.017096	-0.084165	0.320108	0.424914	-0.043930	0.257134	-0.215543	-0.046845	...	-0.219130	0.264653	0.
data/bbc/tech/370.txt	0.207623	0.058697	-0.008874	-0.088409	0.193419	0.046109	-0.107221	0.199647	-0.167632	0.003790	...	-0.054829	0.225892	0.
data/bbc/tech/329.txt	0.022106	0.060943	-0.127390	-0.100214	0.184243	-0.077529	-0.157470	-0.042993	-0.204254	-0.021419	...	-0.337353	0.153419	0.
data/bbc/tech/240.txt	0.259128	-0.108082	0.076262	-0.080416	0.183988	0.329807	0.156697	0.495652	-0.104913	-0.120077	...	-0.218093	0.236378	0.
data/bbc/tech/379.txt	0.071111	-0.112660	0.038746	-0.084503	0.207438	0.231360	-0.015819	0.235174	-0.238940	0.030840	...	-0.330608	0.267756	0.
data/bbc/tech/339.txt	0.080515	0.216040	-0.090722	0.118778	0.336153	0.223334	0.075926	0.472938	-0.046222	0.124439	...	-0.240382	0.015074	-0.
data/bbc/tech/046.txt	0.019151	-0.029814	-0.046270	-0.139506	0.255929	0.230742	-0.011093	0.426066	0.112122	-0.130358	...	-0.285940	0.280919	-0.
data/bbc/tech/140.txt	0.379357	-0.174718	-0.062910	-0.017047	-0.034842	0.260822	0.129207	0.358961	-0.040298	-0.027103	...	-0.144901	0.096271	0.
data/bbc/tech/349.txt	0.241237	-0.013553	0.033077	-0.159186	0.367290	0.414783	0.176808	0.556002	-0.273304	0.108183	...	-0.235471	0.265600	-0.

10 rows × 1024 columns

Figure 4.5: A snapshot of ELMo vectors. ELMo vectors have 1024 elements

To compute the document embeddings from skip-gram, CBOW, and GloVe embeddings, let us write the following function:

```
def generate_document_embeddings(texts, filenames, tokenizer, embeddings):
    """
    This function takes a sequence of tokens and compute the mean
    embedding vector from the word vectors of all the tokens in the
    document """

```

```
doc_embedding_df = []
# Contains document embeddings for all the articles
assert isinstance(embeddings, pd.DataFrame), 'embeddings must be a
pd.DataFrame'

# This is a trick we use to quickly get the text preprocessed by the
# tokenizer
# We first convert text to a sequences, and then back to text, which
# will give the preprocessed tokens
sequences = tokenizer.texts_to_sequences(texts)
preprocessed_texts = tokenizer.sequences_to_texts(sequences)

# For each text,
for text in preprocessed_texts:
    # Make sure we had matches for tokens in the embedding matrix
    assert embeddings.loc[text.split(' '), :].shape[0]>0
    # Compute mean of all the embeddings associated with words
    mean_embedding = embeddings.loc[text.split(' '), :].mean(axis=0)
    # Add that to list
    doc_embedding_df.append(mean_embedding)

# Save the doc embeddings in a dataframe
doc_embedding_df = pd.DataFrame(doc_embedding_df, index=filenames)

return doc_embedding_df
```

The `generate_document_embeddings()` function takes the following arguments:

- `texts` – A list of strings, where each string represents an article
- `filenames` – A list of filenames corresponding to the articles in `texts`
- `tokenizer` – A tokenizer that can process `texts`
- `embeddings` – The embeddings as a `pd.DataFrame`, where each row represents a word vector, indexed by the corresponding token

The function first preprocesses the texts by converting the strings to sequences, and then back to a list of strings. This helps us to use the built-in preprocessing functionalities of the tokenizer to clean the text. Next, each preprocessed string is split by the space character to return a list of tokens. Then we index all the positions in the embeddings matrix that corresponds to all the tokens in the text. Finally, the mean vector is computed for the document by computing the mean of all the chosen embedding vectors.

With that, we can load the embeddings from different algorithms (skip-gram, CBOW, and GloVe), and compute the document embeddings. Here we will only show the process for the skip-gram algorithm. But you can easily extend it to the other algorithms, as they have similar inputs and outputs:

```
# Load the skip-gram embeddings context and target
skipgram_context_embeddings = pd.read_pickle(
    os.path.join('../Ch03-Word-Vectors/skipgram_embeddings',
    'context_embedding.pkl')
)
skipgram_target_embeddings = pd.read_pickle(
    os.path.join('../Ch03-Word-Vectors/skipgram_embeddings',
    'target_embedding.pkl')
)

# Compute the mean of context & target embeddings for better embeddings
skipgram_embeddings = (skipgram_context_embeddings + skipgram_target_
embeddings)/2

# Generate the document embeddings with the average context target
# embeddings
skipgram_doc_embeddings = generate_document_embeddings(news_stories,
filenames, tokenizer, skipgram_embeddings)
```

Now we will see how we can leverage the generated document embedding to train a classifier.

Classifying documents with document embeddings

We will be training a simple multi-class (or a multinomial) logistic regression classifier on this data. The logistic regression model will look as follows:

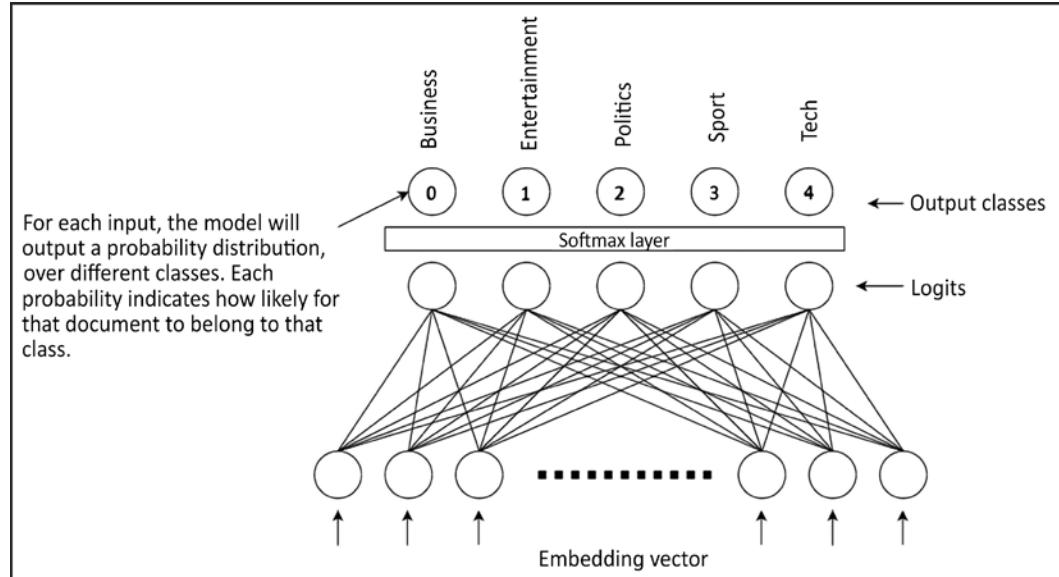


Figure 4.6: This diagram depicts the multinomial logistic regression model. The model takes in an embedding vector and outputs a probability distribution over different available classes

It's a very simple model with a single layer, where the input is the embedding vector (e.g. a 128-element-long vector), and the output is a 5-node softmax layer that will output the likelihood of the input belonging to each category, as a probability distribution.

We will be training several models, as opposed to a single run. This will give us a more consistent result on the performance of the model. To implement the model, we'll be using a popular general-purpose machine learning library called scikit-learn (<https://scikit-learn.org/stable/>). In each run, a multi-class logistic regression classifier is created with the `sklearn.linear_model.LogisticRegression` object. Additionally, in each run:

1. The model is trained on the training inputs and targets
2. The model predicts the class (a value from 0 to 4) for each test input, where the class of an input is the one that has the maximum probability from all classes
3. The model computes the test accuracy using the predicted classes and true classes of the test set

The code looks like the following:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

def get_classification_accuracy(doc_embeddings, train_labels, test_labels,
n_trials):
    """ Train a simple MLP model for several trials and measure test
    accuracy"""

    accuracies = [] # Store accuracies across trials

    # For each trial
    for trial in range(n_trials):
        # Create a MLP classifier
        lr_classifier = LogisticRegression(multi_class='multinomial',
        max_iter=500)

        # Fit the model on training data
        lr_classifier.fit(doc_embeddings.loc[train_labels.index],
        train_labels)

        # Get the predictions for test data
        predictions = lr_classifier.predict(doc_embeddings.loc[test_
        labels.index])

        # Compute accuracy
        accuracies.append(accuracy_score(predictions, test_labels))

    return accuracies

# Get classification accuracy for skip-gram models
skipgram_accuracies = get_classification_accuracy(
    skipgram_doc_embeddings, train_labels, test_labels, n_trials=5
)

print("Skip-gram accuracies: {}".format(skipgram_accuracies))
```

By setting `multi_class='multinomial'`, we are making sure it's a multi-class logistic regression model (or a softmax classifier). This will output:

```
Skip-gram accuracies: [0.882..., 0.882..., 0.881..., 0.882..., 0.884...]
```

When you follow the procedure for all the skip-gram, CBOW, GloVe, and ELMo algorithms, you will see a result similar to the following. This is a box plot diagram. However, as performance is quite similar between trials, you won't see much variation present in the diagram:

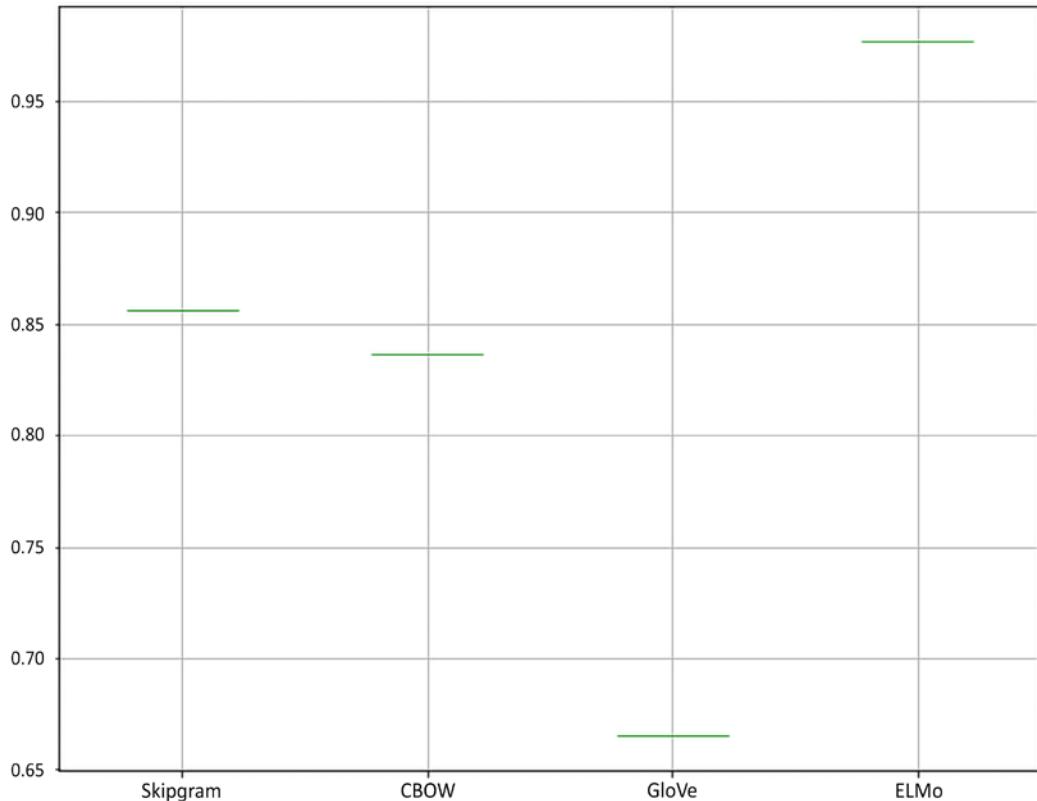


Figure 4.7: Box plot interpreting performance on document classification for different models.

We can see that ELMo is a clear-cut winner, where GloVe performs the worst

We can see that skip-gram achieves around 86% accuracy, followed closely by CBOW, which achieves on-par performance. Surprisingly GloVe achieves performance far below the skip-gram and CBOW, around 66% accuracy.

This could be pointing to a limitation of the GloVe loss function. Unlike, skip-gram and CBOW, which are considered both positive (observed) and negative (unobserved) target and context pairs, GloVe only focuses on observed pairs.

This could be hurting GloVe's ability to generate effective representations of words. Finally, ELMo achieves the best, which is around 98% accuracy. But it is important to keep in mind that ELMo has been trained on a much larger dataset than the BBC dataset, thus it is not fair to compare ELMo with other models just on this number.

In this section, you learned how we can extend word embeddings turned to document embeddings and how these can be used in a downstream classifier model to classify documents. First, you learned about word embeddings using a selected algorithm (e.g. skip-gram, CBOW, and GloVe). Then we created document embeddings by averaging the word embeddings of all the words found in that document. This was the case for the skip-gram, CBOW, and GloVe algorithms. In the case of the ELMo algorithm, we were able to infer document embeddings straight from the model. Later we used these document embeddings to classify some BBC news articles that fall into these categories: entertainment, tech, politics, business, and sports.

Summary

In this chapter, we discussed GloVe—another word embedding learning technique. GloVe takes the current Word2vec algorithms a step further by incorporating global statistics into the optimization, thus increasing the performance.

Next, we learned about a much more advanced algorithm known as ELMo (which stands for Embeddings from Language Models). ELMo provides contextualized representations of words by looking at a word within a sentence or a phrase, not by itself.

Finally, we discussed a real-world application of using word embeddings—document classification. We showed that word embeddings are very powerful and allow us to classify related documents with a simple multi-class logistic regression model reasonably well. ELMo performed the best out of skip-gram, CBOW, and GloVe, due to the vast amount of data it has been trained on.

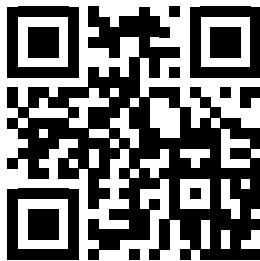
In the next chapter, we will move on to discussing a different family of deep networks that are more powerful in exploiting spatial information present in data, known as **Convolutional Neural Networks (CNNs)**.

Precisely, we will see how CNNs can be used to exploit the spatial structure of sentences to classify them into different classes.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside more than 1000 members at: <https://packt.link/nlp>



5

Sentence Classification with Convolutional Neural Networks

In this chapter, we will discuss a type of neural network known as **Convolutional Neural Networks (CNNs)**. CNNs are quite different from fully connected neural networks and have achieved state-of-the-art performance in numerous tasks. These tasks include image classification, object detection, speech recognition, and of course, sentence classification. One of the main advantages of CNNs is that, compared to a fully connected layer, a convolution layer in a CNN has a much smaller number of parameters. This allows us to build deeper models without worrying about memory overflow. Also, deeper models usually lead to better performance.

We will introduce you to what a CNN is in detail by discussing different components found in a CNN and what makes CNNs different from their fully connected counterparts. Then we will discuss the various operations used in CNNs, such as the convolution and pooling operations, and certain hyperparameters related to these operations, such as filter size, padding, and stride. We will also look at some of the mathematics behind the actual operations. After establishing a good understanding of CNNs, we will look at the practical side of implementing a CNN with TensorFlow. First, we will implement a CNN to classify images and then use a CNN for sentence classification. Specifically, we'll go through the following topics:

- Learning the fundamentals of CNNs
- Classifying images with CNNs
- Classifying sentences with CNNs

Introducing CNNs

In this section, you will learn about CNNs. Specifically, you will first get an understanding of the sort of operations present in a CNN, such as convolution layers, pooling layers, and fully connected layers. Next, we will briefly see how all of these are connected to form an end-to-end model.

It is important to note that the first use case we'll be solving with CNNs is an image classification task. CNNs were originally used to solve computer vision tasks and were adopted for NLP much later. Furthermore, CNNs have a stronger presence in the computer vision domain than the NLP domain, making it easier to explain the underlying concepts in a vision context. For this reason, we will first learn how CNNs are used in computer vision and then move on to NLP.

CNN fundamentals

Now, let's explore the fundamental ideas behind a CNN without delving into too much technical detail. A CNN is a stack of layers, such as convolution layers, pooling layers, and fully connected layers. We will discuss each of these to understand their role in the CNN.

Initially, the input is connected to a set of convolution layers. These convolution layers slide a patch of weights (sometimes called the convolution window or filter) over the input and produce an output by means of the convolution operation. Convolution layers use a small number of weights, organized to cover only a small patch of input in each layer, unlike fully connected neural networks, and these weights are shared across certain dimensions (for example, the width and height dimensions of an image). Also, CNNs use the convolution operations to share the weights from the output by sliding this small set of weights along the desired dimension. What we ultimately get from this convolution operation is illustrated in *Figure 5.1*. If the pattern present in a convolution filter is present in a patch of image, the convolution will output a high value for that location; if not, it will output a low value. Also, by convolving the full image, we get a matrix indicating whether a pattern was present or not in a given location. Finally, we will get a matrix as the convolution output:

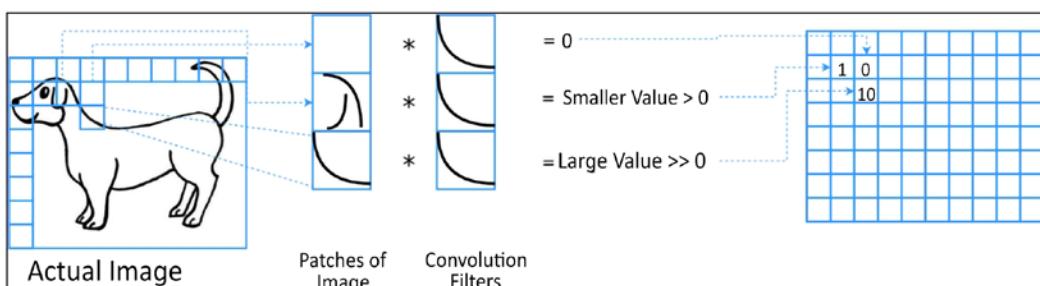


Figure 5.1: What the convolution operation does to an image

Also, these convolution layers are optionally interleaved with pooling/subsampling layers, which reduces the dimensionality of the input. While reducing the dimensionality, we make the translation of CNNs invariant, as well as force the CNN to learn with less information, leading to better generalization and regularization of the model. The dimensionality is reduced by dividing the input into several patches and transforming each patch into a single element. For example, such transformations include picking the maximum element of a patch or averaging all the values in a patch. We will illustrate how pooling can make the translation of CNNs invariant in *Figure 5.2*:

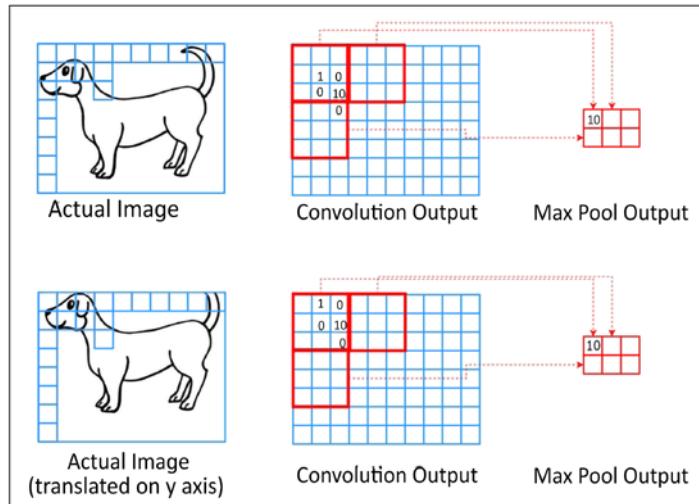


Figure 5.2: How the pooling operation helps to make data translation invariant

Here, we have the original image and an image slightly translated on the y axis. We have convolution output for both images, and you can see that the value **10** appears at slightly different places in the convolution output. However, using max pooling (which takes the maximum value of each thick square), we can get the same output at the end. We will discuss these operations in detail later.

Finally, the output is fed to a set of fully connected layers, which then forward the output to the final classification/regression layer (for example, sentence/image classification). Fully connected layers contain a significant fraction of the total number of weights of the CNN, as convolution layers have a small number of weights. However, it has been found that CNNs perform better with fully connected layers than without them. This could be because convolution layers learn more localized features due to their small size, whereas fully connected layers provide a global picture of how these localized features should be connected together to produce a desirable final output.

Figure 5.3 shows a typical CNN used to classify images:

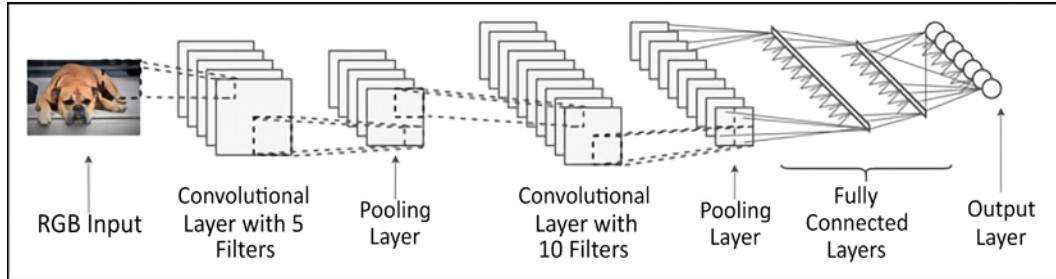


Figure 5.3: A typical CNN architecture

As is evident from the figure, CNNs, by design, preserve the spatial structure of the inputs during learning. In other words, for a two-dimensional input, a CNN will mostly have two-dimensional layers, whereas it will only have fully connected layers close to the output layer. Preserving the spatial structure allows CNNs to exploit valuable spatial information of the inputs and learn about inputs with fewer parameters. The value of spatial information is illustrated in *Figure 5.4*:

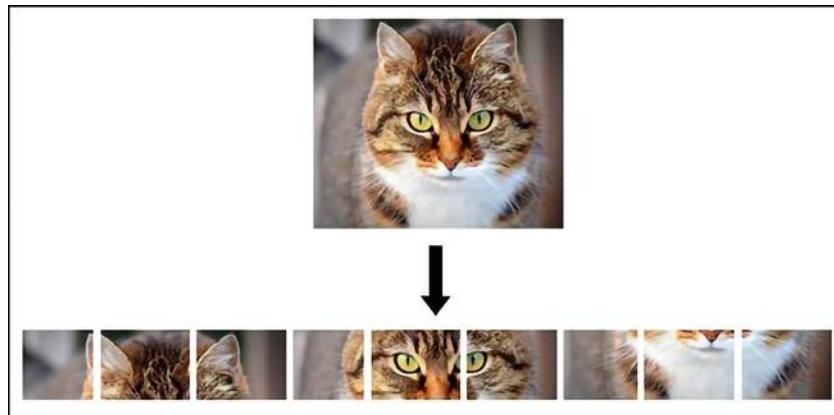


Figure 5.4: Unwrapping an image into a one-dimensional vector loses some of the important spatial information

As you can see, when a two-dimensional image of a cat is unwrapped to be a one-dimensional vector, ears are no longer close to the eyes, and the nose is far away from the eyes as well. This means we have destroyed some of the useful spatial information during the unwrapping. This is why preserving the two-dimensional nature of the inputs is so important.

The power of CNNs

CNNs are a very versatile family of models and have shown a remarkable performance in many types of tasks. Such versatility is attributed to the ability of CNNs to perform feature extraction and learning simultaneously, leading to greater efficiency and generalizability. Let's discuss a few examples of the utility of CNNs.

In the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2020**, which involved classifying images, detecting objects, and localizing objects in an image, CNNs were used to achieve incredible test accuracies. For example, for image-classification tasks, its top-1 test accuracy was approximately 90% for 1,000 different object classes, which means that the CNN was able to correctly identify around 900 different objects correctly.

CNNs also have been used for image segmentation. Image segmentation involves segmenting an image into different areas. For example, in an urbanscape image that includes buildings, a road, vehicles, and passengers, isolating the road from the buildings is a segmentation task. Moreover, CNNs have made incredible strides, demonstrating their performance in NLP tasks such as sentence classification, text generation, and machine translation.

Understanding CNNs

Now that we understand the high level concepts governing CNNs, let's walk through the technical details of a CNN. First, we will discuss the convolution operation and introduce some terminology, such as filter size, stride, and padding. In brief, **filter size** refers to the window size of the convolution operation, **stride** refers to the distance between two movements of the convolution window, and **padding** refers to the way you handle the boundaries of the input. We will also discuss an operation that is known as deconvolution or transposed convolution. Then we will discuss the details of the pooling operation. Finally, we will discuss how to add fully connected layers, which produce the classification or regression output.

Convolution operation

In this section, we will discuss the convolution operation in detail. First, we will discuss the convolution operation without stride and padding, then we will describe the convolution operation with stride, and then we will discuss the convolution operation with padding. Finally, we will discuss something called transposed convolution. For all the operations in this chapter, we consider the index starting from one, and not from zero.

Standard convolution operation

The convolution operation is a central part of CNNs. For an input of size $n \times n$ and a weight patch (also known as a *filter* or a *kernel*) of $m \times m$, where $n \geq m$, the convolution operation slides the patch of weights over the input. Let's denote the input by X , the patch of weights by W , and the output by H . Also, at each location i, j , the output is calculated as follows:

$$h_{i,j} = \sum_{k=1}^m \sum_{l=1}^m w_{k,l} x_{i+k-1, j+l-1}, \text{ where } 1 \leq i, j \leq n - m + 1$$

Here, $x_{i,j}$, $w_{i,j}$, and $h_{i,j}$ denote the value at the $(i,j)^{\text{th}}$ location of X , W , and H , respectively. As already shown by the equation, though the input size is $n \times n$, the output in this case will be $n - m + 1 \times n - m + 1$. Also, m is known as the filter size. This means the width and height of the output will be slightly less than of the original. Let's look at this through a visualization (see *Figure 5.5*):

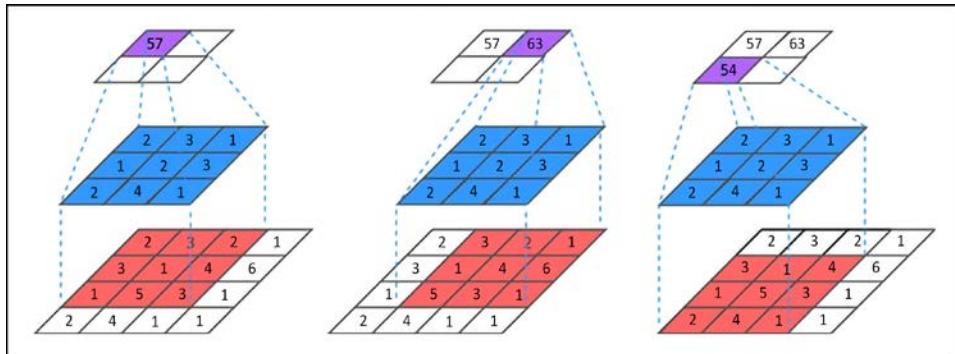


Figure 5.5: The convolution operation with a filter size (m) = 3, stride = 1, and no padding

Note



The output produced by the convolution operation (the rectangle at the top in *Figure 5.5*) is sometimes called a **features map**.

Next let's discuss the stride parameter in convolution.

Convolving with stride

In the preceding example, we shifted the filter by a single step. However, this is not mandatory; we can take large steps or strides while convolving the input. Therefore, the size of the step is known as the stride.

Let's modify the previous equation to include the s_i and s_j strides:

$$h_{i,j} = \sum_{k=1}^m \sum_{l=1}^m w_{k,l} x_{(i-1) \times s_i + k, (j-1) \times s_j + l}$$

where, $1 \leq i \leq \text{floor}[(n-m)/s_i] + 1$ and $\text{floor}[(n-m)/s_j] + 1$

In this case, the output will be smaller as the size of s_i and s_j increases. Comparing *Figure 5.5 (stride = 1)* and *Figure 5.6 (stride = 2)* illustrates the effect of different strides:

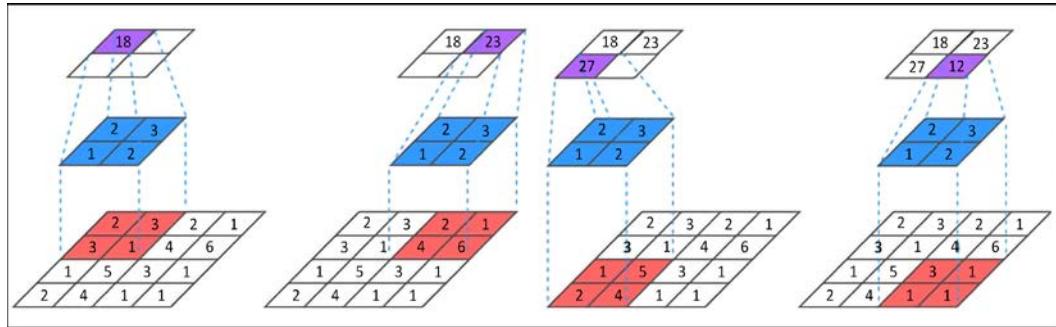


Figure 5.6: The convolution operation with a filter size (m) = 2, stride = 2, and no padding

As you can see, doing convolution with stride helps to reduce the dimensionality of the input similar to a pooling layer. Therefore, sometimes convolution with stride is used instead of pooling in the CNNs as it reduces the computational complexity. Also note that the dimensionality reduction achieved by stride can be tuned or controlled as opposed to the inherent dimensionality reduction from the standard convolution operation. We will now discuss another important concept in convolution known as padding.

Convolving with padding

The inevitable output size reduction resulting from each convolution (without stride) is an undesirable property. This greatly limits the number of layers we can have in a network. Also, it is known that deeper networks perform better than shallow networks. This should not be confused with the dimensionality reduction achieved by stride, as this is a design choice and we can decide to have a stride of 1 if necessary. Therefore, padding is used to circumvent this issue. This is achieved by padding zeros to the boundary of the input so that the output size and the input size are equal. Let's assume a stride of 1:

$$h_{i,j} = \sum_{k=1}^m \sum_{l=1}^m w_{k,l} x_{i+k-(m-1), j+l-(m-1)}, \text{ where } 1 \leq i, j \leq n$$

Here:

$$x_{i,j} = 0 \text{ if } i,j < 1 \text{ or } i,j > n$$

Figure 5.7 depicts the result of the padding:

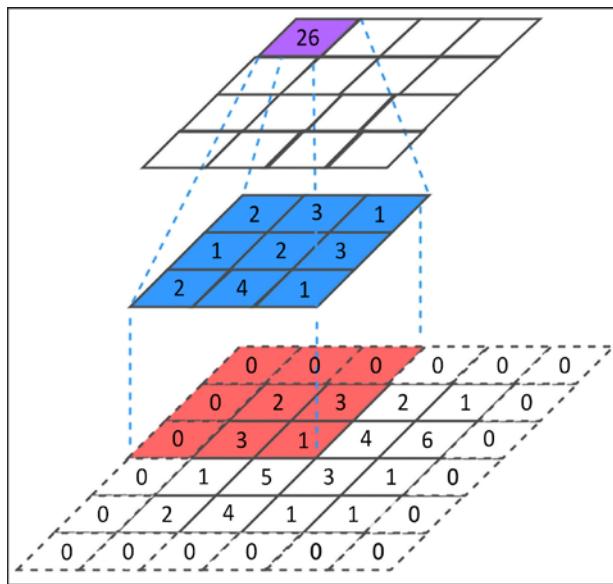


Figure 5.7: The convolution operation with a filter size ($m=3$), stride ($s=1$), and zero padding

We will now discuss the transposed convolution operation.

Transposed convolution

Though the convolution operation looks complicated in terms of mathematics, it can be simplified to a matrix multiplication. For this reason, we can define the transpose of the convolution operation or **deconvolution**, as it is sometimes called. However, we will use the term **transposed convolution** as it sounds more natural. In addition, deconvolution refers to a different mathematical concept. The transposed convolution operation plays an important role in CNNs for the reverse accumulation of the gradients during backpropagation. Let's go through an example.

For an input of size $n \times n$ and a weight patch, or filter, of $m \times m$, where $n \geq m$, the convolution operation slides the patch of weights over the input. Let's denote the input by X , the patch of weights by W , and the output by H . The output H can be calculated as a matrix multiplication as follows.

Let's assume $n = 4$ and $m = 3$ for clarity and unwrap the input X from left to right, top to bottom, resulting in this:

$$x^{(16,1)} = x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4}, \dots, x_{4,1}, x_{4,2}, x_{4,3}, x_{4,4}$$

Let's define a new matrix A from W :

$$A^{(4,16)} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 \end{bmatrix}$$

Then, if we perform the following matrix multiplication, we obtain H :

$$H^{(4,1)} = A^{(4,16)} X^{(16,1)}$$

Now, by reshaping the output $H^{(4,1)}$ to $H^{(2,2)}$ we obtain the convolved output. Now let's project this result back to n and m .

By unwrapping the input $X^{(n,n)}$ to $X^{(n^2,1)}$ and by creating a matrix $A^{((n-m+1)^2, n^2)}$ from w , as we showed earlier, we obtain $H^{((n-m+1)^2, 1)}$, which will then be reshaped to $H^{(n-m+1, n-m+1)}$.

Next, to obtain the transposed convolution, we simply transpose A and arrive at the following:

$$\hat{X}^{(n^2,1)} = (A^T)^{(n^2, (n-m+1)^2)} H^{((n-m+1)^2, 1)}$$

Here, \hat{X} is the resultant output of the transposed convolution.

We end our discussion about the convolution operation here. We discussed the convolution operation, convolution operation with stride, convolution operation with padding, and how to calculate the transposed convolution. Next, we will discuss the pooling operation in more detail.

Pooling operation

The pooling operation, which is sometimes known as the subsampling operation, was introduced to CNNs mainly for reducing the size of the intermediate outputs as well as for making the translation of CNNs invariant. This is preferred over the natural dimensionality reduction caused by convolution without padding, as we can decide where to reduce the size of the output with the pooling layer, in contrast to forcing it to happen every time. Forcing the dimensionality to decrease without padding would strictly limit the number of layers we can have in our CNN models.

We define the pooling operation mathematically in the following sections. More precisely, we will discuss two types of pooling: max pooling and average pooling. First, however, we will define the notation. For an input of size $n \times n$ and a kernel (analogous to the filter of a convolution layer) of size $m \times m$, where $n \geq m$, the convolution operation slides the patch of weights over the input. Let's denote the input by X , the patch of weights by W , and the output by H . Then let us use, $x_{i,j}$, $w_{i,j}$, and $h_{i,j}$ to denote the value at the $(i,j)^{\text{th}}$ location of X , W , and H , respectively. We will now look at specific implementations of pooling commonly used.

Max pooling

The max pooling operation picks the maximum element within the defined kernel of an input to produce the output. The max pooling operation shifts are windows over the input (the middle squares in *Figure 5.8*) and take the maximum at each time. Mathematically, we define the pooling equation as follows:

$$h_{i,j} = \max(\{x_{i,j}, x_{i,j+1}, \dots, x_{i,j+m-1}, x_{i+1,j}, \dots, x_{i+1,j+m-1}, \dots, x_{i+m-1,j}, \dots, x_{i+m-1,j+m-1}\})$$

where, $1 \leq i, j \leq n - m + 1$

Figure 5.8 shows this operation:

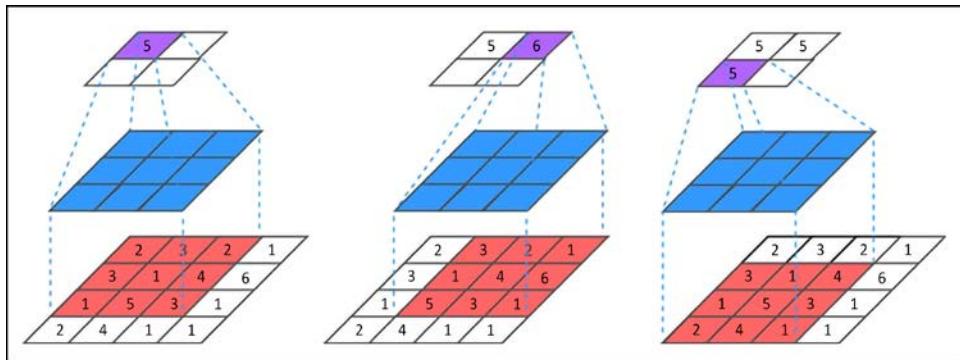


Figure 5.8: The max pooling operation with a filter size of 3, stride of 1, and no padding

Next, let's discuss how to perform max pooling with stride.

Max pooling with stride

Max pooling with stride is similar to convolution with stride. Here is the equation:

$$h_{i,j} = \max \left(\left\{ x_{(i-1) \times s_i + 1, (j-1) \times s_j + 1}, x_{(i-1) \times s_i + 1, (j-1) \times s_j + 2}, \dots, x_{(i-1) \times s_i + 1, (j-1) \times s_j + m}, x_{(i-1) \times s_i + 2, (j-1) \times s_j + 1}, \dots, x_{(i-1) \times s_i + 2, (j-1) \times s_j + m}, \dots, x_{(i-1) \times s_i + m, (j-1) \times s_j + 1}, \dots, x_{(i-1) \times s_i + m, (j-1) \times s_j + m} \right\} \right)$$

where, $1 \leq i \leq \text{floor}[(n-m)/s_i] + 1$ and $1 \leq j \leq \text{floor}[(n-m)/s_j] + 1$

Figure 5.9 shows the result:

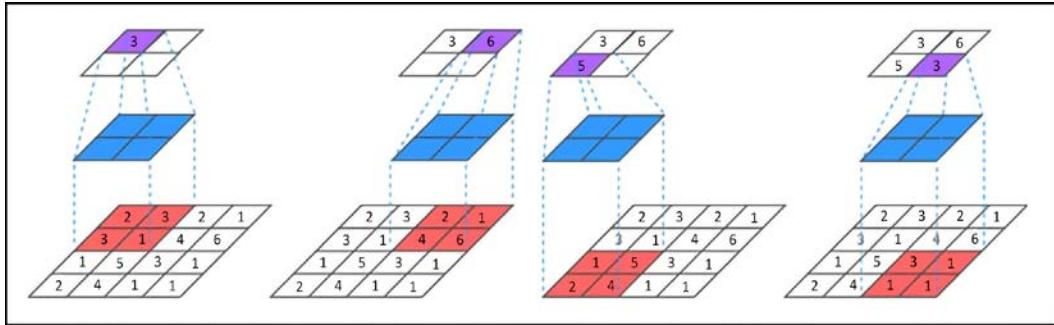


Figure 5.9: The max pooling operation for an input of size ($n=4$) with a filter size of ($m=2$), stride ($s=2$), and no padding

We will discuss another variant of pooling known as average pooling, below.

Average pooling

Average pooling works similar to max pooling, except that instead of only taking the maximum, the average of all the inputs falling within the kernel is taken. Consider the following equation:

$$h_{i,j} = \frac{x_{i,j}, x_{i,j+1}, \dots, x_{i,j+m-1}, x_{i+1,j}, \dots, x_{i+1,j+m-1}, \dots, x_{i+m-1,j}, \dots, x_{i+m-1,j+m-1}}{m \times m} \quad \forall i \geq 1, j \leq n - m + 1$$

The average pooling operation is shown in Figure 5.10:

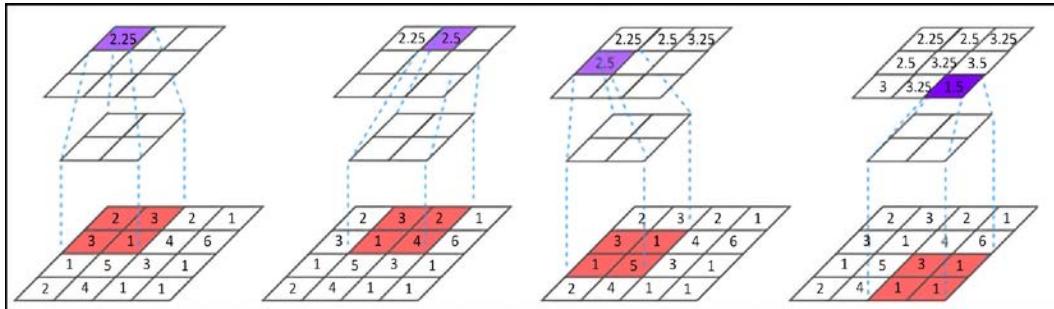


Figure 5.10: The average pooling operation for an input of size ($n=4$) with a filter size of ($m=2$), stride ($s=1$), and no padding

We have so far discussed the operations directly performed on the two-dimensional inputs like images. Next we will discuss how they are connected to one-dimensional fully connected layers.

Fully connected layers

Fully connected layers are a fully connected set of weights from the input to the output. These fully connected weights are able to learn global information as they are connected from each input to each output. Also, having such layers of full connectedness allows us to combine features learned by the convolution layers preceding the fully connected layers, globally, to produce meaningful outputs.

Let's define the output of the last convolution or pooling layer to be of size $p \times o \times d$, where p is the height of the input, o is the width of the input, and d is the depth of the input. As an example, think of an RGB image, which will have a fixed height, fixed width, and a depth of 3 (one depth channel for each RGB component).

Then, for the initial fully connected layer found immediately after the last convolution or pooling layer, the weight matrix will be $w^{(m,p \times o \times d)}$, where *height x width x depth* of the layer output is the number of output units produced by that last layer and m is the number of hidden units in the fully connected layer. Then, during inference (or prediction), we reshape the output of the last convolution/pooling layer to be of size $(p \times o \times d, 1)$ and perform the following matrix multiplication to obtain h :

$$h^{(m \times 1)} = w^{(m,p \times o \times d)} x^{(p \times o \times d, 1)}$$

The resultant fully connected layers will behave as in a fully connected neural network, where you have several fully connected layers and an output layer. The output layer can be a softmax classification layer for a classification problem or a linear layer for a regression problem.

Putting everything together

Now we will discuss how the convolutional, pooling, and fully connected layers come together to form a complete CNN.

As shown in *Figure 5.11*, the convolution, pooling, and fully connected layers come together to form an end-to-end learning model that takes raw data, which can be high-dimensional (for example, RGB images) and produce meaningful output (for example, the class of the object). First, the convolution layers learn the spatial features of the images.

The lower convolution layers learn low-level features such as differently oriented edges present in the images, and the higher layers learn more high-level features such as shapes present in the images (for example, circles and triangles) or bigger parts of an object (for example, the face of a dog, tail of a dog, and front section of a car). The pooling layers in the middle make each of these learned features slightly translation invariant. This means that, in a new image, even if the feature appears a bit offset compared to the location in which the feature appeared in the learned images, the CNN will still recognize that feature. Finally, the fully connected layers combine the high-level features learned by the CNN to produce global representations that will be used by the final output layer to determine the class the object belongs to:

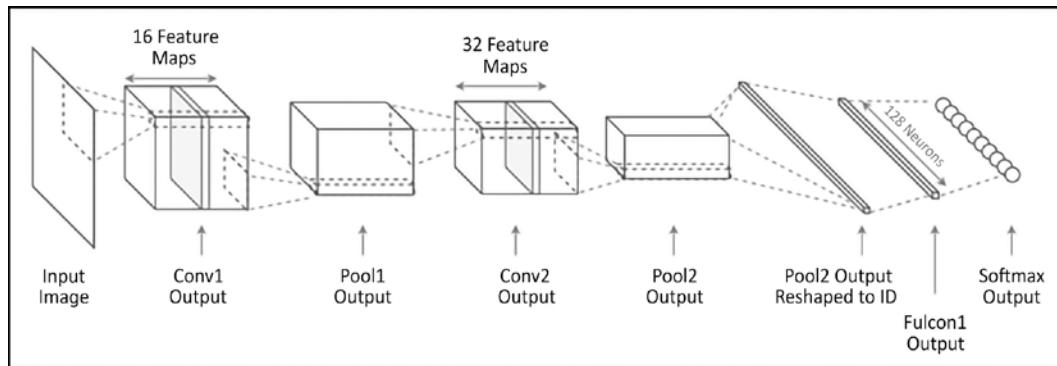


Figure 5.11: Combining convolution layers, pooling layers, and fully connected layers to form a CNN

With a strong conceptual understanding of a CNN, we will now get started on our first use case: classifying images with a CNN model.

Exercise – image classification on Fashion-MNIST with CNN

This will be our first example of using a CNN for a real-world machine learning task. We will classify images using a CNN. The reason for not starting with an NLP task is that applying CNNs to NLP tasks (for example, sentence classification) is not very straightforward. There are several tricks involved in using CNNs for such a task. However, originally, CNNs were designed to cope with image data. Therefore, let's start there, and then find our way through to see how CNNs apply to NLP tasks in the *Using CNNs for sentence classification* section.

About the data

In this exercise, we will use a dataset well-known in the computer vision community: the Fashion-MNIST dataset. Fashion-MNIST was inspired by the famous MNIST dataset (<http://yann.lecun.com/exdb/mnist/>). MNIST is a database of labeled images of handwritten digits from 0 to 9 (i.e. 10 digits). However, due to the simplicity of the MNIST image classification task, test accuracy on MNIST is just shy of 100%. At the time of writing, the popular research benchmarking site [paperswithcode.com](https://paperswithcode.com/sota/image-classification-on-mnist) has published a test accuracy of 99.87% (<https://paperswithcode.com/sota/image-classification-on-mnist>). Because of this, Fashion-MNIST came to life.

Fashion-MNIST consists of images of clothing garments. Our task is to classify each garment into a category (e.g. dress, t-shirt). The dataset contains two sets: the training set, and the test set. We will train on the training set and evaluate the performance of our model on the unseen test dataset. We will further split the training set into two sets: training and validation sets. We will use the validation dataset as a continuous performance monitoring mechanism for our model. We will discuss the details later, but we will see that we can reach up to approximately 88% test accuracy without any special regularization or tricks.

Downloading and exploring the data

The very first task will be to download and explore the data. To download the data, we will simply tap into the `tf.keras.datasets` module, as it provides several datasets to be downloaded conveniently through TensorFlow. To see what other datasets are available, visit https://www.tensorflow.org/api_docs/python/tf/keras/datasets. The full code for this chapter is available in `ch5_image_classification_fashion_mnist.ipynb` in the `Ch05-Sentence-Classification` folder. Simply call the following function to download the data:

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.  
datasets.fashion_mnist.load_data()
```

The data will be downloaded to a default cache directory specified by TensorFlow (for example: `~/.keras/dataset/fasion_minst`).

We will then see the sizes of the data by printing their shapes:

```
print("train_images is of shape: {}".format(train_images.shape))  
print("train_labels is of shape: {}".format(train_labels.shape))  
print("test_images is of shape: {}".format(test_images.shape))  
print("test_labels is of shape: {}".format(test_labels.shape))
```

This will produce:

```
train_images is of shape: (60000, 28, 28)
train_labels is of shape: (60000,)
test_images is of shape: (10000, 28, 28)
test_labels is of shape: (10000,)
```

We can see that we have 60,000 training images, each of size 28x28, and 10,000 testing images of the same dimensions. The labels are simple class IDs ranging from 0 to 9. We will also create a variable to contain the class ID to class name mapping, which will help us during explorations and post-training analysis:

```
# Available at: https://www.tensorflow.org/api_docs/python/tf/keras/
# datasets/fashion_mnist/Load_data
label_map = {
    0: "T-shirt/top", 1: "Trouser", 2: "Pullover", 3: "Dress", 4: "Coat",
    5: "Sandal", 6: "Shirt", 7: "Sneaker", 8: "Bag", 9: "Ankle boot"
}
```

We can also plot the images, which will give the following plot of images (*Figure 5.12*):

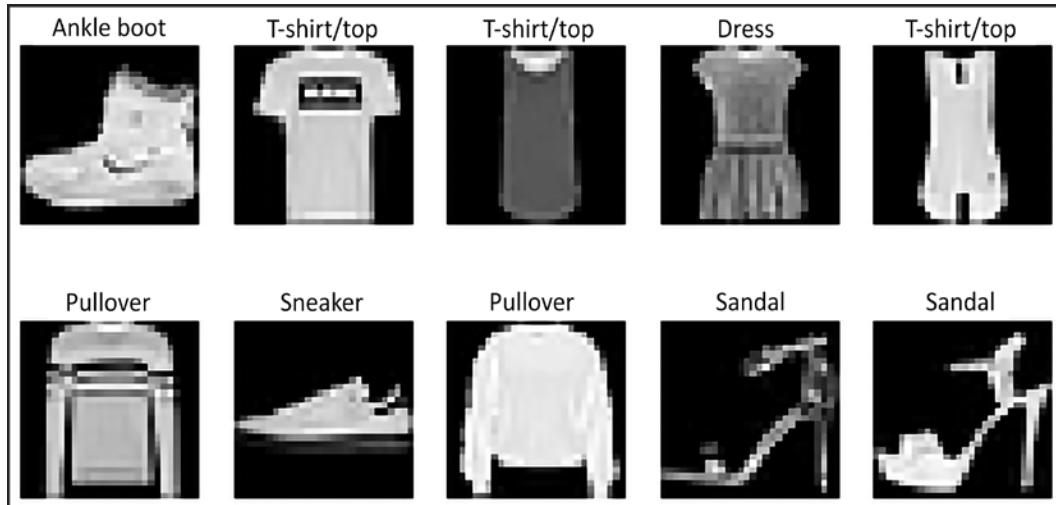


Figure 5.12: An overview of the images found in the Fashion-MNIST dataset

Finally, we are going to extend `train_images` and `test_images` by adding a new dimension (of size 1) to the end of each tensor. Standard implementation of the convolution operation in TensorFlow is designed to work on a four-dimensional input (i.e. batch, height, width, and channel dimensions).

Here, the channel dimension is omitted in the images as they are black and white images. Therefore, to comply with the dimensional requirement of TensorFlow's convolution operation, we add this additional dimension to the images. This is a necessity for using the convolution operation in CNNs. You can do this as follows:

```
train_images = train_images[:, :, :, None]  
test_images = test_images[:, :, :, None]
```

Using the indexing and slicing capabilities available in NumPy, you can simply add a `None` dimension to the tensor when indexing as above. Let's now check the shapes of the tensors:

```
print("train_images is of shape: {}".format(train_images.shape))  
print("test_images is of shape: {}".format(test_images.shape))
```

This gives:

```
train_images is of shape: (60000, 28, 28, 1)  
test_images is of shape: (10000, 28, 28, 1)
```

Let's have a crack at implementing a CNN model that can learn from this data.

Implementing the CNN

In this subsection, we will look at some important code snippets from the TensorFlow implementation of the CNN. The full code is available in `ch5_image_classification_mnist.ipynb` in the `Ch05-Sentence-Classification` folder. First, we will define several important hyperparameters. The code comments are self-explanatory for the purpose of these hyperparameters:

```
batch_size = 100 # This is the typical batch size we've been using  
  
image_size = 28 # This is the width/height of a single image  
  
# Number of color channels in an image. These are black and white images  
n_channels = 1  
  
# Number of different digits we have images for (i.e. classes)  
n_classes = 10
```

With that, we can start to implement the model. We will find inspiration from one of the earliest CNN models, known as LeNet, introduced in the paper *Gradient-Based Learning Applied to Document Recognition* by LeCun et al. (<http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>). This model will be a great start as it is a simple model yet gives a reasonably good performance on the dataset. We will introduce some slight modifications to the original model, because the original model operated on a 32x32-sized image, whereas in our case, the image is a 28x28-sized image.

Let's go through some quick details of the model. It has the following sequence of layers:

- A convolutional layer with a 5x5 kernel, 1x1 stride, and valid padding
- A max pooling layer with a 2x2 kernel, 2x2 stride, and valid pooling
- A convolutional layer with a 5x5 kernel, 1x1 stride, and valid pooling
- A max pooling layer with a 2x2 kernel, 2x2 stride, and valid pooling
- A convolutional layer with a 4x4 kernel, 1x1 stride, and valid pooling
- A layer that flattens the 2D output to a 1D vector
- A Dense layer with 84 nodes
- A final softmax prediction layer with 10 nodes

Here, all the layers except the last have ReLU (Rectified Linear Unit) activation. A convolutional layer in a CNN model generalizes the convolution operation we discussed, to work on multi-channel inputs and produce multi-channel outputs. Let's understand what we meant by that. The original convolution operation we saw operated on a simple 2D plane with a height h and width w . Next, the kernel moves over the plane while producing a single value at each position. This process produces another 2D plane. But in practice, CNN models operate on four-dimensional inputs, i.e. an input of size [batch size, height, width, in channels], and produce an output that is a four-dimensional, i.e. an output of size [batch size, height, width, out channels]. To produce this output, the kernel would need to be a four-dimensional tensor having the dimensions [kernel height, kernel width, in channels, out channels].

It might not be entirely clear why inputs, outputs, and kernels would be in this format. *Figure 5.13* clarifies this.

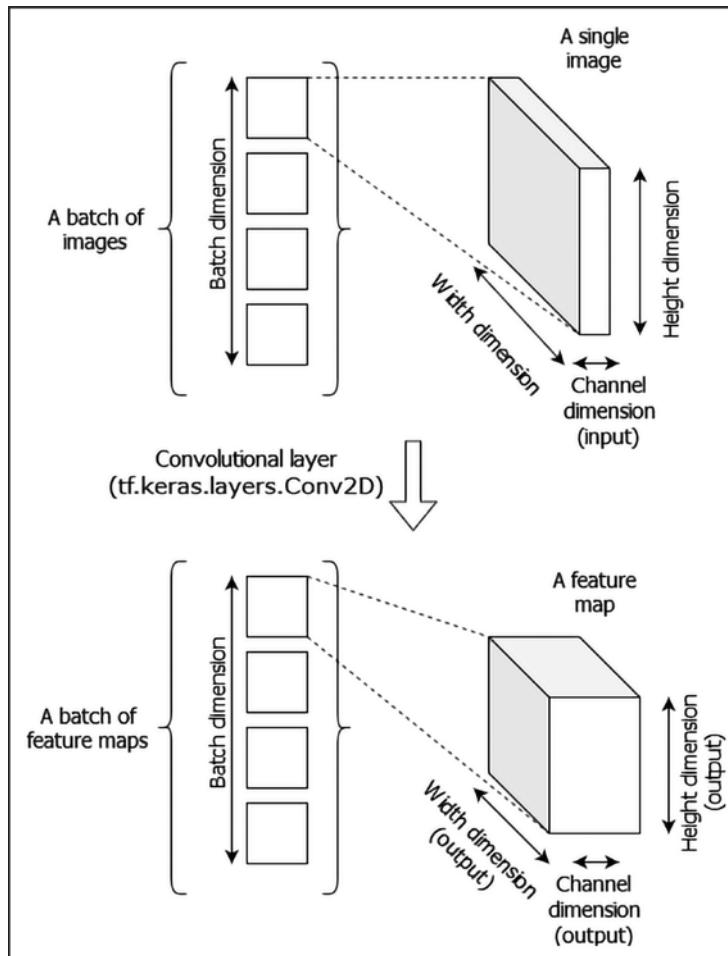


Figure 5.13: How input and output shapes look for a two-dimensional convolution layer

Below, we will outline the full model. Don't worry if you don't understand it at first glance. We will go through line by line to understand how the model comes to be:

```
from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten, Dense
from tensorflow.keras.models import Sequential
import tensorflow.keras.backend as K

K.clear_session()
```

```
lenet_like_model = Sequential([
    # 1st convolutional layer
    Conv2D(
        filters=16, kernel_size=(5,5), strides=(1,1), padding='valid',
        activation='relu',
        input_shape=(image_size,image_size,n_channels)
    ), # in 28x28 / out 24x24
    # 1st max pooling layer
    MaxPool2D(pool_size=(2,2), strides=(2,2), padding='valid'),
    # in 24x24 / out 12x12
    # 2nd convolutional layer
    Conv2D(filters=16, kernel_size=(5,5), strides=(1,1),
           padding='valid', activation='relu'), # in 12x12 / out 8x8
    # 2nd max pooling layer
    MaxPool2D(pool_size=(2,2), strides=(2,2), padding='valid'),
    # in 8x8 / out 4x4
    # 3rd convolutional layer
    Conv2D(filters=120, kernel_size=(4,4), strides=(1,1),
           padding='valid', activation='relu'), # in 4x4 / out 1x1
    # flatten the output of the last layer to suit a fully connected layer
    Flatten(),
    # First dense (fully-connected) layer
    Dense(84, activation='relu'),
    # Final prediction layer
    Dense(n_classes, activation='softmax')
])
```

The very first thing to notice is that we are using the Keras Sequential API. The CNN we are implementing here has a series of layers connected one after the other. Therefore, we will use the simplest API possible. We then have our first convolutional layer. We have already discussed the convolution operation. Let's take the first line:

```
Conv2D(
    filters=16, kernel_size=(5,5), strides=(1,1), padding='valid',
    activation='relu',
    input_shape=(image_size,image_size,n_channels)
)
```

The `tensorflow.keras.layers.Conv2D` layer takes the following argument values in that order:

- `filters` (int): This is the number of output filters (i.e. the number of out channels).
- `kernel_size` (Tuple[int]): This is the (height, width) of the convolution kernel.
- `strides` (Tuple[int]): This denotes the stride on the height and width dimension of the input.
- `padding` (str): This denotes the type of padding (can be 'SAME' or 'VALID').
- `activation` (str): The non-linear activation used.
- `input_shape` (Tuple[int]): The shape of the input. When defining `input_shape`, we do not specify the batch dimension as it's automatically added.

Next, we have the first max-pooling layer, which looks as follows:

```
MaxPool2D(pool_size=(2,2), strides=(2,2), padding='valid')
```

The arguments are quite similar to the ones in `tf.keras.layers.Conv2D`. The `pool_size` argument corresponds to the `kernel_size` argument that specifies the (height, width) of the pool window. Following a similar pattern, the following convolutional and pooling layers are defined. The final convolution layer produces a [batch size, 1, 1, 120]-sized output. The height and width dimensions are equal to 1, because LeNet is designed in a way that the last convolutional kernel has the same height and width as the output. Before this input is fed to a fully connected layer, we need to flatten this output, such that it has the shape [batch size, 120]. This is because a standard Dense layer takes a two-dimensional input. For that, we use the `tf.keras.layers.Flatten()` layer:

```
Flatten(),
```

Finally, we define two Dense layers as follows.

```
Dense(84, activation='relu'),
Dense(n_classes, activation='softmax')
```

As the final step, we will compile the model using the sparse categorical cross-entropy loss and the Adam optimizer. We will also track the accuracy on the data:

```
lenet_like_model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
```

With the data prepared and the model defined fully, we are good to train our model. Model training is as simple as calling one function:

```
lenet_like_model.fit(train_images, train_labels, validation_split=0.2,  
batch_size=batch_size, epochs=5)
```

The `tf.keras.layers.Model.fit()` takes many arguments. But let's only discuss the ones we have used here:

- `x` (`np.ndarray / tf.Tensor / other`): Takes in a tensor that will act as input to the model (implemented as a NumPy array or a TensorFlow tensor). But the accepted values are not limited just to tensors. To see the full list, please refer to https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit.
- `y` (`np.ndarray / tf.Tensor`): Takes in a tensor that will act as the labels (targets) for the model.
- `validation_split (float)`: Setting this argument means a fraction of training data (e.g. 0.2 translates to 20%) will be used as validation data.
- `epochs (int)`: The number of epochs to train the model for.

You can evaluate the trained model on the test data by calling:

```
lenet_like_model.evaluate(test_images, test_labels)
```

Once run, you'll see an output as follows:

```
313/313 [=====] - 1s 2ms/step - loss: 0.3368 -  
accuracy: 0.8806
```

The model should get up to around 88% accuracy when trained.

You just finished learning about the functions that we used to create our first CNN. You learned to use the functions to implement the CNN structure as well as define the loss, minimize the loss, and get predictions for unseen data. We used a simple CNN to see if it could learn to classify clothing items. Also, we were able to achieve an accuracy above 88% with a reasonably simple CNN. Next, we will analyze some of the results produced by the CNN. We will see why the CNN couldn't recognize some of the images correctly.

Analyzing the predictions produced with a CNN

Here, we can randomly pick some correctly and incorrectly classified samples from the test set to evaluate the learning power of CNNs (see *Figure 5.14*).

We can see that for the correctly classified instances, the CNN is very confident about the output, most of the time. This is a good sign that the model is making very confident and accurate decisions. However, when we evaluate the incorrectly classified examples, we can see that some of them are in fact difficult, and even a human can get some of them wrong. For example, for an ankle boot that's classified as a sandal, there is a large black patch that can indicate the presence of straps, which makes it more likely to be a sandal (the third image from the right in the third row). Also, in the fifth image from the right in the third row, it's difficult to say whether it's a shirt or a collared t-shirt:

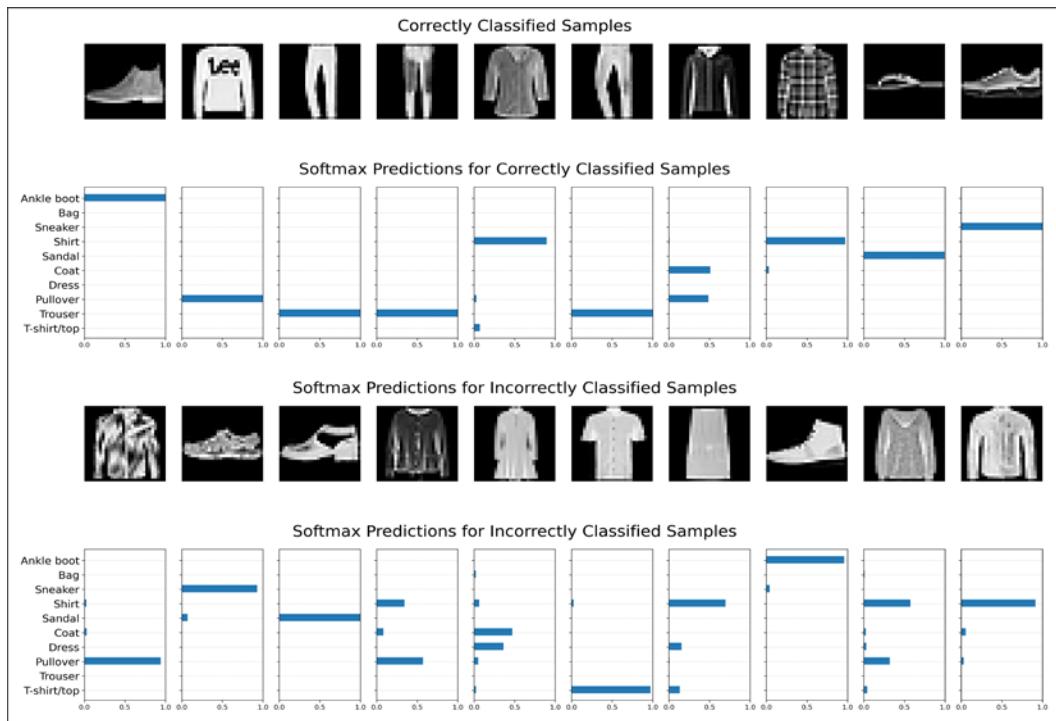


Figure 5.14: Fashion-MNIST correctly classified and misclassified instances

Using CNNs for sentence classification

Though CNNs have mostly been used for computer vision tasks, nothing stops them from being used in NLP applications. But as we highlighted earlier, CNNs were originally designed for visual content. Therefore, using CNNs for NLP tasks requires somewhat more effort. This is why we started out learning about CNNs with a simple computer vision problem. CNNs are an attractive choice for machine learning problems due to the low parameter count of convolution layers. One such NLP application for which CNNs have been used effectively is sentence classification.

In sentence classification, a given sentence should be classified with a class. We will use a question database, where each question is labeled by what the question is about. For example, the question “Who was Abraham Lincoln?” will be a question and its label will be *Person*. For this we will use a sentence classification dataset available at <http://cogcomp.org/Data/QA/QC/>; here you will find several datasets. We are using the set with ~5,500 training questions and their respective labels and 500 testing sentences.

We will use the CNN network introduced in a paper by Yoon Kim, *Convolutional Neural Networks for Sentence Classification*, to understand the value of CNNs for NLP tasks. However, using CNNs for sentence classification is somewhat different from the Fashion-MNIST example we discussed, because operations (for example, convolution and pooling) now happen in one dimension (length) rather than two dimensions (height and width). Furthermore, the pooling operations will also have a different flavor to the normal pooling operation, as we will see soon. You can find the code for this exercise in the `ch5_cnn_sentence_classification.ipynb` file in the Ch5-Sentence-Classification folder. As the first step, we will understand the data.

How data is transformed for sentence classification

Let's assume a sentence of p words. First, we will pad the sentence with some special words (if the length of the sentence is $< n$) to set the sentence length to n words, where $n \geq p$. Next, we will represent each word in the sentence by a vector of size k , where this vector can either be a one-hot-encoded representation, or Word2vec word vectors learned using skip-gram, CBOW, or GloVe. Then a batch of sentences of size b can be represented by a $b \times n \times k$ matrix.

Let's walk through an example. Let's consider the following three sentences:

- *Bob and Mary are friends.*
- *Bob plays soccer.*
- *Mary likes to sing in the choir.*

In this example, the third sentence has the most words, so let's set $n = 7$, which is the number of words in the third sentence. Next, let's look at the one-hot-encoded representation for each word. In this case, there are 13 distinct words. Therefore, we get this:

Bob: 1,0,0,0,0,0,0,0,0,0,0,0,0

and: 0,1,0,0,0,0,0,0,0,0,0,0,0

Mary: 0,0,1,0,0,0,0,0,0,0,0,0,0

Also, $k = 13$ for the same reason. With this representation, we can represent the three sentences as a three-dimensional matrix of size $3 \times 7 \times 13$, as shown in *Figure 5.15*:

		Mary	0	0	1	0	0	0	0	0	0	0	0	s'
		likes	0	0	0	0	0	0	0	1	0	0	0	0
		to	0	0	0	0	0	0	0	0	1	0	0	0
	Bob		1	0	0	0	0	0	0	0	0	0	0	0
	plays		0	0	0	0	0	1	0	0	0	0	0	0
	soccer		0	0	0	0	0	0	1	0	0	0	0	0
Bob			1	0	0	0	0	0	0	0	0	0	0	0
and			0	1	0	0	0	0	0	0	0	0	0	0
Mary			0	0	1	0	0	0	0	0	0	0	0	0
are			0	0	0	1	0	0	0	0	0	0	0	0
friends			0	0	0	0	1	0	0	0	0	0	0	0
PAD			0	0	0	0	0	0	0	0	0	0	0	0
PAD			0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.15: A batch of sentences represented as a sentence matrix

You could also utilize word embeddings instead of one-hot encoding here. Representing each word as a one-hot-encoded feature introduces sparsity and wastes computational memory. By using embeddings, we are enabling the model to learn more compact and powerful word representations than one-hot-encoded representations. This also means that k becomes a hyperparameter (i.e. the embedding size), as opposed to being driven by the size of the vocabulary. This means that, in *Figure 5.15*, each column will be a distributed continuous vector, not a combination of 0s and 1s.

We know that one-hot vectors lead to high-dimensional and highly sparse representations that are sub-optimal. On the other hand, word vectors give richer representations of words. However, learning word vectors is computationally costly. There is another alternative called the hashing trick. The beauty of the hashing trick is that it is extremely simple but gives a powerful and economical alternative that sits between one-hot vectors and word vectors. The idea behind the hashing trick is to use a hash function that converts a given token to an integer.



$$f(<\text{token}>) \rightarrow \text{hash value}$$

Here f is a chosen hash function. Some example popular hash functions are SHA (<https://brilliant.org/wiki/secure-hashing-algorithms/>) and MD5 (<https://searchsecurity.techtarget.com/definition/MD5>). There's also more advanced hashing such as locality-sensitive hashing (<https://www.pinecone.io/learn/locality-sensitive-hashing/>) to give out similar IDs for morphologically similar words. You can easily use the hashing trick via TensorFlow (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/hashing_trick).

Implementation – downloading and preparing data

First we will download the data from the web. The data download functions are provided in the notebook and are simply downloading two files: training and testing data (the paths to the files are retained in `train_filename` and `test_filename`).

If you open these files you will see that they contain a collection of lines of text. Each line has the format:

```
<Category>: <sub-category> <question>
```

There are two pieces of meta information for each question: a category and a sub-category. A category is a macro-level classification, where sub-category is a finer grain identification of the type of the question. There are six categories available: DESC (description-related), ENTY (entity-related), HUM (human-related), ABBR (abbreviation related), NUM (numerical), and LOC (location related). Each category has several sub-categories associated with them. For example, the ENTY category is further broken down to animal, currency, events, food, etc. For our problem, we will be focusing on high-level classification (i.e. six classes), but you could also leverage the same model with minimal changes to classify on the sub-category level.

Once the files are downloaded, we'll read the data into the memory. For that, we will implement the `read_data()` function:

```
def read_data(filename):
    ...
    Read data from a file with given filename
    Returns a list of strings where each string is a lower case word
    ...

    # Holds question strings, categories and sub categories
    # category/sub_category definitions: https://cogcomp.seas.upenn.edu/
    # Data/QA/QC/definition.html
    questions, categories, sub_categories = [], [], []

    with open(filename, 'r', encoding='latin-1') as f:
        # Read each line
        for row in f:
            # Each string has format <cat>:<sub_cat> <question>
            # Split by : to separate cat and (sub_cat + question)
            row_str = row.split(":")
            cat, sub_cat_and_question = row_str[0], row_str[1]
            tokens = sub_cat_and_question.split(' ')
            # The first word in sub_cat_and_question is the sub
            # category rest is the question
            sub_cat, question = tokens[0], ' '.join(tokens[1:])

            questions.append(question.lower().strip())
            categories.append(cat)
            sub_categories.append(sub_cat)

    return questions, categories, sub_categories
```

```
train_questions, train_categories, train_sub_categories = read_data(train_filename)
test_questions, test_categories, test_sub_categories = read_data(test_filename)
```

This function simply goes through each line in the file and separates the question, category, and sub-category, using the format of each line elucidated above. After that, each question, category, and sub-category is written to the lists `questions`, `categories`, and `sub_categories` respectively. Finally, the function returns these lists. With the `questions`, `categories`, and `sub_categories` available for both training and testing data, we will create pandas DataFrames for training and testing data.

pandas DataFrames are an expressive data structure for storing multi-dimensional data. A DataFrame can have indices, columns, and values. Each value has a specific index and a column. It is quite simple to create a DataFrame:

```
# Define training and testing
train_df = pd.DataFrame(
    {'question': train_questions, 'category': train_categories,
     'sub_category': train_sub_categories}
)
test_df = pd.DataFrame(
    {'question': test_questions, 'category': test_categories,
     'sub_category': test_sub_categories}
)
```

We call the `pd.DataFrame` construct with a dictionary. The keys of the dictionary represent columns of the DataFrame, and the values represent the elements in each column. Here we create three columns: `question`, `category`, and `sub_category`.

Figure 5.16 depicts what the `train_df` looks like.

	question	category	sub_category
0	how did serfdom develop in and then leave russ...	DESC	manner
1	what films featured the character popeye doyle ?	ENTY	cremat
2	how can i find a list of celebrities ' real na...	DESC	manner
3	what fowl grabs the spotlight after the chines...	ENTY	animal
4	what is the full form of .com ?	ABBR	exp
5	what contemptible scoundrel stole the cork fro...	HUM	ind
6	what team did baseball 's st. louis browns bec...	HUM	gr
7	what is the oldest profession ?	HUM	title
8	what are liver enzymes ?	DESC	def
9	name the scar-faced bounty hunter of the old w...	HUM	ind

Figure 5.16: A sample of data captured in the pandas DataFrame

We will do a simple shuffle of rows in the training set, to make sure we are not introducing any unintentional ordering in the data:

```
# Shuffle the data for better randomization
train_df = train_df.sample(frac=1.0, random_state=seed)
```

This process will sample 100% of the data from the DataFrame randomly. In other words, it will shuffle the order of the rows. From this point onward, we will not consider the `sub_category` column. We will first map each class label to a class ID:

```
# Generate the Label to ID mapping
unique_cats = train_df["category"].unique()
labels_map = dict(zip(unique_cats, np.arange(unique_cats.shape[0])))
print("Label->ID mapping: {}".format(labels_map))

n_classes = len(labels_map)

# Convert all string Labels to IDs
train_df["category"] = train_df["category"].map(labels_map)
test_df["category"] = test_df["category"].map(labels_map)
```

We first identify the unique values present in the `train_df["category"]`. Then we will create a dictionary by mapping from the unique values to a list of numerical IDs (0 to 5). The `np.arange()` function can be used to generate a series of integers in a specified range (here, the range is from 0 to the length of `unique_cats`). This process will give us the following `labels_map`.

```
Label->ID mapping: {0: 0, 1: 1, 2: 2, 4: 3, 3: 4, 5: 5}
```

Then we simply apply this mapping to the category column of both the train and test DataFrames to convert string labels to numerical labels. The data would look as follows, after the transformation (*Figure 5.17*).

		question	category	sub_category
4343	how can i get started in writing for television ?	0	manner	
2318	what were the achievements of richard nixon ?	1	other	
2808	what was the alternate to vhs ?	1	other	
3217	what country would you visit to ski in the dol...	2	country	
3966	what country imposed the berlin blockade in 19...	2	country	
5189	where on the web is adventours tours from sydn...	2	other	
1675	what u.s. state ends with a g ?	2	state	
1408	what country was sir edmund hillary born in ?	2	country	
4916	what was the name of the u.s. navy gunboat in ...	1	veh	
730	what 19th-century writer had a country estate ...	3	ind	

Figure 5.17: A sample of data in the DataFrame after mapping categories to integers

We create a validation set, stemming from the original training set, to monitor model performance while it trains. We will use the `train_test_split()` function from the scikit-learn library. 10% of the data will be separated as validation data, while 90% is kept as training data.

```
from sklearn.model_selection import train_test_split

train_df, valid_df = train_test_split(train_df, test_size=0.1)
print("Train size: {}".format(train_df.shape))
print("Valid size: {}".format(valid_df.shape))
```

This outputs:

```
Train size: (4906, 3)
Valid size: (546, 3)
```

We can see that approximately 4,900 examples are used as training and the rest as validation. In the next section, we will build a tokenizer to tokenize the questions and assign individual tokens numerical IDs.

Implementation – building a tokenizer

Moving on, now it's time to build a tokenizer that can map words to numerical IDs:

```
from tensorflow.keras.preprocessing.text import Tokenizer

# Define a tokenizer and fit on train data
tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_df["question"].tolist())
```

Here we simply create a `Tokenizer` object and use the `fit_on_texts()` function to train it on the training corpus. In this process, the tokenizer will map words in the vocabulary to IDs. We will convert all of the train, validation, and test inputs to sequences of word IDs. Simply call the `tokenizer.texts_to_sequences()` function with a list of strings, where each string represents a question:

```
# Convert each list of tokens to a list of IDs, using tokenizer's mapping
train_sequences = tokenizer.texts_to_sequences(train_df["question"].
    tolist())
valid_sequences = tokenizer.texts_to_sequences(valid_df["question"].
    tolist())
test_sequences = tokenizer.texts_to_sequences(test_df["question"].
    tolist())
```

It's important to understand that we are feeding our model a batch of questions at a given time. It is very unlikely that all of the questions have the same number of tokens. If all questions do not have the same number of tokens, we cannot form a tensor due to the uneven lengths of different questions. To solve this, we have to pad shorter sequences with special tokens and truncate sequences longer than a specified length. To achieve this we can easily use the `tf.keras.preprocessing.sequence.pad_sequences()` function. It would be worthwhile going through the arguments accepted by this function:

- `sequences` (`List[List[int]]`) – List of list integers; each list of integers is a sequence
- `maxlen` (`int`) – The maximum padding length
- `padding` (`string`) – Whether to pad at the beginning (`pre`) or end (`post`)
- `truncating` (`string`) – Whether to truncate at the beginning (`pre`) or end (`post`)

- `value (int)` – What value is to be used for padding (defaults to 0)

Below we use this function to create sequence matrices for training, validation, and testing data:

```
max_seq_length = 22

# Pad shorter sentences and truncate longer ones (maximum length: max_seq_
# length)
preprocessed_train_sequences = tf.keras.preprocessing.sequence.pad_
sequences(
    train_sequences, maxlen=max_seq_length, padding='post',
    truncating='post'
)
preprocessed_valid_sequences = tf.keras.preprocessing.sequence.pad_
sequences(
    valid_sequences, maxlen=max_seq_length, padding='post',
    truncating='post'
)
preprocessed_test_sequences = tf.keras.preprocessing.sequence.pad_
sequences(
    test_sequences, maxlen=max_seq_length, padding='post',
    truncating='post'
)
```

The reason we picked 22 as the sequence length is through a simple analysis. The 99% percentile of the sequence lengths of the training corpus is equal to 22. Therefore, we have picked that. Another important statistic is that the vocabulary size will be approximately 7,880 words. Now we will discuss the model.

The sentence classification CNN model

Now we will discuss the technical details of the CNN used for sentence classification. First, we will discuss how data or sentences are transformed into a preferred format that can easily be dealt with by CNNs. Next, we will discuss how the convolution and pooling operations are adapted for sentence classification, and finally, we will discuss how all these components are connected.

The convolution operation

If we ignore the batch size, that is, if we assume that we are only processing a single sentence at a time, our data is a $n \times k$ matrix, where n is the number of words per sentence after padding, and k is the dimension of a single word vector. In our example, this would be 7×13 .

Now we will define our convolution weight matrix to be of size $m \times k$, where m is the filter size for a one-dimensional convolution operation. By convolving the input x of size $n \times k$ with a weight matrix W of size $m \times k$, we will produce an output of h of size $1 \times n$ as follows:

$$h_{i,1} = \sum_{j=1}^m \sum_{l=1}^k w_{j,l} x_{i+j-1,l}$$

Here, $w_{i,j}$ is the $(i,j)^{\text{th}}$ element of W and we will pad x with zeros so that h is of size $1 \times n$. Also, we will define this operation more simply, as shown here:

$$h = W * x + b$$

Here, $*$ defines the convolution operation (with padding) and we will add an additional scalar bias b . *Figure 5.18* illustrates this operation:

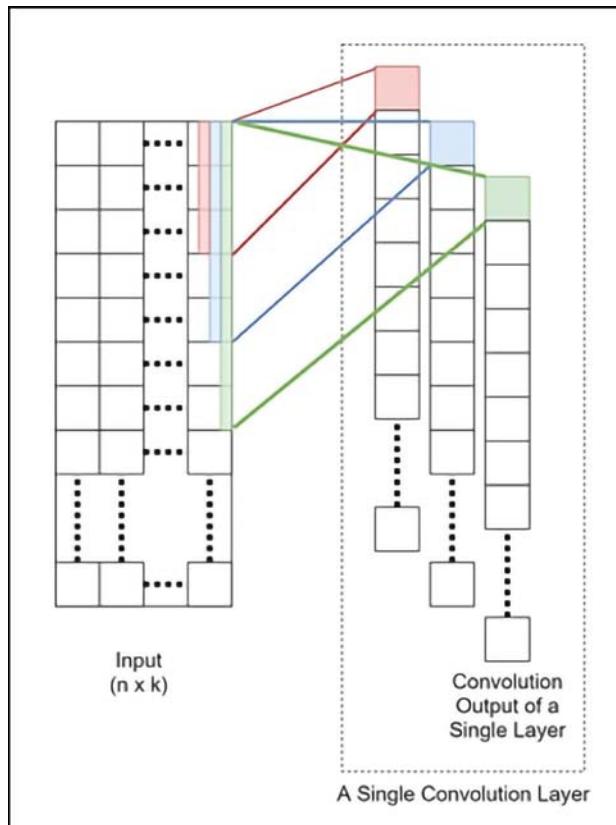


Figure 5.18: A convolution operation for sentence classification. Convolution layers with different kernel widths are used to convolve over the sentence (i.e. sequence of tokens)

Then, to learn a rich set of features, we have parallel layers with different convolution filter sizes. Each convolution layer outputs a hidden vector of size $1 \times n$, and we will concatenate these outputs to form the input to the next layer of size $q \times n$, where q is the number of parallel layers we will use. The larger q is, the better the performance of the model.

The value of convolving can be understood in the following manner. Think about the movie rating learning problem (with two classes, positive or negative), and we have the following sentences:

- *I like the movie, not too bad*
- *I did not like the movie, bad*

Now imagine a convolution window of size 5. Let's bin the words according to the movement of the convolution window.

The sentence *I like the movie, not too bad* gives:

[I, like, the, movie, ',']

[like, the, movie, ',', not]

[the, movie, ',', not, too]

[movie, ',', not, too, bad]

The sentence *I did not like the movie, bad* gives the following:

[I, did, not, like, the]

[did, not, like, the, movie]

[not, like, the, movie, ',']

[like, the, movie, ',', bad]

For the first sentence, windows such as the following convey that the rating is positive:

[I, like, the, movie, ',']

[movie, ',', not, too, bad]

However, for the second sentence, windows such as the following convey negativity in the rating:

[did, not, like, the, movie]

We are able to see such patterns that help to classify ratings thanks to the preserved spatiality. For example, if you use a technique such as *bag-of-words* to calculate sentence representations that lose spatial information, the sentence representations of the above two sentences would be highly similar. The convolution operation plays an important role in preserving the spatial information of the sentences.

Having q different layers with different filter sizes, the network learns to extract the rating with different size phrases, leading to an improved performance.

Pooling over time

The pooling operation is designed to subsample the outputs produced by the previously discussed parallel convolution layers. This is achieved as follows.

Let's assume the output of the last layer h is of size $q \times n$. The pooling over time layer would produce an output h' of size $q \times 1$ output. The precise calculation would be as follows:

$$h'_{i,1} = \{\max(h^{(i)})\}, \text{ where } 1 \leq i \leq q$$

Here, $h^{(i)} = W^{(i)} * x + b$ and $h^{(i)}$ is the output produced by the i^{th} convolution layer and $W^{(i)}$ is the set of weights belonging to that layer. Simply put, the pooling over time operation creates a vector by concatenating the maximum element of each convolution layer.

We will illustrate this operation in *Figure 5.19*:

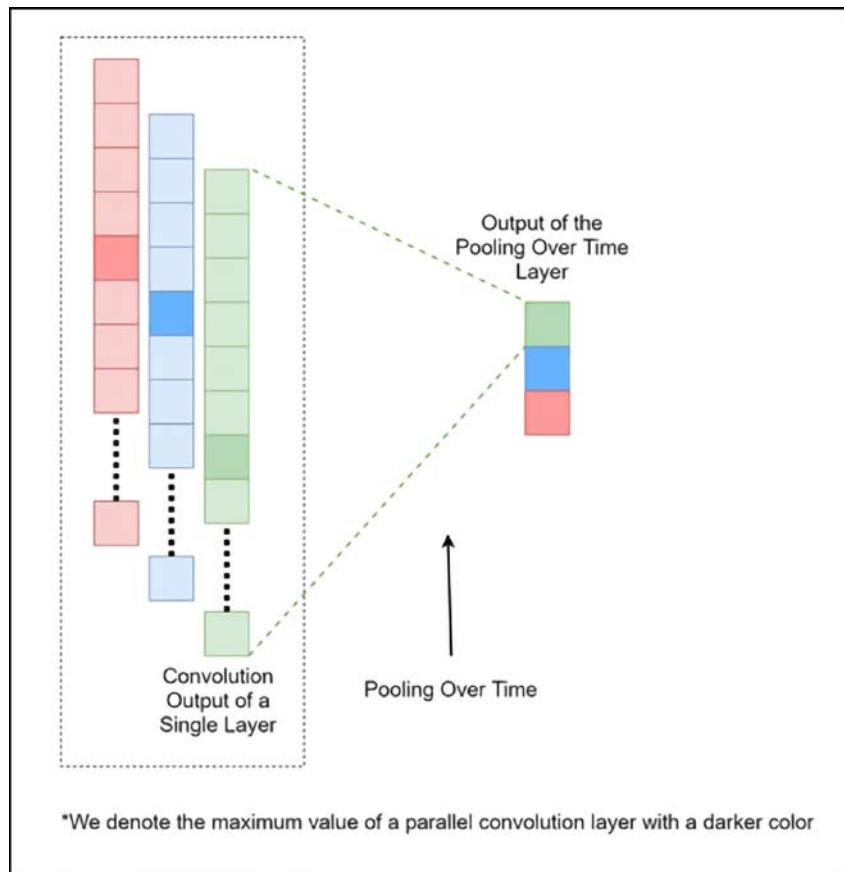


Figure 5.19: The pooling over time operation for sentence classification

By combining these operations, we finally arrive at the architecture shown in *Figure 5.20*:

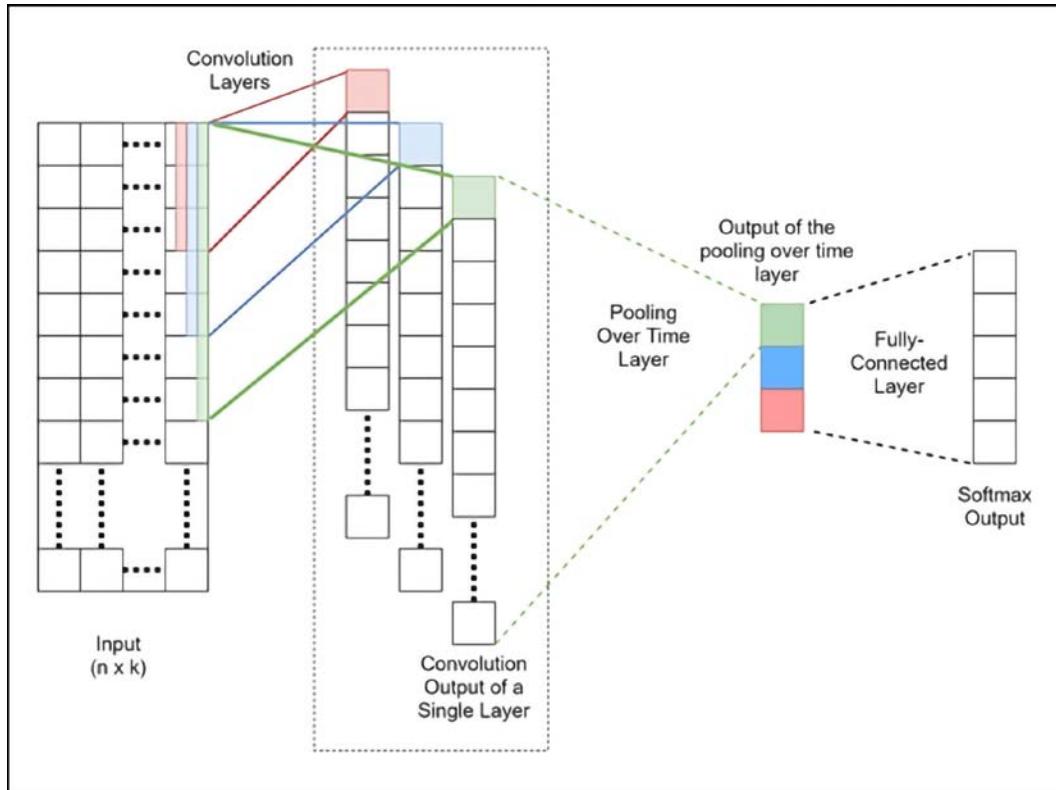


Figure 5. 20: A sentence classification CNN architecture. The pool of convolution layers having different kernel widths produces a set of output sequences. They are fed into the Pooling Over Time Layer that produces a compact representation of that input. This is finally connected to a classification layer with softmax activation

Implementation – sentence classification with CNNs

We are off implementing the model in TensorFlow 2. As a prerequisite, let's import several necessary modules from TensorFlow:

```
import tensorflow.keras.backend as K
import tensorflow.keras.layers as layers
import tensorflow.keras.regularizers as regularizers
from tensorflow.keras.models import Model
```

Clear the running session to make sure previous runs are not interfering with the current run:

```
K.clear_session()
```

Before we start, we will be using the Functional API from Keras. The reason for this is that the model we will be building here cannot be built with the Sequential API, due to intricate pathways present in the model. Let's start off by creating an input layer:

```
Input Layer takes word IDs as inputs
word_id_inputs = layers.Input(shape=(max_seq_length,), dtype='int32')
```

The input layer simply takes a batch of `max_seq_length` word IDs. That is, a batch of sequences, where each sequence is padded/truncated to a max length. We specify the `dtype` as `int32`, since they are word IDs. Next, we define an embedding layer, from which we will look up embeddings corresponding to the word IDs coming through the `word_id_inputs` layer:

```
# Get the embeddings of the inputs / out [batch_size, sent_length,
# output_dim]
embedding_out = layers.Embedding(input_dim=n_vocab, output_dim=64)(word_
id_inputs)
```

This is a randomly initialized embedding layer. It contains a large matrix of size [`n_vocab`, 64], where each row represents the word vector of the word indexed by that row number. The embeddings will be jointly learned with the model, while the model is trained on the supervised task. For the next part, we will define three different one-dimensional convolution layers with three different kernel (filter) sizes of 3, 4, and 5, having 100 feature maps each:

```
# For all Layers: in [batch_size, sent_length, emb_size] / out [batch_
# size, sent_length, 100]
conv1_1 = layers.Conv1D(
    100, kernel_size=3, strides=1, padding='same',
    activation='relu'
)(embedding_out)
conv1_2 = layers.Conv1D(
    100, kernel_size=4, strides=1, padding='same',
    activation='relu'
)(embedding_out)
conv1_3 = layers.Conv1D(
    100, kernel_size=5, strides=1, padding='same',
    activation='relu'
)(embedding_out)
```

An important distinction to make here is that we are using one-dimensional convolution as opposed to the two-dimensional convolution we used in the earlier exercise. However, most of the concepts remain the same. The main difference is that, unlike `tf.keras.layers.Conv2D`, which works on four-dimensional inputs, `tf.keras.layers.Conv1D` operates on three-dimensional inputs (i.e. inputs with shape `[batch_size, width, in_channels]`). In other words, the convolution kernel moves only in one direction over the inputs. Each of these layers produces a `[batch_size, sentence_length, 100]`-sized output. Afterward, these outputs are concatenated on the last axis to produce a single tensor:

```
# in previous conv outputs / out [batch_size, sent_length, 300]
conv_out = layers.concatenate(axis=-1)([conv1_1, conv1_2, conv1_3])
```

Subsequently, the new tensor of size `[batch_size, sentence_length, 300]` will be used to perform the pooling over time operation. We can implement the pooling over time operation by defining a one-dimensional max-pooling layer (i.e. `tf.keras.layers.MaxPool1D`) with a window as wide as the sequence length. This will produce a single value as the output, for each feature map in `conv_out`:

```
# Pooling over time operation.
# This is doing the max pooling over sequence length
# in other words, each feature map results in a single output
# in [batch_size, sent_length, 300] / out [batch_size, 1, 300]
pool_over_time_out = layers.MaxPool1D(pool_size=max_seq_length,
padding='valid')(conv_out)
```

Here we get a `[batch_size, 1, 300]`-sized output after performing the operation. Next, we will convert this output to a `[batch_size, 300]`-sized output, by using the `tf.keras.layers.Flatten` layer. The Flatten layer simply collapses all the dimensions (except the batch dimension) to a single dimension:

```
# Flatten the unit Length dimension
flatten_out = layers.Flatten()(pool_over_time_out)
```

Finally, `flatten_out` is passed to a Dense layer that has `n_classes` (i.e. six) nodes as the output and has a softmax activation:

```
# Compute the final output
out = layers.Dense(
    n_classes, activation='softmax',
```

```
    kernel_regularizer=regularizers.l1(0.001)
)(flatten_out)
```

Note the use of the `kernel_regularizer` argument. We can use this argument to add any special regularization (e.g. L1 or L2 regularization) to a given layer. Finally, we define a model as,

```
# Define the model  
cnn_model = Model(inputs=word_id_inputs, outputs=out)
```

and compile the model with the desired loss function, an optimizer, and metrics:

```
# Compile the model with loss/optimzier/metrics
cnn_model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
```

You can view the model by running the following line:

```
cnn_model.summary()
```

which gives,

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	[(None, 22)]	0	
embedding (Embedding)	(None, 22, 64)	504320	input_1[0][0]
conv1d (Conv1D)	(None, 22, 100)	19300	embedding[0][0]
conv1d_1 (Conv1D)	(None, 22, 100)	25700	embedding[0][0]
conv1d_2 (Conv1D)	(None, 22, 100)	32100	embedding[0][0]
concatenate (Concatenate)	(None, 22, 300)	0	conv1d[0][0] conv1d_1[0][0]

```

conv1d_2[0][0]

max_pooling1d (MaxPooling1D) (None, 1, 300)      0      concatenate[0][0]
=====
flatten (Flatten)           (None, 300)          0      max_pooling1d[0][0]
=====
dense (Dense)              (None, 6)            1806   flatten[0][0]
=====
Total params: 583,226
Trainable params: 583,226
Non-trainable params: 0
=====
```

Next, we will train the model on the data we already prepared.

Training the model

Since we have done the hard yard at the beginning, by making sure the data is transformed, training the model is simple. All we need to do is call the `tf.keras.layers.Model.fit()` function. However, let's leverage a few techniques to improve model performance. This will be done by leveraging a built-in callback of TensorFlow. The technique we'll be using is known as "decaying the learning rate." The idea is to reduce the learning rate (by some fraction) whenever the model has stopped to improve performance. The following callback assists us to do this:

```

# Call backs
lr_reduce_callback = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss', factor=0.1, patience=3, verbose=1,
    mode='auto', min_delta=0.0001, min_lr=0.000001
)
```

The parameters can be set as you wish, to control the learning rate reduction. Let's understand the arguments above:

- `monitor` (`str`) – Which metric to monitor in order to decay the learning rate. We will monitor the validation loss
- `factor` (`float`) – By how much to reduce the learning rate. For example, a factor of 0.1 means that the learning rate will be reduced by 10 times (e.g. 0.01 will be stepped down to 0.001)

- `patience` (`int`) – How many epochs to wait without an improvement, before reducing the learning rate
- `mode` (`string`) – Whether to look for an increase or decrease of the metric; ‘auto’ means that the direction will be determined by looking at the metric name
- `min_delta` (`float`) – How much of an increase/decrease to consider as an improvement
- `min_lr` (`float`) – Minimum learning rate (floor)

Let’s train the model:

```
# Train the model
cnn_model.fit(
    preprocessed_train_sequences, train_labels,
    validation_data=(preprocessed_valid_sequences, valid_labels),
    batch_size=128,
    epochs=25,
    callbacks=[lr_reduce_callback]
)
```

We will see the accuracy quickly going up and the validation accuracy plateauing around 88%. Here’s a snippet of the output produced:

```
Epoch 1/50
39/39 [=====] - 1s 9ms/step - loss: 1.7147 -
accuracy: 0.3063 - val_loss: 1.3912 - val_accuracy: 0.5696
Epoch 2/50
39/39 [=====] - 0s 6ms/step - loss: 1.2268 -
accuracy: 0.6052 - val_loss: 0.7832 - val_accuracy: 0.7509

...
Epoch 00015: ReduceLROnPlateau reducing learning rate to
1.0000000656873453e-06.
Epoch 16/50
39/39 [=====] - 0s 6ms/step - loss: 0.0487 -
accuracy: 0.9999 - val_loss: 0.3639 - val_accuracy: 0.8846
Restoring model weights from the end of the best epoch.
Epoch 00016: early stopping
```

Next, let's test the model on the testing dataset:

```
cnn_model.evaluate(preprocessed_test_sequences, test_labels, return_
dict=True)
```

Evaluating the test data as given in the exercise gives us a test accuracy of close to 88% (for 500 test sentences) in this sentence classification task.

Here we end our discussion about using CNNs for sentence classification. We first discussed how one-dimensional convolution operations combined with a special pooling operation called *pooling over time* can be used to implement a sentence classifier based on the CNN architecture. Finally, we discussed how to use TensorFlow to implement such a CNN and saw that it in fact performs well in sentence classification.

It can be useful to know how the problem we just solved can be useful in the real world. Assume that you have a large document about the history of Rome in your hand, and you want to find out about Julius Caesar without reading the whole document. In this situation, the sentence classifier we just implemented can be used as a handy tool to summarize the sentences that only correspond to a person, so you don't have to read the whole document.

Sentence classification can be used for many other tasks as well; one common use of this is classifying movie reviews as positive or negative, which is useful for automating the computation of movie ratings. Another important application of sentence classification can be seen in the medical domain, where it is used to extract clinically useful sentences from large documents containing large amounts of text.

Summary

In this chapter, we discussed CNNs and their various applications. First, we went through a detailed explanation of what CNNs are and their ability to excel at machine learning tasks. Next we decomposed the CNN into several components, such as convolution and pooling layers, and discussed in detail how these operators work. Furthermore, we discussed several hyperparameters that are related to these operators such as filter size, stride, and padding.

Then, to illustrate the functionality of CNNs, we walked through a simple example of classifying images of garments. We also did a bit of analysis to see why the CNN fails to recognize some images correctly.

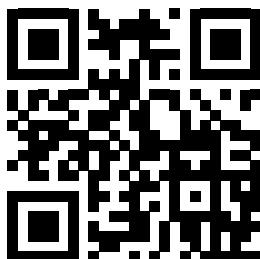
Finally, we started talking about how CNNs are applied for NLP tasks. Concretely, we discussed an altered architecture of CNNs that can be used to classify sentences. We then implemented this particular CNN architecture and tested it on an actual sentence classification task.

In the next chapter, we will move on to one of the most popular types of neural networks used for many NLP tasks – **Recurrent Neural Networks (RNNs)**.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside
more than 1000 members at: <https://packt.link/nlp>



6

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a special family of neural networks that are designed to cope with sequential data (that is, time-series data), such as stock market prices or a sequence of texts (for example, variable-length sentences). RNNs maintain a state variable that captures the various patterns present in sequential data; therefore, they are able to model sequential data. In comparison, conventional feed-forward neural networks do not have this ability unless the data is represented with a feature representation that captures the important patterns present in the sequence. However, coming up with such feature representations is extremely difficult. Another alternative for feed-forward models to model sequential data is to have a separate set of parameters for each position in time/sequence so that the set of parameters assigned to a certain position learns about the patterns that occur at that position. This will greatly increase the memory requirement for your model.

However, as opposed to having a separate set of parameters for each position like feed-forward networks, RNNs share the same set of parameters over time. Sharing parameters over time is an important part of RNNs and in fact is one of the main enablers for learning temporal patterns. Then the state variable is updated over time for each input we observe in the sequence. These parameters shared over time, combined with the state vector, are able to predict the next value of a sequence, given the previously observed values of the sequence. Furthermore, since we process a single element of a sequence at a time (for example, one word in a document at a time), RNNs can process data of arbitrary lengths without padding data with special tokens.

In this chapter, we will dive into the details of RNNs. First, we will discuss how an RNN can be formed by starting with a simple feed-forward model.

After this we will discuss the basic functionality of an RNN. We will also delve into the underlying equations, such as output calculation and parameter update rules of RNNs, and discuss several variants of applications of RNNs: one-to-one, one-to-many, and many-to-many RNNs. We will walk through an example of using RNNs to identify named entities (e.g. person names, organization, etc.), which has valuable downstream use cases like building knowledge bases. We will discuss a more complex RNN model that can read text both forward and backward, and uses convolutional layers to increase the model accuracy. This chapter will cover this through the following main topics:

- Understanding RNNs
- Backpropagation Through Time
- Applications of RNNs
- Named Entity Recognition (NER) with RNNs
- NER with character and token embeddings

Understanding RNNs

In this section, we will discuss what an RNN is by starting with a gentle introduction, and then move on to more in-depth technical details. We mentioned earlier that RNNs maintain a state variable that evolves over time as the RNN sees more data, thus giving it the power to model sequential data. In particular, this state variable is updated over time by a set of recurrent connections. The existence of recurrent connections is the main structural difference between an RNN and a feed-forward network. The recurrent connections can be understood as links between a series of memories that the RNN learned in the past, connecting to the current state variable of the RNN. In other words, the recurrent connections update the current state variable with respect to the past memory the RNN has, enabling the RNN to make a prediction based on the current input as well as the previous inputs.



The term RNN is sometimes used to refer to the family of recurrent models, which has many different models. In other words, it is sometimes used as a generalization of a specific RNN variant. Here, we are using the term RNN to refer to one of the earliest implementations of an RNN model known as the Elman network.

In the upcoming section, we will discuss the following topics. First, we will discuss how we can start by representing a feed-forward network as a computational graph.

Then we will see through an example why a feed-forward network might fail at a sequential task. Then we will adapt that feed-forward graph to model sequential data, which will give us the basic computational graph of an RNN. We will also discuss the technical details (for example, update rules) of an RNN. Finally, we will discuss the details of how we can train RNN models.

The problem with feed-forward neural networks

To understand the limits of feed-forward neural networks and how RNNs address them, let's imagine a sequence of data:

$$x = \{x_1, x_2, \dots, x_T\}, y = \{y_1, y_2, \dots, y_T\}$$

Next, let's assume that, in the real world, x and y are linked in the following relationship:

$$h_t = g_1(x_t, h_{t-1})$$

$$y_t = g_2(h_t)$$

Here, g_1 and g_2 are transformations (e.g. multiplying with a weight matrix followed by a non-linear transformation). This means that the current output y_t depends on the current state h_t , where h_t is calculated with the current input x_t and previous state h_{t-1} . The state encodes information about previous inputs observed historically by the model.

Now, let's imagine a simple feed-forward neural network, which we will represent with the following:

$$y_t = f(x_t; \theta)$$

Here, y_t is the predicted output for some input x_t .

If we use a feed-forward neural network to solve this task, the network will have to produce $\{y_1, y_2, \dots, y_T\}$ one at a time, by taking $\{x_1, x_2, \dots, x_T\}$ as inputs, one at a time. Now, let's consider the problem we face in this solution for a time-series problem.

The predicted output y_t at time t of a feed-forward neural network depends only on the current input x_t . In other words, it does not have any knowledge about the inputs that led to x_t (that is, $\{x_1, x_2, \dots, x_{t-1}\}$). For this reason, a feed-forward neural network will fail at a task where the current output not only depends on the current input but also on the previous inputs. Let's understand this through an example.

Say we need to train a neural network to fill in missing words. We have the following phrase, and we would like to predict the next word:

James has a cat and it likes to drink ____.

If we are to process one word at a time and use a feed-forward neural network, we will only have the input *drink* and this is not enough at all to understand the phrase or even to understand the context (the word *drink* can appear in many different contexts). One can argue that we can achieve good results by processing the full sentence in a single go. Even though this is true, such an approach has limitations such as processing very long sentences. However, there is a new family of models known as Transformers that are processing the full sequences of data with fully-connected layers, and have been surpassing the performance of sequential models. We will have a separate chapter on these models later.

Modeling with RNNs

On the other hand, we can use an RNN to find a solution to this problem. We will start with the data we have:

$$x = \{x_1, x_2, \dots, x_T\}, y = \{y_1, y_2, \dots, y_T\}$$

Assume that we have the following relationship:

$$h_t = g_1(x_t, h_{t-1})$$

$$y_t = g_2(h_t)$$

Now, let's replace g_1 with a function approximator $f_1(x_t, h_{t-1}; \theta)$ parametrized by θ that takes the current input x_t and the previous state of the system h_{t-1} as the input and produces the current state h_t . Then, we will replace g_2 with $f_2(h_t; \varphi)$, which takes the current state of the system h_t to produce y_t . This gives us the following:

$$h_t = f_1(x_t, h_{t-1}; \theta)$$

$$y_t = f_2(h_t; \varphi)$$

We can think of $f_1 \circ f_2$ as an approximation of the true model that generates x and y . To understand this more clearly, let's now expand the equation as follows:

$$y_t = f_2(f_1(x_t, h_{t-1}; \theta); \varphi)$$

For example, we can represent y_4 as follows:

$$y_4 = f_2(f_1(x_4, h_3; \theta); \varphi)$$

Also, by expansion we get the following (omitting θ and φ for clarity):

$$y_4 = f_2(f_1(x_4, f_2(f_1(x_3, f_2(f_1(x_2, f_2(f_1(x_1, h_0))))))))$$

This can be illustrated in a graph, as shown in *Figure 6.1*:

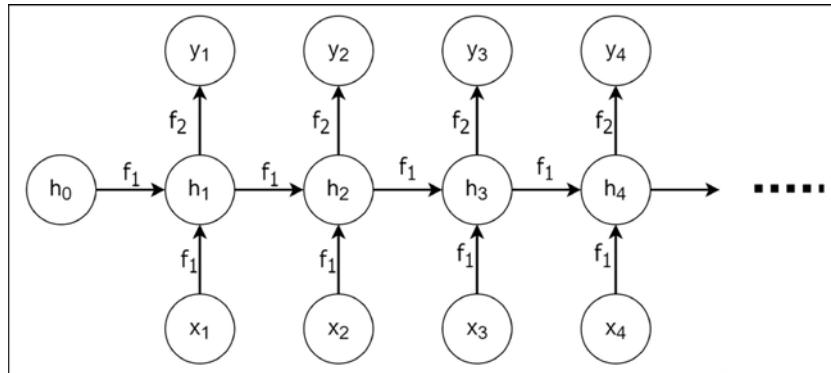


Figure 6.1: The relationship between x_t and y_t expanded

We can generally summarize the diagram, for any given time step t , as shown in *Figure 6.2*:

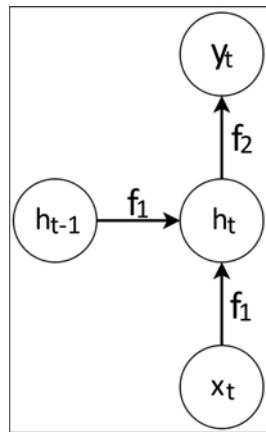


Figure 6.2: A single-step calculation of an RNN structure

However, it should be understood that h_{t-1} in fact is what h_t was before receiving x_t . In other words, h_{t-1} is h_t before one time step.

Therefore, we can represent the calculation of h_t with a recurrent connection, as shown in *Figure 6.3*:

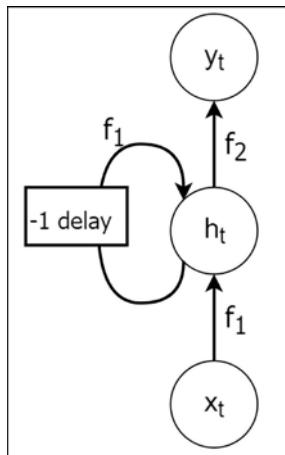


Figure 6.3: A single-step calculation of an RNN with the recurrent connection

The ability to summarize a chain of equations mapping $\{x_1, x_2, \dots, x_T\}$ to $\{y_1, y_2, \dots, y_T\}$ as in *Figure 6.3* allows us to write any y_t in terms of x_t , h_{t-1} , and h_t . This is the key idea behind an RNN.

Technical description of an RNN

Let's now have an even closer look at what makes an RNN and define the mathematical equations for the calculations taking place within an RNN. Let's start with the two functions we derived as function approximators for learning y_t from x_t :

$$h_t = f_1(x_t, h_{t-1}; \theta)$$

$$y_t = f_2(h_t; \varphi)$$

As we have seen, a neural network is composed of a set of weights and biases and some nonlinear activation function. Therefore, we can write the preceding relation as shown here:

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

Here, \tanh is the \tanh activation function, and U is a weight matrix of size $m \times d$, where m is the number of hidden units and d is the dimensionality of the input. Also, W is a weight matrix of size $m \times m$ that creates the recurrent link from h_{t-1} to h_t . The y_t relation is given by the following equation:

$$y_t = \text{softmax}(Vh_t)$$

Here, V is a weight matrix of size $c \times m$ and c is the dimensionality of the output (this can be the number of output classes). In *Figure 6.4*, we illustrate how these weights form an RNN. The arrows represent the direction that the data flows in the network:

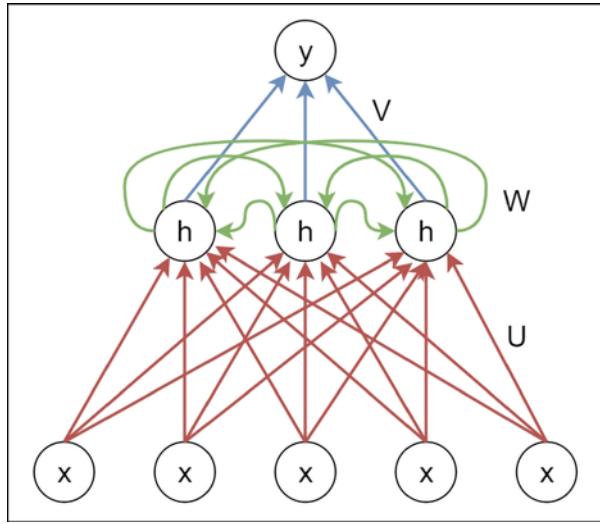


Figure 6.4: The structure of an RNN

So far, we have seen how we can represent an RNN with a graph of computational nodes, with edges denoting computations. Also, we looked at the actual mathematics behind an RNN. Let's now look at how to optimize (or train) the weights of an RNN to learn from sequential data.

Backpropagation Through Time

For training RNNs, a special form of backpropagation, known as **Backpropagation Through Time (BPTT)**, is used. To understand BPTT, however, first we need to understand how BP works. Then we will discuss why BP cannot be directly applied to RNNs, but how BP can be adapted for RNNs, resulting in BPTT. Finally, we will discuss two major problems present in BPTT.

How backpropagation works

Backpropagation is the technique that is used to train a feed-forward neural network. In backpropagation, you do the following:

- Calculate a prediction for a given input

- Calculate an error, E , of the prediction by comparing it to the actual label of the input (for example, mean squared error and cross-entropy loss)
- Update the weights of the feed-forward network to minimize the loss calculated in step 2, by taking a small step in the opposite direction of the gradient $\partial E / \partial w_{ij}$ for all w_{ij} , where w_{ij} is the j^{th} weight of the i^{th} layer

To understand the above computations more clearly, consider the feed-forward network depicted in *Figure 6.5*. This has two single weights, w_1 and w_2 , and calculates two outputs, h and y , as shown in the following figure. We assume no nonlinearities in the model for simplicity:

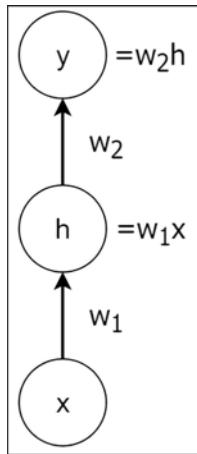


Figure 6.5: Computations of a feed-forward network

We can calculate $\frac{\partial E}{\partial w_1}$ using the chain rule as follows:

$$\frac{\partial E}{\partial w_1} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial w_1}$$

This simplifies to the following:

$$\frac{\partial E}{\partial w_1} = \frac{\partial(y - l)^2}{\partial y} \frac{\partial(w_2 h)}{\partial h} \frac{\partial(w_1 x)}{\partial w_1}$$

Here, l is the correct label for the data point x . Also, we are assuming the mean squared error as the loss function. Everything here is defined, and it is quite straightforward to calculate $\frac{\partial E}{\partial w_1}$.

Why we cannot use BP directly for RNNs

Now, let's try the same for the RNN in *Figure 6.6*. Now we have an additional recurrent weight w_3 . We have omitted the time components of inputs and outputs for the clarity of the problem we are trying to emphasize:

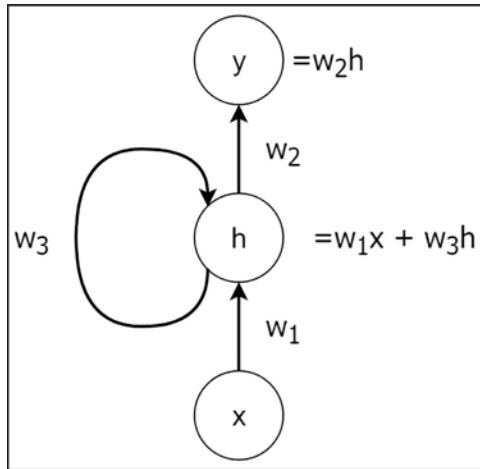


Figure 6.6: Computations of an RNN

Let's see what happens if we apply the chain rule to calculate $\frac{\partial E}{\partial w_3}$:

$$\frac{\partial E}{\partial w_3} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial w_3}$$

This becomes the following:

$$\frac{\partial E}{\partial w_3} = \frac{\partial(y - l)^2}{\partial y} \frac{\partial(w_2 h)}{\partial h} \left(\frac{\partial(w_1 x)}{\partial w_3} + \frac{\partial(w_3 h)}{\partial w_3} \right)$$

The term $\frac{\partial(w_3 h)}{\partial w_3}$ here creates problems because it is a recursive term. You end up with an infinite number of derivative terms, as h is recursive (that is, calculating h includes h itself) and h is not a constant and dependent on w_3 . This is solved by unrolling the input sequence x over time, creating a copy of the RNN for each input x_t and calculating derivatives for each copy separately, and collapsing those updates into one, by summing up the gradients, to calculate the weight update. We will discuss the details of this process next.

Backpropagation Through Time – training RNNs

The trick to calculating backpropagation for RNNs is to consider not a single input, but the full input sequence. Then, if we calculate $\frac{\partial E}{\partial w_3}$ at time step 4, we will get the following:

$$\frac{\partial E}{\partial w_3} = \sum_{j=1}^3 \frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial h_4}{\partial h_j} \frac{\partial h_j}{\partial w_3}$$

This means that we need to calculate the sum of gradients for all the time steps up to the fourth time step. In other words, we will first unroll the sequence so that we can calculate $\frac{\partial h_4}{\partial h_j}$ and $\frac{\partial h_j}{\partial w_3}$ for each time step j . This is done by creating four copies of the RNN. So, to calculate $\frac{\partial h_t}{\partial h_j}$, we need $t-j+1$ copies of the RNN. Then we will roll up the copies to a single RNN by summing up gradients with respect to all previous time steps to get the gradient, and update the RNN with the gradient $\frac{\partial E}{\partial w_3}$.

However, this becomes costly as the number of time steps increases. For more computational efficiency, we can use **Truncated Backpropagation Through Time (TBPTT)** to optimize recurrent models, which is an approximation of BPTT.

Truncated BPTT – training RNNs efficiently

In TBPTT, we only calculate the gradients for a fixed number of T time steps (in contrast to calculating it up to the very beginning of the sequence as in BPTT). More specifically, when calculating $\frac{\partial E}{\partial w_3}$, for time step t , we only calculate derivatives down to $t-T$ (that is, we do not compute derivatives up to the very beginning):

$$\frac{\partial E}{\partial w_3} = \sum_{j=t-T}^{t-1} \frac{\partial L}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_j} \frac{\partial h_j}{\partial w_3}$$

This is much more computationally efficient than standard BPTT. In standard BPTT, for each time step t , we calculate derivatives up to the very beginning of the sequence. But this gets computationally infeasible as the sequence length becomes larger and larger (for example, this could occur when processing a long text document word by word). However, in truncated BPTT, we only calculate the derivatives for a fixed number of steps backward, and as you can imagine, the computational cost does not change as the sequence becomes larger.

Limitations of BPTT – vanishing and exploding gradients

Having a way to calculate gradients for recurrent weights and having a computationally efficient approximation such as TBPTT does not enable us to train RNNs without trouble. Something else can go wrong with the calculations.

To see why, let's expand a single term in $\frac{\partial E}{\partial w_3}$, which is as follows:

$$\frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial h_4}{\partial h_1} \frac{\partial h_1}{\partial w_3} = \frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial (w_1 x + w_3 h_3)}{\partial h_1} \frac{\partial (w_1 x + w_3 h_0)}{\partial w_3}$$

Since we know that the issues of backpropagation arise from the recurrent connections, let's ignore the $w_i x$ terms and consider the following:

$$\frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial (w_3 h_3)}{\partial h_1} \frac{\partial (w_3 h_0)}{\partial w_3}$$

By simply expanding h_3 and doing simple arithmetic operations we can show this:

$$= \frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} h_0 w_3^3$$

We see that for just four time steps we have a term w_3^3 . So at the n^{th} time step, it would become w_3^{n-1} . Say we initialized w_3 to be very small (say 0.00001) at $n=100$ time step; the gradient would be infinitesimally small (of scale 10^{-500}). Also, since computers have limited precision in representing a number, this update would be ignored (that is, arithmetic underflow). This is called the **vanishing gradient**.

Solving the vanishing gradient is not very straightforward. There are no easy ways of rescaling the gradients so that they will properly propagate through time. A few techniques used in practice to solve the problem of vanishing gradients are to use careful initialization of weights (for example, the Xavier initialization), or to use momentum-based optimization methods (that is, in addition to the current gradient update, we add an additional term, which is the accumulation of all the past gradients known as the **velocity term**). However, more principled approaches to solving the vanishing gradient problem, such as different structural modifications to the standard RNN, have been introduced, as we will see in *Chapter 7, Understanding Long Short-Term Memory Networks*.

On the other hand, say that we initialized w_3 to be very large (say 1000.00). Then at the $n=100$ time step, the gradients would be massive (of scale 10^{300}).

This leads to numerical instabilities and you will get values such as `Inf` or `Nan` (that is, not a number) in Python. This is called the **exploding gradient**.

Gradient explosion can also take place due to the complexity of the loss surface of a problem. Complex nonconvex loss surfaces are very common in deep neural networks due to both the dimensionality of inputs as well as the large number of parameters (weights) present in the models.

Figure 6.7 illustrates the loss surface of an RNN and highlights the presence of walls with very high curvature. If the optimization method comes in contact with such a wall, then the gradients will explode or overshoot, as shown by the solid line in the image. This can either lead to very poor loss minimization, numerical instabilities, or both. A simple solution to avoid gradient explosion in such situations is to clip the gradients to a reasonably small value when it is larger than some threshold. The dashed line in the figure shows what happens when we clip the gradient at some small value. (Gradient clipping is covered in the paper *On the difficulty of training recurrent neural networks*, Pascanu, Mikolov, and Bengio, International Conference on Machine Learning (2013): 1310-1318.)

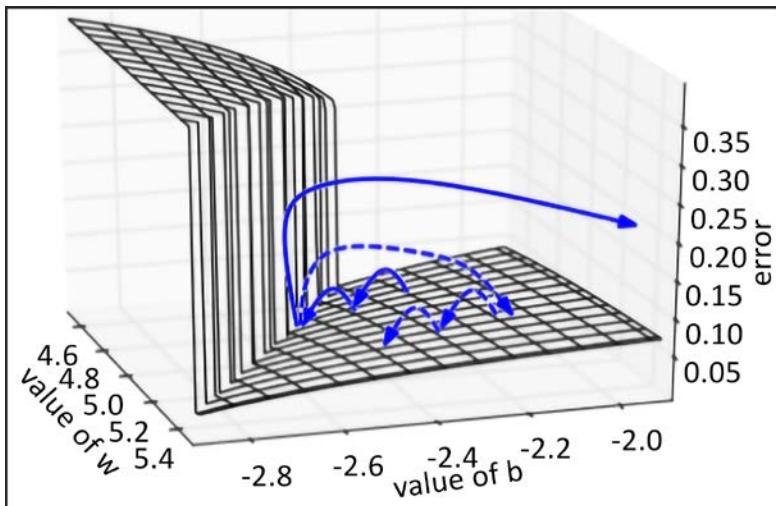


Figure 6.7: The gradient explosion phenomenon. Source: This figure is from the paper ‘On the difficulty of training recurrent neural networks’ by Pascanu, Mikolov, and Bengio

Here we conclude our discussion about BPTT, which adapts backpropagation for RNNs. Next we will discuss various ways that RNNs can be used to solve applications. These applications include sentence classification, image captioning, and machine translation. We will categorize the RNNs into several different categories such as one-to-one, one-to-many, many-to-one, and many-to-many.

Applications of RNNs

So far, we have only talked about one-to-one-mapped RNNs, where the current output depends on the current input as well as the previously observed history of inputs. This means that there exists an output for the sequence of previously observed inputs and the current input. However, in the real world, there can be situations where there is only one output for a sequence of inputs, a sequence of outputs for a single input, and a sequence of outputs for a sequence of inputs where the sequence sizes are different. In this section, we will look at several different settings of RNN models and the applications they would be used in.

One-to-one RNNs

In one-to-one RNNs, the current input depends on the previously observed inputs (see *Figure 6.8*). Such RNNs are appropriate for problems where each input has an output, but the output depends both on the current input and the history of inputs that led to the current input. An example of such a task is stock market prediction, where we output a value for the current input, and this output also depends on how the previous inputs have behaved. Another example would be scene classification, where each pixel in an image is labeled (for example, labels such as car, road, and person). Sometimes x_{t+1} can be the same as y_t for some problems. For example, in text generation problems, the previously predicted word becomes an input to predict the next word. The following figure depicts a one-to-one RNN:

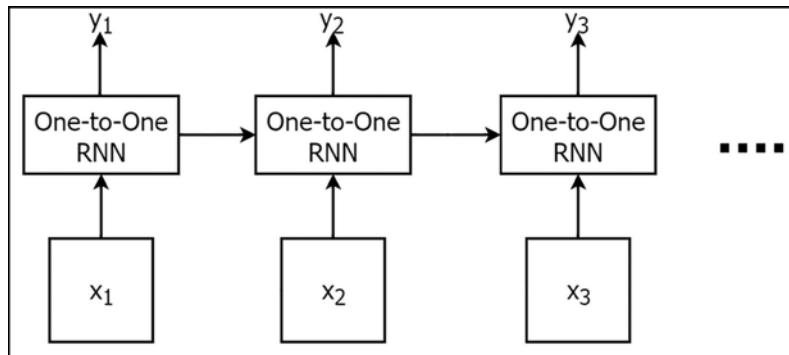


Figure 6.8: One-to-one RNNs having temporal dependencies

One-to-many RNNs

A one-to-many RNN would take a single input and output a sequence (see *Figure 6.9*). Here, we assume the inputs to be independent of each other.

That is, we do not need information about previous inputs to make a prediction about the current input. However, the recurrent connections are needed because, although we process a single input, the output is a sequence of values that depends on the previous output values. An example task where such an RNN would be used is an image captioning task. For example, for a given input image, the text caption can consist of five or ten words. In other words, the RNN will keep predicting words until it outputs a meaningful phrase describing the image. The following figure depicts a one-to-many RNN:

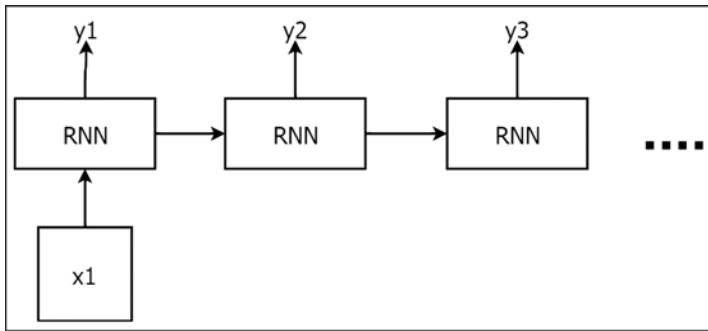


Figure 6.9: A one-to-many RNN

Many-to-one RNNs

Many-to-one RNNs take an input of arbitrary length and produce a single output for the sequence of inputs (see *Figure 6.10*). Sentence classification is one such task that can benefit from a many-to-one RNN. A sentence is represented to the model as a sequence of words of arbitrary length. The model takes it as the input and produces an output, classifying the sentence into one of a set of predefined classes. Some specific examples of sentence classification are as follows:

- Classifying movie reviews as positive or negative statements (that is, sentiment analysis)
- Classifying a sentence depending on what the sentence describes (for example, person, object, or location)

Another application of many-to-one RNNs is classifying large-scale images by processing only a patch of images at a time and moving the window over the whole image.

The following figure depicts a many-to-one RNN:

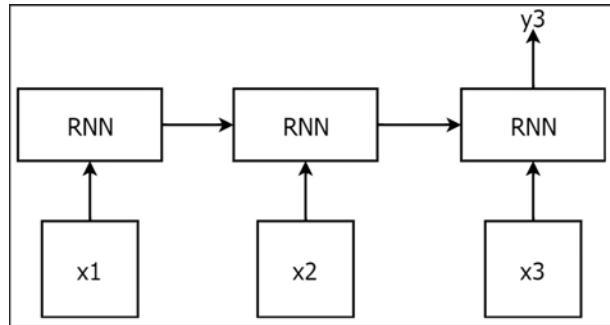


Figure 6.10: A many-to-one RNN

Many-to-many RNNs

Many-to-many RNNs (or Sequences-to-Sequence, seq2seq for short) often produce arbitrary-length outputs from arbitrary-length inputs (see *Figure 6.11*). In other words, inputs and outputs do not have to be of the same length. This is particularly useful in machine translation, where we translate a sentence from one language to another. As you can imagine, one sentence in a certain language does not always align with a sentence from another language. Another such example is chatbots, where the chatbot reads a sequence of words (that is, a user request) and outputs a sequence of words (that is, the answer). The following figure depicts a many-to-many RNN:

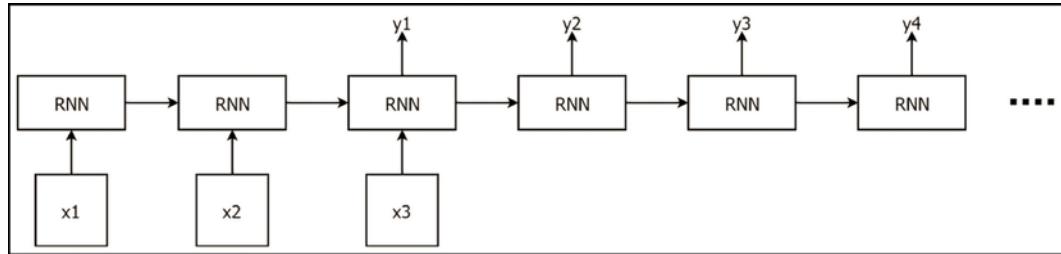


Figure 6.11: A many-to-many RNN

We can summarize the different types of applications of feed-forward networks and RNNs as follows:

Algorithm	Description	Applications
One-to-one RNNs	These take a single input and give a single output. Current input depends on the previously observed input(s).	Stock market prediction, scene classification, and text generation
One-to-many RNNs	These take a single input and give an output consisting of an arbitrary number of elements	Image captioning
Many-to-one RNNs	These take a sequence of inputs and give a single output.	Sentence classification (considering a single word as a single input)
Many-to-many RNNs	These take a sequence of arbitrary length as inputs and output a sequence of arbitrary length.	Machine translation, chatbots

Next, we will learn how to use RNNs to identify various entities mentioned in a text corpus.

Named Entity Recognition with RNNs

Now let's look at our first task: using an RNN to identify named entities in a text corpus. This task is known as **Named Entity Recognition (NER)**. We will be using a modified version of the well-known **CoNLL 2003** (which stands for **Conference on Computational Natural Language Learning - 2003**) dataset for NER.

CoNLL 2003 is available for multiple languages, and the English data was generated from a Reuters Corpus that contains news stories published between August 1996 and August 1997. The database we'll be using is found at <https://github.com/ZihanWangKi/CrossWeigh> and is called **ConLLPP**. It is a more closely curated version than the original CoNLL, which contains errors in the dataset induced by incorrectly understanding the context of a word. For example, in the phrase "*Chicago won ...*" Chicago was identified as a location, whereas it is in fact an organization. This exercise is available in `ch06_rnns_for_named_entity_recognition.ipynb` in the Ch06-Recurrent-Neural-Networks folder.

Understanding the data

We have defined a function called `download_data()`, which can be used to download the data. We will not go into the details of it as it simply downloads several files and places them in a data folder. Once the download finishes, you'll have three files:

- data\conllpp_train.txt – Training set, contains 14041 sentences
- data\conllpp_dev.txt – Validation set, contains 3250 sentences
- data\conllpp_test.txt – Test set, contains 3452 sentences

Next up, we will read the data and convert it into a specific format that suits our model. But before that, we need to see what our data looks like originally:

```
-DOCSTART- -X- -X- 0

EU NNP B-NP B-ORG
rejects VBZ B-VP 0
German JJ B-NP B-MISC
call NN I-NP 0
to TO B-VP 0
boycott VB I-VP 0
British JJ B-NP B-MISC
lamb NN I-NP 0
. . 0 0

The DT B-NP 0
European NNP I-NP B-ORG
Commission NNP I-NP I-ORG
said VBD B-VP 0
...
to TO B-PP 0
sheep NN B-NP 0
. . 0 0
```

As you can see, the document has a single word in each line along with the associated tags of that word. These tags are in the following order:

1. The Part-of-speech (POS) tag (e.g. noun - NN, verb - VB, determinant - DT, etc.)
2. Chunk tag – A chunk is a segment of text made of one or more tokens (for example, NP represents a noun phrase such as “The European Commission”)
3. Named entity tag (e.g. Location, Organization, Person, etc.)

Both chunk tags and named entity tags have a B- and I- prefix (e.g. B-ORG or I-ORG). These prefixes are there to differentiate the starting token of an entity/chunk from the continuing token of an entity/chunk.

There are also five types of entities in the dataset:

- Location-based entities (LOC)
- Person-based entities (PER)
- Organization-based entities (ORG)
- Miscellaneous entities (MISC)
- Non-entities (O)

Finally, there's an empty line between separate sentences.

Now let's look at the code that loads the data we downloaded into memory, so that we can start using it:

```
def read_data(filename):
    ...
    Read data from a file with given filename
    Returns a list of sentences (each sentence a string),
    and list of ner labels for each string
    ...

    print("Reading data ...")
    # master lists - Holds sentences (list of tokens),
    # ner_labels (for each token an NER Label)
    sentences, ner_labels = [], []

    # Open the file
    with open(filename,'r',encoding='latin-1') as f:
        # Read each line
        is_sos = True
        # We record at each line if we are seeing the beginning of a
        # sentence

        # Tokens and Labels of a single sentence, flushed when encountered
        # a new one
        sentence_tokens = []
        sentence_labels = []
        i = 0
        for row in f:
```

```
# If we are seeing an empty line or -DOCSTART- that's a new line
if len(row.strip()) == 0 or row.split(' ')[0] == '-DOCSTART-':
    is_sos = False
# Otherwise keep capturing tokens and labels
else:
    is_sos = True
    token, _, _, ner_label = row.split(' ')
    sentence_tokens.append(token)
    sentence_labels.append(ner_label.strip())

# When we reach the end / or reach the beginning of next
# add the data to the master lists, flush the temporary one
if not is_sos and len(sentence_tokens)>0:
    sentences.append(' '.join(sentence_tokens))
    ner_labels.append(sentence_labels)
    sentence_tokens, sentence_labels = [], []

print('\tDone')
return sentences, ner_labels
```

Here, we will store all the sentences (as a list of strings in `sentences`) and all the labels associated with each token in the sentences (as a list of lists in `ner_labels`). We will read the file line by line. We will maintain a Boolean called `is_sos` that indicates whether we are at the start of a sentence. We will also have two temporary lists (`sentence_tokens` and `sentence_labels`) that will accumulate the tokens and the NER labels of the current sentence. When we are at the start of a sentence, we reset these temporary lists. Otherwise, we keep writing each token and NER label we see in the file to these temporary lists. We can now run this function on the train, validation, and test corpora we have:

```
# Train data
train_sentences, train_labels = read_data(train_filepath)
# Validation data
valid_sentences, valid_labels = read_data(dev_filepath)
# Test data
test_sentences, test_labels = read_data(test_filepath)
```

We will print a few samples and see what we have with:

```
# Print some data
print('\nSample data\n')
for v_sent, v_labels in zip(valid_sentences[:5], valid_labels[:5]):
    print("Sentence: {}".format(v_sent))
    print("Labels: {}".format(v_labels))
    print('\n')
```

This produces:

```
Sentence: West Indian all-rounder Phil Simmons took four for 38 on Friday
as Leicestershire beat Somerset by an innings and 39 runs in two days to
take over at the head of the county championship .
```

```
Labels: ['B-MISC', 'I-MISC', 'O', 'B-PER', 'I-PER', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'B-ORG', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```

```
Sentence: Their stay on top , though , may be short-lived as title rivals
Essex , Derbyshire and Surrey all closed in on victory while Kent made up
for lost time in their rain-affected match against Nottinghamshire .
```

```
Labels: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'B-ORG', 'O', 'B-ORG', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O',
'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-ORG', 'O']
```

```
Sentence: After bowling Somerset out for 83 on the opening morning at
Grace Road , Leicestershire extended their first innings by 94 runs before
being bowled out for 296 with England discard Andy Caddick taking three
for 83 .
```

```
Labels: ['O', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'B-LOC', 'I-LOC', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'B-LOC', 'O', 'B-PER', 'I-PER', 'O', 'O',
'O', 'O', 'O']
```

One of the unique characteristics of NER tasks is the class imbalance. That is, not all classes will have a roughly equal number of samples. As you can probably guess, in a corpus, there are more non-named entities than named entities. This leads to a significant class imbalance among labels. Therefore, let's have a look at the distribution of samples among different classes:

```
from itertools import chain

# Print the value count for each label
print("Training data label counts")
print(pd.Series(chain(*train_labels)).value_counts())
```

To analyze the data, we will first convert the NER labels into a pandas Series object. This can be done by simply calling the `pd.Series()` construct on `train_labels`, `valid_labels`, and `test_labels`. But remember that these were lists of lists, where each inner list represents the NER tags for all the tokens in a sentence. To create a flat list, we can use the `chain()` function from the built-in Python library `itertools`. It will chain several lists together to form a single list. After that, we call the `value_counts()` function on that pandas Series. This will return a new list, where the indices are unique labels found in the original Series and the values are the counts of occurrences of each label. This gives us:

```
Training data label counts
O          169578
B-LOC      7140
B-PER      6600
B-ORG      6321
I-PER      4528
I-ORG      3704
B-MISC     3438
I-LOC      1157
I-MISC     1155
dtype: int64
```

As you can see, O labels are several magnitudes higher than the volume of other labels. We need to keep this in mind when training the model. Subsequently, we will analyze the sequence length (i.e. number of tokens) of each sentence. We need this information later to pad our sentences to a fixed length.

```
pd.Series(train_sentences).str.split().str.len().
describe(percentiles=[0.05, 0.95])
```

Here, we create a pandas Series, where each item has the length of a sentence after splitting each sentence into a list of tokens.

Then we will look at the 5% and 95% percentiles of those lengths. This produces:

```
count    14041.000000
mean      14.501887
std       11.602756
min       1.000000
5%        2.000000
50%       10.000000
95%       37.000000
max      113.000000
dtype: float64
```

We can see that 95% of our sentences have 37 tokens or less.

Processing data

Now it's time to process the data. We will keep the sentences in the same format, i.e. a list of strings where each string represents a sentence. This is because we will integrate text processing right into our model (as opposed to doing it externally). For labels, we have to do several changes. Remember labels are a list of lists, where the inner lists represent labels for all the tokens in each sentence. Specifically we will do the following:

- Convert the class labels to class IDs
- Pad the sequences of labels to a specified maximum length
- Generate a mask that indicates the padded labels, so that we can use this information to disregard the padded labels during model training

First let's write a function to get a class label to class ID mapping. This function leverages pandas' `unique()` function to get the unique labels in the training set and generate a mapping of integers to unique labels found.

```
def get_label_id_map(train_labels):
    # Get the unique list of labels
    unique_train_labels = pd.Series(chain(*train_labels)).unique()
    # Create a class Label -> class ID mapping
    labels_map = dict(
        zip(unique_train_labels,
            np.arange(unique_train_labels.shape[0])))
    print("labels_map: {}".format(labels_map))
    return labels_map
```

If you run this with:

```
labels_map = get_label_id_map(train_labels)
```

Then you will get:

```
labels_map: {'B-ORG': 0, 'O': 1, 'B-MISC': 2, 'B-PER': 3, 'I-PER': 4,
'B-LOC': 5, 'I-ORG': 6, 'I-MISC': 7, 'I-LOC': 8}
```

We write a function called `get_padded_int_labels()` that will take sequences of class labels and return sequences of padded class IDs, with the option to return a mask indicating padded labels. This function takes the following arguments:

- `labels` (`List[List[str]]`) – A list of lists of strings, where each string is a class label of the string type
- `labels_map` (`Dict[str, int]`) – A dictionary mapping a string label to a class ID of type integer
- `max_seq_length` (`int`) – A maximum length to be padded to (longer sequences will be truncated at this length)
- `return_mask` (`bool`) – Whether to return the mask showing padded labels or not

Let's now look at the code that performs the aforementioned operations:

```
def get_padded_int_labels(labels, labels_map, max_seq_length,
return_mask=True):

    # Convert string labels to integers
    int_labels = [[labels_map[x] for x in one_seq] for one_seq in
    labels]

    # Pad sequences
    if return_mask:
        # If we return mask, we first pad with a special value (-1) and
        # use that to create the mask and later replace -1 with '0'
        padded_labels = np.array(
            tf.keras.preprocessing.sequence.pad_sequences(
                int_labels, maxlen=max_seq_length, padding='post',
```

```

        truncating='post', value=-1
    )
)

# mask filter
mask_filter = (padded_labels != -1)
# replace -1 with '0' s ID
padded_labels[~mask_filter] = labels_map['0']
return padded_labels, mask_filter.astype('int')

else:
    padded_labels = np.array(ner_pad_sequence_func(int_labels,
    value=labels_map['0']))
    return padded_labels

```

You can see the first step in the function converts all the string labels in `labels` to integer labels using the `labels_map`. Next we get the padded sequences with the `tf.keras.preprocessing.sequence.pad_sequences()` function. We discussed this function in detail in the previous chapter. Essentially, it will pad (with a specified value) and truncate arbitrary-length sequences, to return fixed-length sequences. We are instructing the function to do both padding and truncating at the end of sequences, and to pad with a special value of `-1`. Then we can simply generate the mask as a boolean filter where `padded_labels` is not equal to `-1`. Thus, the positions where original labels exist will have a value of `1` and the rest will have `0`. However, we have to convert the `-1` values to a class ID found in the `labels_map`. We will give them the class ID of the label `0` (i.e. others).

From our findings in the previous chapter, we will set the maximum sequence length to `40`. Remember that the 95% percentile fell at the length of 37 words:

```
max_seq_length = 40
```

And now we will generate processed labels and masks for all of the training, validation, and testing data:

```

# Convert string labels to integers for all train/validation/test data
# Pad train/validation/test data
padded_train_labels, train_mask = get_padded_int_labels(
    train_labels, labels_map, max_seq_length, return_mask=True
)
padded_valid_labels, valid_mask = get_padded_int_labels(

```

```
    valid_labels, labels_map, max_seq_length, return_mask=True
)
padded_test_labels, test_mask = get_padded_int_labels(
    test_labels, labels_map, max_seq_length, return_mask=True
)
```

Finally, we will print the processed labels and masks of the first two sequences:

```
# Print some labels IDs  
print(padded_train_labels[:2])  
print(train_mask[:2])
```

Which returns:

You can see that the mask is indicating the true labels and padded ones clearly. Next, we will define some hyperparameters of the model.

Defining hyperparameters

Now let's define several hyperparameters needed for our RNN, as shown here:

- `max_seq_length` – Denotes the maximum length for a sequence. We infer this from our training data during data exploration. It is important to have a reasonable length for sequences, as otherwise, memory can explode, due to the unrolling of the RNN.
 - `embedding_size` – The dimensionality of token embeddings. Since we have a small corpus, a value < 100 will suffice.
 - `rnn_hidden_size` – The dimensionality of hidden layers in the RNN. Increasing dimensionality of the hidden layer usually leads to better performance. However, note that increasing the size of the hidden layer causes all three sets of internal weights (that is, U , W , and V) to increase as well, thus resulting in a high computational footprint.

- `n_classes` – Number of unique output classes present.
- `batch_size` – The batch size for training data, validation data, and test data. A higher batch size often leads to better results as we are seeing more data during each optimization step, but just like unrolling, this causes a higher memory requirement.
- `epochs` – The number of epochs to train the model for.

These are defined below:

```
# The maximum length of sequences
max_seq_length = 40

# Size of token embeddings
embedding_size = 64

# Number of hidden units in the RNN Layer
rnn_hidden_size = 64

# Number of output nodes in the last layer
n_classes = 9

# Number of samples in a batch
batch_size = 64

# Number of epochs to train
epochs = 3
```

Now we will define the model.

Defining the model

We will define the model here. Our model will have an embedding layer, followed by a simple RNN layer, and finally a dense prediction layer. One thing to note in the work we have done so far is that, unlike in previous chapters, we haven't yet defined a `Tokenizer` object. Although the `Tokenizer` has been an important part of our NLP pipeline to convert each token (or word) into an ID, there's a big downside to using an external tokenizer. After training the model, if you forget to save the tokenizer along with the model, your machine learning model becomes useless: to combat this, during inference, you would need to map each word to the exact ID it was mapped to during training.

This is a significant risk the tokenizer poses. In this chapter, we will seek an alternative, where we will integrate the tokenization mechanism right into our model, so that we don't need to worry about it later. *Figure 6.12* depicts the overall architecture of the model:

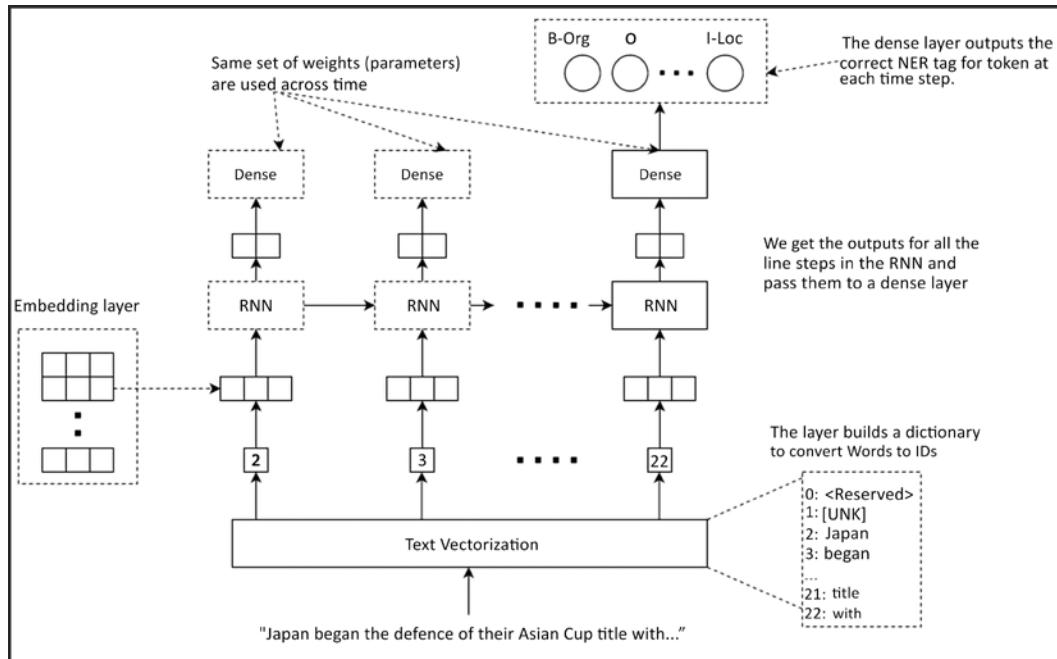


Figure 6.12: Overall architecture of the model. The text vectorization layer tokenizes the text and converts it into word IDs. Next, each token is fed as an input at each timestep of the RNN.

Finally, the RNN predicts a label for each token at every time step

Introduction to the TextVectorization layer

The `TextVectorization` layer can be thought of as a modernized tokenizer that can be plugged into the model. Here, we will play around just with the `TextVectorization` layer, without the overhead of the complexity from the rest of the model. First, we will import the `TextVectorization` layer:

```
from tensorflow.keras.layers.experimental.preprocessing import
TextVectorization
```

Now we will define a simple text corpus:

```
toy_corpus = ["I went to the market on Sunday", "The Market was empty."]
```

We can instantiate a text vectorization layer as follows:

```
toy_vectorization_layer = TextVectorization()
```

After instantiating, you need to fit this layer on some data. This way, just like the tokenizer we used previously, it can learn a word-to-numerical ID mapping. For this, we invoke the `adapt()` method of the layer, by passing the corpus of text as an input:

```
# Fit it on a corpus of data
toy_vectorization_layer.adapt(toy_corpus)
```

We can generate the tokenized output as follows:

```
toy_vectorized_output = toy_vectorization_layer(toy_corpus)
```

Which will have:

```
[[ 9  4  6  2  3  8  7]
 [ 2  3  5 10  0  0  0]]
```

We can also see the vocabulary the layer has learned:

```
Vocabulary: ['', '[UNK]', 'the', 'market', 'went', 'was', 'to', 'sunday',
'on', 'i', 'empty']
```

We can see that the layer has done some pre-processing (e.g. turned words to lowercase and removed punctuation). Next let's see how we can limit the size of the vocabulary. We can do this with the `max_tokens` argument:

```
toy_vectorization_layer = TextVectorization(max_tokens=5)
toy_vectorization_layer.adapt(toy_corpus)
toy_vectorized_output = toy_vectorization_layer(toy_corpus)
```

If you convert the `toy_corpus` to word IDs, you will see:

```
[[1 4 1 2 3 1 1]
 [2 3 1 1 0 0 0]]
```

The vocabulary will be as follows:

```
Vocabulary: ['', '[UNK]', 'the', 'market', 'went']
```

We can now see that there are only five elements in the vocabulary, just like we specified. Now if you need to skip the text pre-processing that happens within the layer, you can do so by setting the `standardize` argument to `None` in the layer:

```
toy_vectorization_layer = TextVectorization(standardize=None)
toy_vectorization_layer.adapt(toy_corpus)
toy_vectorized_output = toy_vectorization_layer(toy_corpus)
```

This will produce:

```
[[12 2 4 5 7 6 10]
 [ 9 11 3 8 0 0 0]]
```

The vocabulary will look as follows:

```
Vocabulary: ['', '[UNK]', 'went', 'was', 'to', 'the', 'on', 'market',
 'empty.', 'The', 'Sunday', 'Market', 'I']
```

Finally, we can also control the padding/truncation of sequences with the `output_sequence_length` command. For example, the following command will pad/truncate sequences at length 4:

```
toy_vectorization_layer = TextVectorization(output_sequence_length=4)
toy_vectorization_layer.adapt(toy_corpus)
toy_vectorized_output = toy_vectorization_layer(toy_corpus)
```

This will produce:

```
[[ 9 4 6 2]
 [ 2 3 5 10]]
```

Here the vocabulary is:

```
Vocabulary: ['', '[UNK]', 'the', 'market', 'went', 'was', 'to', 'sunday',
 'on', 'i', 'empty']
```

Now you have a good understanding of the arguments and what they do in the `TextVectorization` layer. Let's now discuss the model.

Defining the rest of the model

First we will import the necessary modules:

```
import tensorflow.keras.layers as layers
import tensorflow.keras.backend as K
from tensorflow.keras.layers.experimental.preprocessing import
TextVectorization
```

We will define an input layer that has a single column (i.e. each sentence represented as a single unit) and has `dtype=tf.string`:

```
# Input Layer
word_input = tf.keras.layers.Input(shape=(1,), dtype=tf.string)
```

Next, we will define a function that takes a corpus, a maximum sequence length, and a vocabulary size, and returns the trained `TextVectorization` layer and the vocabulary size:

```
def get_fitted_token_vectorization_layer(corpus, max_seq_length,
                                         vocabulary_size=None):
    """ Fit a TextVectorization layer on given data """

    # Define a text vectorization layer
    vectorization_layer = TextVectorization(
        max_tokens=vocabulary_size, standardize=None,
        output_sequence_length=max_seq_length,
    )
    # Fit it on a corpus of data
    vectorization_layer.adapt(corpus)

    # Get the vocabulary size
    n_vocab = len(vectorization_layer.get_vocabulary())

    return vectorization_layer, n_vocab
```

The function does what we have already described. However, pay attention to the various arguments we have set for the vectorization layer. We are passing the vocabulary size as `max_tokens`; we are setting the `standardize` to `None`. This is an important setting. When performing NER, keeping the case of characters is very important. Typically, an entity starts with an uppercase letter (e.g. the name of a person or organization). Therefore, we should preserve the case in the text.

Finally, we also set the `output_sequence_length` to the sequence length we found during the analysis. With that, we create the text vectorization layer as follows:

```
# Text vectorization Layer
vectorize_layer, n_vocab = get_fitted_token_vectorization_layer(train_
sentences, max_seq_length)
```

Then pass the `word_input` to the `vectorize_layer` and get the output:

```
# Vectorized output (each word mapped to an int ID)
vectorized_out = vectorize_layer(word_input)
```

The output from the `vectorize_layer` (i.e `vectorized_out`) will be sent to an embedding layer. This embedding layer is a randomly initialized embedding layer, which will have an output dimensionality of `embedding_size`:

```
# Look up embeddings for the returned IDs
embedding_layer = layers.Embedding(
    input_dim=n_vocab,
    output_dim=embedding_size,
    mask_zero=True
)(vectorized_out)
```

Until now, we dealt with feed-forward networks. Outputs of feed-forward networks did not have a time dimension. But if you look at the output from the `TextVectorization` layer, it will be a [batch size, sequence length] - sized output. When this output goes through an embedding layer, the output would be a [batch size, sequence length, embedding size]-shaped tensor. In other words, there is an additional time dimension included in the output of the embedding layer.

Another difference is the introduction of the `mask_true` argument. Masking is used to mask uninformative words added to sequences (e.g. the padding token added to make sentences a fixed length), as they do not contribute to the final outcome. Masking is a commonly used technique in sequence learning. To learn more about masking, please read the information box below.

Masking in sequence learning



Naturally, text has arbitrary lengths. For example, sentences in a corpus would have a wide variety of token lengths. But deep networks process tensors with fixed dimensions. To bring arbitrary-length sentences to constant length, we pad these sequences with some special value (e.g. 0). However, these padded values are synthetic, and only serve as a way to ensure the correct input shape. They should not contribute to the final loss or evaluation metrics. To ignore them during loss calculation and evaluation, “masking” is used. The idea is to multiply the loss resulting from padded timesteps with a zero, essentially cutting them off from the final loss.

It would be cumbersome to manually perform masking when training a model. But in TensorFlow, most layers support masking. For example, in the embedding layer, to ignore padded values (which will be zeros), all you need to do is set `mask_true=True`.

When you enable masking in a layer, it will propagate the mask to the downstream layers, flowing down until the loss computations. In other words, you only need to enable masking at the start of the model (as we have done at the embedding layer) and the rest is taken care of by TensorFlow.

Following this, we will define the core layer of our model, the RNN:

```
# Define a simple RNN Layer, it returns an output at each position
rnn_layer = layers.SimpleRNN(
    units=rnn_hidden_size, return_sequences=True
)

rnn_out = rnn_layer(embedding_layer)
```

You can implement a vanilla RNN by simply calling `tf.keras.layers.SimpleRNN`. Here we pass two important arguments. There are other useful arguments besides the two discussed here, however, they will be covered in later chapters with more complex variants of RNNs:

- `units` (int) – This defines the hidden output size of the RNN model. The larger this is, the more representational power the model will have.
- `return_sequences` (bool) – Whether to return outputs from all the timesteps, or to return only the last output. For NER tasks, we need to label every single token. Therefore we need to return outputs for all the time steps.

The `rnn_layer` takes a `[batch_size, sequence_length, embedding_size]`-sized tensor and returns a `[batch_size, sequence_length, rnn_hidden_size]`-sized tensor. Finally, the time-distributed output from the RNN will go to a Dense layer with `n_classes` output nodes and a `softmax` activation:

```
dense_layer = layers.Dense(n_classes, activation='softmax')
dense_out = dense_layer(rnn_out)
```

Finally, we can define the final model as follows. It takes a batch of string sentences as the input, and returns a batch of sequences of labels as the output:

```
model = tf.keras.Model(inputs=word_input, outputs=dense_out)
```

We have now finished building the model. Next, we will discuss the loss function and the evaluation metrics.

Evaluation metrics and the loss function

During our previous discussion, we alluded to the fact that NER tasks carry a high class imbalance. It is quite normal for text to have more non-entity-related tokens than entity-related tokens. This leads to large amounts of other (0) type labels and fewer of the remaining types. We need to take this into consideration when training the model and evaluating the model. We will address the class imbalance in two ways:

- We will create a new evaluation metric that is resilient to class imbalance
- We will use sample weights to penalize more frequent classes and boost the importance of rare classes

In this section, we will only address the former. The latter will be addressed in the next section. We will define a modified version of the accuracy. This is called a macro-averaged accuracy. In macro averaging, we compute accuracies for each class separately, and then average it. Therefore, the class imbalance is ignored when computing the accuracy. When computing standard metrics like accuracy precision or recall, there are different types of averaging available. To learn more about these, read the information box below.

Different types of metric averaging

There are different types of averaging available for metrics. You can read one such example of these averaging available in scikit-learn explained at https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html. Consider a simple binary classification example with the following confusion matrix results:



		Predicted		Support
		0	1	
True	0	35	5	40
	1	25	0	25

Figure 6.13: Example confusion matrix results

- **micro** – Computes a global metric, ignoring the differences in class distribution. e.g. $35/65 = \sim 54\%$
- **macro** – Computes the metric for each class separately and computes the mean. e.g. $(35/40 + 0/25)/2 = \sim 43.7\%$
- **weighted** – Computes the metric for each class separately and weighs it by support (i.e. number of true labels for each class). e.g. $(35/40) * 40 + (0/25) * 25 / 65 = \sim 54\%$

Here you can see the micro and weighted return the same result. This is because the denominator of the accuracy computation is the same as the support. Therefore, they cancel out in the weighted averaging. However for other metrics such as precision and recall you will get different values.

Below we define the function to compute macro accuracy using a batch of true targets (`y_true`) and predictions (`y_pred`). `y_true` will have the shape [batch_size, sequence_length] and `y_pred` will have the shape [batch_size, sequence_length, n_classes]:

```
def macro_accuracy(y_true, y_pred):

    # [batch size, time] => [batch size * time]
    y_true = tf.cast(tf.reshape(y_true, [-1]), 'int32')
    # [batch size, sequence length, n_classes] => [batch size * time]
    y_pred = tf.cast(tf.reshape(tf.argmax(y_pred, axis=-1), [-1]),
                     'int32')

    sorted_y_true = tf.sort(y_true)
    sorted_inds = tf.argsort(y_true)

    sorted_y_pred = tf.gather(y_pred, sorted_inds)

    sorted_correct = tf.cast(tf.math.equal(sorted_y_true,
                                           sorted_y_pred), 'int32')

    # We are adding one to make sure there are no division by zero
    correct_for_each_label =
        tf.cast(tf.math.segment_sum(sorted_correct, sorted_y_true),
               'float32') + 1
    all_for_each_label =
        tf.cast(tf.math.segment_sum(tf.ones_like(sorted_y_true),
                                   sorted_y_true), 'float32') + 1

    mean_accuracy =
        tf.reduce_mean(correct_for_each_label/all_for_each_label)

    return mean_accuracy
```

It is important to note that we have to write this function using TensorFlow operations, so that they are executed as a graph. Even though TensorFlow 2 has migrated toward more imperative style execution operations, there still are remnants of the declarative style introduced by TensorFlow 1.

First we flatten `y_true` so that it's a vector. Next we get the predicted label from `y_pred` using the `tf.argmax()` function and flatten the predicted labels to a vector. The two flattened structures will have the same number of elements. Then we sort `y_true`, so that same-labeled elements are close together.

We take the indices of the original data after sorting and then use the `tf.gather()` function to order `y_pred` in the same order as `y_true`. In other words, `sorted_y_true` and `sorted_y_pred` still have the same correspondence with each other. The `tf.gather()` function takes a tensor and a set of indices and orders the passed tensor in the order of the indices. For more information about `tf.gather()` refer to https://www.tensorflow.org/api_docs/python/tf/gather.

Then we compute `sorted_correct`, which is a simple indicator function that switches on if the corresponding element in `sorted_y_true` and `sorted_y_pred` are the same, and if not stays off. Then we use the `tf.math.segment_sum()` function to compute a segmented sum of correctly predicted samples. Samples belonging to each class are considered a single segment (`correct_for_each_label`). The `segment_sum()` function takes two arguments: `data` and `segment_ids`. For example, if the `data` is `[0, 1, 2, 3, 4, 5, 6, 7]` and `segment_ids` are `[0, 0, 0, 1, 1, 2, 3, 3]`, then the segment sum would be `[0+1+2, 3+4, 5, 6+7] = [3, 7, 5, 13]`.

Then we do the same for a vector of 1s. In this case, we get the number of true samples present for each class in the batch of data (`all_for_each_label`). Note that we are adding a 1 at the end. This is to avoid division by 0 in the next step. Finally, we divide `correct_for_each_label` by `all_for_each_label`, which gives us a vector containing the accuracy of each class. With that we compute the mean accuracy, which is the macro-averaged accuracy.

Finally we wrap this function in a `MeanMetricWrapper` that will produce a `tf.keras.metrics.Metric` object that we can pass to the `model.compile()` function:

```
mean_accuracy_metric = tf.keras.metrics.MeanMetricWrapper(fn=macro_accuracy, name='macro_accuracy')
```

Compile the model by calling:

```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=[mean_accuracy_metric])
```

Next, we will train the model with the data prepared.

Training and evaluating RNN on NER task

Let's train our model on the data we have prepared. But first, we need to define a function to tackle the class imbalance in our dataset. We will pass sample weights to the `model.fit()` function. To compute sample weights, we will first define a function called `get_class_weights()` that computes `class_weights` for each class. Next we will pass the class weights to another function, `get_sample_weights_from_class_weights()`, which will generate sample weights:

```
def get_class_weights(train_labels):

    label_count_ser = pd.Series(chain(*train_labels)).value_counts()
    label_count_ser = label_count_ser.min()/label_count_ser

    label_id_map = get_label_id_map(train_labels)
    label_count_ser.index = label_count_ser.index.map(label_id_map)
    return label_count_ser.to_dict()
```

The first function, `get_class_weights()`, takes a `train_labels` (a list of list of class IDs). Then we create a pandas Series object with `train_labels`. Note that we are using a function called `chain` from the built-in `itertools` library, which will flatten `train_labels` to a list of class IDs. The Series object contains frequency counts of each class label that appears in the train dataset. Next to compute weights, we divide the minimum frequency element-wise from other frequencies. In other words, if the frequency for class label c is denoted by $\text{freq}(c)$, and the total label set is denoted by C , the weight for class c is computed as:

$$w_c = \frac{\min\{\text{freq}(l) \forall l \in C\}}{\text{freq}(c)}$$

Finally, the output is converted into a dictionary that has class IDs as keys and class weights as values. Next we need to convert the `class_weights` to `sample_weights`. We simply perform a dictionary lookup element-wise on each label to generate a sample weight from `class_weights`. The `sample_weights` will be the same shape as the `train_labels` as there's one weight for each sample:

```
def get_sample_weights_from_class_weights(labels, class_weights):
    """ From the class weights generate sample weights """
    return np.vectorize(class_weights.get)(labels)
```

We can use NumPy's `np.vectorize()` function to achieve this. `np.vectorize()` takes in a function (e.g. `class_weights.get()` is the key lookup function provided by Python) and applies that on all elements, which gives us the sample weights. Call the functions we defined above to generate the actual weights:

```
train_class_weights = get_class_weights(train_labels)
print("Class weights: {}".format(train_class_weights))

# Get sample weights (we cannot use class_weight with TextVectorization
# layer)
```

```
train_sample_weights = get_sample_weights_from_class_weights(padded_train_labels, train_class_weights)
```

After we have the sample weights at our disposal, we can train our model. You can view the `class_weights` by printing them out. This will give:

```
labels_map: {
    'B-ORG': 0,
    'O': 1,
    'B-MISC': 2,
    'B-PER': 3,
    'I-PER': 4,
    'B-LOC': 5,
    'I-ORG': 6,
    'I-MISC': 7,
    'I-LOC': 8
}

Class weights: {
    1: 0.006811025015037328,
    5: 0.16176470588235295,
    3: 0.17500000000000002,
    0: 0.18272425249169436,
    4: 0.25507950530035334,
    6: 0.31182505399568033,
    2: 0.33595113438045376,
    8: 0.9982713915298186,
    7: 1.0
}
```

You can see the class `Other` has the lowest weight (because it's the most frequent), and the class `I-MISC` has the highest as it's the least frequent. Now we will train our model using the prepared data:

```
# Make train_sequences an array
train_sentences = np.array(train_sentences)

# Training the model
model.fit(
    train_sentences, padded_train_labels,
```

```
        sample_weight=train_sample_weights,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(np.array(valid_sentences),
        padded_valid_labels)
    )
```

You should get an accuracy of around 78-79% without any special performance optimization tricks. Next you can evaluate the model on test data with:

```
model.evaluate(np.array(test_sentences), padded_test_labels)
```

This will give a test accuracy of around 77%. Since the validation accuracy and test accuracy are on par, we can say that the model has generalized well. But to make sure, let's visually inspect a few samples from the test set.

Visually analyzing outputs

To analyze the output, we will use the first five sentences in the test set:

```
n_samples = 5
visual_test_sentences = test_sentences[:n_samples]
visual_test_labels = padded_test_labels[:n_samples]
```

Next predict using the model and convert those predictions to predicted class IDs:

```
visual_test_predictions = model.predict(np.array(visual_test_sentences))
visual_test_pred_labels = np.argmax(visual_test_predictions, axis=-1)
```

We will create a reversed `labels_map` that has a mapping from label ID to label string:

```
rev_labels_map = dict(zip(labels_map.values(), labels_map.keys()))
```

Finally, we will print out the results:

```
for i, (sentence, sent_labels, sent_preds) in enumerate(zip(visual_test_sentences,
    visual_test_labels, visual_test_pred_labels)):
    n_tokens = len(sentence.split())
    print("Sample:\t", "\t".join(sentence.split()))
    print("True:\t", "\t".join([rev_labels_map[i] for i in
    sent_labels[:n_tokens]]))
    print("Pred:\t", "\t".join([rev_labels_map[i] for i in
```

```

    sent_preds[:n_tokens]]))
print("\n")

```

This will print out:

Sample:	SOCCER	-	JAPAN	GET	LUCKY	WIN	,	CHINA
IN	SURPRISE	DEFEAT	.					
True:	0	0	B-LOC	0	0	0	B-LOC	0
	0							0
Pred:	0	0	B-MISC	0	0	0	B-PER	0
	0						B-LOC	
Sample:	Nadim	Ladki						
True:	B-PER	I-PER						
Pred:	B-LOC	0						
Sample:	AL-AIN	,	United	Arab	Emirates	1996-12-06		
True:	B-LOC	0	B-LOC	I-LOC	I-LOC	0		
Pred:	B-LOC	0	B-LOC	I-LOC	I-LOC	I-ORG		
Sample:	Japan	began	the	defence	of	their	Asian	
Cup	title	with	a	lucky	2-1	win	against	Syria
a	Group	C	championship	match	on	Friday	.	in
True:	B-LOC	0	0	0	0	B-MISC	I-MISC	0
0	0	0	0	B-LOC	0	0	0	0
0								
Pred:	B-LOC	I-LOC	0	0	0	B-MISC	I-MISC	
I-MISC	0	0	0	0	0	B-LOC	0	0
0	0	0	0					

It can be seen that our model is doing a decent job. It is good at identifying locations but is struggling at identifying the names of people. Here we end our discussion about the basic RNN solution that performs NER. In the next section, we will make the model more complex, giving it the ability to understand text better by providing more fine-grained details. Let's understand how we can improve our model.

NER with character and token embeddings

Nowadays, recurrent models used to solve the NER task are much more sophisticated than having just a single embedding layer and an RNN model. They involve using more advanced recurrent models like **Long Short-Term Memory (LSTM)**, **Gated Recurrent Units (GRUs)**, etc. We will set aside the discussion about these advanced models for several upcoming chapters. Here we will focus our discussion on a technique that provides the model embeddings at multiple scales, enabling it to understand language better. That is, instead of relying only on token embeddings, also use character embeddings. Then a token embedding is generated with the character embeddings by shifting a convolutional window over the characters in the token. Don't worry if you don't understand the details yet. The following sections will go into specific details of the solution. This exercise is available in `ch06_rnn_for_named_entity_recognition.ipynb` in the Ch06-Recurrent-Neural-Networks folder.

Using convolution to generate token embeddings

A combination of character embeddings and a convolutional kernel can be used to generate token embeddings (*Figure 6.14*). The method will be as follows:

- Pad each token (e.g. word) to a predefined length
- Look up the character embeddings for the characters in the token from an embedding layer
- Shift a convolutional kernel over the sequence of character embeddings to generate a token embedding

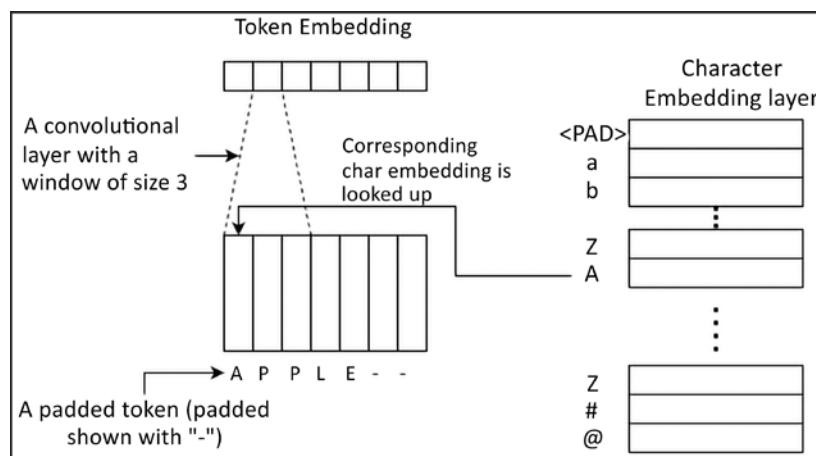


Figure 6.14: How token embeddings are generated using character embeddings and the convolution operation

The very first thing we need to do is analyze the statistics around how many characters there are for a token in our corpus. Similar to how we did it previously, we can do this with pandas:

```
vocab_ser = pd.Series(
    pd.Series(train_sentences).str.split().explode().unique()
)
vocab_ser.str.len().describe(percentiles=[0.05, 0.95])
```

In computing `vocab_ser`, the first part (i.e. `pd.Series(train_sentences).str.split()`) will result in a pandas Series, whose elements are a list of tokens (each token in the sentence is an item of that list). Next, `explode()` will convert the Series of a list of tokens into a Series of tokens, by converting each token into a separate item in the Series. Finally we take only the unique tokens in that Series. Here we end up with a pandas Series where each item is a unique token.

We will now use the `str.len()` function to get the length of each token (i.e. the number of characters) and look at the 95% percentile in that. We will get the following:

count	23623.000000
mean	6.832705
std	2.749288
min	1.000000
5%	3.000000
50%	7.000000
95%	12.000000
max	61.000000
dtype:	float64

We can see around 95% of our words have less than or equal to 12 characters. Next, we will write a function to pad shorter tokens:

```
def prepare_corpus_for_char_embeddings(tokenized_sentences, max_seq_length):
    """ Pads each sequence to a maximum length """
    proc_sentences = []
    for tokens in tokenized_sentences:
        if len(tokens) >= max_seq_length:
            proc_sentences.append([[t] for t in
                                  tokens[:max_seq_length]])
        else:
            proc_sentences.append([[t] for t in
```

```

    tokens+[ ' ']*(max_seq_length-len(tokens))]

return proc_sentences

```

The function takes a set of tokenized sentences (i.e. each sentence as a list of tokens, not a string) and a maximum sequence length. Note that this is the maximum sequence length we used previously, not the new token length we discussed. This function would then do the following:

- For longer sentences, only return the `max_seq_length` tokens
- For shorter sentences, append “as a token until `max_seq_length` is reached

Let's run this function on a small toy dataset:

```

# Define sample data
data = ['aaaa bb c', 'd eee']

# Pad sequences
tokenized_sentences = prepare_corpus_for_char_embeddings([d.split() for d
in data], 3)

```

This will return:

```
Padded sequence: [[[ 'aaaa'], ['bb'], ['c']], [[ 'd'], ['eee'], ['']]]
```

We will now define a new `TextVectorization` layer that can cope with the changes we introduced to the data. Instead of tokenizing on the token level, the new `TextVectorization` layer must tokenize on the character level. For this we need to make a few changes. We will again write a function to contain this vectorization layer:

```

def get_fitted_char_vectorization_layer(corpus, max_seq_length, max_token_
length, vocabulary_size=None):
    """ Fit a TextVectorization layer on given data """

    def _split_char(token):
        return tf.strings.bytes_split(token)

    # Define a text vectorization layer
    vectorization_layer = TextVectorization(
        standardize=None,
        split=_split_char,
        output_sequence_length=max_token_length,
    )

```

```

tokenized_sentences = [sent.split() for sent in corpus]
padded_tokenized_sentences =
prepare_corpus_for_char_embeddings(tokenized_sentences,
max_seq_length)

# Fit it on a corpus of data
vectorization_layer.adapt(padded_tokenized_sentences)

# Get the vocabulary size
n_vocab = len(vectorization_layer.get_vocabulary())

return vectorization_layer, n_vocab

```

We first define a function called `_split_char()` that takes a token (as a `tf.Tensor`) and returns a char-tokenized tensor. For example, `_split_char(tf.constant(['abcd']))` would return `<tf.RaggedTensor [[b'a', b'b', b'c', b'd']]>`. Then we define a `TextVectorization` layer that will use this newly defined function as the way to split the data it gets. We will also define `output_sequence_length` as `max_token_length`. Then we create `tokenized_sentences`, a list of list of strings, and pad it using the `prepare_corpus_for_char_embeddings()` function we defined earlier. Finally we use the `TextVectorization` layer's `adapt()` function to fit it with the data we prepared. Two key differences between the previous token-based text vectorizer and this char-based text vectorizer are in the input dimensions and the final output dimensions:

- Token-based vectorizer – Takes in a `[batch_size, 1]`-sized input and produces a `[batch_size, sequence_length]`-sized output
- Char-based vectorizer – Takes in a `[batch_size, sequence_length, 1]`-sized input and produces a `[batch_size, sequence_length, token_length]`-sized output

Now we are equipped with the ingredients to implement our new and improved NER classifier.

Implementing the new NER model

With a good conceptual understanding of the model, let's implement the new NER model. We will first define some hyperparameters, followed by defining a text vectorizer as before. However, our `TextVectorization` will be more complex in this section, as we have several different levels of tokenization taking place (e.g. char-level and token-level). Finally we define the RNN-based model that produces the output.

Defining hyperparameters

First, we will define the two hyperparameters as follows:

```
max_seq_length = 40  
max_token_length = 12
```

Defining the input layer

We then define an input layer with the data type `tf.strings` as before:

```
# Input Layer (tokens)  
word_input = tf.keras.layers.Input(shape=(1,), dtype=tf.string)
```

The inputs to this layer would be a batch of sentences, where each sentence is a string.

Defining the token-based TextVectorization layer

Then we define the token-level TextVectorization layer just like we did above:

```
# Text vectorize layer (token)  
token_vectorize_layer, n_token_vocab = get_fitted_token_vectorization_  
layer(train_sentences, max_seq_length)  
  
# Vectorized output (each word mapped to an int ID)  
token_vectorized_out = token_vectorize_layer(word_input)
```

Defining the character-based TextVectorization layer

For the character-level vectorization layer we will employ the `get_fitted_char_vectorization_layer()` function we defined above:

```
# Text vectorize layer (char)  
char_vectorize_layer, n_char_vocab = get_fitted_char_vectorization_  
layer(train_sentences, max_seq_length, max_token_length)
```

Next, we will discuss the inputs for this layer.

Processing the inputs for the char_vectorize_layer

We will use the same `word_input` for this new vectorization layer as well. However, using the same input means we need to introduce some interim pre-processing to get the input to the correct format intended for this layer. Remember that the input to this layer needs to be a [batch size, sequence length, 1]-sized tensor.

This means the sentences need to be tokenized to a list of tokens. For that we will use the `tf.keras.layers.Lambda()` layer and the `tf.strings.split()` function:

```
tokenized_word_input = layers.Lambda(
    lambda x: tf.strings.split(x).to_tensor(default_value='',
    shape=[None, max_seq_length, 1])
)(word_input)
char_vectorized_out = char_vectorize_layer(tokenized_word_input)
```

The `Lambda` layer is used as a way to create a layer from a custom TensorFlow/Keras function, which may not be available as a standard layer in Keras. Here we are using a `Lambda` layer to define a layer that will tokenize a passed input to a list of tokens. Furthermore, the `tf.strings.split()` function returns a ragged tensor. In a typical tensor, all the dimensions need to have a constant size. A ragged tensor is a special tensor whose dimensions are not fixed. For example, since a list of sentences is highly unlikely to have the same number of tokens, this results in a ragged tensor. But TensorFlow will complain if you try to go forward with a `tf.RaggedTensor` as most layers do not support these tensors. Therefore, we need to convert this to a standard tensor using the `to_tensor()` function. We can pass a shape to this function and it will make sure the shape of the resulting tensor will be the defined shape (by means of padding and truncations).

A key thing to pay attention to is how the shapes of the input-output tensors are transformed at each layer. For example, we started off with a `[batch_size, 1]`-sized tensor that went into the `Lambda` layer to be transformed to a `[batch_size, sequence_length, 1]`-sized layer. Finally, the `char_vectorize_layer` transforms this into a `[batch_size, sequence_length, token_length]`-sized tensor.

We will then define an embedding layer, with which we will look up embeddings for the resulting char IDs coming from the `char_vectorize_layer`:

```
# Produces a [batch_size, seq_length, token_length, emb_size]
char_embedding_layer = layers.Embedding(input_dim=n_char_vocab, output_
dim=32, mask_zero=True)(char_vectorized_out)
```

This layer produces a `[batch_size, sequence_length, token_length, 32]`-sized tensor, with a char embedding vector for each character in the tensor. Now it's time to perform convolution on top of this output.

Performing convolution on the character embeddings

We will define a 1D convolution layer with a kernel size of 5 (i.e. convolutional window size), a stride of 1, 'same' padding, and a ReLU activation. We then feed the output from the previous section to this:

```
# A 1D convolutional Layer that will generate token embeddings by shifting
# a convolutional kernel over the sequence of chars in each token (padded)
char_token_output = layers.Conv1D(filters=1, kernel_size=5, strides=1,
padding='same', activation='relu')(char_embedding_layer)
```

This layer typically takes a [batch size, width, in channels]-sized tensor. However, in our case, we have a four-dimensional input. This means, our Conv1D layer is going to behave in a time-distributed fashion. Put in another way, it will take an input with a temporal dimension (i.e. sequence length dimension) and produce an output with that dimension intact. In other words, it takes our input of shape [batch size, sequence length, token length, 32 (in channels)] and produces a [batch size, sequence length, token length, 1 (out channels)]-sized output. You can see that the convolution only operates on the last two dimensions, while keeping the first two as they are.

Another way to think about this is, ignore the batch and sequence dimensions and visualize how convolution would work on the width and in channel dimensions. Then apply the same operation element-wise to other dimensions, while considering the operation on 2D [width, in channel] tensors as a single unit of computation.

Remember that we have a [batch size, sequence length, token length, 1]-sized output. This has an extra dimension of 1 at the end. We will write a simple Lambda layer to get rid of this dimension:

```
# There is an additional dimension of size 1 (out channel dimension) that
# we need to remove
char_token_output = layers.Lambda(lambda x: x[:, :, :, 0])(char_token_
output)
```

To get the final output embedding (i.e. a combination of token- and character-based embeddings), we concatenate the two embeddings on the last axis. This would result in a 48 element-long vector (i.e. 32 element-long token embedding + 12 element-long char-based token embedding):

```
# Concatenate the token and char embeddings
concat_embedding_out = layers.concatenate([token_embedding_out, char_
token_output])
```

The rest of the model, we will keep it the same. First define an RNN layer and pass the `concat_embedding_out` as an input:

```
# Define a simple bidirectional RNN Layer, it returns an output at each
# position
rnn_layer_1 = layers.SimpleRNN(
    units=64, activation='tanh', use_bias=True, return_sequences=True
)

rnn_out_1 = rnn_layer_1(concat_embedding_out)
```

Remember that we have set `return_sequences=True`, which means it will produce an output at each time step, as opposed to only at the last time step. Next, we define the final Dense layer, which has `n_classes` output nodes (i.e. 9) and a `softmax` activation:

```
# Defines the final prediction layer
dense_layer = layers.Dense(n_classes, activation='softmax')
dense_out = dense_layer(rnn_out_1)
```

We define the model and compile it like before:

```
# Defines the model
char_token_embedding_rnn = tf.keras.Model(inputs=word_input,
outputs=dense_out)

# Define a macro accuracy measure
mean_accuracy_metric = tf.keras.metrics.MeanMetricWrapper(fn=macro_
accuracy, name='macro_accuracy')

# Compile the model with a Loss optimizer and metrics
char_token_embedding_rnn.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics=[mean_accuracy_metric])
```

This is our final model. The key difference in this model compared to the previous solution is that it used two different embedding types. A standard token-based embedding layer and a complex, char-based embedding that was leveraged to generate token embeddings using the convolution operation. Now let's train the model.

Model training and evaluation

Model training is identical to the training we did for the standard RNN model, so we will not discuss it further:

```
# Make train_sequences an array
train_sentences = np.array(train_sentences)
# Get sample weights (we cannot use class_weight with TextVectorization
# layer)
train_sample_weights = get_sample_weights_from_class_weights(padded_train_
labels, train_class_weights)

# Training the model
char_token_embedding_rnn.fit(
    train_sentences, padded_train_labels,
    sample_weight=train_sample_weights,
    batch_size=64,
    epochs=3,
    validation_data=(np.array(valid_sentences), padded_valid_labels)
)
```

You should get around a ~2% validation accuracy and a ~1% test accuracy boost after these modifications.

Other improvements you can make

Here we will discuss several improvements you can make to uplift the model performance even further.

- **More RNN layers** – Adding more stacked RNN layers. By adding more hidden RNN layers, we can allow the model to learn more refined latent representations, leading to better performance. An example usage is shown below:

```
rnn_layer_1 = layers.SimpleRNN(
    units=64, activation='tanh', use_bias=True, return_
    sequences=True
)
rnn_out_1 = rnn_layer_1(concat_embedding_out)
rnn_layer_2 = layers.SimpleRNN(
```

```

        units=32, activation='tanh', use_bias=True, return_
sequences=True
)
rnn_out_1 = rnn_layer_1(rnn_out_1)

```

- **Make the RNN layer bidirectional** – The RNN models we discussed so far are uni-directional, i.e. looks at the sequence of text from forward to backward. However a different variant known as bi-directional RNNs looks at the sequence in both directions, i.e. forward to backward and backward to forward. This leads to better language understanding in models and inevitably better performance. We will discuss this variant in more detail in the upcoming chapters. An example usage is shown below:

```

rnn_layer_1 = layers.Bidirectional(layers.SimpleRNN(
    units=64, activation='tanh', use_bias=True, return_
sequences=True
))

```

- **Incorporate regularization techniques** – You can leverage L2 regularization and dropout techniques to avoid overfitting and improve generalization of the model.
- **Use early stopping and learning rate reduction to reduce overfitting** – During model training, use early stopping (i.e. training the model only until the validation accuracy is improving) and learning rate reduction (i.e. gradually reducing the learning rate over the epochs).

We recommend experimenting with some of these techniques yourself to see how they can maximize the performance of your RNNs.

Summary

In this chapter, we looked at RNNs, which are different from conventional feed-forward neural networks and more powerful in terms of solving temporal tasks.

Specifically, we discussed how to arrive at an RNN from a feed-forward neural network type structure.

We assumed a sequence of inputs and outputs, and designed a computational graph that can represent the sequence of inputs and outputs.

This computational graph resulted in a series of copies of functions that we applied to each individual input-output tuple in the sequence. Then, by generalizing this model to any given single time step t in the sequence, we were able to arrive at the basic computational graph of an RNN. We discussed the exact equations and update rules used to calculate the hidden state and the output.

Next we discussed how RNNs are trained with data using BPTT. We examined how we can arrive at BPTT with standard backpropagation as well as why we can't use standard backpropagation for RNNs. We also discussed two important practical issues that arise with BPTT—vanishing gradient and exploding gradient—and how these can be solved on the surface level.

Then we moved on to the practical applications of RNNs. We discussed four main categories of RNNs. One-to-one architectures are used for tasks such as text generation, scene classification, and video frame labeling. Many-to-one architectures are used for sentiment analysis, where we process the sentences/phrases word by word (compared to processing a full sentence in a single go, as we saw in the previous chapter). One-to-many architectures are common in image captioning tasks, where we map a single image to an arbitrarily long sentence phrase describing the image. Many-to-many architectures are leveraged for machine translation tasks.

We solved the task of NER with RNNs. In NER, the problem is to, given a sequence of tokens, predict a label for each token. The label represents an entity (e.g. organization, location, person, etc.). For this we used embeddings as well as an RNN to process each token while considering the sequence of tokens as a time-series input. We also used a text vectorization layer to convert tokens into word IDs. A key benefit of the text vectorization layer is that it is built as a part of the model, unlike the tokenizer we used before.

Finally, we looked at how we can adopt character embeddings and the convolution operation to generate token embeddings. We used these new token embeddings along with standard word embeddings to improve model accuracy.

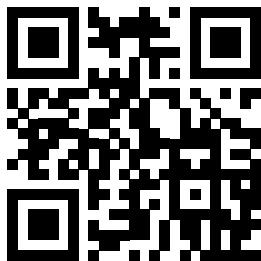
In the next chapter, we will discuss a more powerful RNN model known as **Long Short-Term Memory (LSTM)** networks that further reduces the adverse effect of the vanishing gradient, and thus produces much better results.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside

more than 1000 members at: <https://packt.link/nlp>



7

Understanding Long Short-Term Memory Networks

In this chapter, we will discuss the fundamentals behind a more advanced RNN variant known as **Long Short-Term Memory Networks (LSTMs)**. Here, we will focus on understanding the theory behind LSTMs, so we can discuss their implementation in the next chapter. LSTMs are widely used in many sequential tasks (including stock market prediction, language modeling, and machine translation) and have proven to perform better than older sequential models (for example, standard RNNs), especially given the availability of large amounts of data. LSTMs are designed to avoid the problem of the vanishing gradient that we discussed in the previous chapter.

The main practical limitation posed by the vanishing gradient is that it prevents the model from learning long-term dependencies. However, by avoiding the vanishing gradient problem, LSTMs have the ability to store memory for longer than ordinary RNNs (for hundreds of time steps). In contrast to RNNs, which only maintain a single hidden state, LSTMs have many more parameters as well as better control over what memory to store and what to discard at a given training step. For example, RNNs are not able to decide which memory to store and which to discard, as the hidden state is forced to be updated at every training step.

Specifically, we will discuss what an LSTM is at a very high level and how the functionality of LSTMs allows them to store long-term dependencies. Then we will go into the actual underlying mathematical framework governing LSTMs and discuss an example to highlight why each computation matters. We will also compare LSTMs to vanilla RNNs and see that LSTMs have a much more sophisticated architecture that allows them to surpass vanilla RNNs in sequential tasks.

Revisiting the problem of the vanishing gradient and illustrating it through an example will lead us to understand how LSTMs solve the problem.

Thereafter, we will discuss several techniques that have been introduced to improve the predictions produced by a standard LSTM (for example, improving the quality/variety of generated text in a text generation task). For example, generating several predictions at once instead of predicting them one by one can help to improve the quality of generated predictions. We will also look at **bidirectional LSTMs (BiLSTMs)**, which are an extension to the standard LSTM, that have greater capabilities for capturing the patterns present in a sequence than a standard LSTM.

Finally, we will discuss two recent LSTM variants. First, we will look at **peephole connections**, which introduce more parameters and information to the LSTM gates, allowing LSTMs to perform better. Next, we will discuss **Gated Recurrent Units (GRUs)**, which are gaining increasing popularity as they have a much simpler structure compared to standard LSTMs and also do not degrade performance.

Specifically, this chapter will cover the following main topics:

- Understanding Long Short-Term Memory Networks
- How LSTMs solve the vanishing gradient problem
- Improving LSTMs
- Other variants of LSTMs



Transformer models have emerged as a more powerful alternative for sequence learning. Transformer models deliver better performance as these models have access to the full history of the sequence at a given step, whereas LSTM models can only see the previous output at a given step. We will discuss Transformer models in detail in *Chapter 10, Transformers* and *Chapter 11, Image Captioning with Transformers*. However, it's still worth learning about LSTMs as they have laid the foundation for next-generation models like Transformers. Additionally, LSTMs are still used to some extent, especially when working on time-series problems in memory-constrained environments.

Understanding Long Short-Term Memory Networks

In this section, we will first explain how an LSTM cell operates. We will see that in addition to the hidden states, a gating mechanism is in place to control information flow inside the cell.

Then we will work through a detailed example and see how gates and states help at various stages of the example to achieve desired behaviors, finally leading to the desired output. Finally, we will compare an LSTM against a standard RNN to learn how an LSTM differs from a standard RNN.

What is an LSTM?

LSTMs can be seen as a more complex and capable family of RNNs. Though LSTMs are a complicated beast, the underlying principles of LSTMs are as same as of RNNs; they process a sequence of items by working on one input at a time in a sequential order. An LSTM is mainly composed of five different components:

- **Cell state:** This is the internal cell state (that is, memory) of an LSTM cell
- **Hidden state:** This is the external hidden state exposed to other layers and used to calculate predictions
- **Input gate:** This determines how much of the current input is read into the cell state
- **Forget gate:** This determines how much of the previous cell state is sent into the current cell state
- **Output gate:** This determines how much of the cell state is output into the hidden state

We can wrap the RNN to a cell architecture as follows: the cell will output some state (with a nonlinear activation function) that is dependent on the previous cell state and the current input. However, in RNNs, the cell state is continuously updated with every incoming input. This behavior is quite undesirable for storing long-term dependencies.

LSTMs can decide when to add, update, or forget information stored in each neuron in the cell state. In other words, LSTMs are equipped with a mechanism to keep the cell state unchanged (if warranted for better performance), giving them the ability to store long-term dependencies.

This is achieved by introducing a gating mechanism. LSTMs possess gates for each operation the cell needs to perform. The gates are continuous (often sigmoid functions) between 0 and 1, where 0 means no information flows through the gate and 1 means all the information flows through the gate. An LSTM uses one such gate for each neuron in the cell. As explained in the introduction, these gates control the following:

- How much of the current input is written to the cell state (input gate)
- How much information is forgotten from the previous cell state (forget gate)
- How much information is output into the final hidden state from the cell state (output gate)

Figure 7.1 illustrates this functionality for a hypothetical scenario. Each gate decides how much of various data (for example, the current input, the previous hidden state, or the previous cell state) flows into the states (that is, the final hidden state or the cell state). The thickness of each line represents how much information is flowing from/to that gate (in some hypothetical scenarios). For example, in this figure, you can see that the input gate is allowing more from the current input than from the previous final hidden state, where the forget gate allows more from the previous final hidden state than from the current input:

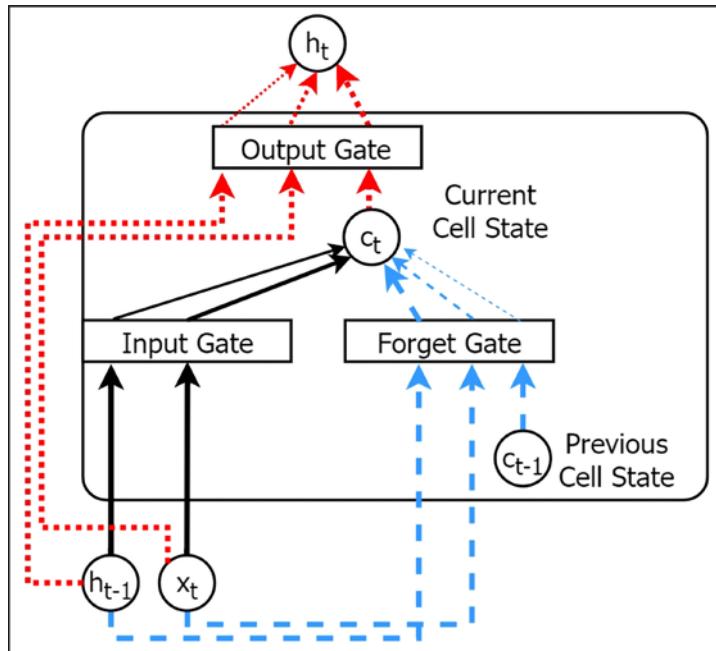


Figure 7.1: An abstract view of the data flow in an LSTM

LSTMs in more detail

Here we will walk through the actual mechanism of LSTMs. We will first briefly discuss the overall view of an LSTM cell and then start discussing each of the computations crunched within an LSTM cell, along with an example of text generation.

As we discussed earlier, LSTMs have a gating mechanism composed of the following three gates:

- **Input gate:** A gate that outputs values between 0 (the current input is not written to the cell state) and 1 (the current input is fully written to the cell state). Sigmoid activation is used to squash the output to between 0 and 1.

- **Forget gate:** A sigmoidal gate that outputs values between 0 (the previous cell state is fully forgotten for calculating the current cell state) and 1 (the previous cell state is fully read in when calculating the current cell state).
- **Output gate:** A sigmoidal gate that outputs values between 0 (the current cell state is fully discarded for calculating the final state) and 1 (the current cell state is fully used when calculating the final hidden state).

This can be shown as in *Figure 7.2*. This is a very high-level diagram, and some details have been omitted in order to avoid clutter. We present LSTMs, both with loops and without loops, to improve understanding. The figure on the right-hand side depicts an LSTM with loops, and the one on the left-hand side shows the same LSTM with the loops unfolded so that no loops are present in the model:

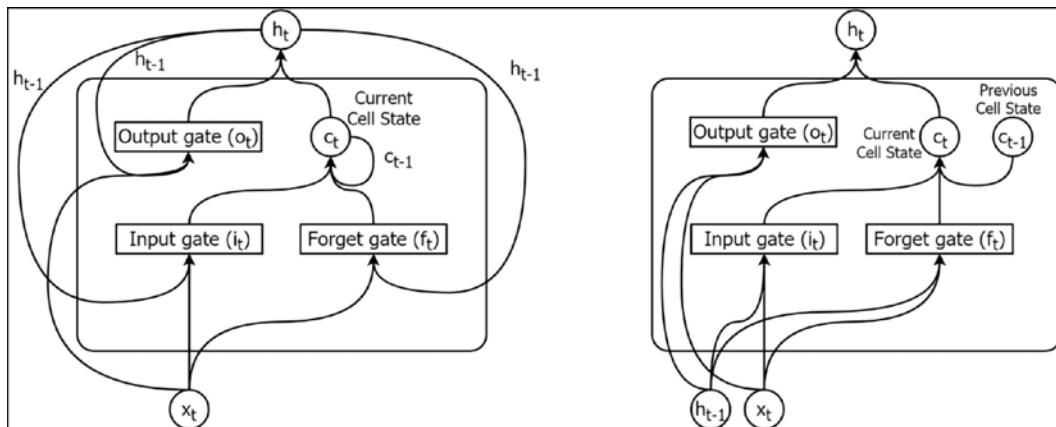


Figure 7.2: An LSTM with recurrent links (that is, loops) (right) and an LSTM with recurrent links unfolded (left)

Now, to get a better understanding of LSTMs, let's consider a language modeling example. We will discuss the actual update rules and equations side by side with the example to ground our understanding of LSTMs better.

Let's consider an example of generating text starting from the following sentence:

John gave Mary a puppy.

The story that we output should be about *John*, *Mary*, and the *puppy*. Let's assume our LSTM outputs two sentences following the given sentence:

John gave Mary a puppy. _____.

The following is the output given by our LSTM:

John gave Mary a puppy. It barks very loudly. They named it Luna.

We are still far from outputting realistic phrases such as these. However, LSTMs can learn relationships such as between nouns and pronouns. For example, *it* is related to the *puppy*, and *they* to *John* and *Mary*. Then, it should learn the relationship between the noun/pronoun and the verb. For example, for *it*, the verb should have an s at the end. We illustrate these relationships/dependencies in *Figure 7.3*. As we can see, both long-term (for example, *Luna* --> *puppy*) and short-term (for example, *It* -->*barks*) dependencies are present in this phrase. The solid arrows depict links between nouns and pronouns and dashed arrows show links between nouns/pronouns and verbs:

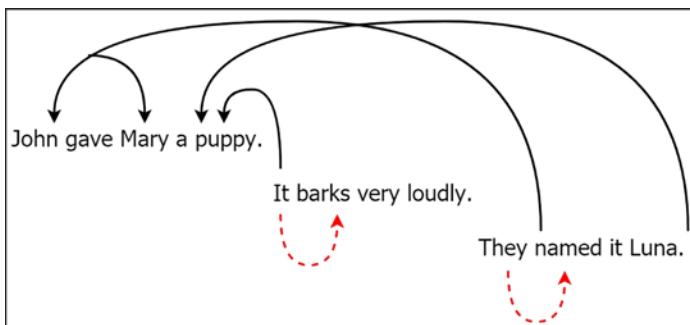


Figure 7.3: Sentences given and predicted by the LSTM with various relationships between words highlighted

Now let's consider how LSTMs, using their various operations, can model such relationships and dependencies to output sensible text, given a starting sentence.

The input gate (i_t) takes the current input (x_t) and the previous final hidden state (h_{t-1}) as the input and calculates i_t , as follows:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

The input gate i_t can be understood as the calculation performed at the hidden layer of a single-hidden-layer standard RNN with the sigmoidal activation. Remember that we calculated the hidden state of a standard RNN as follows:

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

Therefore, the calculation of i_t of the LSTM looks quite analogous to the calculation of h_t of a standard RNN, except for the change in the activation function and the addition of bias.

After the calculation, a value of 0 for i_t will mean that no information from the current input will flow to the cell state, where a value of 1 means that all the information from the current input will flow to the cell state.

Next, another value (which is called **candidate value**) is calculated as follows, which is fed in to calculate the current cell state later. This value will be treated as a potential candidate for the final cell state of this time step:

$$\tilde{c}_t = \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

We can visualize these calculations in *Figure 7.4*:

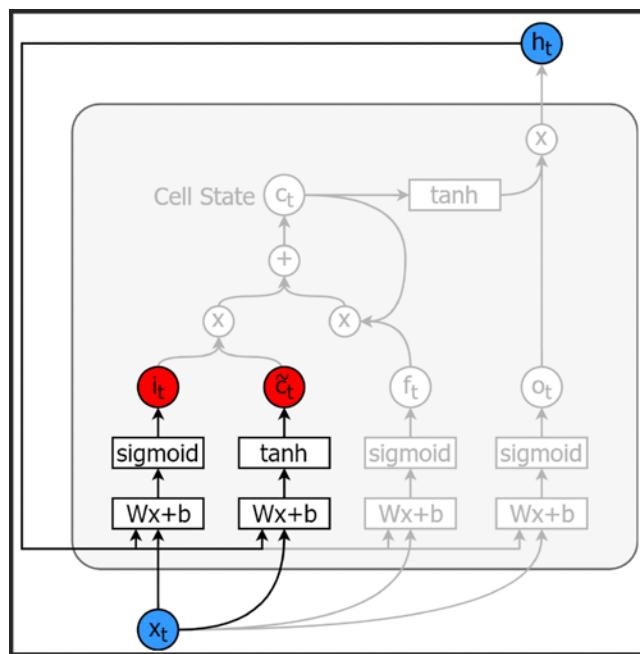


Figure 7.4: Calculation of i_t and \tilde{c}_t (in bold) in the context of all the calculations (grayed out) that take place in an LSTM

In our example, at the very beginning of the learning, the input gate needs to be highly activated, as the model has no prior knowledge of the task. The first word that the LSTM outputs is *it*. Also, in order to do so, the LSTM must learn that *puppy* is also referred to as *it*. Let's assume our LSTM has five neurons to store the state. We would like the LSTM to store the information that *it* refers to *puppy*. Another piece of information we would like the LSTM to learn (in a different neuron) is that the present tense verb should have an *s* at the end of the verb when the pronoun *it* is used.

One more thing the LSTM needs to know is that the *puppy barks loud*. Figure 7.5 illustrates how this knowledge might be encoded in the cell state of the LSTM. Each circle represents a single neuron (that is, a hidden unit) of the cell state:

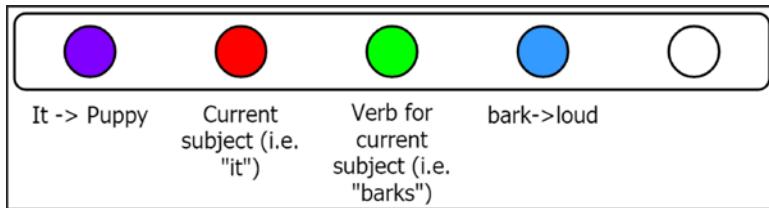


Figure 7.5: The knowledge that should be encoded in the cell state to output the first sentence

With this information, we can output the first new sentence:

John gave Mary a puppy. It barks very loudly.

Next, the forget gate is calculated as follows:

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

The forget gate does the following. A value of 0 for the forget gate means that no information from c_{t-1} will be passed to calculate c_t , and a value of 1 means that all the information of c_{t-1} will propagate into the calculation of c_t . It may sound counter-intuitive, as switching on the forget gate causes the model to remember from the previous step and vice versa. But to respect the original naming conventions and design, we'll continue to use them as they are.

Now we will see how the forget gate helps in predicting the next sentence:

They named it Luna.

Now, as you can see, the new relationship we are looking at is between *John* and *Mary* and *they*. Therefore, we no longer need information about *it* and how the verb *bark* behaves, as the subjects are *John* and *Mary*. We can use the forget gate in combination with the current subject *they* and the corresponding verb *named* to replace the information stored in the **Current subject** and **Verb for current subject** neurons (see Figure 7.6):

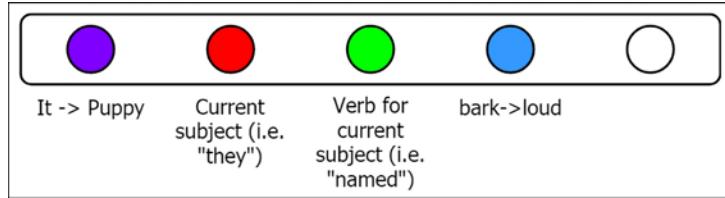


Figure 7.6: The knowledge in the third neuron from the left (*it* -> *barks*) is replaced with new information (*they* -> *named*)

In terms of the values of weights, we illustrate this transformation in Figure 7.7. We do not change the state of the neuron maintaining the *it* -> *puppy* relationship, because *puppy* appears as an object in the last sentence. This is done by setting weights connecting *it* -> *puppy* from c_{t-1} to c_t to 1. Then we will replace the neurons maintaining the current subject and verb information with a new subject and verb. This is achieved by setting the forget weights of f_t , for that neuron, to 0. Then we will set the weights of i_t , connecting the current subject and verb to the corresponding state neurons, to 1. We can think of \tilde{c}_t (the candidate value) as a potential candidate for the cell's memory, as it contains information from the current input x_t :

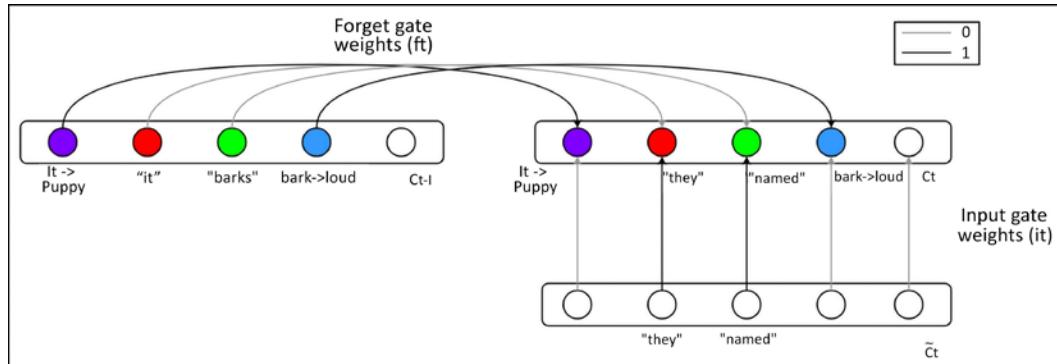


Figure 7.7: How the cell state c_t is calculated with the previous state c_{t-1} and the candidate value \tilde{c}_t

The current cell state will be updated as follows:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

In other words, the current state is a combination of the following:

- What information to forget/remember from the previous cell state
- What information to add/discard to the current input

Next, in *Figure 7.8*, we highlight what we have calculated so far with respect to all the calculations that are taking place inside an LSTM:

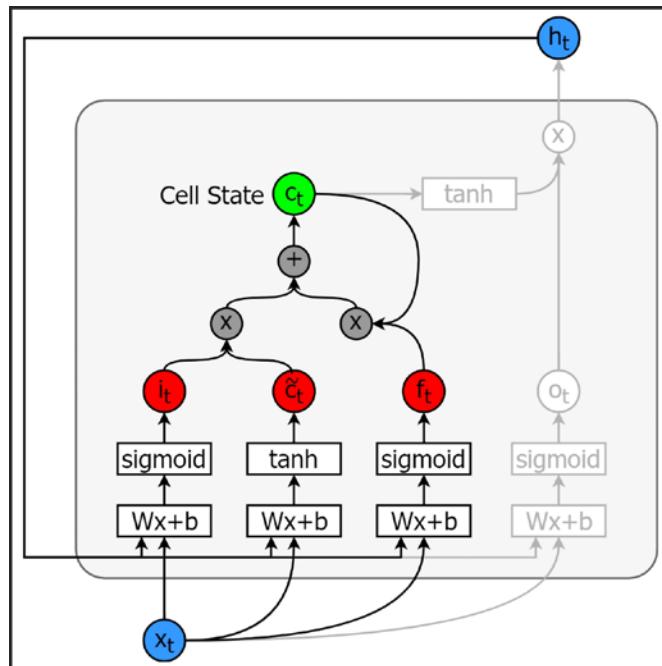


Figure 7.8: Calculations covered so far, including i_t , f_t , \tilde{c}_t , and c_t

After learning the full cell state, it would look like *Figure 7.9*:

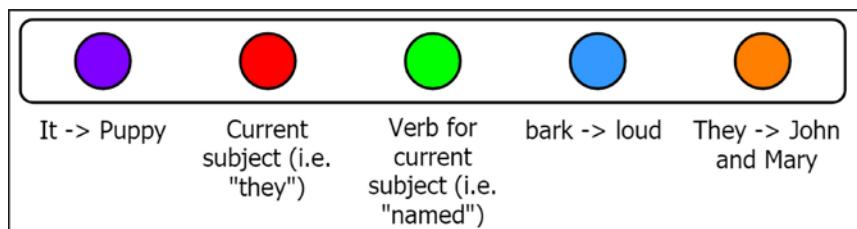


Figure 7.9: The full cell state will look like this after outputting both the sentences

Next, we will look at how the final state of the LSTM cell (h_t) is computed:

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

$$h_t = o_t \tanh(c_t)$$

In our example, we want to output the following sentence:

They named it Luna.

For this, we do **not** need the second to last neuron to compute this sentence, as it contains information about how the puppy barks, whereas this sentence is about the name of the puppy. Therefore, we can ignore this neuron (containing the *bark -> loud* relationship) during the predictions of the last sentence. This is exactly what o_t does; it ignores the unnecessary memory and only retrieves the related memory from the cell state when calculating the final output of the LSTM cell. Also, in *Figure 7.10*, we illustrate what a full LSTM cell would look like at a glance:

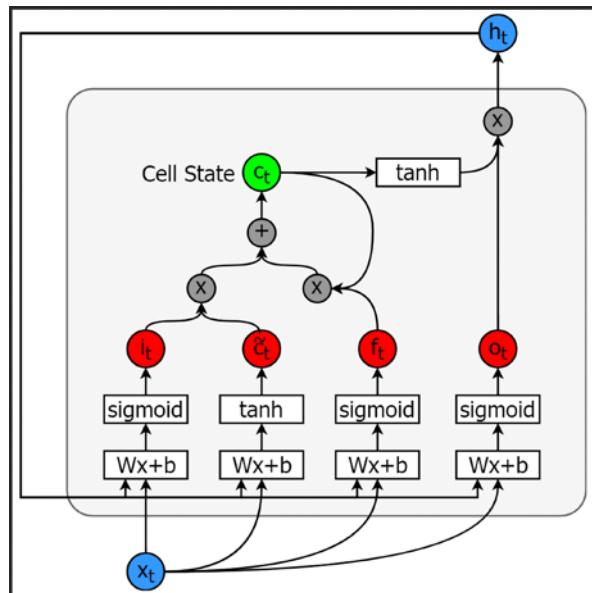


Figure 7.10: What the full LSTM looks like

Here, we summarize all the equations relating to the operations taking place within an LSTM cell:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

$$\tilde{c}_t = \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

$$h_t = o_t \tanh(c_t)$$

Now in the bigger picture, for a sequential learning problem, we can unroll the LSTM cells over time to show how they would link together so that they receive the previous state of the cell to compute the next state, as shown in *Figure 7.11*:

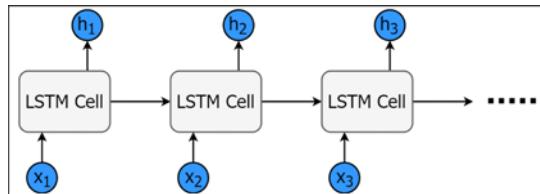


Figure 7.11: How LSTMs would be linked over time

However, this is not adequate to do something useful. We typically use machine learning models to solve a task formulated as a classification or regression problem. As you can see, we still don't have an output layer to output predictions. But if we want to use what the LSTM actually learned, we need a way to extract the final output from the LSTM. Therefore, we will fit a softmax layer (with weights W_s and bias b_s) on top of the LSTM. The final output is obtained using the following equation:

$$y_t = \text{softmax}(W_s h_t + b_s)$$

Now the final picture of the LSTM with the softmax layer looks like *Figure 7.12*:

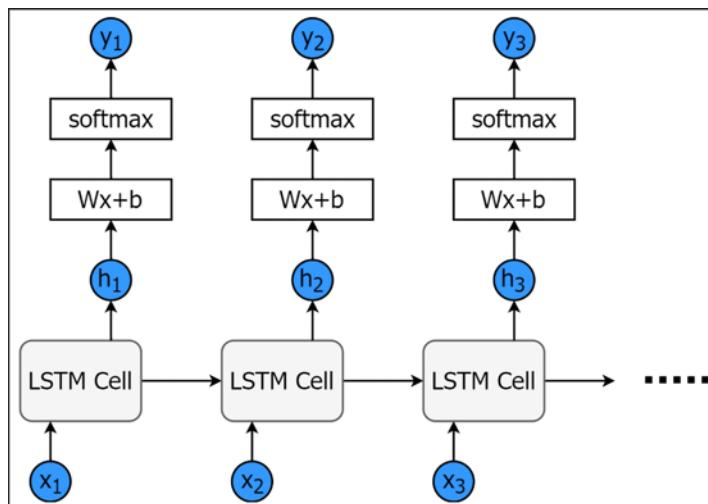


Figure 7.12: LSTMs with a softmax output layer linked over time

With the softmax head attached to the LSTM, it can now perform a given classification task end to end. Now let's compare and contrast LSTMs and the standard RNN model we discussed in the previous chapter.

How LSTMs differ from standard RNNs

Let's now investigate how LSTMs compare to standard RNNs. An LSTM has a more intricate structure compared to a standard RNN. One of the primary differences is that an LSTM has two different states: a cell state c_t and a final hidden state h_t . However, an RNN only has a single hidden state h_t . The next primary difference is that, since an LSTM has three different gates, an LSTM has much more control over how the current input and the previous cell state are handled when computing the final hidden state h_t .

Having the two different states is quite advantageous. With this mechanism, we can decouple the model's short-term and long-term memory. In other words, even when the cell state is changing quickly, the final hidden state will still be changed more slowly. So, while the cell state is learning both short-term and long-term dependencies, the final hidden state can reflect either only the short-term dependencies, only the long-term dependencies, or both.

Next, the gating mechanism is composed of three gates: the input, forget, and output gates.

It is quite evident that this is a more principled approach (especially compared to the standard RNNs) that permits better control over how much the current input and the previous cell state contribute to the current cell state. Also, the output gate gives better control over how much the cell state contributes to the final hidden state.

In *Figure 7.13*, we compare schematic diagrams of a standard RNN and an LSTM to emphasize the difference in terms of the functionality of the two models:

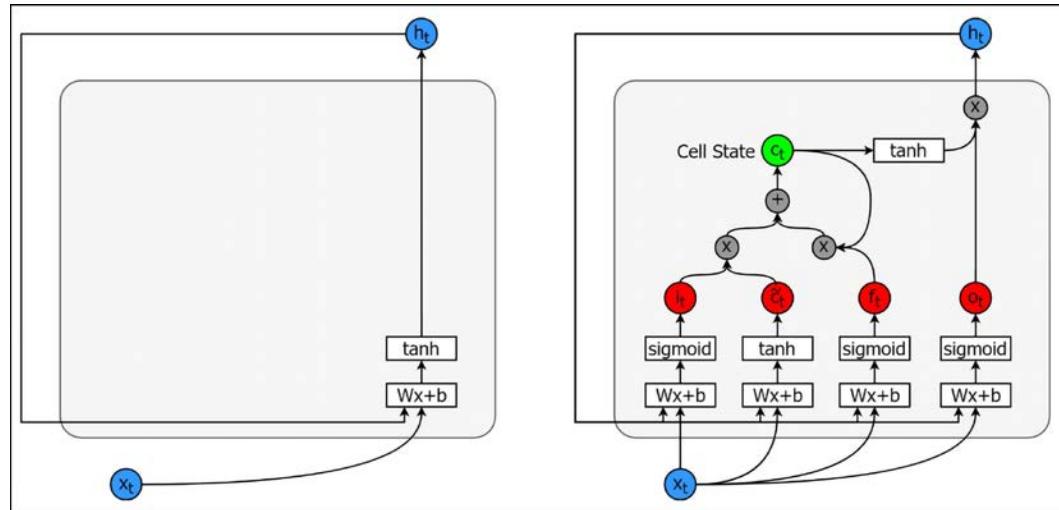


Figure 7.13: A side-by-side comparison of a standard RNN and an LSTM cell

In summary, with the design of maintaining two different states, an LSTM can learn both short-term and long-term dependencies, which helps solve the problem of the vanishing gradient, which we'll discuss in the following section.

How LSTMs solve the vanishing gradient problem

As we discussed earlier, even though RNNs are theoretically sound, in practice they suffer from a serious drawback. That is, when **Backpropagation Through Time (BPTT)** is used, the gradient diminishes quickly, which allows us to propagate the information of only a few time steps. Consequently, we can only store the information of very few time steps, thus possessing only short-term memory. This in turn limits the usefulness of RNNs in real-world sequential tasks.

Often, useful and interesting sequential tasks (such as stock market predictions or language modeling) require the ability to learn and store long-term dependencies. Think of the following example for predicting the next word:

John is a talented student. He is an A-grade student and plays rugby and cricket. All the other students envy ____.

For us, this is a very easy task. The answer would be *John*. However, for an RNN, this is a difficult task. We are trying to predict an answer that lies at the very beginning of the text. Also, to solve this task, we need a way to store long-term dependencies in the state of the RNN. This is exactly the type of task LSTMs are designed to solve.

In *Chapter 6, Recurrent Neural Networks*, we discussed how a vanishing/exploding gradient can appear without any nonlinear functions present. We will now see that it could still happen even with the nonlinear term present. For this, we will derive the term $\partial h_t / \partial h_{t-k}$ for a standard RNN and $\partial c_t / \partial c_{t-k}$ for an LSTM network to understand the differences. This is the crucial term that causes the vanishing gradient, as we learned in the previous chapter.

Let's assume the hidden state is calculated as follows for a standard RNN:

$$h_t = \sigma(W_x x_t + W_h h_{t-1})$$

To simplify the calculations, we can ignore the current input related terms and focus on the recurrent part, which will give us the following equation:

$$h_t = \sigma(W_h h_{t-1})$$

If we calculate $\partial h_t / \partial h_{t-k}$ for the preceding equations, we will get the following:

$$\begin{aligned} \partial h_t / \partial h_{t-k} &= \prod_{i=0}^{k-1} W_h \sigma(W_h h_{t-k+i}) (1 - \sigma(W_h h_{t-k+i})) \\ \partial h_t / \partial h_{t-k} &= W_h^k \prod_{i=0}^{k-1} \sigma(W_h h_{t-k+i}) (1 - \sigma(W_h h_{t-k+i})) \end{aligned}$$

Now let's see what happens when $W_h h_{t-k+i} \ll 0$ or $W_h h_{t-k+i} \gg 0$ (which will happen as learning continues). In both cases, $\partial h_t / \partial h_{t-k}$ will start to approach 0, giving rise to the vanishing gradient. Even when $W_h h_{t-k+i} = 0$, where the gradient is maximum (0.25) for sigmoid activation, when multiplied for many time steps, the overall gradient becomes quite small. Moreover, the term W_h^k (possibly due to bad initialization) can cause exploding or vanishing of the gradients as well. However, compared to the gradient vanishing due to $W_h h_{t-k+i} \ll 0$ or $W_h h_{t-k+i} \gg 0$, the gradient vanishing/explosion caused by the term W_h^k is relatively easy to solve (with careful initialization of weights and gradient clipping).

Now let's look at an LSTM cell. More specifically, we'll look at the cell state, given by the following equation:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

This is the product of all the forget gate applications happening in the LSTM. However, if you calculate $\partial c_t / \partial c_{t-k}$ in a similar way for LSTMs (that is, ignoring the $W_{fx}x_t$ terms and b_f , as they are non-recurrent), we get the following:

$$\frac{\partial c_t}{\partial c_{t-k}} = \prod_{i=0}^{k-1} \sigma(W_{fh}h_{t-k+i})$$

In this case, though the gradient will vanish if $W_h h_{t-k+i} \ll 0$, on the other hand, if $W_h h_{t-k+i} \gg 0$, the derivative will decrease much slower than it would in a standard RNN. Therefore, we have one alternative, where the gradient will not vanish. Also, as the squashing function is used, the gradients will not explode due to $\partial c_t / \partial c_{t-k}$ being large (which is the thing likely to be the cause of a gradient explosion). In addition, when $W_h h_{t-k+i} \gg 0$, we get a maximum gradient close to 1, meaning that the gradients will not rapidly decrease as we saw with RNNs (when the gradient is at maximum). Finally, there is no term such as W_h^k in the derivation. However, derivations are trickier for $\partial h_t / \partial h_{t-k}$. Let's see if such terms are present in the derivation of $\partial h_t / \partial h_{t-k}$. If you calculate the derivatives of this, you will get something of the following form:

$$\frac{\partial h_t}{\partial h_{t-k}} = \frac{\partial(o_t \tanh(c_t))}{\partial h_{t-k}}$$

Once you solve this, you will get something of this form:

$$\tanh(\cdot) \sigma(\cdot)[1 - \sigma(\cdot)]w_{oh} + \sigma(\cdot)[1 - \tanh^2(\cdot)]\{c_{t-1}\sigma(\cdot)[1 - \sigma(\cdot)]w_{fh} + \sigma(\cdot)[1 - \tanh^2(\cdot)]w_{ch} + \tanh(\cdot)\sigma(\cdot)[1 - \sigma(\cdot)]w_{ih}\}$$

We do not care about the content within $\sigma(\cdot)$ or $\tanh(\cdot)$, because no matter the value, it will be bounded by (0,1) or (-1,1). If we further reduce the notation by replacing the $\sigma(\cdot)$, $[1 - \sigma(\cdot)]$, $\tanh(\cdot)$ and $[1 - \tanh^2(\cdot)]$ terms with a common notation such as $\gamma(\cdot)$, we get something of this form:

$$\gamma(\cdot)w_{oh} + \gamma(\cdot)[c_{t-1}\gamma(\cdot)w_{fh} + \gamma(\cdot)w_{ch} + \gamma(\cdot)w_{ih}]$$

Alternatively, we get the following (assuming that the outside $\gamma(\cdot)$ gets absorbed by each $\gamma(\cdot)$ term present within the square brackets):

$$\gamma(\cdot)w_{oh} + c_{t-1}\gamma(\cdot)w_{fh} + \gamma(\cdot)w_{ch} + \gamma(\cdot)w_{ih}$$

This will give the following:

$$\frac{\partial h_t}{\partial h_{t-k}} \approx \prod_{i=0}^{k-1} \gamma(\cdot)w_{oh} + c_{t-1}\gamma(\cdot)w_{fh} + \gamma(\cdot)w_{ch} + \gamma(\cdot)w_{ih}$$

This means that though the term $\partial c_t / \partial c_{t-k}$ is safe from any W_h^k terms, $\partial h_t / \partial h_{t-k}$ is not. Therefore, we must be careful when initializing the weights of the LSTM and we should use gradient clipping as well.



Note

However, h_t of LSTMs being unsafe from vanishing gradient is not as crucial as it is for RNNs, because c_t still can store the long-term dependencies without being affected by vanishing gradient, and h_t can retrieve the long-term dependencies from c_t , if required to.

Improving LSTMs

Having a model backed up by solid foundations does not always guarantee pragmatic success when used in the real world. Natural language is quite complex. Sometimes seasoned writers struggle to produce quality content. So we can't expect LSTMs to magically output meaningful, well-written content all of a sudden. Having a sophisticated design—allowing for better modeling of long-term dependencies in the data—does help, but we need more techniques during inference to produce better text. Therefore, numerous extensions have been developed to help LSTMs perform better at the prediction stage. Here we will discuss several such improvements: greedy sampling, beam search, using word vectors instead of a one-hot-encoded representation of words, and using bidirectional LSTMs. It is important to note that these optimization techniques are not specific to LSTMs; rather, any sequential model can benefit from them.

Greedy sampling

If we try to always predict the word with the highest probability, the LSTM will tend to produce very monotonic results. For example, due to the frequent occurrence of stop words (e.g. *the*), it may repeat them many times before switching to another word.

One way to get around this is to use **greedy sampling**, where we pick the predicted best n and sample from that set. This helps to break the monotonic nature of the predictions.

Let's consider the first sentence of the previous example:

John gave Mary a puppy.

Say, we start with the first word and want to predict the next four words:

John _____.

If we attempt to choose samples deterministically, the LSTM might output something like the following:

John gave Mary gave John.

However, by sampling the next word from a subset of words in the vocabulary (most highly probable ones), the LSTM is forced to vary the prediction and might output the following:

John gave Mary a puppy.

Alternatively, it might give the following output:

John gave puppy a puppy.

However, even though greedy sampling helps to add more flavor/diversity to the generated text, this method does not guarantee that the output will always be realistic, especially when outputting longer sequences of text. Now we will see a better search technique that actually looks ahead several steps before predictions.

Beam search

Beam search is a way of helping with the quality of the predictions produced by the LSTM. In this, the predictions are found by solving a search problem. Particularly, we predict several steps ahead for multiple candidates at each step. This gives rise to a tree-like structure with candidate sequences of words (*Figure 7.14*). The crucial idea of beam search is to produce the b outputs (that is, $y_t, y_{t+1}, \dots, y_{t+b}$) at once instead of a single output y . Here, b is known as the **length** of the beam, and the b outputs produced are known as the **beam**. More technically, we pick the beam that has the highest joint probability $P(y_t, y_{t+1}, \dots, y_{t+b} | x_t)$ instead of picking the highest probable $P(y_t | x_t)$. We are looking farther into the future before making a prediction, which usually leads to better results.

Let's understand beam search through the previous example:

John gave Mary a puppy.

Say, we are predicting word by word and initially we have the following:

John _____.

Let's assume hypothetically that our LSTM produces the example sentence using beam search. Then the probabilities for each word might look like what we see in *Figure 7.14*. Let's assume beam length $b = 2$, and we will consider the $n = 3$ best candidates at each stage of the search.

The search tree would look like the following figure:

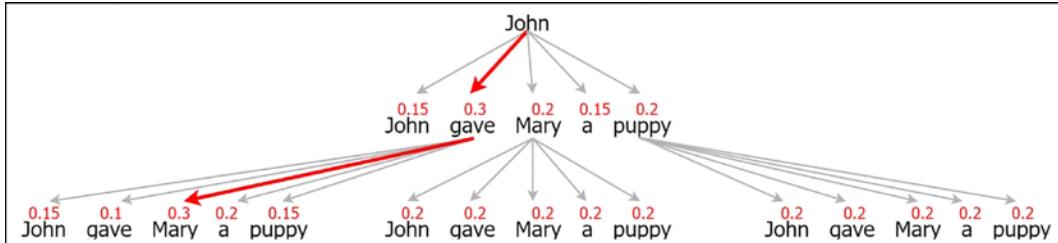


Figure 7.14: The search space of beam search for a $b=2$ and $n=3$

We start with the word *John* and get the probabilities for all the words in the vocabulary. In our example, as $n = 3$, we pick the best three candidates for the next level of the tree: *gave*, *Mary*, and *puppy*. (Note that these might not be the candidates found by an actual LSTM and are only used as an example.) Then from these selected candidates, the next level of the tree is grown. And from that, we will pick the best three candidates, and the search will repeat until we reach a depth of b in the tree.

The path that gives the highest joint probability (that is, $P(\text{gave}, \text{Mary} | \text{John}) = 0.09$) is highlighted with heavier arrows. Also, this is a better prediction mechanism, as it would return a higher probability, or a reward, for a phrase such as *John gave Mary* than *John Mary John* or *John John gave*.

Note that the outputs produced by both greedy sampling and beam search are identical in our example, which is a simple sentence containing five words. However, this is not the case when we scale this to output a small paragraph. Then the results produced by beam search will be much more realistic and meaningful than the ones produced by greedy sampling.

Using word vectors

Another popular way of improving the performance of LSTMs is to use word vectors instead of using one-hot-encoded vectors as the input to the LSTM. Let's understand the value of this method through an example. Let's assume that we want to generate text starting from some random word. In our case, it would be the following:

John _____.

We have already trained our LSTM on the following sentences:

John gave Mary a puppy. Mary has sent Bob a kitten.

Let's also assume that we have the word vectors positioned as shown in *Figure 7.15*. Remember that semantically similar words will have vectors placed close to each other:

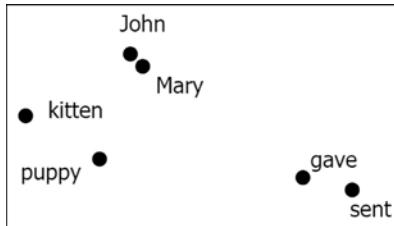


Figure 7.15: Assumed word vectors' topology in two-dimensional space

The word embeddings of these words, in their numerical form, might look like the following:

kitten: [0.5, 0.3, 0.2]

puppy: [0.49, 0.31, 0.25]

gave: [0.1, 0.8, 0.9]

It can be seen that $\text{distance}(\text{kitten}, \text{puppy}) < \text{distance}(\text{kitten}, \text{gave})$. However, if we use one-hot encoding, they would be as follows:

kitten: [1, 0, 0, ...]

puppy: [0, 1, 0, ...]

gave: [0, 0, 1, ...]

Then, $\text{distance}(\text{kitten}, \text{puppy}) = \text{distance}(\text{kitten}, \text{gave})$. As we can already see, one-hot-encoded vectors do not capture the proper relationship between words and see all the words are equally distanced from each other. However, word vectors are capable of capturing such relationships and are more suitable to represent text for machine learning models.

Using word vectors, the LSTM will learn to exploit relationships between words better. For example, with word vectors, LSTM will learn the following:

John gave Mary a kitten.

This is quite close to the following:

John gave Mary a puppy.

Also, it is quite different from the following:

John gave Mary a gave.

However, this would not be the case if one-hot-encoded vectors are used.

Bidirectional LSTMs (BiLSTMs)

Making LSTMs bidirectional is another way of improving the quality of the predictions of an LSTM. By this we mean training the LSTM with text read in both directions: from the beginning to the end and the end to the beginning. So far during the training of the LSTM, we would create a dataset as follows.

Consider the following two sentences:

John gave Mary a _____. It barks very loudly.

At this stage, there is data missing in one of the sentences that we would want our LSTM to fill sensibly.

If we read from the beginning up to the missing word, it would be as follows:

John gave Mary a ____.

This does not provide enough information about the context of the missing word to fill the word properly. However, if we read in both directions, it would be the following:

John gave Mary a ____.

_____. It barks very loudly.

If we created data with both these pieces, it is adequate to predict that the missing word should be something like *dog* or *puppy*. Therefore, certain problems can benefit significantly from reading data from both sides. BiLSTMs also help in multilingual problems as different languages can have very different sentence structures.



Another application of BiLSTMs is neural machine translation, where we translate a sentence of a source language to a target language. As there is no specific alignment between the translation of one language to another, having access to both sides of a given token in the source language can greatly help to understand the context better, thus producing better translations. As an example, consider a translation task of translating Filipino to English. In Filipino, sentences are usually written having *verb-object-subject* in that order, whereas in English, it is *subject-verb-object*. In this translation task, it will be extremely helpful to read sentences both forward and backward to make a good translation.

A BiLSTM is essentially two separate LSTM networks. One network learns data from the beginning to the end, and the other network learns data from the end to the beginning. In *Figure 7.16*, we illustrate the architecture of a BiLSTM network.

Training occurs in two phases. First, the solid-colored network is trained with data created by reading the text from the beginning to the end. This network represents the normal training procedure used for standard LSTMs. Secondly, the dashed network is trained with data generated by reading the text in the reversed direction. Then, at the inference phase, we use both the solid and dashed states' information (by concatenating both states and creating a vector) to predict the missing word:

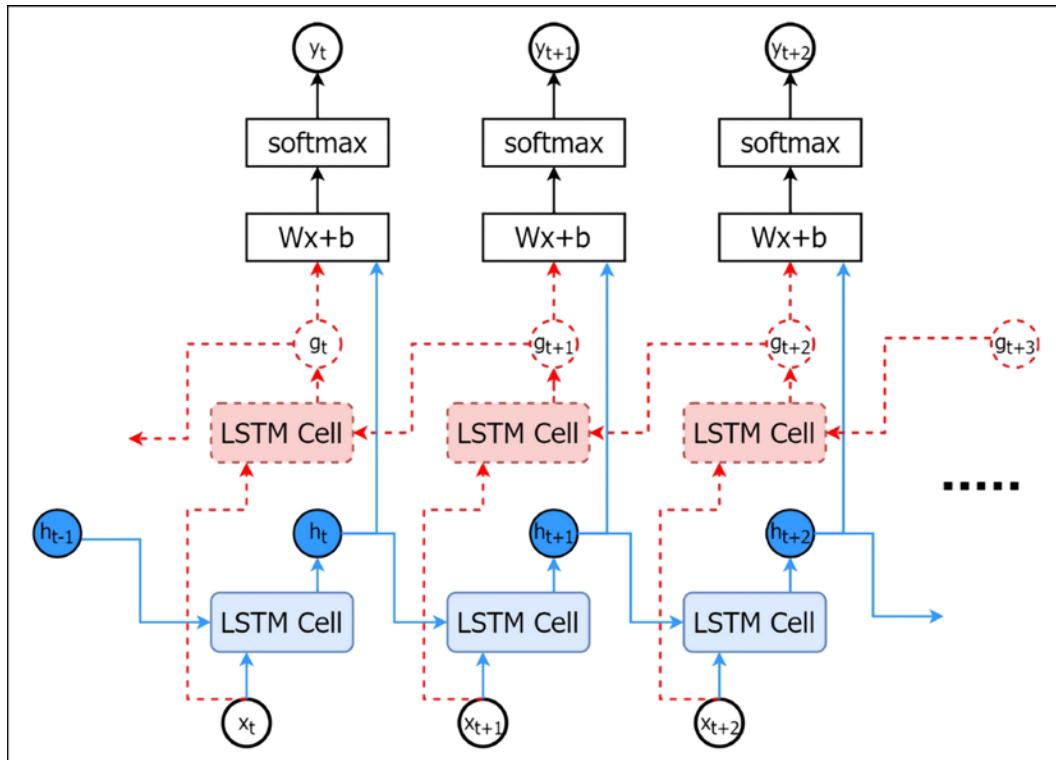


Figure 7.16: A schematic diagram of a BiLSTM

In this section, we discussed several different ways to improve the performance of LSTM models. This involved employing better prediction strategies to introduce structural changes such as word vectors and BiLSTMs.

Other variants of LSTMs

Though we will mainly focus on the standard LSTM architecture, many variants have emerged that either simplify the complex architecture found in standard LSTMs, produce better performance, or both. We will look at two variants that introduce structural modifications to the cell architecture of LSTMs: peephole connections and GRUs.

Peephole connections

Peephole connections allow gates to see not only the current input and the previous final hidden state, but also the previous cell state. This increases the number of weights in the LSTM cell. Having such connections has been shown to produce better results. The equations would look like these:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + W_{ic}c_{t-1} + b_i)$$

$$\tilde{c}_t = \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + W_{fc}c_{t-1} + b_f)$$

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + W_{oc}c_t + b_o)$$

$$h_t = o_t \tanh(c_t)$$

Let's briefly look at how this helps the LSTM perform better. So far, the gates see the current input and final hidden state but not the cell state. However, in this configuration, if the output gate is close to zero, even when the cell state contains information crucial to better performance, the final hidden state will be close to zero. Thus, the gates will not take the hidden state into consideration during calculation. Including the cell state directly in the gate calculation equation allows more control over the cell state, and it can perform well even in situations where the output gate is close to zero.

We illustrate the architecture of the LSTM with peephole connections in *Figure 7.17*. We have grayed all the existing connections in a standard LSTM and the newly added connections are shown in black:

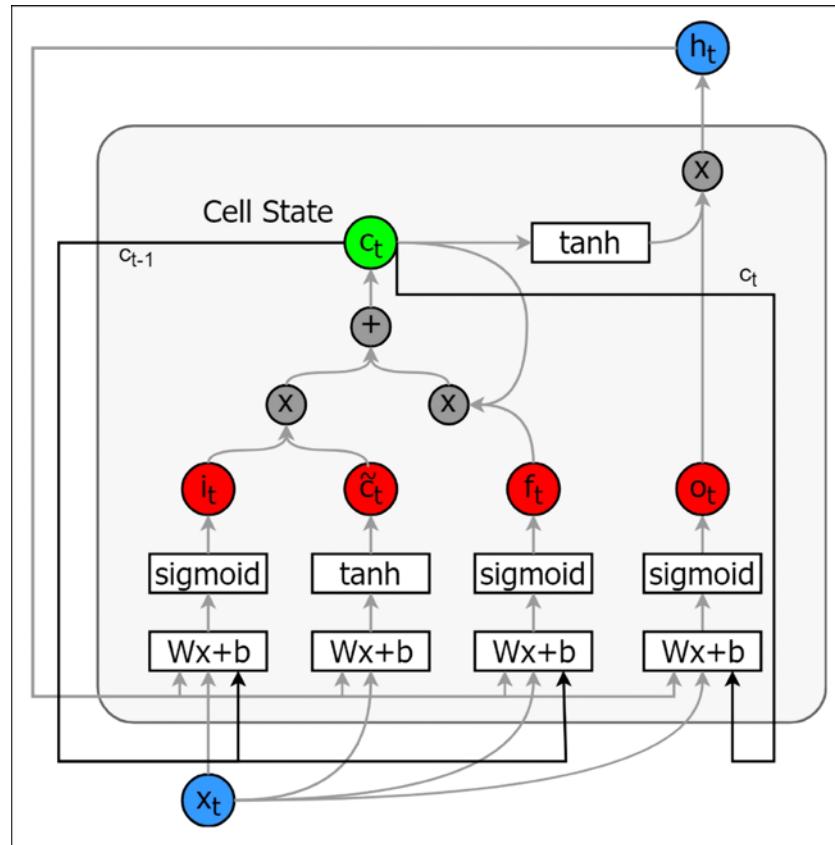


Figure 7.17: An LSTM with peephole connections (the peephole connections are shown in black while the other connections are grayed out)

Gated Recurrent Units

GRUs can be seen as a simplification of the standard LSTM architecture. As we have seen already, an LSTM has three different gates and two different states. This alone requires a large number of parameters even for a small state size. Therefore, scientists have investigated ways to reduce the number of parameters. GRUs are a result of one such endeavor.

There are several main differences in GRUs compared to LSTMs.

First, GRUs combine two states, the cell state and the final hidden state, into a single hidden state h_t . Now, as a side effect of this simple modification of not having two different states, we can get rid of the output gate. Remember, the output gate was merely deciding how much of the cell state is read into the final hidden state. This operation greatly reduces the number of parameters in the cell.

Next, GRUs introduce a reset gate that, when it's close to 1, takes the full previous state information in when computing the current state. Also, when the reset gate is close to 0, it ignores the previous state when computing the current state:

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r)$$

$$\tilde{h}_t = \tanh(W_{hx}x_t + W_{hh}(r_t h_{t-1}) + b_h)$$

Then, GRUs combine the input and forget gates into one *update gate*. The standard LSTM has two gates known as the input and forget gates. The input gate decides how much of the current input is read into the cell state, and the forget gate determines how much of the previous cell state is read into the current cell state. Mathematically, this can be shown as follows:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

GRUs combine these two operations into a single gate known as the update gate. If the update gate is 0, then the full state information of the previous cell state is pushed into the current cell state, where none of the current input is read into the state. If the update gate is 1, then all of the current input is read into the current cell state and none of the previous cell state is propagated into the current cell state. In other words, the input gate i_t becomes inverse of the forget gate, that is, $1 - f_t$:

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z)$$

$$h_t = z_t \tilde{h}_t + (1 - z_t)h_{t-1}$$

Now let's bring all the equations into one place. The GRU computations would look like this:

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r)$$

$$\tilde{h}_t = \tanh(W_{hx}x_t + W_{hh}(r_t h_{t-1}) + b_h)$$

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z)$$

$$h_t = z_t \tilde{h}_t + (1 - z_t)h_{t-1}$$

This is much more compact than LSTMs. In *Figure 7.18*, we can visualize a GRU cell (left) and an LSTM cell (right) side by side:

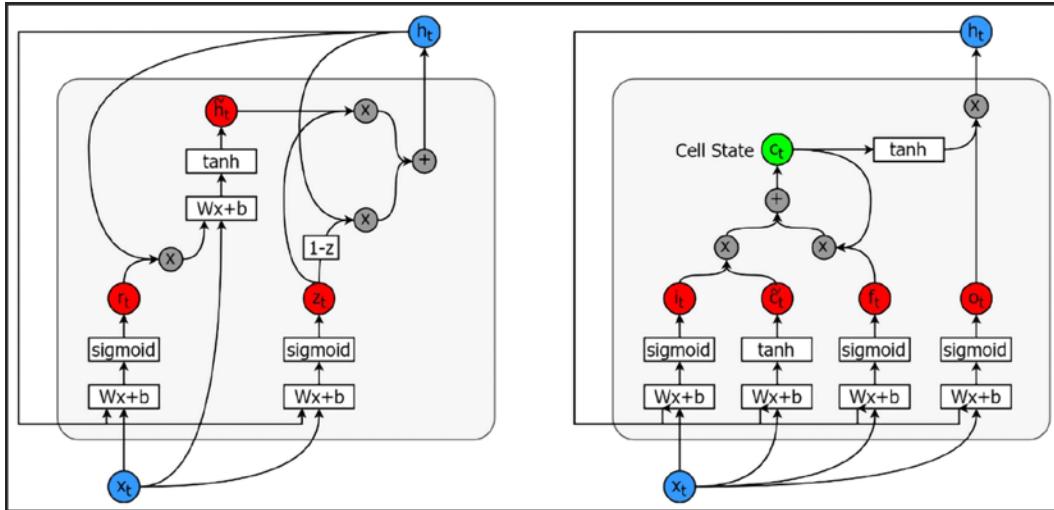


Figure 7.18: A side-by-side comparison of a GRU (left) and the standard LSTM (right)

In this section, we learned two variants of the LSTM: LSTMs with peepholes and GRUs. GRUs have become a popular choice over LSTMs, due to their simplicity and on-par performance with more complex LSTMs.

Summary

In this chapter, you learned about LSTM networks. First, we discussed what an LSTM is and its high-level architecture. We also delved into the detailed computations that take place in an LSTM and discussed the computations through an example.

We saw that an LSTM is composed mainly of five different things:

- **Cell state:** The internal cell state of an LSTM cell
- **Hidden state:** The external hidden state used to calculate predictions
- **Input gate:** This determines how much of the current input is read into the cell state

- **Forget gate:** This determines how much of the previous cell state is sent into the current cell state
- **Output gate:** This determines how much of the cell state is output into the hidden state

Having such a complex structure allows LSTMs to capture both short-term and long-term dependencies quite well.

We compared LSTMs to vanilla RNNs and saw that LSTMs are actually capable of learning long-term dependencies as an inherent part of their structure, whereas RNNs can fail to learn long-term dependencies. Afterward, we discussed how LSTMs solve the vanishing gradient with its complex structure.

Then we discussed several extensions that improve the performance of LSTMs. First, a very simple technique we called greedy sampling, in which, instead of always outputting the best candidate, we randomly sample a prediction from a set of best candidates. We saw that this improves the diversity of the generated text. After that, we looked at a more complex search technique called beam search. With this, instead of making a prediction for a single time step into the future, we predict several time steps into the future and pick the candidates that produce the best joint probability. Another improvement involved seeing how word vectors can help improve the quality of the predictions of an LSTM. Using word vectors, LSTMs can learn more effectively to replace semantically similar words during prediction (for example, instead of outputting *dog*, LSTM might output *cat*), leading to more realism and correctness of the generated text. The final extension we considered was BiLSTMs or bidirectional LSTMs. A popular application of BiLSTMs is filling missing words in a phrase. BiLSTMs read the text in both directions, from the beginning to the end and the end to the beginning. This gives more context as we are looking at both the past and future before predicting.

Finally, we discussed two variants of vanilla LSTMs: peephole connections and GRUs. Vanilla LSTMs, when calculating the gates, only look at the current input and the hidden state. With peephole connections, we make the gate computations dependent on all: the current input, and the hidden and cell states.

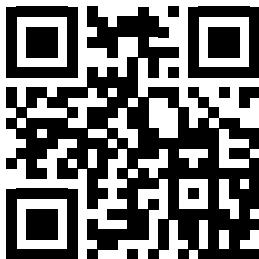
GRUs are a much more elegant variant of vanilla LSTMs that simplify LSTMs without compromising on performance. GRUs have only two gates and a single state, whereas vanilla LSTMs have three gates and two states.

In the next chapter, we will see all these different architectures in action with implementations of each of them and see how well they perform in text generation tasks.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside more than 1000 members at: <https://packt.link/nlp>



8

Applications of LSTM – Generating Text

Now that we have a good understanding of the underlying mechanisms of LSTMs, such as how they solve the problem of the vanishing gradient and update rules, we can look at how to use them in NLP tasks. LSTMs are employed for tasks such as text generation and image caption generation. For example, language modeling is at the core of any NLP task, as the ability to model language effectively leads to effective language understanding. Therefore, this is typically used for pretraining downstream decision support NLP models. By itself, language modeling can be used to generate songs (<https://towardsdatascience.com/generating-drake-rap-lyrics-using-language-models-and-lstms-8725d71b1b12>), movie scripts (<https://builtin.com/media-gaming/ai-movie-script>), etc.

The application that we will cover in this chapter is building an LSTM that can write new folk stories. For this task, we will download translations of some folk stories by the Grimm brothers. We will use these stories to train an LSTM and then ask it to output a fresh new story. We will process the text by breaking it into character-level bigrams (n -grams where $n=2$) and make a vocabulary out of the unique bigrams. Note that representing bigrams as one-hot-encoded vectors is very ineffective for machine learning models, as it forces the model to treat each bigram as an independent unit of text that is entirely different from other bigrams. But bigrams do share semantics, where certain bigrams co-occur where certain ones would not. One-hot encoding will ignore this important property, which is undesirable. To leverage this property in our modeling, we will use an embedding layer and jointly train it with the model.

We will also explore ways to implement previously described techniques such as greedy sampling or beam search for improving the quality of predictions. Afterward, we will see how we can implement time-series models other than standard LSTMs, such as GRUs.

Specifically, this chapter will cover the following main topics:

- Our data
- Implementing the language model
- Comparing LSTMs to LSTMs with peephole connections and GRUs
- Improving sequential models – beam search
- Improving LSTMs – generating text with words instead of n-grams

Our data

First, we will discuss the data we will use for text generation and various preprocessing steps employed to clean the data.

About the dataset

First, we will understand what the dataset looks like so that when we see the generated text, we can assess whether it makes sense, given the training data. We will download the first 100 books from the website <https://www.cs.cmu.edu/~spok/grimmtmp/>. These are translations of a set of books (from German to English) by the Grimm brothers.

Initially, we will download all 209 books from the website with an automated script, as follows:

```
url = 'https://www.cs.cmu.edu/~spok/grimmtmp/'  
dir_name = 'data'  
  
def download_data(url, filename, download_dir):  
    """Download a file if not present, and make sure it's the right  
    size."""  
  
    # Create directories if doesn't exist  
    os.makedirs(download_dir, exist_ok=True)  
  
    # If file doesn't exist download  
    if not os.path.exists(os.path.join(download_dir, filename)):  
        filepath, _ = urlretrieve(url + filename,
```

```
        os.path.join(download_dir,filename))
else:
    filepath = os.path.join(download_dir, filename)

return filepath

# Number of files and their names to download
num_files = 209
filenames = [format(i, '03d')+'.txt' for i in range(1,num_files+1)]

# Download each file
for fn in filenames:
    download_data(url, fn, dir_name)

# Check if all files are downloaded
for i in range(len(filenames)):
    file_exists = os.path.isfile(os.path.join(dir_name,filenames[i]))
    assert file_exists
print('{} files found.'.format(len(filenames)))
```

We will now show example text snippets extracted from two randomly picked stories. The following is the first snippet:

Then she said, my dearest benjamin, your father has had these coffins made for you and for your eleven brothers, for if I bring a little girl into the world, you are all to be killed and buried in them. And as she wept while she was saying this, the son comforted her and said, weep not, dear mother, we will save ourselves, and go hence. But she said, go forth into the forest with your eleven brothers, and let one sit constantly on the highest tree which can be found, and keep watch, looking towards the tower here in the castle. If I give birth to a little son, I will put up a white flag, and then you may venture to come back. But if I bear a daughter, I will hoist a red flag, and then fly hence as quickly as you are able, and may the good God protect you.

The second text snippet is as follows:

Red-cap did not know what a wicked creature he was, and was not at all afraid of him.

“Good-day, little red-cap,” said he.

“Thank you kindly, wolf.”

“Whither away so early, little red-cap?”

“To my grandmother’s.”

“What have you got in your apron?”

“Cake and wine. Yesterday was baking-day, so poor sick grandmother is to have something good, to make her stronger.”

“Where does your grandmother live, little red-cap?”

“A good quarter of a league farther on in the wood. Her house stands under the three large oak-trees, the nut-trees are just below. You surely must know it,” replied little red-cap.

The wolf thought to himself, what a tender young creature. What a nice plump mouthful, she will be better to eat than the old woman.

We now understand what our data looks like. With that understanding, let us move on to processing our data further.

Generating training, validation, and test sets

We will segregate the stories we downloaded into three sets: training, validation, and test files. We will use the content in each set of files as the training, validation, and test data. We will use scikit-learn’s `train_test_split()` function to do so.

```
from sklearn.model_selection import train_test_split

# Fix the random seed so we get the same output everytime
random_state = 54321

filenames = [os.path.join(dir_name, f) for f in os.listdir(dir_name)]

# First separate train and valid+test data
train_filenames, test_and_valid_filenames = train_test_split(filenames,
test_size=0.2, random_state=random_state)

# Separate valid+test data to validation and test data
valid_filenames, test_filenames = train_test_split(test_and_valid_
filenames, test_size=0.5, random_state=random_state)
```

```
# Print out the sizes and some sample filenames
for subset_id, subset in zip(['train', 'valid', 'test'], (train_filenames,
    valid_filenames, test_filenames)):
    print("Got {} files in the {} dataset (e.g.
    {})".format(len(subset), subset_id, subset[:3]))
```

The `train_test_split()` function takes an iterable (e.g. list, tuple, array, etc.) as an input and splits it into two sets based on a defined split ratio. In this case, the input is a list of filenames and we first make a split of 80%-20% training and [validation + test] data. Then we further split the `test_and_valid_filenames` 50%-50% to generate test and validation sets. Note how we also pass a random seed to the `train_test_split` function to make sure we get the same split over multiple runs.

This code will output the following text:

```
Got 167 files in the train dataset (e.g. ['data\\117.txt', 'data\\133.
txt', 'data\\069.txt'])
Got 21 files in the valid dataset (e.g. ['data\\023.txt', 'data\\078.txt',
'data\\176.txt'])
Got 21 files in the test dataset (e.g. ['data\\129.txt', 'data\\207.txt',
'data\\170.txt'])
```

We can see that from our 209 files, we have roughly 80% of files allocated as training data, 10% as validation data, and the final 10% as testing data.

Analyzing the vocabulary size

We will be using bigrams (i.e. n-grams with $n=2$) to train our language model. That is, we will split the story into units of two characters. Furthermore, we will convert all characters to lowercase to reduce the input dimensionality. Using character-level bigrams helps us to language model with a reduced vocabulary, leading to faster model training. For example:

The king was hunting in the forest.

would break down to a sequence of bigrams as follows:

`['th', 'e', 'ki', 'ng', 'w', 'as', ...]`

Let's find out how large the vocabulary is. For that, we first define a `set` object. Next, we go through each training file, read the content, and store that as a string in the variable `document`.

Finally, we update the set object with all the bigrams in the string containing each story. We get the bigrams by traversing the string two characters at a time:

```
bigram_set = set()

# Go through each file in the training set
for fname in train_filenames:
    document = [] # This will hold all the text
    with open(fname, 'r') as f:
        for row in f:
            # Convert text to lower case to reduce input dimensionality
            document.append(row.lower())

    # From the list of text we have, generate one long string
    # (containing all training stories)
    document = " ".join(document)

    # Update the set with all bigrams found
    bigram_set.update([document[i:i+2] for i in range(0,
        len(document), 2)])

# Assign to a variable and print
n_vocab = len(bigram_set)
print("Found {} unique bigrams".format(n_vocab))
```

This would print:

Found 705 unique bigrams

We have a vocabulary of 705 bigrams. It would have been a lot more if we decided to treat each word as a unit, as opposed to character-level bigrams.

Defining the tf.data pipeline

We will now define a fully fledged data pipeline that is capable of reading the files from the disk and transforming the content into a format or structure that can be used to train the model. The `tf.data` API in TensorFlow allows you to define data pipelines that can manipulate data in specific ways to suite machine learning models. For that we will define a function called `generate_tf_dataset()` that takes:

- `filenames` – A list of filenames containing the text to be used for the model
- `ngram_width` – Width of the n-grams to be extracted
- `window_size` – Length of the sequence of n-grams to be used to generate a single data point for the model
- `batch_size` – Size of the batch
- `shuffle` – (defaults to `False`) Whether to shuffle the data or not

For example assume an `ngram_width` of 2, `batch_size` of 1, and `window_size` of 5. This function would take the string “*the king was hunting in the forest*” and output:

```
Batch 1: ["th", "e ", "ki", " ng", " w"] -> ["e ", "ki", "ng", " w", "as"]
Batch 2: ["as", " h", "un", "ti", "ng"] -> [" h", "un", "ti", "ng", " i"]
...

```

The left list in each batch represents the input sequence, and the right list represents the target sequence. Note how the right list is simply the left one shifted one to the right. Also note how there’s no overlap between the inputs in the two records. *Figure 8.1* illustrates the high-level process:

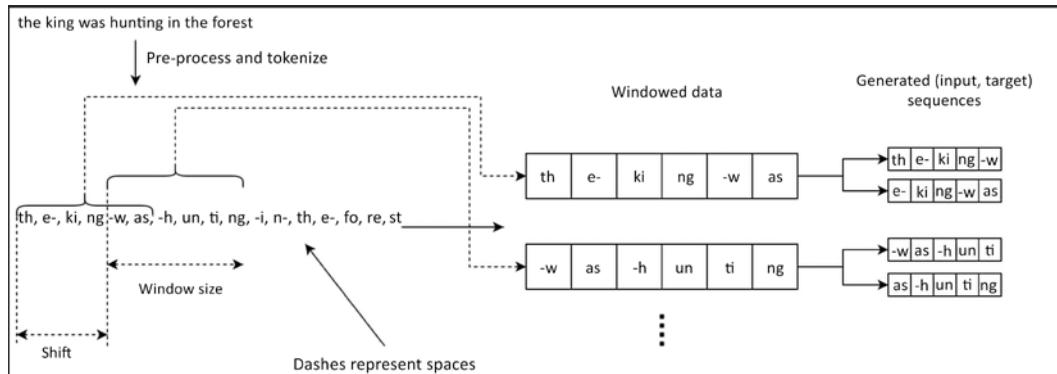


Figure 8.1: The high-level steps of the data transformation we will be implementing with the `tf.data` API

Let’s discuss the specifics of how the pipeline is implemented using TensorFlow’s `tf.data` API. We define the code to generate the data pipeline as a reusable function:

```
def generate_tf_dataset(filenames, ngram_width, window_size, batch_size,
shuffle=False):
    """ Generate batched data from a list of files specified """

```

```
# Read the data found in the documents
documents = []
for f in filenames:
    doc = tf.io.read_file(f)
    doc = tf.strings.ngrams()      # Generate ngrams from the string
    tf.strings.bytes_split(
        # Create a list of chars from a string
        tf.strings.regex_replace(
            # Replace new lines with space
            tf.strings.lower()    # Convert string to lower case
            doc
            ), "\n", " "
        )
    ),
    ngram_width, separator=''
)
documents.append(doc.numpy().tolist())

# documents is a list of list of strings, where each string is a story
# From that we generate a ragged tensor
documents = tf.ragged.constant(documents)
# Create a dataset where each row in the ragged tensor would be a
# sample
doc_dataset = tf.data.Dataset.from_tensor_slices(documents)
# We need to perform a quick transformation - tf.strings.ngrams
# would generate all the ngrams (e.g. abcd -> ab, bc, cd) with
# overlap, however for our data we do not need the overlap, so we need
# to skip the overlapping ngrams
# The following line does that
doc_dataset = doc_dataset.map(lambda x: x[::ngram_width])

# Here we are using a window function to generate windows from text
# For a text sequence with window_size 3 and shift 1 you get
# e.g. ab, cd, ef, gh, ij, ... -> [ab, cd, ef], [cd, ef, gh], [ef,
# gh, ij], ...
# each of these windows is a single training sequence for our model
doc_dataset = doc_dataset.flat_map(
```

```
    lambda x: tf.data.Dataset.from_tensor_slices(  
        x  
    ).window(  
        size=window_size+1, shift=int(window_size * 0.75)  
    ).flat_map(  
        lambda window: window.batch(window_size+1,  
        drop_remainder=True)  
    )  
)  
  
# From each windowed sequence we generate input and target tuple  
# e.g. [ab, cd, ef] -> ([ab, cd], [cd, ef])  
doc_dataset = doc_dataset.map(lambda x: (x[:-1], x[1:]))  
  
# Batch the data  
doc_dataset = doc_dataset.batch(batch_size=batch_size)  
  
# Shuffle the data if required  
doc_dataset = doc_dataset.shuffle(buffer_size=batch_size*10) if  
shuffle else doc_dataset  
  
# Return the data  
return doc_dataset
```

Let's now discuss the above code in more detail. First we go through each file in the `filenames` variable and read the content in each with:

```
doc = tf.io.read_file(f)
```

After the content is read, we generate n-grams from that using the `tf.strings.ngrams()` function. However, this function expects a list of chars as opposed to a string.

Therefore, we convert the string into a list of chars with the `tf.strings.bytes_split()` function. Additionally, we perform several preprocessing steps, such as:

- Converting text to lowercase with `tf.strings.lower()`
- Replacing new-line characters (`\n`) with space to have a continuous stream of words

Each of these stories is stored in a list object (`documents`). It is important to note that, `tf.strings.ngrams()` produces all possible n-grams for a given n-gram length. In other words, consecutive n-grams would overlap. For example, the sequence “*The king was hunting*” with an n-gram length of 2 would produce `["Th", "he", "e ", " k", ...]`. Therefore, we will need an extra processing step later to remove the overlapping n-grams from the sequence. After all of them are read and processed, we create a `RaggedTensor` object from the documents:

```
documents = tf.ragged.constant(documents)
```

A `RaggedTensor` is a special type of tensor that can have dimensions that accept arbitrarily sized inputs. For example, it is almost impossible that all the stories would have the same number of n-grams in each as they vary from each other a lot. In this case, we will have arbitrarily long sequences of n-grams representing our stories. Therefore, we can use a `RaggedTensor` to store these arbitrarily sized sequences.

`tf.RaggedTensor` objects are a special type of tensor that can have variable-sized dimensions. You can read more about ragged tensors at https://www.tensorflow.org/api_docs/python/tf/RaggedTensor. There are many ways to define a ragged tensor.

We can define a ragged tensor by passing a nested list containing values to the `tf.ragged.constant()` function:

```
a = tf.ragged.constant([[1, 2, 3], [1,2], [1]])
```

We can also define a flat sequence of values and define where to split the rows:



```
b = tf.RaggedTensor.from_row_splits([1,2,3,4,5,6,7], row_splits=[0, 3, 3, 6, 7])
```

Here, each value in the `row_splits` argument defines where the subsequent row in the resulting tensor ends. For example, the first row will contain elements from index 0 to 3 (i.e. 0, 1, 2). This will output:

```
<tf.RaggedTensor [[1, 2, 3], [], [4, 5, 6], [7]]>
```

You can get the shape of the tensor using `b.shape`, which will return:

```
[4, None]
```

Next, we create a `tf.data.Dataset` from the tensor with the `tf.data.Dataset.from_tensor_slices()` function.

This function simply produces a dataset, where a single item in the dataset would be a row of the provided tensor. For example, if you provide a standard tensor of shape [10, 8, 6], it will produce 10 samples of shape [8, 6]:

```
doc_dataset = tf.data.Dataset.from_tensor_slices(documents)
```

Here, we simply get rid of the overlapping n-grams by taking only every n^{th} n-gram in the sequence:

```
doc_dataset = doc_dataset.map(lambda x: x[::ngram_width])
```

We will then use the `tf.data.Dataset.window()` function to create shorter, fixed-length windowed sequences from each story:

```
doc_dataset = doc_dataset.flat_map(  
    lambda x: tf.data.Dataset.from_tensor_slices(  
        x  
    ).window(  
        size=window_size+1, shift=int(window_size * 0.75)  
    ).flat_map(  
        lambda window: window.batch(window_size+1,  
            drop_remainder=True)  
    )  
)
```

From each window, we generate input and target pairs, as follows. We take all the n-grams except the last as inputs and all the n-grams except the first as targets. This way, at each time step, the model will be predicting the next n-gram given all the previous n-grams. The shift determines how much we shift the window at each iteration. Having some overlap between records make sure the model doesn't treat the story as independent windows, which may lead to poor performance. We will maintain around 25% overlap between two consecutive sequences:

```
doc_dataset = doc_dataset.map(lambda x: (x[:-1], x[1:])))
```

We shuffle the data using `tf.data.Dataset.shuffle()` and batch the data with a predefined batch size. Note that we have to specify a `buffer_size` for the `shuffle()` function. `buffer_size` determines how much data is retrieved before shuffling. The more data you buffer, the better the shuffling would be, but also the worse the memory consumption would be:

```
doc_dataset = doc_dataset.shuffle(buffer_size=batch_size*10) if shuffle  
else doc_dataset  
doc_dataset = doc_dataset.batch(batch_size=batch_size)
```

Finally, we specify the necessary hyperparameters and generate three datasets: training, validation, and testing:

```
ngram_length = 2
batch_size = 256
window_size = 128

train_ds = generate_tf_dataset(train_filenames, ngram_length, window_size,
batch_size, shuffle=True)
valid_ds = generate_tf_dataset(valid_filenames, ngram_length, window_size,
batch_size)
test_ds = generate_tf_dataset(test_filenames, ngram_length, window_size,
batch_size)
```

Let's generate some data and look at the data generated by this function:

```
ds = generate_tf_dataset(train_filenames, 2, window_size=10, batch_
size=1).take(5)

for record in ds:
    print(record[0].numpy(), '->', record[1].numpy())
```

This returns:

```
[[b'th' b'er' b'e ' b'wa' b's ' b'on' b'ce' b' u' b'po' b'n ']] -> [[b'er'
b'e ' b'wa' b's ' b'on' b'ce' b' u' b'po' b'n ' b'a ']]
[[b' u' b'po' b'n ' b'a ' b'ti' b'me' b' a' b' s' b'he' b'ph']] -> [[b'po'
b'n ' b'a ' b'ti' b'me' b' a' b' s' b'he' b'ph' b'er']]
[[b' s' b'he' b'ph' b'er' b'd ' b'bo' b'y ' b'wh' b'os' b'e ']] -> [[b'he'
b'ph' b'er' b'd ' b'bo' b'y ' b'wh' b'os' b'e ' b'fa']]
```

...

Here, you can see that the target sequence is just the input sequence shifted one to the right. The b in front of the characters denotes that the characters are stored as bytes. Next, we will look at how we can implement the model.

Implementing the language model

Here, we will discuss the details of the LSTM implementation.

First, we will discuss the hyperparameters that are used for the LSTM and their effects.

Thereafter, we will discuss the parameters (weights and biases) required to implement the LSTM. We will then discuss how these parameters are used to write the operations taking place within the LSTM. This will be followed by understanding how we will sequentially feed data to the LSTM. Next, we will discuss how to train the model. Finally, we will investigate how we can use the learned model to output predictions, which are essentially bigrams that will eventually add up to a meaningful story.

Defining the TextVectorization layer

We discussed the TextVectorization layer and used it in *Chapter 6, Recurrent Neural Networks*. We'll be using the same text vectorization mechanism to tokenize text. In summary, the TextVectorization layer provides you with a convenient way to integrate text tokenization (i.e. converting strings into a list of tokens that are represented by integer IDs) into the model as a layer.

Here, we will define a TextVectorization layer to convert the sequences of n-grams to sequences of integer IDs:

```
import tensorflow.keras.layers as layers
import tensorflow.keras.models as models

# The vectorization layer that will convert string bigrams to IDs
text_vectorizer = tf.keras.layers.TextVectorization(
    max_tokens=n_vocab, standardize=None,
    split=None, input_shape=(window_size,))
)
```

Note that we are defining several important arguments, such as the `max_tokens` (size of the vocabulary), the `standardize` argument to not perform any text preprocessing, the `split` argument to not perform any splitting, and finally, the `input_shape` argument to inform the layer that the input will be a batch of sequences of n-grams. With that, we have to train the text vectorization layer to recognize the available n-grams and map them to unique IDs. We can simply pass our training `tf.data` pipeline to this layer to learn the n-grams.

```
text_vectorizer.adapt(train_ds)
```

Next, let's print the words in the vocabulary to see what this layer has learned:

```
text_vectorizer.get_vocabulary()[:10]
```

This will output:

```
['', '[UNK]', 'e', 'he', 't', 'th', 'd', 'a', ' ', 'h']
```

Once the `TextVectorization` layer is trained, we have to modify our training, validation, and testing data pipelines slightly. Remember that our data pipelines output sequences of n-gram strings as inputs and targets. We need to convert the target sequences to sequences of n-gram IDs so that a loss can be computed. For that we will simply pass the targets in the datasets through the `text_vectorizer` layer using the `tf.data.Dataset.map()` functionality:

```
train_ds = train_ds.map(lambda x, y: (x, text_vectorizer(y)))
valid_ds = valid_ds.map(lambda x, y: (x, text_vectorizer(y)))
```

Next, we will look at the LSTM-based model we'll be using. We'll go through various components of the model such as the embedding layer, LSTM layers, and the final prediction layer.

Defining the LSTM model

We will define a simple LSTM-based model. Our model will have:

- The previously trained `TextVectorization` layer
- An embedding layer randomly initialized and jointly trained with the model
- Two LSTM layers each with 512 and 256 nodes respectively
- A fully-connected hidden layer with 1024 nodes and ReLU activation
- The final prediction layer with `n_vocab` nodes and `softmax` activation

Since the model is quite straightforward with the layers defined sequentially, we will use the Sequential API to define this model:

```
import tensorflow.keras.backend as K

K.clear_session()

lm_model = models.Sequential([
    text_vectorizer,
    layers.Embedding(n_vocab+2, 96),
    layers.LSTM(512, return_state=False, return_sequences=True),
    layers.LSTM(256, return_state=False, return_sequences=True),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.5),
```

```
    layers.Dense(n_vocab, activation='softmax')  
])
```

We start by calling `K.clear_session()`, which is a function that clears the current TensorFlow session (e.g. layers and variables defined and their states). Otherwise, if you run multiple times in a notebook, it will create an unnecessary number of layers and variables. Additionally, let's look at the parameters of the LSTM layer in more detail:

- `return_state` – Setting this to `False` means that the layer outputs only the final output, whereas if set to `True`, it will return state vectors along with the final output of the layer. For example, for an LSTM layer, setting `return_state=True` means you'll get three outputs: the final output, cell state, and hidden state. Note that the final output and the hidden state will be identical in this case.
- `return_sequences` – Setting this to `true` will cause the layer to output the full output sequences, as opposed to just the last output. For example, setting this to `false` will give you a $[b, n]$ -sized output where b is the batch size and n is the number of nodes in the layer. If `true`, it will output a $[b, t, n]$ -sized output, where t is the number of time steps.

You can see a summary of this model by executing:

```
lm_model.summary()
```

which returns:

Model: "sequential"		
Layer (type)	Output Shape	Param #
text_vectorization (TextVec torization)		0
embedding (Embedding)	(None, 128, 96)	67872
lstm (LSTM)	(None, 128, 512)	1247232
lstm_1 (LSTM)	(None, 128, 256)	787456
dense (Dense)	(None, 128, 1024)	263168

dropout (Dropout)	(None, 128, 1024)	0
dense_1 (Dense)	(None, 128, 705)	722625
<hr/>		
Total params:	3,088,353	
Trainable params:	3,088,353	
Non-trainable params:	0	

Next, let's look at the metrics we can use to track model performance and finally compile the model with appropriate loss, optimizer, and metrics.

Defining metrics and compiling the model

For our language model, we have to define a performance metric that we can use to demonstrate how good the model is. We have typically seen accuracy being used widely as a general-purpose evaluation metric across different ML tasks. However, accuracy might not be cut out for this task, mainly because it relies on the model choosing the exact word/bigram for a given time step as in the dataset. However, languages are complex and there can be many different choices to generate the next word/bigram given a text. Therefore, NLP practitioners rely on a metric known as **perplexity**, which measures how “perplexed” or “surprised” the model was to see a $t+1$ bigram given $1:t$ bigrams.

Perplexity computation is simple. It's simply the entropy to the power of two. Entropy is a measure of the uncertainty or randomness of an event. The more uncertain the outcome of the event, the higher the entropy (to learn more about entropy visit <https://machinelearningmastery.com/what-is-information-entropy/>). Entropy is computed as:

$$H(X) = - \sum_{x \in X} p(x) \log(p(x))$$

In machine learning, to optimize ML models, we measure the difference between the predicted probability distribution versus the target probability distribution for a given sample. For that, we use cross-entropy, an extension of entropy for two distributions:

$$\text{CategoricalCrossEntropy}(\hat{y}_l, y_l) = - \sum_{c=1}^C y_{l,c} \log(\hat{y}_{l,c})$$

Finally, we define perplexity as:

$$\text{Perplexity} = 2^{H(X)}$$

To learn more about the relationship between cross entropy and perplexity visit <https://thegradient.pub/understanding-evaluation-metrics-for-language-models/>.

In TensorFlow, we define a custom `tf.keras.metrics.Metric` object to compute perplexity. We are going to use `tf.keras.metrics.Mean` as our super-class as it already knows how to compute and track the mean value of a given metric:

```
class PerplexityMetric(tf.keras.metrics.Mean):

    def __init__(self, name='perplexity', **kwargs):
        super().__init__(name=name, **kwargs)
        self.cross_entropy =
            tf.keras.losses.SparseCategoricalCrossentropy(
                from_logits=False, reduction='none')

    def _calculate_perplexity(self, real, pred):

        # The next 4 Lines zero-out the padding from Loss
        # calculations, this follows the logic from:
        # https://www.tensorflow.org/beta/tutorials/text/transformer#Loss_
        # and_metrics
        loss_ = self.cross_entropy(real, pred)
        # Calculating the perplexity steps:
        step1 = K.mean(loss_, axis=-1)
        perplexity = K.exp(step1)

    return perplexity

    def update_state(self, y_true, y_pred, sample_weight=None):
        perplexity = self._calculate_perplexity(y_true, y_pred)
        super().update_state(perplexity)
```

Here we are simply computing the cross-entropy loss for a given batch of predictions and targets using the built-in `SparseCategoricalCrossentropy` loss object. Then we raise it to the power of exponential to get the perplexity. We will now compile our model using:

- Sparse categorical cross-entropy as our loss function

- Adam as our optimizer
- Accuracy and perplexity as our metrics

```
lm_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy', PerplexityMetric()])
```

Here, the perplexity metric will be tracked during model training and validation and be printed out, similar to the accuracy metric.

Training the model

It's time to train our model. Since we have done all the heavy-lifting required (e.g. reading files, preprocessing and transforming text, and compiling the model), all we have to do is call our model with the `fit()` function:

```
lm_model.fit(train_ds, validation_data=valid_ds, epochs=60)
```

Here we are passing `train_ds` (training data pipeline) as the first argument and `valid_ds` (validation data pipeline) for the `validation_data` argument, and setting the training to run for 60 epochs. Once the model is trained, let us evaluate it on the test dataset by simply calling:

```
lm_model.evaluate(test_ds)
```

This gives the following output:

```
5/5 [=====] - 0s 45ms/step - loss: 2.4742 -
accuracy: 0.3968 - perplexity: 12.3155
```

You might have slight variations in the metrics you see, but it should roughly converge to the same value.

Defining the inference model

During training, we trained our model and evaluated it on sequences of bigrams. This works for us because during training and evaluation, we have the full text available to us. However, when we need to generate new text, we do not have anything available to us. Therefore, we have to make adjustments to our trained model so that it can generate text from scratch.

The way we do this is by defining a recursive model that takes the current time step's output of the model as the input to the next time step. This way we can keep predicting words/bigrams for an infinite number of steps. We provide the initial seed as a random word/bigram picked from the corpus (or even a sequence of bigrams).

Figure 8.2 illustrates how the inference model works.

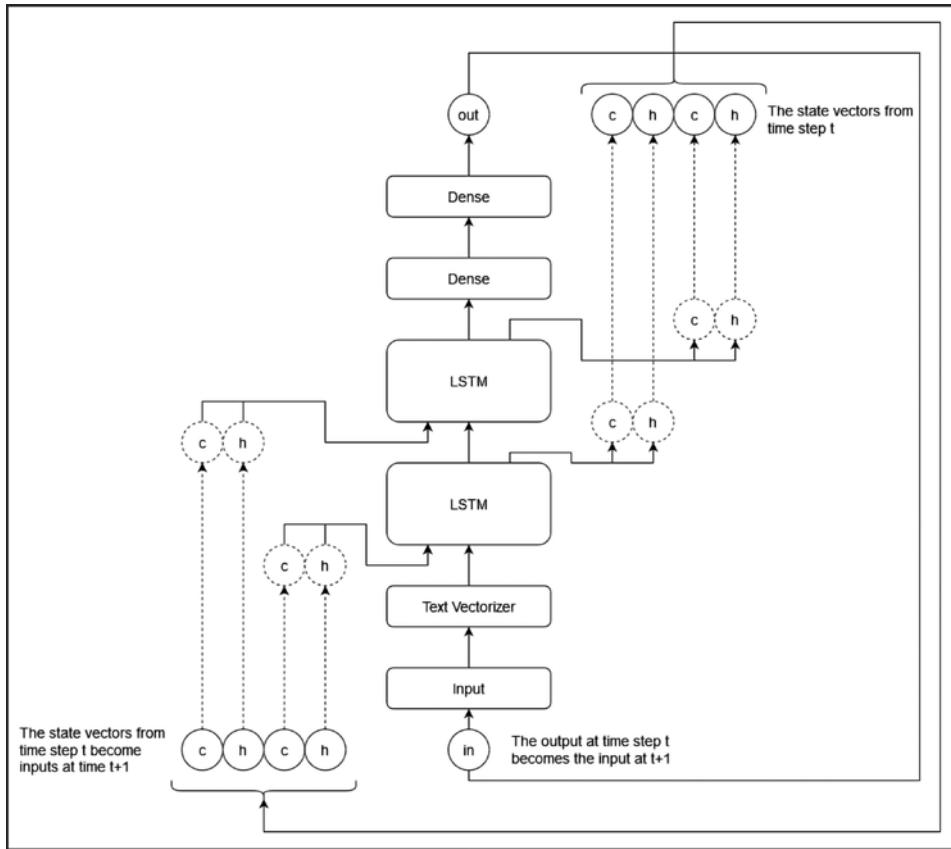


Figure 8.2: The operational view of the inference model we'll be building from our trained model

Our inference model is going to be comparatively more sophisticated, as we need to design an iterative process to generate text using previous predictions as inputs. Therefore, we will be using Keras's Functional API to implement the model:

```
# Define inputs to the model
inp = tf.keras.layers.Input(dtype=tf.string, shape=(1,))

inp_state_c_lstm = tf.keras.layers.Input(shape=(512,))
inp_state_h_lstm = tf.keras.layers.Input(shape=(512,))
inp_state_c_lstm_1 = tf.keras.layers.Input(shape=(256,))
```

```
inp_state_h_lstm_1 = tf.keras.layers.Input(shape=(256,))

text_vectorized_out = lm_model.get_layer('text_vectorization')(inp)

# Define embedding layer and output
emb_layer = lm_model.get_layer('embedding')
emb_out = emb_layer(text_vectorized_out)

# Defining a LSTM Layers and output
lstm_layer = tf.keras.layers.LSTM(512, return_state=True, return_
sequences=True)
lstm_out, lstm_state_c, lstm_state_h = lstm_layer(emb_out, initial_
state=[inp_state_c_lstm, inp_state_h_lstm])

lstm_1_layer = tf.keras.layers.LSTM(256, return_state=True, return_
sequences=True)
lstm_1_out, lstm_1_state_c, lstm_1_state_h = lstm_1_layer(lstm_out,
initial_state=[inp_state_c_lstm_1, inp_state_h_lstm_1])

# Defining a Dense layer and output
dense_out = lm_model.get_layer('dense')(lstm_1_out)

# Defining the final Dense layer and output
final_out = lm_model.get_layer('dense_1')(dense_out)

# Copy the weights from the original model
lstm_layer.set_weights(lm_model.get_layer('lstm').get_weights())
lstm_1_layer.set_weights(lm_model.get_layer('lstm_1').get_weights())

# Define final model
infer_model = tf.keras.models.Model(
    inputs=[inp, inp_state_c_lstm, inp_state_h_lstm,
    inp_state_c_lstm_1, inp_state_h_lstm_1],
    outputs=[final_out, lstm_state_c, lstm_state_h, lstm_1_state_c,
    lstm_1_state_h])
```

We start by defining an input layer that takes an input having one time step.

Note that we are defining the `shape` argument. This means it can accept an arbitrarily sized batch of data (as long as it has one time step). We also define several other inputs to maintain the states of the LSTM layers we have. This is because we have to maintain state vectors of LSTM layers explicitly as we are recursively generating outputs from the model:

```
inp = tf.keras.layers.Input(dtype=tf.string, shape=(1,))

inp_state_c_lstm = tf.keras.layers.Input(shape=(512,))
inp_state_h_lstm = tf.keras.layers.Input(shape=(512,))
inp_state_c_lstm_1 = tf.keras.layers.Input(shape=(256,))
inp_state_h_lstm_1 = tf.keras.layers.Input(shape=(256,))
```

Next we retrieve the trained model's `text_vectorization` layer and transform the text to integer IDs using it:

```
text_vectorized_out = lm_model.get_layer('text_vectorization')(inp)
```

Then we obtain the embeddings layer of the train model and use it to generate the embedding output:

```
emb_layer = lm_model.get_layer('embedding')
emb_out = emb_layer(text_vectorized_out)
```

We will create a fresh new LSTM layer to represent the first LSTM layer in the trained model. This is because the inference LSTM layers will have slight differences to the trained LSTM layers. Therefore, we will define new layers and copy the trained weights over later. We set the `return_state` argument to `True`. By setting this to true we get three outputs when we call the layer with an input: the final output, the cell state, and the final state vector. Note how we are also passing another argument called `initial_state`. The `initial_state` needs to be a list of tensors: the cell state and the final state vector, in that order. We are passing the input layers as those states and will populate them accordingly during runtime:

```
lstm_layer = tf.keras.layers.LSTM(512, return_state=True, return_
sequences=True)
lstm_out, lstm_state_c, lstm_state_h = lstm_layer(emb_out, initial_
state=[inp_state_c_lstm, inp_state_h_lstm])
```

Similarly, the second LSTM layer will be defined. We get the dense layers and replicate the fully connected layers found in the trained model. Note that we don't use `softmax` in the last layer.

This is because at inference time softmax is only an overhead, as we only need the output class with the highest output score (i.e. it doesn't need to be a probability distribution):

```
# Defining a Dense Layer and output
dense_out = lm_model.get_layer('dense')(lstm_1_out)

# Defining the final Dense Layer and output
final_out = lm_model.get_layer('dense_1')(dense_out)
```

Don't forget to copy the weights of the trained LSTM layers to our newly created LSTM layers:

```
lstm_layer.set_weights(lm_model.get_layer('lstm').get_weights())
lstm_1_layer.set_weights(lm_model.get_layer('lstm_1').get_weights())
```

Finally, we define the model:

```
infer_model = tf.keras.models.Model(
    inputs=[inp, inp_state_c_lstm, inp_state_h_lstm,
    inp_state_c_lstm_1, inp_state_h_lstm_1],
    outputs=[final_out, lstm_state_c, lstm_state_h, lstm_1_state_c,
    lstm_1_state_h])
```

Our model takes a sequence of 1 bigram as the input, along with state vectors of both LSTM layers, and outputs the final prediction probabilities and the new state vectors of both LSTM layers. Let us now generate new text from the model.

Generating new text with the model

We'll use our new inference model to generate a story. We will define an initial seed that we will use to generate a story. Here, we take the first phrase from one of the test files. Then we use it to generate text recursively, by using the predicted bigram at time t as the input at time $t+1$. We will run this for 500 steps:

```
text = ["When adam and eve were driven out of paradise, they were
compelled to build a house for themselves on barren ground"]

seq = [text[0][i:i+2] for i in range(0, len(text[0]), 2)]

# build up model state using the given string
print("Making predictions from a {} element long input".format(len(seq)))
```

```
vocabulary = infer_model.get_layer("text_vectorization").get_vocabulary()
index_word = dict(zip(range(len(vocabulary)), vocabulary))

# Reset the state of the model initially
infer_model.reset_states()

# Defining the initial state as all zeros
state_c = np.zeros(shape=(1,512))
state_h = np.zeros(shape=(1,512))
state_c_1 = np.zeros(shape=(1,256))
state_h_1 = np.zeros(shape=(1,256))

# Recursively update the model by assigning new state to state
for c in seq:
    #print(c)
    out, state_c, state_h, state_c_1, state_h_1 = infer_model.predict(
        [np.array([[c]]), state_c, state_h, state_c_1, state_h_1]
    )

# Get final prediction after feeding the input string
wid = int(np.argmax(out[0],axis=-1).ravel())
word = index_word[wid]
text.append(word)

# Define first input to generate text recursively from
x = np.array([[word]])

# Code Listing 10.7
for _ in range(500):

    # Get the next output and state
    out, state_c, state_h, state_c_1, state_h_1 =
        infer_model.predict([x, state_c, state_h, state_c_1, state_h_1])

    # Get the word id and the word from out
    out_argsort = np.argsort(out[0], axis=-1).ravel()
    wid = int(out_argsort[-1])
    word = index_word[wid]
```

```

# If the word ends with space, we introduce a bit of randomness
# Essentially pick one of the top 3 outputs for that timestep
# depending on their likelihood
if word.endswith(' '):
    if np.random.normal()>0.5:
        width = 5
        i = np.random.choice(list(range(-width,0)),
        p=out.argsort[-width:]/out.argsort[-width:].sum())
        wid = int(out.argsort[i])
        word = index_word[wid]

# Append the prediction
text.append(word)

# Recursively make the current prediction the next input
x = np.array([[word]])

# Print the final output
print('\n')
print('*'*60)
print("Final text: ")
print(''.join(text))

```

Notice how we are recursively using the variables `x`, `state_c`, `state_h`, `state_c_1`, and `state_h_1` to generate and assign new values.

```

out, state_c, state_h, state_c_1, state_h_1 =
infer_model.predict([x, state_c, state_h, state_c_1, state_h_1 ])

```

Moreover, we will use a simple condition to diversify the inputs we are generating:

```

if word.endswith(' '):
    if np.random.normal()>0.5:
        width = 5
        i = np.random.choice(list(range(-width,0)),
        p=out.argsort[-width:]/out.argsort[-width:].sum())
        wid = int(out.argsort[i])
        word = index_word[wid]

```

Essentially, if the predicted bigram ends with the ' ' character, we will choose the next bigram randomly, from the top five bigrams. Each bigram will be chosen according to its predicted likelihood. Let's see what the output text looks like:

```
When adam and eve were driven out of paradise, they were compelled to  
build a house for themselves on barren groundy the king's daughter and  
said, i will so the king's daughter angry this they were and said, "i will  
so the king's daughter. the king's daughter.' they were to the forest of  
the stork. then the king's daughters, and they were to the forest of the  
stork, and, and then they were to the forest. ...
```

It seems our model is able to generate actual words and phrases that make sense. Next we will investigate how the text generated from standard LSTMs compares to other models, such as LSTMs with peepholes and GRUs.

Comparing LSTMs to LSTMs with peephole connections and GRUs

Now we will compare LSTMs to LSTMs with peepholes and GRUs in the text generation task. This will help us to compare how well different models (LSTMs with peepholes and GRUs) perform in terms of perplexity. Remember that we prefer perplexity over accuracy, as accuracy assumes there's only one correct token given a previous input sequence. However, as we have learned, language is complex and there can be many different correct ways to generate text given previous inputs. This is available as an exercise in `ch08_lstm_for_text_generation.ipynb` located in the Ch08-Language-Modelling-with-LSTMs folder.

Standard LSTM

First, we will reiterate the components of a standard LSTM. We will not repeat the code for standard LSTMs as it is identical to what we discussed previously. Finally, we will see some text generated by an LSTM.

Review

Here, we will revisit what a standard LSTM looks like. As we already mentioned, an LSTM consists of the following:

- **Input gate** – This decides how much of the current input is written to the cell state

- **Forget gate** – This decides how much of the previous cell state is written to the current cell state
- **Output gate** – This decides how much information from the cell state is exposed to output into the external hidden state

In *Figure 8.3*, we illustrate how each of these gates, inputs, cell states, and the external hidden states are connected:

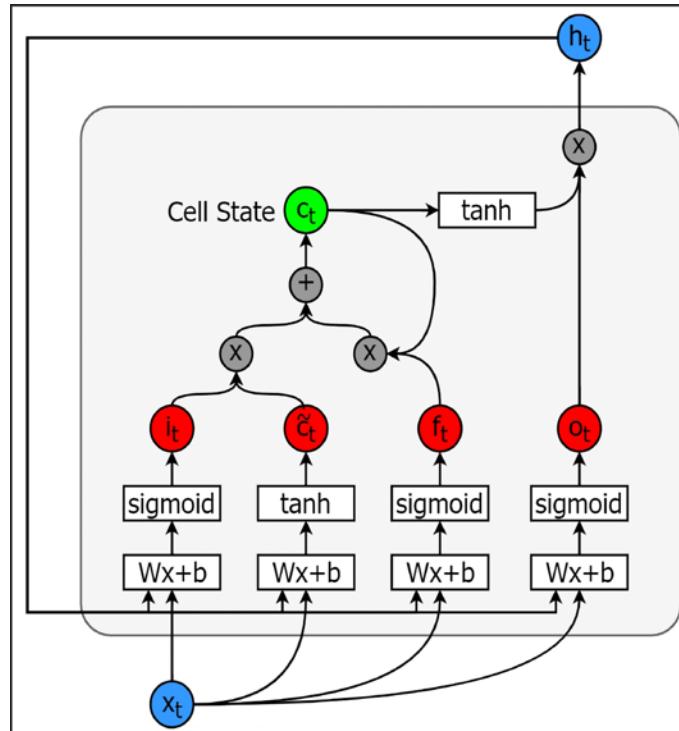


Figure 8.3: An LSTM cell

Gated Recurrent Units (GRUs)

Here we will first briefly delineate what a GRU is composed of, followed by showing the code for implementing a GRU cell. Finally, we look at some code generated by a GRU cell.

Review

Let's briefly revisit what a GRU is. A GRU is an elegant simplification of the operations of an LSTM. A GRU introduces two different modifications to an LSTM (see *Figure 8.4*):

- It connects the internal cell state and the external hidden state into a single state

- Then it combines the input gate and the forget gate into one update gate

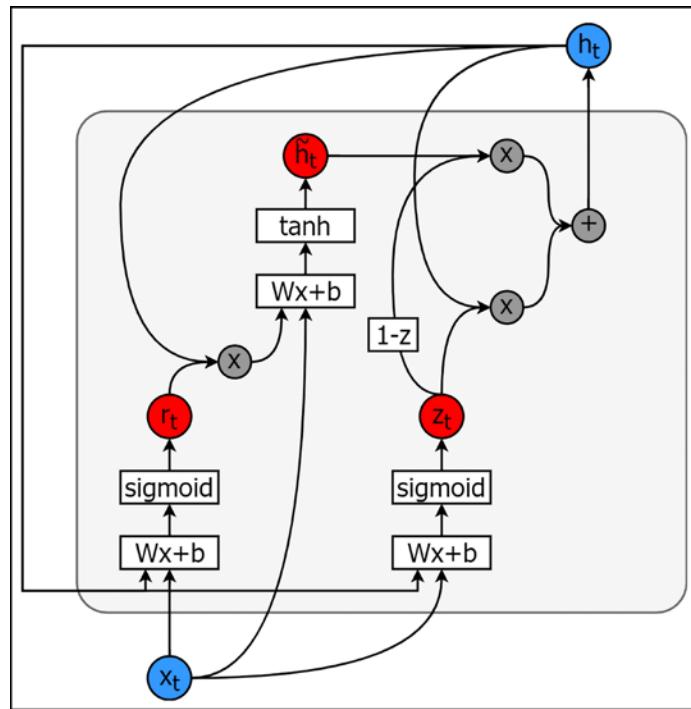


Figure 8.4: A GRU cell

The GRU model uses a simpler gating mechanism than the LSTM. However, it still manages to capture important capabilities such as memory updates, forgets, etc.

The model

Here we will define a GRU-based language model:

```
text_vectorizer = tf.keras.layers.TextVectorization(
    max_tokens=n_vocab, standardize=None,
    split=None, input_shape=(window_size,))
)

# Train the model on existing data
text_vectorizer.adapt(train_ds)

lm_gru_model = models.Sequential([
    text_vectorizer,
```

```

    layers.Embedding(n_vocab+2, 96),
    layers.GRU(512, return_sequences=True),
    layers.GRU(256, return_sequences=True),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(n_vocab, activation='softmax')
)

```

The training code is identical to how we trained the LSTM-based model. Therefore, we won't duplicate our discussion here. Next we'll look at a slightly different variant of LSTM models.

LSTMs with peepholes

Here we will discuss LSTMs with peepholes and how they are different from a standard LSTM. After that, we will discuss their implementation.

Review

Now, let's briefly look at LSTMs with peepholes. Peepholes are essentially a way for the gates (input, forget, and output) to directly see the cell state, instead of waiting for the external hidden state (see *Figure 8.5*):

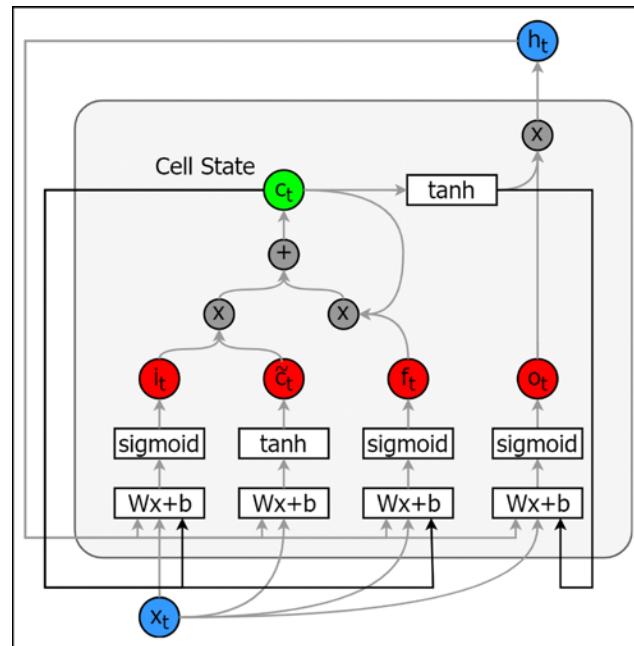


Figure 8.5: An LSTM with peepholes

The code

Note that we're using an implementation of the peephole connections that are diagonal. We found that nondiagonal peephole connections (proposed by Gers and Schmidhuber in their paper *Recurrent Nets that Time and Count, Neural Networks, 2000*) hurt performance more than they help, for this language modeling task. Therefore, we're using a different variation that uses diagonal peephole connections, as used by Sak, Senior, and Beaufays in their paper *Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling, Proceedings of the Annual Conference of the International Speech Communication Association*.

Fortunately, we have this technique implemented as an RNNCell object in tensorflow-addons. Therefore, all we need to do is wrap this PeepholeLSTMCell object in a layers.RNN object to produce the desired layer. The following is the code implementation:

```
text_vectorizer = tf.keras.layers.TextVectorization(  
    max_tokens=n_vocab, standardize=None,  
    split=None, input_shape=(window_size,),  
)  
  
# Train the model on existing data  
text_vectorizer.adapt(train_ds)  
  
lm_peephole_model = models.Sequential([  
    text_vectorizer,  
    layers.Embedding(n_vocab+2, 96),  
    layers.RNN(  
        tfa.rnn.PeepholeLSTMCell(512),  
        return_sequences=True  
,  
    layers.RNN(  
        tfa.rnn.PeepholeLSTMCell(256),  
        return_sequences=True  
,  
    layers.Dense(1024, activation='relu'),  
    layers.Dropout(0.5),  
    layers.Dense(n_vocab, activation='softmax')  
)
```

Now let's look at the training and validation perplexities of different models and how they change over time.

Training and validation perplexities over time

In *Figure 8.6*, we have plotted the behavior of perplexity over time for LSTMs, LSTMs with peepholes, and GRUs. We can see that GRUs are a clear-cut winner in terms of performance. This can be attributed to the innovative simplification of LSTM cells found in GRU cells. But it looks like GRU model does overfit quite heavily. Therefore, it's important to use techniques such as early stopping to prevent such behavior. We can see that LSTMs with peepholes haven't given us much advantage in terms of performance. But it is important to keep in mind that we are using a relatively small dataset.

For larger, more complex datasets, the performance might vary. We will leave experimenting with GRU cells for the reader and continue with the LSTM model:

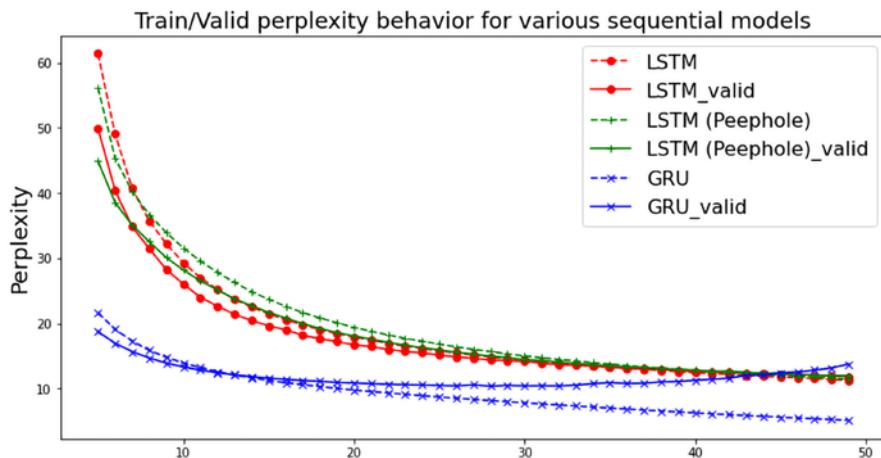


Figure 8.6: Perplexity change for training data over time (LSTMs, LSTM (peephole), and GRUs)

Note



The current literature suggests that among LSTMs and GRUs, there is no clear winner and a lot depends on the task (refer to the paper *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*, Chung and others, NIPS 2014 Workshop on Deep Learning, December 2014 at <https://arxiv.org/abs/1412.3555>).

In this section, we discussed three different models: standard LSTMs, GRUs, and LSTMs with peepholes.

The results clearly indicate that, for this dataset, GRUs outperform other variants. In the next section, we will discuss techniques that can enhance the predictive power of sequential models.

Improving sequential models – beam search

As we saw earlier, the generated text can be improved. Now let's see if beam search, which we discussed in *Chapter 7, Understanding Long Short-Term Memory Networks*, might help to improve the performance. The standard way to predict from a language model is by predicting one step at a time and using the prediction from the previous time step as the new input. In beam search, we predict several steps ahead before picking an input.

This enables us to pick output sequences that may not look as attractive if taken individually, but are better when considered as a sequence. The way beam search works is by, at a given time, predicting m^n output sequences or beams. m is known as the beam width and n is the beam depth. Each output sequence (or a beam) is n bigrams predicted into the future. We compute the joint probability of each beam by multiplying individual prediction probabilities of the items in that beam. We then pick the beam with the highest joint probability as our output sequence for that given time step. Note that this is a greedy search, meaning that we will calculate the best candidates at each depth of the tree iteratively, as the tree grows. It should be noted that this search will not result in the globally best beam. *Figure 8.7* shows an example. We will indicate the best beam candidates (and their probabilities) with bold font and arrows:

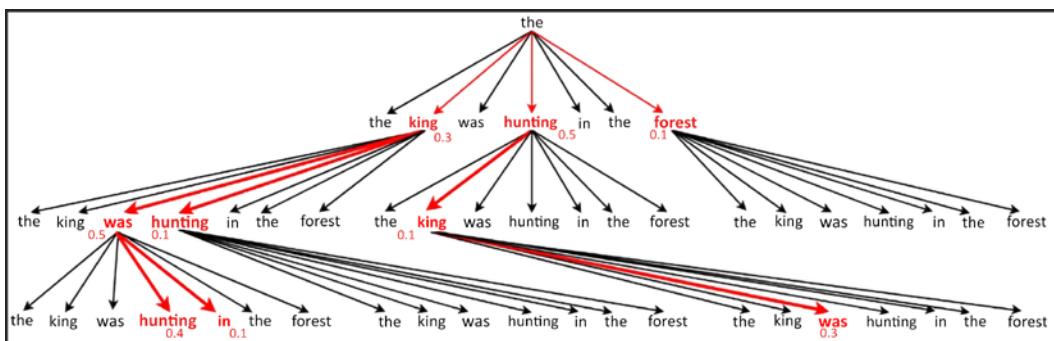


Figure 8.7: A beam search illustrating the requirement for updating beam states at each step. Each number underneath the word represents the probability of that word being chosen. For the words not in bold, you can assume the probabilities are negligible

We can see that in the first step, the word “*hunting*” has the highest probability. However, if we perform a beam search with a beam depth of 3, we get the sequence [“*king*”, “*was*”, “*hunting*”] with a joint probability of $0.3 * 0.5 * 0.4 = 0.06$ as the best beam.

This is higher than a beam that would start from the word “*hunting*” (which has a joint probability of $0.5 * 0.1 * 0.3 = 0.015$).

Implementing beam search

We implement beam search as a recursive function. But first we will implement a function that performs a single step of our recursive function called `beam_one_step()`. This function simply takes a model, an input, and states (from the LSTM) and produces the output and new states.

```
def beam_one_step(model, input_, states):
    """ Perform the model update and output for one step"""
    out = model.predict([input_, *states])
    output, new_states = out[0], out[1:]
    return output, new_states
```

Next, we write the main recursive function that performs beam search. This function takes the following arguments:

- `model` – An inference-based language model
- `input_` – The initial input
- `states` – The initial state vectors
- `beam_depth` – The search depth of the beam
- `beam_width` – The search width of the beam (i.e. number of candidates considered at a given depth)

Let's now discuss the function:

```
def beam_search(model, input_, states, beam_depth=5, beam_width=3):
    """ Defines an outer wrapper for the computational function of
    beam search """

    vocabulary =
        infer_model.get_layer("text_vectorization").get_vocabulary()
    index_word = dict(zip(range(len(vocabulary)), vocabulary))

    def recursive_fn(input_, states, sequence, log_prob, i):
        """ This function performs actual recursive computation of the
        long string"""


```

```
if i == beam_depth:  
    """ Base case: Terminate the beam search """  
    results.append((list(sequence), states, np.exp(log_prob)))  
    return sequence, log_prob, states  
  
else:  
    """ Recursive case: Keep computing the output using the  
    previous outputs"""  
    output, new_states = beam_one_step(model, input_, states)  
  
    # Get the top beam_width candidates for the given depth  
    top_probs, top_ids = tf.nn.top_k(output, k=beam_width)  
    top_probs, top_ids = top_probs.numpy().ravel(),  
    top_ids.numpy().ravel()  
  
    # For each candidate compute the next prediction  
    for p, wid in zip(top_probs, top_ids):  
        new_log_prob = log_prob + np.log(p)  
  
        # we are going to penalize joint probability whenever  
        # the same symbol is repeating  
        if len(sequence)>0 and wid == sequence[-1]:  
            new_log_prob = new_log_prob + np.log(1e-1)  
  
        sequence.append(wid)  
        _ = recursive_fn(np.array([[index_word[wid]]]),  
                        new_states, sequence, new_log_prob, i+1)  
        sequence.pop()  
  
    results = []  
    sequence = []  
    log_prob = 0.0  
    recursive_fn(input_, states, sequence, log_prob, 0)  
  
    results = sorted(results, key=lambda x: x[2], reverse=True)  
  
return results
```

The `beam_search()` function in fact defines a nested recursive function (`recursive_fn`) that accumulates the outputs as it is called and stores the results in a list called `results`. The `recursive_fn()` does the following. If the function has been called a number of times equal to the `beam_depth`, then it returns the current result. If the number of function calls hasn't reached the predefined depth, for a given depth index, then the `recursive_fn()`:

- Computes the new output and states using the `beam_one_step()` function
- Gets the IDs and probabilities of the top bigram candidates
- Computes the joint probability of each beam in the log space (in log space we get better numerical stability for smaller probability values)
- Finally, we call the same function with the new inputs, new state, and the next depth index

With that you can simply call the `beam_search()` function to get beams of predictions from the inference model. Let's look at how we can do that next.

Generating text with beam search

Here we will only show the part where we iteratively call `beam_search()` to generate new text. For the full code refer to `ch08_lstm_for_text_generation.ipynb`.

```
for i in range(50):
    print('.', end='')

# Get the results from beam search
result = beam_search(infer_model, x, states, 5, 5)

# Get one of the top 10 results based on their likelihood
n_probs = np.array([p for _, _, p in result[:10]])
p_j = np.random.choice(list(range(n_probs.size)),
p=n_probs/n_probs.sum())
best_beam_ids, states, _ = result[p_j]
x = np.array([[index_word[best_beam_ids[-1]]]])

text.extend([index_word[w] for w in best_beam_ids])
```

We simply call the function `beam_search()` with `infer_model`, current input `x`, current states `states`, `beam depth`, and `beam width`, and update `x` and `states` to reflect the winning beam. Then the model will iteratively use the winning beam to produce the next beam.

Let's see how our LSTM performs with beam search:

When adam and eve were driven out of paradise, they were compelled to build a house for themselves on barren groundr, said the king's daughter went out of the king's son to the king's daughter, and then the king's daughter went into the world, and asked the hedgehog's daughter that the king was about to the forest, and there was on the window, and said, "if you will give her that you have been and said, i will give him the king's daughter, but when she went to the king's sister, and when she was still before the window, and said to himself, and when he said to her father, and that he had nothing and said to hi

Here's what the standard LSTM with greedy sampling (i.e. predicting one word at a time) outputs:

When adam and eve were driven out of paradise, they were compelled to build a house for themselves on barren groundr, and then this they were all the third began to be able to the forests, and they were. the king's daughter was no one was about to the king's daughter to the forest of them to the stone. then the king's daughter was, and then the king's daughter was nothing-eyes, and the king's daughter was still, and then that had there was about through the third, and the king's daughters was seems to the king's daughter to the forest of them to the stone for them to the forests, and that it was not been to be ables, and the king's daughter wanted to be and said, ...

Compared to the text produced by the LSTM, this text seems to have more variation in it while keeping the text grammatically consistent as well. So, in fact, beam search helps to produce quality predictions compared to predicting one word at a time. But still, there are instances where words together don't make much sense. Let's see how we can improve our LSTM further.

Improving LSTMs – generating text with words instead of n-grams

Here we will discuss ways to improve LSTMs. We have so far used bigrams as our basic unit of text. But you would get better results by incorporating words, as opposed to bigrams. This is because using words reduces the overhead of the model by alleviating the need to learn to form words from bigrams. We will discuss how we can employ word vectors in the code to generate better-quality text compared to using bigrams.

The curse of dimensionality

One major limitation stopping us from using words instead of n-grams as the input to our LSTM is that this will drastically increase the number of parameters in our model. Let's understand this through an example. Consider that we have an input of size *500* and a cell state of size *100*. This would result in a total of approximately *240K* parameters (excluding the softmax layer), as shown here:

$$= \sim 4x(500 \times 100 + 100 \times 100 + 100) = \sim 240k$$

Let's now increase the size of the input to *1000*. Now the total number of parameters would be approximately *440K*, as shown here:

$$= \sim 4x(1000 \times 100 + 100 \times 100 + 100) = \sim 440k$$

As you can see, for an increase of 500 units of the input dimensionality, the number of parameters has grown by 200,000. This not only increases the computational complexity but also increases the risk of overfitting due to the large number of parameters. So, we need ways of restricting the dimensionality of the input.

Word2vec to the rescue

As you will remember, not only can Word2vec give a lower-dimensional feature representation of words compared to one-hot encoding, but it also gives semantically sound features. To understand this, let's consider three words: *cat*, *dog*, and *volcano*. If we one-hot encode just these words and calculate the Euclidean distance between them, it would be the following:

$$\text{distance}(\text{cat}, \text{volcano}) = \text{distance}(\text{cat}, \text{dog})$$

However, if we learn word embeddings, it would be the following:

$$\text{distance}(\text{cat}, \text{volcano}) > \text{distance}(\text{cat}, \text{dog})$$

We would like our features to represent the latter, where similar things have a lower distance than dissimilar things. Consequently, the model will be able to generate better-quality text.

Generating text with Word2vec

The structure of the model remains more or less the same as what we have discussed. It is only the units of text we would consider that changes.

Figure 8.8 depicts the overall architecture of LSTM-Word2vec:

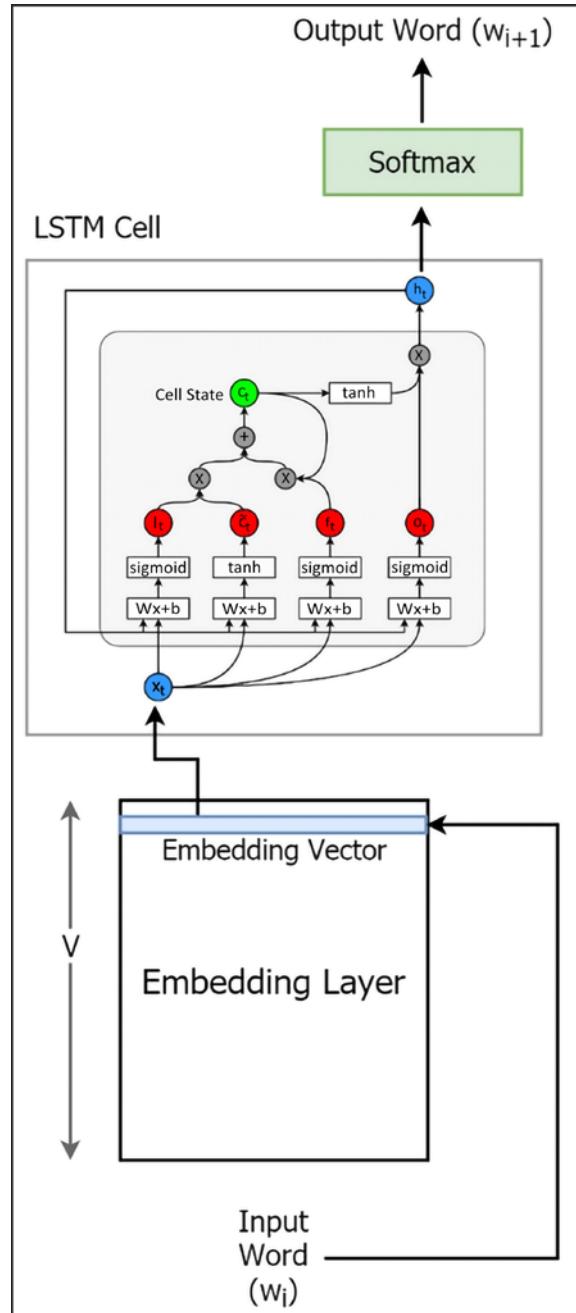


Figure 8.8: The structure of a language modeling LSTM using word vectors

You have a few options when it comes to using word vectors. You can either:

- Randomly initialize the vectors and jointly learn them during the task
- Train the embeddings using a word vector algorithm (e.g. Word2vec, GloVe, etc.) beforehand
- Use pretrained word vectors freely available to download, to initialize the embedding layer

Note

Below we list a few freely available pretrained word vectors. Word vectors found by learning from a text corpus with billions of words are freely available to be downloaded and used:



- **Word2vec:** <https://code.google.com/archive/p/word2vec/>
- **Pretrained GloVe word vectors:** <https://nlp.stanford.edu/projects/glove/>
- **fastText word vectors:** <https://github.com/facebookresearch/fastText>

We end our discussion on language modeling here.

Summary

In this chapter, we looked at the implementations of the LSTM algorithm and other various important aspects to improve LSTMs beyond standard performance. As an exercise, we trained our LSTM on the text of stories by the Grimm brothers and asked the LSTM to output a fresh new story. We discussed how to implement an LSTM model with code examples extracted from exercises.

Next, we had a technical discussion about how to implement LSTMs with peepholes and GRUs. Then we did a performance comparison between a standard LSTM and its variants. We saw that the GRUs performed the best compared to LSTMs with peepholes and LSTMs.

Then we discussed some of the various improvements possible for enhancing the quality of outputs generated by an LSTM. The first improvement was beam search. We looked at an implementation of beam search and covered how to implement it step by step. Then we looked at how we can use word embeddings to teach our LSTM to output better text.

In conclusion, LSTMs are very powerful machine learning models that can capture both long-term and short-term dependencies.

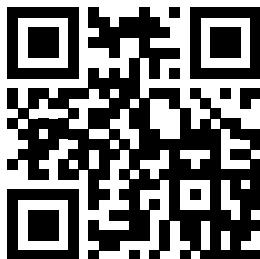
Moreover, beam search in fact helps to produce more realistic-looking textual phrases compared to predicting one at a time.

In the next chapter, we will look at how sequential models can be used to solve a more complex type of problem known as sequence-to-sequence problems. Specifically, we will look at how we can perform machine translation by formulating it as a sequence-to-sequence problem.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside
more than 1000 members at: <https://packt.link/nlp>



9

Sequence-to-Sequence Learning – Neural Machine Translation

Sequence-to-sequence learning is the term used for tasks that require mapping an arbitrary-length sequence to another arbitrary-length sequence. This is one of the most sophisticated tasks in NLP, which involves learning many-to-many mappings. Examples of this task include **Neural Machine Translation (NMT)** and creating chatbots. NMT is where we translate a sentence from one language (source language) to another (target language). Google Translate is an example of an NMT system. Chatbots (that is, software that can communicate with/answer a person) are able to converse with humans in a realistic manner. This is especially useful for various service providers, as chatbots can be used to find answers to easily solvable questions that customers might have, instead of redirecting them to human operators.

In this chapter, we will learn how to implement an NMT system. However, before diving directly into such recent advances, we will first briefly visit some **Statistical Machine Translation (SMT)** methods, which preceded NMT and were the state-of-the-art systems until NMT caught up. Next, we will walk through the steps required for building an NMT. Finally, we will learn how to implement a real NMT system that translates from German to English, step by step.

Specifically, this chapter will cover the following main topics:

- Machine translation
- A brief historical tour of machine translation
- Understanding neural machine translation
- Preparing data for the NMT system

- Defining the model
- Training the NMT
- The BLEU score – evaluating the machine translation systems
- Visualizing attention patterns
- Inference with NMT
- Other applications of Seq2Seq models – chatbots

Machine translation

Humans often communicate with each other by means of a language, compared to other communication methods (for example, gesturing). Currently, more than 6,000 languages are spoken worldwide. Furthermore, learning a language to a level where it is easily understandable to a native speaker of that language is a difficult task to master. However, communication is essential for sharing knowledge, socializing, and expanding your network. Therefore, language acts as a barrier to communicating with people in different parts of the world. This is where **Machine Translation (MT)** comes in. MT systems allow the user to input a sentence in their own tongue (known as the source language) and output a sentence in a desired target language.

The problem with MT can be formulated as follows. Say we are given a sentence (or a sequence of words) W_s belonging to a source language S , defined by the following:

$$W_s = \{w_1, w_2, w_3, \dots, w_L\}$$

Here, $W_s \in S$.

The source language would be translated to a sentence W_T , where T is the target language and is given by the following:

$$W_T = \{w'_1, w'_2, w'_3, \dots, w'_M\}$$

Here, $W_T \in T$.

W_T is obtained through the MT system, which outputs the following:

$$p(W_T | W_s) \quad \forall W_T \in W_T^*$$

Here, W_T^* is the pool of possible translation candidates found by the algorithm for the source sentence. Also, the best candidate from the pool of candidates is given by the following equation:

$$W_T^{best} = argmax_{W_T \in W_T^*} (p(W_T | W_s); \theta)$$

Here, θ is the model parameters. During training, we optimize the model to maximize the probability of some known target translations for a set of corresponding source translations (that is, training data).

So far, we have discussed the formal setup of the language translation problem that we're interested in solving. Next, we will walk through the history of MT to get a feel of how people tried solving this in the early days.

A brief historical tour of machine translation

Here, we will discuss the history of MT. The inception of MT involved rule-based systems. Then, more statistically sound MT systems emerged. **Statistical Machine Translation (SMT)** used various measures of statistics of a language to produce translations to another language. Then came the era of NMT. NMT currently holds state-of-the-art performance in most machine learning tasks compared with other methods.

Rule-based translation

NMT came long after statistical machine learning, and statistical machine learning has been around for more than half a century now. The inception of SMT methods dates back to 1950-60, when during one of the first recorded projects, the *Georgetown-IBM experiment*, more than 60 Russian sentences were translated to English. To give some perspective, this attempt is almost as old as the invention of the transistor.

One of the initial techniques for MT was word-based machine translation. This system performed word-to-word translations using bilingual dictionaries. However, as you can imagine, this method has serious limitations. The obvious limitation is that word-to-word translation is not a one-to-one mapping between different languages. In addition, word-to-word translation may lead to incorrect results as it does not consider the context of a given word. The translation of a given word in the source language can change depending on the context in which it is used. To understand this with a concrete example, let's look at the translation example from English to French in *Figure 9.1*. You can see that in the given two English sentences, a single word changes. However, this creates drastic changes in the translation:

She loves cats	She loves him
Elle aime les chats	Elle l'aime

Figure 9.1: Translations (English to French) between languages are not one-to-one mappings between words

In the 1960s, the **Automatic Language Processing Advisory Committee (ALPAC)** released a report, *Languages and machines: computers in translation and linguistics, National Academy of the Sciences (1966)*, on MT's prospects. The conclusion was this:

There is no immediate or predictable prospect of useful machine translation.

This was because MT was slower, less accurate, and more expensive than human translation at the time. This delivered a huge blow to MT advancements, and almost a decade passed in silence.

Next came corpora-based MT, where an algorithm was trained using tuples of source sentences, and the corresponding target sentence was obtained through a parallel corpus, that is, the parallel corpus would be of the format [$(\langle \text{source_sentence_1} \rangle, \langle \text{target_sentence_1} \rangle), (\langle \text{source_sentence_2} \rangle, \langle \text{target_sentence_2} \rangle), \dots]$]. The parallel corpus is a large text corpus formed as tuples, consisting of text from the source language and the corresponding translation of that text. An illustration of this is shown in *Table 9.1*. It should be noted that building a parallel corpus is much easier than building bilingual dictionaries and they are more accurate because the training data is richer than word-to-word training data. Furthermore, instead of directly relying on manually created bilingual dictionaries, a bilingual dictionary (that is, the transition models) of two languages can be built using the parallel corpus. A transition model shows how likely a target word/phrase is to be the correct translation, given the current source word/phrase. In addition to learning the transition model, corpora-based MT also learns the word alignment models. A word alignment model can represent how words in a phrase from the source language correspond to the translation of that phrase. An example of parallel corpora and a word alignment model is depicted in *Figure 9.2*:

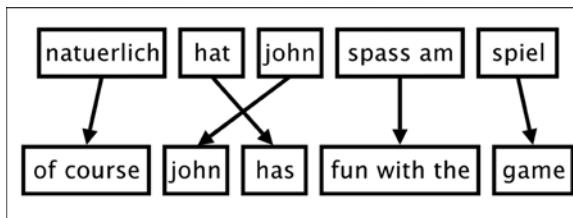


Figure 9.2: Word alignment between two different languages

An illustration of an example parallel corpora is shown in *Table 9.1*:

Source language sentences (English)	Target language sentences (French)
I went home	Je suis allé à la maison
John likes to play guitar	John aime jouer de la guitare
He is from England	Il est d'Angleterre
...	...

Table 9.1: Parallel corpora for English and French sentences

Another approach was interlingual machine translation, which involved translating the source sentence into a language-neutral **interlingua** (that is, a metalanguage), and then generating the translated sentence out of the interlingua. More specifically, an interlingual machine translation system consists of two important components, an analyzer and a synthesizer. The analyzer will take the source sentence and identify agents (for example, nouns), actions (for example, verbs), and so on, and also how they interact with each other. Next, these identified elements are represented by means of an interlingual lexicon. An example of an interlingual lexicon can be made with the **synsets** (that is, the group of synonyms sharing a common meaning) available in WordNet. Then, from this interlingual representation, the synthesizer will create the translation. Since the synthesizer knows the nouns, verbs, and so on through the interlingual representation, it can generate the translation in the target language by incorporating language-specific grammar rules.

Statistical Machine Translation (SMT)

Next, more statistically sound systems started emerging. One of the pioneering models of this era was IBM Models 1-5, which did word-based translation. However, as we discussed earlier, word translations do not match one to one from the source language to a target language (for example, compound words and morphology). Eventually, researchers started experimenting with phrase-based translation systems, which made some notable advances in machine translation.

Phrase-based translation works in a similar way to word-based translation, except that it uses phrases of a language as the atomic units of translation instead of individual words. This is a more sensible approach as it makes modeling the one-to-many, many-to-one, or many-to-many relationships between words easier. The main goal of phrase-based translation is to learn a phrase-translation model that contains a probability distribution of different candidate target phrases for a given source phrase. As you can imagine, this method involves maintaining huge databases of various phrases in two languages. A reordering step for phrases is also performed as there is no monotonic ordering of words between a sentence from one language and one in another.

An example of this is shown in *Figure 9.2*; if the words were monotonically ordered between languages, there would not be crosses between word mappings.

One of the limitations of this approach is that the decoding process (finding the best target phrase for a given source phrase) is expensive. This is due to the size of the phrase database, as well as the fact that a source phrase often contains multiple target language phrases. To alleviate the burden, syntax-based translations arose.

In syntax-based translation, the source sentence is represented by a syntax tree. In *Figure 9.3*, NP represents a noun phrase, VP a verb phrase, and S a sentence. Then a **reordering phase** takes place, where the tree nodes are reordered to change the order of subject, verb, and object, depending on the target language. This is because the sentence structure can change depending on the language (for example, in English it is *subject-verb-object*, whereas in Japanese it is *subject-object-verb*). The reordering is decided according to something known as the **r-table**. The r-table contains the likelihood probabilities for the tree nodes to be changed to some other order:

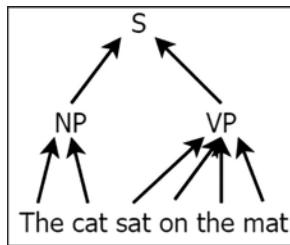


Figure 9.3: Syntax tree for a sentence

An **insertion phase** then takes place. In the insertion phase, we stochastically insert a word into each node of the tree. This is due to the assumption that there is an invisible NULL word, and it generates target words at random positions of the tree. Also, the probability of inserting a word is determined by something called the **n-table**, which is a table that contains probabilities of inserting a particular word into the tree.

Next, the **translation phase** occurs, where each leaf node is translated to the target word in a word-by-word manner. Finally, the translated sentence is read off the syntax tree, to construct the target sentence.

Neural Machine Translation (NMT)

Finally, around the year 2014, NMT systems were introduced. NMT is an end-to-end system that takes a full sentence as an input, performs certain transformations, and then outputs the translated sentence for the corresponding source sentence.

Therefore, NMT eliminates the need for the feature engineering required for machine translation, such as building phrase translation models and building syntax trees, which is a big win for the NLP community. Also, NMT has outperformed all the other popular MT techniques in a very short period, just two to three years. In *Figure 9.4*, we depict the results of various MT systems reported in the MT literature. For example, 2016 results are obtained from Sennrich and others in their paper *Edinburgh Neural Machine Translation Systems for WMT 16*, *Association for Computational Linguistics, Proceedings of the First Conference on Machine Translation, August 2016*: 371-376, and from Williams and others in their paper *Edinburgh's Statistical Machine Translation Systems for WMT16*, *Association for Computational Linguistics, Proceedings of the First Conference on Machine Translation, August 2016*: 399-410. All the MT systems are evaluated with the BLEU score. The BLEU score denotes the number of n-grams (for example, unigrams and bigrams) of candidate translation that matched in the reference translation. So the higher the BLEU score, the better the MT system. We'll discuss the BLEU metric in detail later in the chapter. There is no need to highlight that NMT is a clear-cut winner:

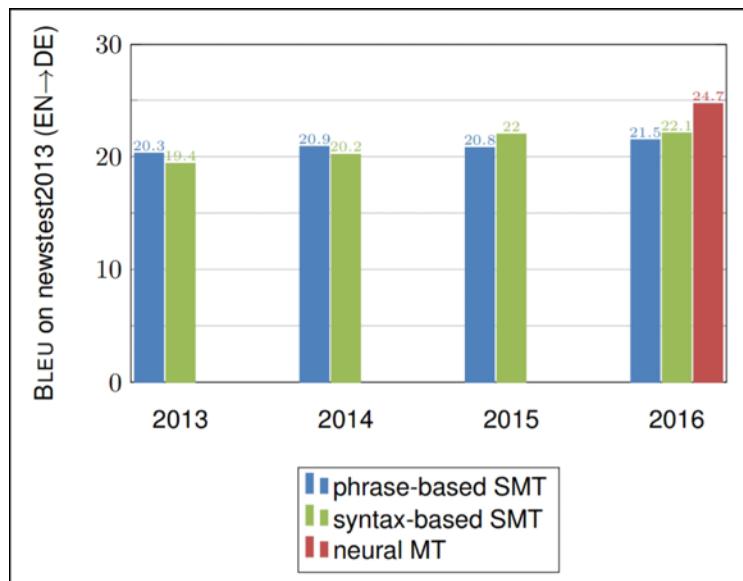


Figure 9.4: Comparison of statistical machine translation system to NMT systems. Courtesy of Rico Sennrich

A case study assessing the potential of NMT systems is available in *Is Neural Machine Translation Ready for Deployment? A Case Study on 30 Translation Directions*, Junczys-Dowmunt, Hoang and Dwojak, *Proceedings of the Ninth International Workshop on Spoken Language Translation, Seattle* (2016).

The study looks at the performance of different systems on several translation tasks between various languages (English, Arabic, French, Russian, and Chinese). The results also support that NMT systems (NMT 1.2M and NMT 2.4M) perform better than SMT systems (PB-SMT and Hiero).

Figure 9.5 shows several statistics for a set from a 2017 state-of-the-art machine translator. This is from a presentation, *State of the Machine Translation, Intento, Inc, 2017*, produced by Konstantin Savenkov, cofounder and CEO of Intento. We can see that the performance of the MT produced by DeepL (<https://www.deepl.com>) appears to be competing closely with other MT giants, including Google. The comparison includes MT systems such as DeepL (NMT), Google (NMT), Yandex (NMT-SMT hybrid), Microsoft (has both SMT and NMT), IBM (SMT), Prompt (rule-based), and SYSTRAN (rule-based/SMT hybrid). The graph clearly shows that NMT systems are leading the current MT advancements. The LEPOR score is used to assess different systems. LEPOUR is a more advanced metric than BLEU, and it attempts to solve the **language bias problem**. The language bias problem refers to the phenomenon that some evaluation metrics (such as BLEU) perform well for certain languages, but perform poorly for others.

However, it should also be noted that the results do contain some bias due to the averaging mechanism used in this comparison. For example, Google Translate has been averaged over a larger set of languages (including difficult translation tasks), whereas DeepL has been averaged over a smaller and relatively easier subset of languages. Therefore, we should not conclude that the DeepL MT system is always better than the Google MT system. Nevertheless, the overall results provide a general comparison of the performance of the current NMT and SMT systems:

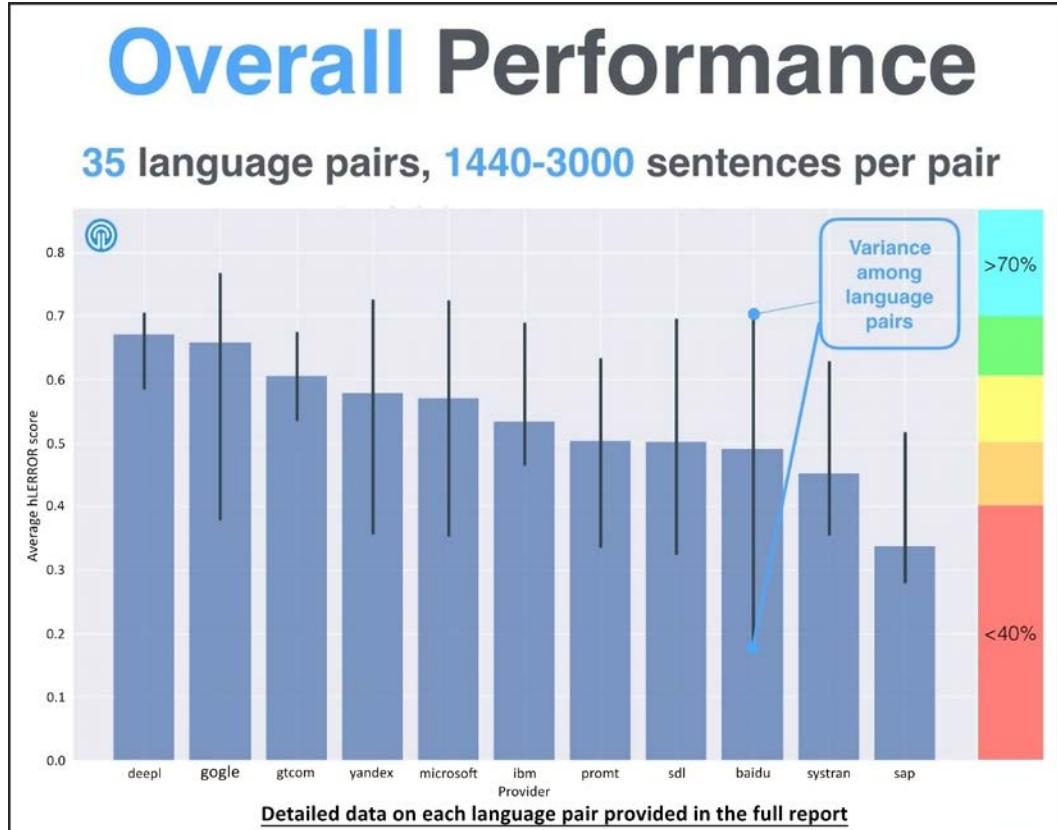


Figure 9.5: Performance of various MT systems. Courtesy of Intento, Inc.

We saw that NMT has already outperformed SMT systems in very few years, and is the current state of the art. We will now move on to discussing the details and architecture of an NMT system. Finally, we will be implementing an NMT system from scratch.

Understanding neural machine translation

Now that we have an appreciation for how machine translation has evolved over time, let's try to understand how state-of-the-art NMT works. First, we will take a look at the model architecture used by neural machine translators and then move on to understanding the actual training algorithm.

Intuition behind NMT systems

First, let's understand the intuition underlying an NMT system's design. Say you are a fluent English and German speaker and were asked to translate the following sentence into German:

I went home

This sentence translates to the following:

Ich ging nach Hause

Although it might not have taken more than a few seconds for a fluent person to translate this, there is a certain process that produces the translation. First, you read the English sentence, and then you create a thought or concept about what this sentence represents or implies, in your mind. And finally, you translate the sentence into German. The same idea is used for building NMT systems (see *Figure 9.6*). The encoder reads the source sentence (that is, similar to you reading the English sentence). Then the encoder outputs a context vector (the context vector corresponds to the thought/concept you imagined after reading the sentence). Finally, the decoder takes in the context vectors and outputs the translation in German:

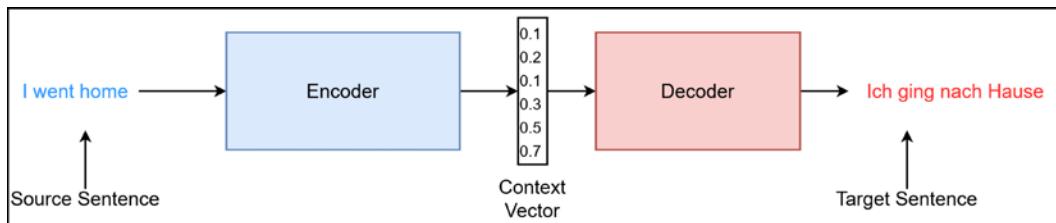


Figure 9.6: Conceptual architecture of an NMT system

NMT architecture

Now we will look at the architecture in more detail. The sequence-to-sequence approach was originally proposed by Sutskever, Vinyals, and Le in their paper *Sequence to Sequence Learning with Neural Networks, Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2: 3104-3112*.

From the diagram in *Figure 9.6*, we can see that there are two major components in the NMT architecture. These are called the encoder and decoder. In other words, NMT can be seen as an encoder-decoder architecture. The **encoder** converts a sentence from a given source language into a thought vector (i.e. a contextualized representation), and the **decoder** decodes or translates the thought into a target language. As you can see, this shares some features with the interlingual machine translation method we briefly talked about. This explanation is illustrated in *Figure 9.7*. The left-hand side of the context vector denotes the encoder (which takes a source sentence word by word to train a time-series model). The right-hand side denotes the decoder, which outputs word by word (while using the previous word as the current input) the corresponding translation of the source sentence. We will also use embedding layers (for both the source and target languages) where the semantics of the individual tokens will be learned and fed as inputs to the models:

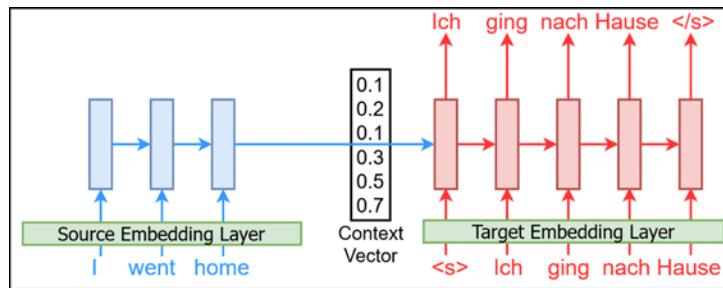


Figure 9.7: Unrolling the source and target sentences over time

With a basic understanding of what NMT looks like, let's formally define the objective of the NMT. The ultimate objective of an NMT system is to maximize the log likelihood, given a source sentence x_s and its corresponding y_t . That is, to maximize the following:

$$\frac{1}{N} \sum_{i=1}^N \log P(y_T | x_s)$$

Here, N refers to the number of source and target sentence inputs we have as training data.

Then, during inference, for a given source sentence, x_s^{infer} , we will find the y_T^{best} translation using the following:

$$Y_T^{best} = argmax_{y \in Y_T} P(y_T | x_s^{infer}) = argmax_{y \in Y_T} \prod_{i=1}^M P(y_T^i | x_s^{infer})$$

Here, y_T^i is the predicted token at the i^{th} time step and Y_T is the set of possible candidate sentences.

Before we examine each part of the NMT architecture, let's define the mathematical notation to understand the system more concretely. As our sequential model, we will choose a **Gated Recurrent Unit (GRU)**, as it is simpler than an LSTM and performs comparatively well.

Let's define the encoder GRU as GRU_{enc} and the decoder GRU as GRU_{dec} . At the time step t , let's define the output state of a general GRU as h_t . That is, feeding the input x_t into the GRU produces h_t :

$$h_t = \text{GRU}(x_t | x_1, x_2, \dots, x_{t-1})$$

Now, we will talk about the embedding layer, the encoder, the context vector, and finally, the decoder.

The embedding layer

We have already seen the power of word embeddings. Here, we can also leverage embeddings to improve model performance. We will be using two-word embedding layers, Emb_s , for the source language and Emb_t for the target language. So, instead of feeding x_t directly into the GRU, we will be getting $\text{Emb}(x_t)$. However, to avoid excessive notation, we will assume $x_t = \text{Emb}(x_t)$.

The encoder

As mentioned earlier, the encoder is responsible for generating a thought vector or a context vector that represents what is meant by the source language. For this, we will use a GRU-based network (see *Figure 9.8*):

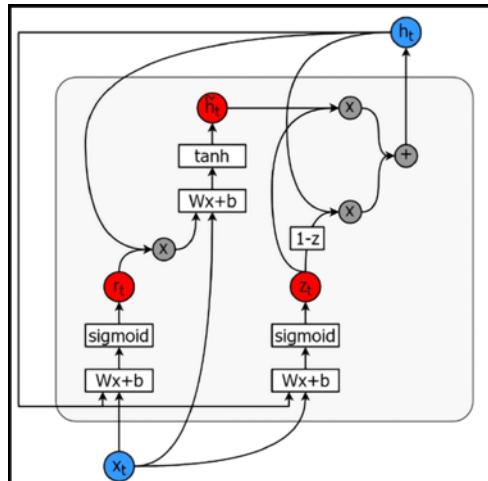


Figure 9.8: A GRU cell

The encoder is initialized with h at time step 0 (h_0) with a zero vector by default. The encoder takes a sequence of words, $x_s = \{x_s^1, x_s^2, \dots, x_s^L\}$, as the input and calculates a context vector, $v = h_L$, where v is the final external hidden state obtained after processing the final element x_s^L , of the sequence x_s . We represent this as the following:

$$h_L = \text{GRU}_{enc}(x_s^L | x_s^1, x_s^2, \dots, x_s^{L-1})$$

$$v = h_L$$

The context vector

The idea of the context vector (v) is to represent a sentence of a source language concisely. Also, in contrast to how the encoder's state is initialized (that is, it is initialized with zeros), the context vector becomes the initial state for the decoder GRU. In other words, the decoder GRU doesn't start with an initial state of zeros, but with the context vector as its initial state. This creates a linkage between the encoder and the decoder and makes the whole model end-to-end differentiable. We will talk about this in more detail next.

The decoder

The decoder is responsible for decoding the context vector into the desired translation. Our decoder is an RNN as well. Though it is possible for the encoder and decoder to share the same set of weights, it is usually better to use two different networks for the encoder and the decoder. This increases the number of parameters in our model, allowing us to learn the translations more effectively.

First, the decoder's states are initialized with the context vector, i.e. $v = h_L$, as shown here: $h_0 = v$.

Here, h_0 is the initial state vector of the decoder (GRU_{dec}).

This (v) is the crucial link that connects the encoder with the decoder to form an end-to-end computational chain (see in *Figure 9.6* that the only thing shared by the encoder and decoder is v). Also, this is the only piece of information that is available to the decoder about the source sentence.

Then we will compute the m^{th} prediction of the translated sentence with the following:

$$h_m = \text{GRU}_{dec}(y_T^{m-1} | v, y_T^1, y_T^2, \dots, y_T^{m-2})$$

$$y_T^m = \text{softmax}(w_{\text{softmax}} h_m + b_{\text{softmax}})$$

The full NMT system with the details of how the GRU cell in the encoder connects to the GRU cell in the decoder, and how the softmax layer is used to output predictions, is shown in *Figure 9.9*:

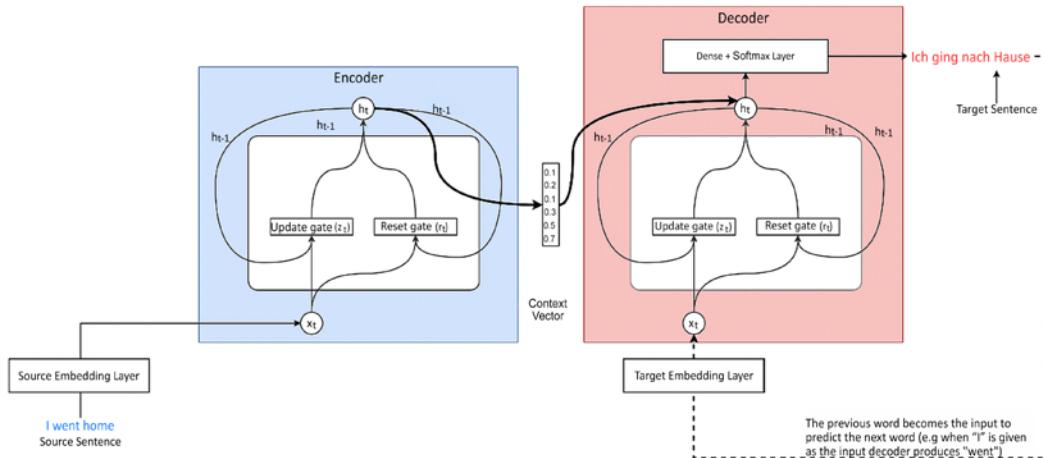


Figure 9.9: The encoder-decoder architecture with the GRUs. Both the encoder and the decoder have a separate GRU component. Additionally, the decoder has a fully-connected (dense) layer and a softmax layer that produce the final predictions.

In the next section, we will go through the steps required to prepare data for our model.

Preparing data for the NMT system

In this section, we will understand the data and learn about the process for preparing data for training and predicting from the NMT system. First, we will talk about how to prepare training data (that is, the source sentence and target sentence pairs) to train the NMT system, followed by inputting a given source sentence to produce the translation of the source sentence.

The dataset

The dataset we'll be using for this chapter is the WMT-14 English-German translation data from <https://nlp.stanford.edu/projects/nmt/>. There are ~4.5 million sentence pairs available. However, we will use only 250,000 sentence pairs due to computational feasibility. The vocabulary consists of the 50,000 most common English words and the 50,000 most common German words, and words not found in the vocabulary will be replaced with a special token, <unk>. You will need to download the following files:

- **train.de** – File containing German sentences
- **train.en** – File containing English sentences

- vocab.50K.de – File containing German vocabulary
- vocab.50K.en – File containing English vocabulary

train.de and train.en contain parallel sentences in German and English, respectively. Once downloaded we will load the sentences as follows:

```
n_sentences = 250000

# Loading English sentences
original_en_sentences = []
with open(os.path.join('data', 'train.en'), 'r', encoding='utf-8') as en_file:
    for i, row in enumerate(en_file):
        if i >= n_sentences: break
        original_en_sentences.append(row.strip().split(" "))

# Loading German sentences
original_de_sentences = []
with open(os.path.join('data', 'train.de'), 'r', encoding='utf-8') as de_file:
    for i, row in enumerate(de_file):
        if i >= n_sentences: break
        original_de_sentences.append(row.strip().split(" "))
```

If you print the data you just loaded, for the two languages, you would have sentences like the following:

```
English: a fire restant repair cement for fire places , ovens , open
fireplaces etc .
German: feuerfester Reparaturkitt für Feuerungsanlagen , Öfen , offene
Feuerstellen etc .

English: Construction and repair of highways and ...
German: Der Bau und die Reparatur der Autostraßen ...

English: An announcement must be commercial character .
German: die Mitteilungen sollen den geschäftlichen kommerziellen Charakter
tragen .
```

Adding special tokens

The next step is to add a few special tokens to the start and end of our sentences. We will add <s> to mark the start of a sentence and </s> to mark the end of a sentence. We can easily achieve this using the following list comprehension:

```
en_sentences = [[<s>]+sent+[</s>] for sent in original_en_sentences]
de_sentences = [[<s>]+sent+[</s>] for sent in original_de_sentences]
```

This will give us:

```
English: <s> a fire restant repair cement for fire places , ovens , open
fireplaces etc . </s>
German: <s> feuerfester Reparaturkitt für Feuerungsanlagen , Öfen , offene
Feuerstellen etc. </s>

English: <s> Construction and repair of highways and ... </s>
German: <s> Der Bau und die Reparatur der Autostraßen ... </s>

English: <s> An announcement must be commercial character . </s>
German: <s> die Mitteilungen sollen den geschäftlichen kommerziellen
Charakter tragen . </s>
```

This is a very important step for Seq2Seq models. <s> and </s> tokens serve an extremely important role during model inference. As you will see, at inference time, we will be using the decoder to predict one word at a time, by using the output of the previous time step as an input. This way we can predict for an arbitrary number of time steps. Using <s> as the starting token gives us a way to signal to the decoder that it should start predicting tokens from the target language. Next, if we do not use the </s> token to mark the end of a sentence, we cannot signal the decoder to end a sentence. This can lead the model to enter an infinite loop of predictions.

Splitting training, validation, and testing datasets

We need to split our dataset into three parts: a training set, a validation set, and a testing set. Specifically, let's use 80% of sentences to train the model, 10% as validation data, and the remaining 10% as testing data:

```
from sklearn.model_selection import train_test_split

train_en_sentences, valid_test_en_sentences, train_de_sentences, valid_
test_de_sentences = train_test_split(
```

```
    np.array(en_sentences), np.array(de_sentences), test_size=0.2
)

valid_en_sentences, valid_de_sentences, test_en_sentences, test_de_
sentences = train_test_split(
    valid_test_en_sentences, valid_test_de_sentences, test_size=0.5)
```

Defining sequence lengths for the two languages

A key statistic we have to understand at this point is how long, generally, the sentences in our corpus are. It is quite likely that the two languages will have different sentence lengths. To learn the statistics of this, we'll be using the pandas library in the following way:

```
pd.Series(train_en_sentences).str.len().describe(percentiles=[0.05, 0.5,
0.95])
```

Here, we are first converting the `train_en_sentences` to a `pd.Series` object. `pd.Series` is an indexed series of values (an array). Here, each value is a list of tokens belonging to each sentence. Calling `.str.len()` will give us the length of each list of tokens. Finally, the `describe` method will give important statistics such as mean, standard deviation, and percentiles. Here, we are specifically asking for 5%, 50%, and 95% percentiles.

Note that we are only using the training data for this calculation. If you include validation or test datasets in these calculations, we may be leaking data about validation and test data. Therefore, it's best to only use the training dataset for these calculations.

The result from the previous code gives us:

```
Sequence lengths (English)
count    40000.000000
mean      25.162625
std       13.857748
min       6.000000
5%        9.000000
50%       22.000000
95%       53.000000
max      100.000000
dtype: float64
```

We can get the same for the German sentences the following way:

```
pd.Series(train_de_sentences).str.len().describe(percentiles=[0.05, 0.5, 0.95])
```

This gives us:

```
Sequence lengths (German)
count    40000.000000
mean      22.882550
std       12.574325
min       6.000000
5%        9.000000
50%       20.000000
95%       47.000000
max      100.000000
dtype: float64
```

Here we can see that 95% of English sentences have 53 tokens, where 95% of German sentences have 47 tokens.

Padding the sentences

Next, we need to pad our sentences. For this, we will use the `pad_sequences()` function provided in Keras. This function takes in values for the following arguments:

- `sequences` – A list of strings/IDs representing the text corpus. Each document can either be a list of strings or a list of integers
- `maxlen` – The maximum length to pad for (defaults to `None`)
- `dtype` – The type of data (defaults to '`int32`')
- `padding` – The side to pad short sequences (defaults to '`'pre'`')
- `truncating` – The side to truncate long sequences from (defaults to '`'pre'`')

- `value` – The values to pad with (defaults to 0.0)

We will use this function as follows:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

train_en_sentences_padded = pad_sequences(train_en_sentences, maxlen=n_
en_seq_length, value=unk_token, dtype=object, truncating='post',
padding='post')
valid_en_sentences_padded = pad_sequences(valid_en_sentences, maxlen=n_
en_seq_length, value=unk_token, dtype=object, truncating='post',
padding='post')
test_en_sentences_padded = pad_sequences(test_en_sentences, maxlen=n_
en_seq_length, value=unk_token, dtype=object, truncating='post',
padding='post')

train_de_sentences_padded = pad_sequences(train_de_sentences, maxlen=n_
de_seq_length, value=unk_token, dtype=object, truncating='post',
padding='post')
valid_de_sentences_padded = pad_sequences(valid_de_sentences, maxlen=n_
de_seq_length, value=unk_token, dtype=object, truncating='post',
padding='post')
test_de_sentences_padded = pad_sequences(test_de_sentences, maxlen=n_
de_seq_length, value=unk_token, dtype=object, truncating='post',
padding='post')
```

We are padding all of the training, validation, and test sentences in both English and German. We will use the recently found sequence lengths as the padding/truncating length.

Reversing the source sentence

We can also perform a special trick on the source sentences. Say we have the sentence ABC in the source language, which we want to translate to $\alpha\beta\gamma\phi$ in the target language. We will first reverse the source sentences so that the sentence ABC is read as CBA . This means that in order to translate ABC to $\alpha\beta\gamma\phi$, we need to feed in CBA . This improves the performance of our model significantly, especially when the source and target languages share the same sentence structure (for example, *subject-verb-object*).

Let's try to understand why this helps. Mainly, it helps to build good *communication* between the encoder and the decoder. Let's start from the previous example. We will concatenate the source and target sentences:

$ABC\alpha\beta\gamma$



If you calculate the distance (that is, the number of words separating two words) from A to α or B to β , they will be the same. However, consider this when you reverse the source sentence, as shown here:

$CBA\alpha\beta\gamma\phi$

Here, A is very close to α and so on. Also, to build good translations, building good communications at the very start is important. This simple trick can possibly help NMT systems to improve their performance.

Note that the source sentence reversing step is a subjective preprocessing step. This might not be necessary for some translational tasks. For example, if your translation task is to translate from Japanese (which is often written in *subject-object-verb* format) to Filipino (often written *verb-subject-object*), then reversing the source sentence might actually cause harm rather than helping. This is because by reversing the text in Japanese, you are increasing the distance between the starting element of the target sentence (that is, the verb (Japanese)) and the corresponding source language entity (that is, the verb (Filipino)).

Next let's define our encoder-decoder model.

Defining the model

In this section, we will define the model from end to end.

We are going to implement an encoder-decoder based NMT model equipped with additional techniques to boost performance. Let's start off by converting our string tokens to IDs.

Converting tokens to IDs

Before we jump to the model, we have one more text processing operation remaining, that is, converting the processed text tokens into numerical IDs. We are going to use a `tf.keras.layers.Layer` to do this. Particularly, we'll be using the `StringLookup` layer to create a layer in our model that converts each token into a numerical ID. As the first step, let us load the vocabulary files provided in the data. Before doing so, we will define the variable `n_vocab` to denote the size of the vocabulary for each language:

```
n_vocab = 25000 + 1
```

Originally, each vocabulary contains 50,000 tokens. However, we'll take only half of this to reduce the memory requirement. Note that we allow one extra token as there's a special token `<unk>` to denote **out-of-vocabulary (OOV)** words. With a 50,000-token vocabulary, it is quite easy to run out of memory due to the size of the final prediction layer we'll build. While cutting back on the size of the vocabulary, we have to make sure that we preserve the most common 25,000 words. Fortunately, each vocabulary file is organized such that words are ordered by their frequency of occurrence (high to low). Therefore, we just need to read the first 25,001 lines from the file:

```
en_vocabulary = []
with open(os.path.join('data', 'vocab.50K.en'), 'r', encoding='utf-8') as en_file:
    for ri, row in enumerate(en_file):
        if ri >= n_vocab: break
    en_vocabulary.append(row.strip())
```

Then we do the same for the German vocabulary:

```
de_vocabulary = []
with open(os.path.join('data', 'vocab.50K.de'), 'r', encoding='utf-8') as de_file:
    for ri, row in enumerate(de_file):
        if ri >= n_vocab: break
    de_vocabulary.append(row.strip())
```

Each of the vocabularies contain the special OOV token <unk> as the first line. We'll pop that out of the en_vocabulary and de_vocabulary lists as we need this for the next step:

```
en_unk_token = en_vocabulary.pop(0)
de_unk_token = de_vocabulary.pop(0)
```

Here's how we can define our English StringLookup layer:

```
en_lookup_layer = tf.keras.layers.StringLookup(
    vocabulary=en_vocabulary, oov_token=en_unk_token,
    mask_token=pad_token, pad_to_max_tokens=False
)
```

Let's understand the arguments provided to this layer:

- **vocabulary** – Contains a list of words that are found in the corpus (except certain special tokens that will be discussed below)
- **oov_token** – A special out-of-vocabulary token that will be used to replace tokens not listed in the vocabulary
- **mask_token** – A special token that will be used to mask inputs (e.g. uninformative padded tokens)
- **pad_to_max_tokens** – If padding should occur to bring arbitrary-length sequences in a batch of data to the same length

Similarly, we define a lookup layer for the German language:

```
de_lookup_layer = tf.keras.layers.StringLookup(
    vocabulary=de_vocabulary, oov_token=de_unk_token,
    mask_token=pad_token, pad_to_max_tokens=False
)
```

With the groundwork laid out, we can start building the encoder.

Defining the encoder

We start the encoder with an input layer. The input layer will take in a batch of sequences of tokens. Each sequence of tokens is n_en_seq_length elements long. Remember that we padded or truncated the sentences to make sure all of them have a fixed length of n_en_seq_length:

```
encoder_input = tf.keras.layers.Input(shape=(n_en_seq_length,), dtype=tf.string)
```

Next we use the previously defined `StringLookup` layer to convert the string tokens into word IDs. As we saw, the `StringLookup` layer can take a list of unique words (i.e. a vocabulary) and create a lookup operation to convert a given token into a numerical ID:

```
encoder_wid_out = en_lookup_layer(encoder_input)
```

With the tokens converted into IDs, we route the generated word IDs to a token embedding layer. We pass in the size of the vocabulary (derived from the `en_lookup_layer`'s `get_vocabulary()` method) and the embedding size (128) and finally we ask the layer to mask any zero-valued inputs as they don't contain any information:

```
en_full_vocab_size = len(en_lookup_layer.get_vocabulary())
encoder_emb_out = tf.keras.layers.Embedding(en_full_vocab_size, 128, mask_zero=True)(encoder_wid_out)
```

The output of the embedding layer is stored in `encoder_emb_out`. Next we define a GRU layer to process the sequence of English token embeddings:

```
encoder_gru_out, encoder_gru_last_state = tf.keras.layers.GRU(256, return_sequences=True, return_state=True)(encoder_emb_out)
```

Note how we are setting both the `return_sequences` and `return_state` arguments to True. To recap, `return_sequences` returns the full sequence of hidden states as the output (instead of returning only the last), where `return_state` returns the last state of the model as an additional output. We need both these outputs to build the rest of our model. For example, we need to pass the last state of the encoder to the decoder as the initial state. For that, we need the last state of the encoder (stored in `encoder_gru_last_state`). We will discuss the purpose of this in more detail as we go. We now have everything to define the encoder part of our model. It takes in a batch of sequences of string tokens and returns the full sequence of GRU hidden states as the output.

```
encoder = tf.keras.models.Model(inputs=encoder_input, outputs=encoder_gru_out)
```

With the encoder defined, let's build the decoder.

Defining the decoder

Our decoder will be more complex than the encoder. The objective of the decoder is, given the last encoder state and the previous token the decoder predicted, predict the next token. For example, for the German sentence:

<ss> ich ging zum Laden </s>

We define:

Input	<s>	ich	ging	zum	Laden
Output	ich	ging	zum	Laden	</s>

This technique is known as **teacher forcing**. In other words, the decoder is leveraging previous tokens of the target itself to predict the next token. This makes the translation task easier for the model. We can understand this phenomenon as follows. Say a teacher asks a kindergarten student to complete the following sentence, given just the first word:

I _____

This means that the child needs to pick a subject, verb, and object; know the syntax of the language; understand the grammar rules of the language; and so on. Therefore, the likelihood of the child producing an incorrect sentence is high.

However, if we ask the child to produce it word by word, they might do a better job at coming up with a sentence. In other words, we ask the child to produce the next word given the following:

I _____

Then we ask them to fill in the blank given:

I like _____

And continue in the same fashion:

I like to ___, *I like to fly* ___, *I like to fly kites* ___

This way, the child can do a better job at producing a correct and meaningful sentence. We can adopt the same approach to alleviate the difficulty of the translation task, as shown in *Figure 9.10*:

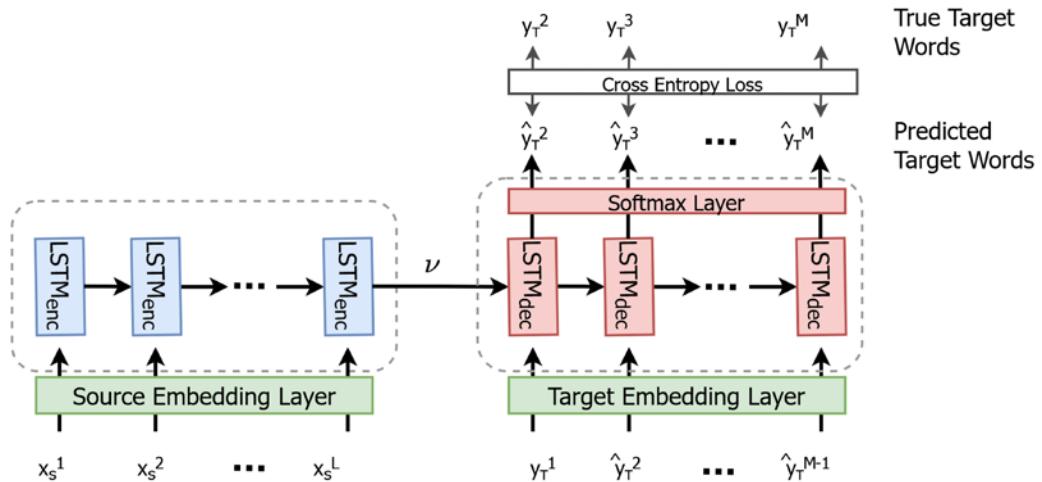


Figure 9.10: The teacher forcing mechanism. The darker arrows in the inputs depict newly introduced input connections to the decoder. The right-hand side figure shows how the decoder GRU cell changes.

To feed in previous tokens predicted by the decoder, we need an input layer for the decoder. When formulating the decoder inputs and outputs this way, for a sequence of tokens with length n , the input and output are $n-1$ tokens long:

```
decoder_input = tf.keras.layers.Input(shape=(n_de_seq_length-1,),  
dtype=tf.string)
```

Next, we use the `de_lookup_layer` defined earlier to convert tokens to IDs:

```
decoder_wid_out = de_lookup_layer(decoder_input)
```

Similar to the encoder, let's define an embedding layer for the German language:

```
de_full_vocab_size = len(de_lookup_layer.get_vocabulary())  
decoder_emb_out = tf.keras.layers.Embedding(de_full_vocab_size, 128, mask_  
zero=True)(decoder_wid_out)
```

We define a GRU layer in the decoder that will take the token embeddings and produce hidden outputs:

```
decoder_gru_out = tf.keras.layers.GRU(256, return_sequences=True)(decoder_emb_out, initial_state=encoder_gru_last_state)
```

Note that we are passing the encoder's last state to a special argument called `initial_state` in the GRU's `call()` method. This ensures that the decoder uses the encoder's last state to initialize its memory.

The next step of our journey takes us to one of the most important concepts in machine learning, ‘attention.’ So far, the decoder had to rely on the encoder’s last state as the ‘only’ input/signal about the source language. This is like asking to summarize a sentence using a single word. Generally, when doing so, you lose a lot of the meaning and message in this conversion. Attention alleviates this problem.

Attention: Analyzing the encoder states

Instead of relying just on the encoder’s last state, attention enables the decoder to analyze the complete history of state outputs. The decoder does this at every step of the prediction and creates a weighted average of all the state outputs depending on what it needs to produce at that step. For example, in the translation *I went to the shop* -> *ich ging zum Laden*, when predicting the word *ging*, the decoder will pay more attention to the first part of the English sentence than the latter.

There have been many different implementations of attention over the years. It’s important to properly emphasize the need for attention in NMT systems. As you have learned previously, the context, or thought vector, that resides between the encoder and the decoder is a performance bottleneck (see *Figure 9.11*):

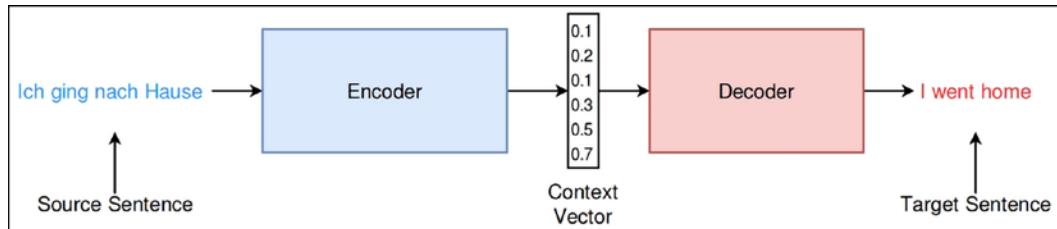


Figure 9.11: The encoder-decoder architecture

To understand why this is a bottleneck, let's imagine translating the following English sentence:

I went to the flower market to buy some flowers

This translates to the following:

Ich ging zum Blumenmarkt, um Blumen zu kaufen

If we are to compress this into a fixed-length vector, the resulting vector needs to contain these:

- Information about the subject (*I*)
- Information about the verbs (*buy* and *went*)
- Information about the objects (*flowers* and *flower market*)
- Interaction of the subjects, verbs, and objects with each other in the sentence

Generally, the context vector has a size of 128 or 256 elements. Reliance on the context vector to store all this information with a small-sized vector is very impractical and an extremely difficult requirement for the system. Therefore, most of the time, the context vector fails to provide the complete information required to make a good translation. This results in an underperforming decoder that suboptimally translates a sentence.

To make the problem worse, during decoding the context vector is observed only in the beginning. Thereafter, the decoder GRU must memorize the context vector until the end of the translation. This becomes more and more difficult for long sentences.

Attention sidesteps this issue. With attention, the decoder will have access to the full state history of the encoder for each decoding time step. This allows the decoder to access a very rich representation of the source sentence. Furthermore, the attention mechanism introduces a softmax layer that allows the decoder to calculate a weighted mean of the past observed encoder states, which will be used as the context vector for the decoder. This allows the decoder to pay different amounts of attention to different words at different decoding steps.

Figure 9.12 shows a conceptual breakdown of the attention mechanism:

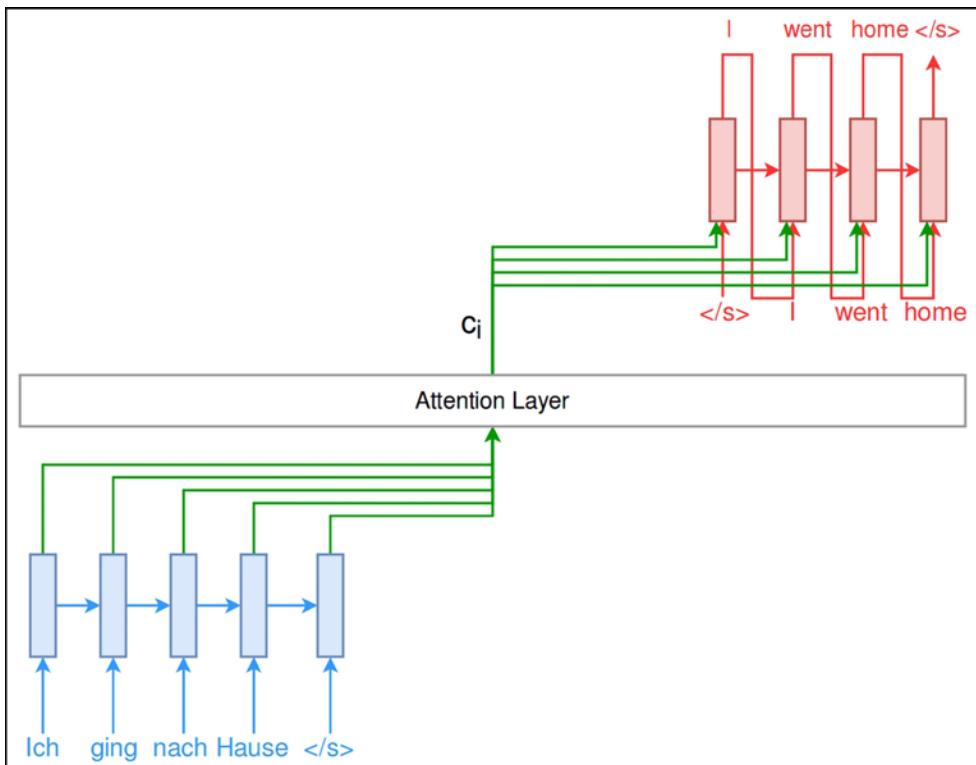


Figure 9.12: Conceptual attention mechanism in NMT

Next, let's look at how we can compute attention.

Computing Attention

Now let's investigate the actual implementation of the attention mechanism in detail. For this, we will use the Bahdanau attention mechanism introduced in the paper *Neural Machine Translation by Learning to Jointly Align and Translate*, by Bahdanau et al. We will discuss the original attention mechanism here. However, we'll be implementing a slightly different version of it, due to the limitations of TensorFlow. For consistency with the paper, we will use the following notations:

- Encoder's j^{th} hidden state: h_j
- i^{th} target token: y_i
- i^{th} decode hidden state in the i^{th} time step: s_i
- Context vector: c_i

Our decoder GRU is a function of an input y_i and a previous step's hidden state s_{i-1} . This can be represented as follows:

$$\text{GRU}_{dec} = f(y_i, s_{i-1})$$

Here, f represents the actual update rules used to calculate y_i and s_{i-1} . With the attention mechanism, we are introducing a new time-dependent context vector c_i for the i^{th} decoding step. The c_i vector is a weighted mean of the hidden states of all the unrolled encoder steps. A higher weight will be given to the j^{th} hidden state of the encoder if the j^{th} word is more important for translating the i^{th} word in the target language. This means the model can learn which words are important at which time step, regardless of the directionality of the two languages or alignment mismatches. Now the decoder GRU becomes this:

$$\text{GRU}_{dec} = f(y_i, s_{i-1}, c_i)$$

Conceptually, the attention mechanism can be thought of as a separate layer and illustrated as in *Figure 9.13*. As shown, attention functions as a layer. The attention layer is responsible for producing c_i for the i^{th} time step of the decoding process.

Let's now see how to calculate c_i :

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

Here, L is the number of words in the source sentence, and α_{ij} is a normalized weight representing the importance of the j^{th} encoder hidden state for calculating the i^{th} decoder prediction. This is calculated using what is known as an energy value. We represent e_{ij} as the energy of the encoder's j^{th} position for predicting the decoder's i^{th} position. e_{ij} is computed using a small fully connected network as follows:

$$e_{ij} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)$$

In other words, e_{ij} is calculated with a multilayer perceptron whose weights are v_a , W_a , and U_a , and s_{i-1} (decoder's previous hidden state from $(i-1)^{\text{th}}$ time step) and h_j (encoder's j^{th} hidden output) are the inputs to the network. Finally, we compute the normalized energy values (i.e. weights) using softmax normalization over all encoder timesteps:

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

The attention mechanism is shown in *Figure 9.13*:

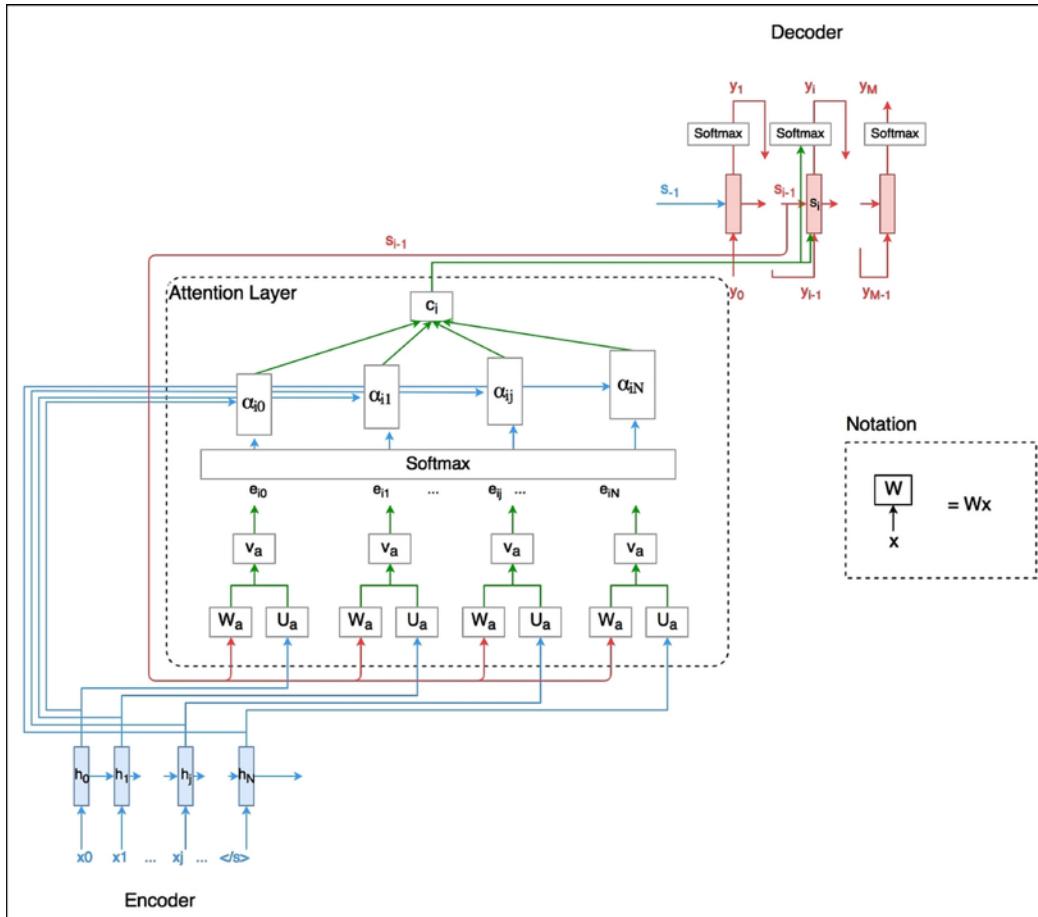


Figure 9.13: The attention mechanism

Implementing Attention

We said above that we'll be implementing a slightly different variation of Bahdanau attention. This is because TensorFlow currently does not support an attention mechanism that can be iteratively computed for each time step, similar to how an RNN works. Therefore, we are going to decouple the attention mechanism from the GRU model and have it computed separately. We will concatenate the attention output with the hidden output of the GRU layer and feed it to the final prediction layer. In other words, we are not feeding attention output to the GRU model, but directly to the prediction layer. This is depicted in *Figure 9.14*:

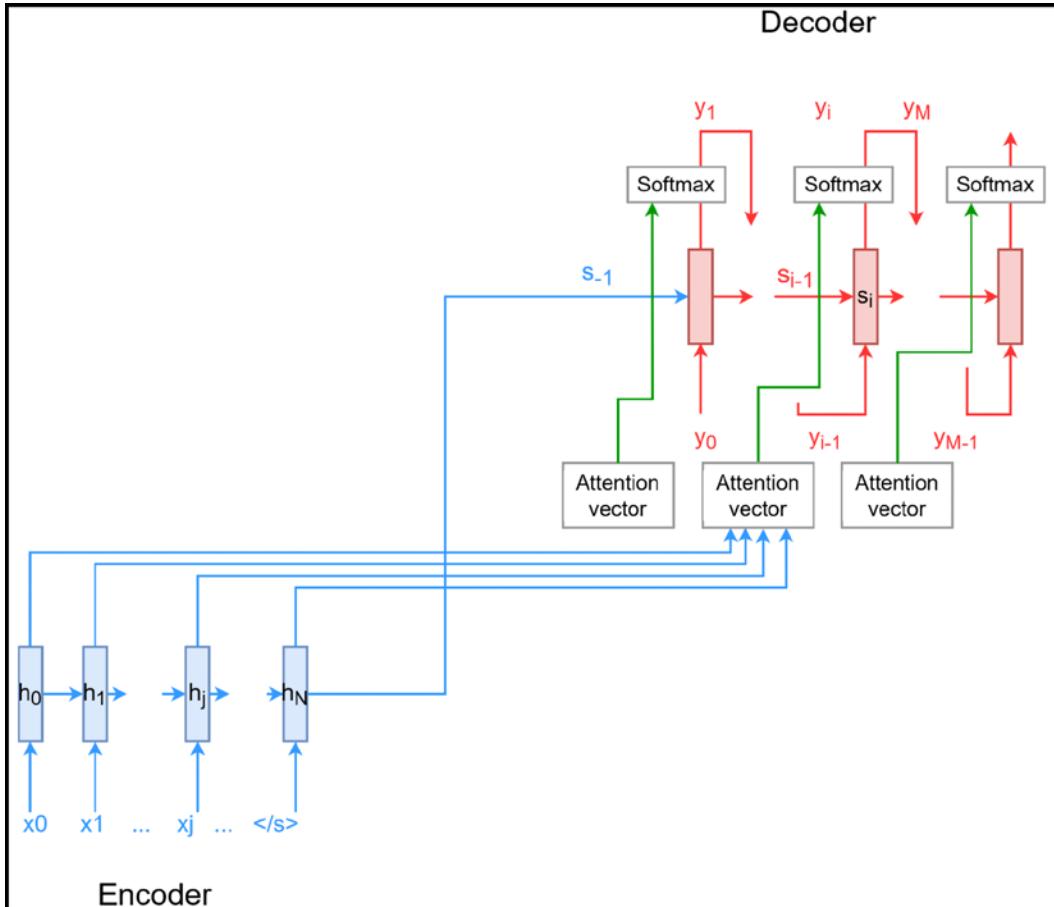


Figure 9.14: The attention mechanism employed in this chapter

To implement attention, we are going to use the sub-classing API of Keras. We'll define a class called `BahdanauAttention` (which inherits from the `Layer` class) and override two functions in that:

- `__init__()` – Defines the layer's initialization logic
- `call()` – Defines the computational logic of the layer

Our defined class would look like this. But don't worry, we'll be going through those two functions in detail below:

```
class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
```

```

super().__init__()

# Weights to compute Bahdanau attention
self.Wa = tf.keras.layers.Dense(units, use_bias=False)
self.Ua = tf.keras.layers.Dense(units, use_bias=False)

self.attention =
tf.keras.layers.AdditiveAttention(use_scale=True)

def call(self, query, key, value, mask,
return_attention_scores=False):

    # Compute 'Wa.h'.
    wa_query = self.Wa(query)

    # Compute 'Ua.hs'.
    ua_key = self.Ua(key)

    # Compute masks
    query_mask = tf.ones(tf.shape(query)[:-1], dtype=bool)
    value_mask = mask

    # Compute the attention
    context_vector, attention_weights = self.attention(
        inputs = [wa_query, value, ua_key],
        mask=[query_mask, value_mask, value_mask],
        return_attention_scores = True,
    )

    if not return_attention_scores:
        return context_vector
    else:
        return context_vector, attention_weights

```

First, we'll be looking at the `__init__()` function.

Here, you can see that we are defining three layers: weight matrix W_a , weight matrix U_a , and finally the `AdditiveAttention` layer, which contains the attention computation logic we discussed above. The `AdditiveAttention` layer takes in a query, value and a key. The query is the decoder states, and the value and key are all of encoder states produced.

We will discuss this layer in more detail soon. We'll discuss the details of this layer below. Next let's look at the computations defined in the `call()` function:

```
def call(self, query, key, value, mask, return_attention_scores=False):

    # Compute 'Wa.ht'
    wa_query = self.Wa(query)

    # Compute 'Ua.hs'
    ua_key = self.Ua(key)

    # Compute masks
    query_mask = tf.ones(tf.shape(query)[:-1], dtype=bool)
    value_mask = mask

    # Compute the attention
    context_vector, attention_weights = self.attention(
        inputs = [wa_query, value, ua_key],
        mask=[query_mask, value_mask, value_mask],
        return_attention_scores = True,
    )

    if not return_attention_scores:
        return context_vector
    else:
        return context_vector, attention_weights
```

The first thing to note is that this function takes a query, a key, and a value. These three elements will drive the attention computation. In Bahdanau attention, you can think of the key and value as being the same thing. The query will represent each decoder GRU's hidden states for each time step, and the value (or key) will represent each encoder GRU's hidden states for each time step. In other words, we are querying an output for each decoder position based on values provided by the encoder's hidden states.

Let's recap the computations we have to perform:

$$e_{ij} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)$$

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

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

First we compute `wa_query` (represents $W_a s_{i-1}$) and `ua_key` (represents $U_a h_j$). Next, we propagate these values to the attention layer. The `AdditiveAttention` layer (https://www.tensorflow.org/api_docs/python/tf/keras/layers/AdditiveAttention) performs the following steps:

1. Reshapes `wa_query` from `[batch_size, Tq, dim]` to shape `[batch_size, Tq, 1, dim]` and `ua_key` from `[batch_size, Tv, dim]` shape to `[batch_size, 1, Tv, dim]`.
2. Calculates scores with shape `[batch_size, Tq, Tv]` as: `scores = tf.reduce_sum(tf.tanh(query + key), axis=-1)`.
3. Uses scores to calculate a distribution with shape `[batch_size, Tq, Tv]` using softmax activation: `distribution = tf.nn.softmax(scores)`.
4. Uses `distribution` to create a linear combination of value with shape `[batch_size, Tq, dim]`.
5. Returns `tf.matmul(distribution, value)`, which represents a weighted average of all encoder states (i.e. `value`)

Here, you can see that *step 2* performs the first equation, *step 3* performs the second equation, and finally *step 4* performs the third equation. Another thing worth noting is that *step 2* does not mention v_a from the first equation. v_a is essentially a weight matrix with which we compute the dot product. We can introduce this weight matrix by setting `use_scale=True` when defining the `AdditiveAttention` layer:

```
self.attention = tf.keras.layers.AdditiveAttention(use_scale=True)
```

Another important argument is the `return_attention_scores` argument when calling the `AdditiveAttention` layer. This gives us the distribution weight matrix defined in *step 3*. We will use this to visualize where the model was paying attention when decoding the translation.

Defining the final model

With the attention mechanism understood and implemented, let's continue our implementation of the decoder. We will get the attention output sequence, with one attended output for each time step.

Moreover, we'll get the attention weights distribution matrix, which we'll use to visualize attention patterns against inputs and outputs:

```
decoder_attn_out, attn_weights = BahdanauAttention(256)(  
    query=decoder_gru_out, key=encoder_gru_out, value=encoder_gru_out,  
    mask=(encoder_wid_out != 0),  
    return_attention_scores=True  
)
```

When defining attention, we'll also pass a mask that denotes which tokens need to be ignored when computing outputs (e.g. padded tokens). Combine the attention output and the decoder's GRU output to create a single concatenated input for the prediction layer:

```
context_and_rnn_output = tf.keras.layers.concatenate([decoder_  
attn_out, decoder_gru_out])
```

Finally, the prediction layer takes the concatenated attention's context vector and the GRU output to produce probability distributions over the German tokens for each timestep:

```
# Final prediction layer (size of the vocabulary)  
decoder_out = tf.keras.layers.Dense(full_de_vocab_size,  
activation='softmax')(context_and_rnn_output)
```

With the encoder and the decoder fully defined, let's define the end-to-end model:

```
seq2seq_model = tf.keras.models.Model(inputs=[encoder.inputs, decoder_  
input], outputs=decoder_out)  
  
seq2seq_model.compile(loss='sparse_categorical_crossentropy',  
optimizer='adam', metrics='accuracy')
```

We are also going to define a secondary model called the `attention_visualizer`:

```
attention_visualizer = tf.keras.models.Model(inputs=[encoder.inputs,  
decoder_input], outputs=[attn_weights, decoder_out])
```

The `attention_visualizer` can generate attention patterns for a given set of inputs. This is a handy way to know if the model is paying attention to the correct words during the decoding process. This visualizer model will be used once the full model is trained. We will now look at how we can train our model.

Training the NMT

Now that we have defined the NMT architecture and preprocessed the training data, it is quite straightforward to train the model. Here, we will define and illustrate (see *Figure 9.15*) the exact process used for training:

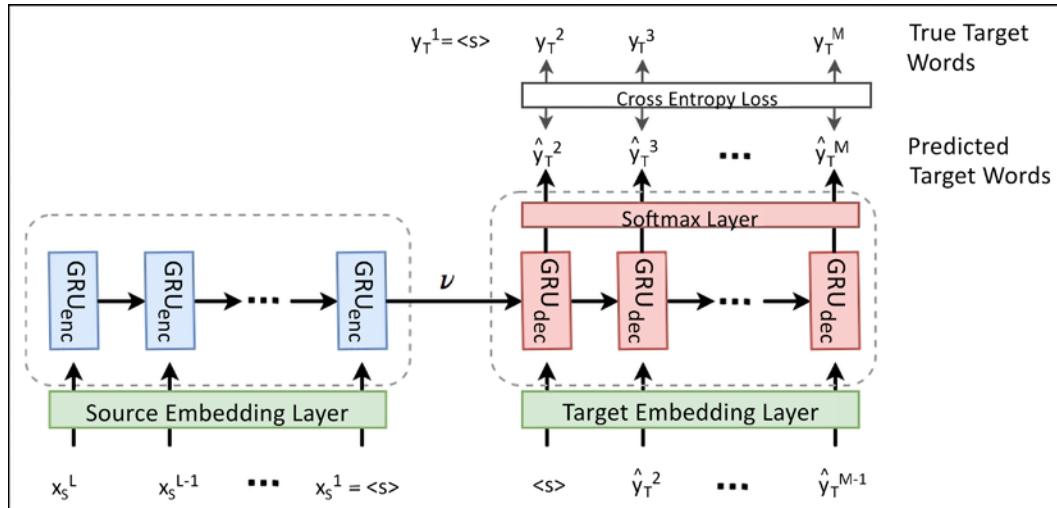


Figure 9.15: The training procedure for NMT

For the model training, we're going to define a custom training loop, as there is a special metric we'd like to track. Unfortunately, this metric is not a readily available TensorFlow metric. But before that, there are several utility functions we need to define:

```
def prepare_data(de_lookup_layer, train_xy, valid_xy, test_xy):
    """ Create a data dictionary from the dataframes containing data
    """
    data_dict = {}
    for label, data_xy in zip(['train', 'valid', 'test'], [train_xy, valid_xy, test_xy]):

        data_x, data_y = data_xy
        en_inputs = data_x
        de_inputs = data_y[:, :-1]
        de_labels = de_lookup_layer(data_y[:, 1:]).numpy()
        data_dict[label] = {'encoder_inputs': en_inputs,
```

```
'decoder_inputs': de_inputs, 'decoder_labels': de_labels}

return data_dict
```

The `prepare_data()` function takes the source sentence and target sentence pairs and generates encoder and decoder inputs and decoder labels. Let's understand the arguments:

- `de_lookup_layer` – The `StringLookup` layer of the German language
- `train_xy` – A tuple containing tokenized English sentences and tokenized German sentences in the training set, respectively
- `valid_xy` – Similar to `train_xy` but for validation data
- `test_xy` – Similar to `train_xy` but for test data

For each training, validation, and test dataset, this function generates the following:

- `encoder_inputs` – Tokenized English sentences as in the preprocessed dataset
- `decoder_inputs` – All tokens except the last of each German sentence
- `decoder_labels` – All token IDs except the first of each German sentence, where token IDs are generated by the `de_lookup_layer`

So, you can see that `decoder_labels` will be `decoder_inputs` shifted one token to the left. Next we define the `shuffle_data()` function, which will shuffle a provided set of data:

```
def shuffle_data(en_inputs, de_inputs, de_labels, shuffle_inds=None):
    """ Shuffle the data randomly (but all of inputs and labels at
    ones)"""

    if shuffle_inds is None:
        # If shuffle_inds are not passed create a shuffling
        # automatically
        shuffle_inds =
            np.random.permutation(np.arange(en_inputs.shape[0]))
    else:
        # Shuffle the provided shuffle_inds
        shuffle_inds = np.random.permutation(shuffle_inds)

    # Return shuffled data
    return (en_inputs[shuffle_inds], de_inputs[shuffle_inds],
            de_labels[shuffle_inds]), shuffle_inds
```

The logic here is quite straightforward. We take the `encoder_inputs`, `decoder_inputs`, and `decoder_labels` (generated by the `prepare_data()` step) with `shuffle_inds`. If `shuffle_inds` is `None`, we generate a random permutation of the indices. Otherwise, we generate a random permutation of the `shuffle_inds` provided. Finally, we index all of the data according to the shuffled index. We can then train the model:

```
Def train_model(model, en_lookup_layer, de_lookup_layer, train_xy,
    valid_xy, test_xy, epochs, batch_size, shuffle=True, predict_bleu_at_
    training=False):
    """ Training the model and evaluating on validation/test sets """

    # Define the metric
    bleu_metric = BLEUMetric(de_vocabulary)

    # Define the data
    data_dict = prepare_data(de_lookup_layer, train_xy, valid_xy,
        test_xy)

    shuffle_inds = None

    for epoch in range(epochs):

        # Reset metric Logs every epoch
        if predict_bleu_at_training:
            blue_log = []
            accuracy_log = []
            loss_log = []

            # ===== #
            #           Train Phase
            # ===== #

        # Shuffle data at the beginning of every epoch
        if shuffle:
            (en_inputs_raw,de_inputs_raw,de_labels), shuffle_inds =
            shuffle_data(
                data_dict['train']['encoder_inputs'],
```

```
        data_dict['train']['decoder_inputs'],
        data_dict['train']['decoder_labels'],
        shuffle_inds
    )
else:
    (en_inputs_raw,de_inputs_raw,de_labels) = (
        data_dict['train']['encoder_inputs'],
        data_dict['train']['decoder_inputs'],
        data_dict['train']['decoder_labels'],
    )
# Get the number of training batches
n_train_batches = en_inputs_raw.shape[0]//batch_size

prev_loss = None
# Train one batch at a time
for i in range(n_train_batches):
    # Status update
    print("Training batch {}/{}".format(i+1, n_train_batches),
          end='\r')

    # Get a batch of inputs (english and german sequences)
    x = [en_inputs_raw[i*batch_size:(i+1)*batch_size],
         de_inputs_raw[i*batch_size:(i+1)*batch_size]]
    # Get a batch of targets (german sequences offset by 1)
    y = de_labels[i*batch_size:(i+1)*batch_size]

    loss, accuracy = model.evaluate(x, y, verbose=0)

    # Check if any samples are causing NaNs
    check_for_nans(loss, model, en_lookup_layer,
                    de_lookup_layer)

    # Train for a single step
    model.train_on_batch(x, y)

    # Update the epoch's log records of the metrics
    loss_log.append(loss)
```

```
accuracy_log.append(accuracy)

if predict_bleu_at_training:
    # Get the final prediction to compute BLEU
    pred_y = model.predict(x)
    bleu_log.append(bleu_metric.calculate_bleu_from_
predictions(y, pred_y))

print("")
print("\nEpoch {} / {}".format(epoch+1, epochs))
if predict_bleu_at_training:
    print(f"\t(train) loss: {np.mean(loss_log)} - accuracy:
{np.mean(accuracy_log)} - bleu: {np.mean(bleu_log)}")
else:
    print(f"\t(train) loss: {np.mean(loss_log)} - accuracy:
{np.mean(accuracy_log)}")

# ===== #
#           Validation Phase
# ===== #

val_en_inputs = data_dict['valid']['encoder_inputs']
val_de_inputs = data_dict['valid']['decoder_inputs']
val_de_labels = data_dict['valid']['decoder_labels']

val_loss, val_accuracy, val_bleu = evaluate_model(
    model, de_lookup_layer, val_en_inputs, val_de_inputs,
    val_de_labels, batch_size
)

# Print the evaluation metrics of each epoch
print("\t(valid) loss: {} - accuracy: {} - bleu:
{}".format(val_loss, val_accuracy, val_bleu))

# ===== #
#           Test Phase
# ===== #
```

```
test_en_inputs = data_dict['test']['encoder_inputs']
test_de_inputs = data_dict['test']['decoder_inputs']
test_de_labels = data_dict['test']['decoder_labels']

test_loss, test_accuracy, test_bleu = evaluate_model(
    model, de_lookup_layer, test_en_inputs, test_de_inputs,
    test_de_labels, batch_size
)

print("\n(test) loss: {} - accuracy: {} - bleu:
{}".format(test_loss, test_accuracy, test_bleu))
```

During model training, we do the following:

- Prepare encoder and decoder inputs and decoder outputs using the `prepare_data()` function
- For each epoch:
 - Shuffle the data if the flag `shuffle` is set to True
 - For each iteration:
 - Get a batch of data from prepared inputs and outputs
 - Evaluate that batch using `model.evaluate` to get the loss and accuracy
 - Check if any of the samples are giving `nan` values (useful as a debugging step)
 - Train on the batch of data
 - Compute the BLEU score if the flag `predict_bleu_at_training` is set to True
 - Evaluate the model on validation data to get validation loss and accuracy
 - Compute the validation BLEU score
- Compute the loss, accuracy, and BLEU score on test data

You can see that we are computing a new metric called the BLEU metric. BLEU is a special metric used to measure performance in sequence-to-sequence problems. It tries to maximize the correctness of n-grams of tokens, rather than measuring it on individual tokens (e.g. accuracy). The higher the BLEU score, the better. You will learn more about how the BLEU score is calculated in the next section. You can see the logic defined in the `BLEUMetric` object in the code.

In this, we are mostly doing the preprocessing of text to remove uninformative tokens, so that the BLEU score is not overestimated. For example, if we include the <pad> token, you will see high BLEU scores, as there are long sequences of <pad> tokens for short sentences. To compute the BLEU score, we'll be using a third-party implementation available at <https://github.com/tensorflow/nmt/blob/master/nmt/scripts/bleu.py>.



Note

If you have a large batch size, you may see TensorFlow throwing an exception starting out as:

```
Resource exhausted: OOM when allocating tensor with ...
```

In this case, you may need to restart the notebook kernel, reduce the batch size, and rerun the code.

Another thing we do, but haven't discussed, is check for NaN (i.e. not-a-number) values. It can be very frustrating to see your loss value being NaN at the end of a training cycle. This is done by using the `check_for_nan()` function. This function will print out any specific data points that caused NaN values, so you have a much better idea of what caused it. You can find the implementation of the `check_for_nan()` function in the code.



Note

In 2021, the current state-of-the-art BLEU score for German to English translation is 35.14 (<https://paperswithcode.com/sota/machine-translation-on-wmt2014-english-german>).

Once the model is fully trained, you should see a BLEU score of around 15 for validation and test data. This is quite good, given that we used a very small proportion of the data (i.e. 250,000 sentences from more than 4 million) and a relatively simpler model compared to the state-of-the-art models.

Improving NMT performance with deep GRUs



One obvious improvement we can do is to increase the number of layers by stacking GRUs on top of each other, thereby creating a deep GRUs. For example, the Google NMT system uses eight LSTM layers stacked upon each other (*Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, Wu and others, Technical Report (2016)*). Though this hampers the computational efficiency, having more layers greatly improves the neural network's ability to learn the syntax and other linguistic characteristics of the two languages.

Next, let's understand how the BLEU score is calculated in detail.

The BLEU score – evaluating the machine translation systems

BLEU stands for **Bilingual Evaluation Understudy** and is a way of automatically evaluating machine translation systems. This metric was first introduced in the paper *BLEU: A Method for Automatic Evaluation of Machine Translation, Papineni and others, Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July 2002: 311-318*. We will be using an implementation of the BLEU score found at <https://github.com/tensorflow/nmt/blob/master/nmt/scripts/bleu.py>. Let's understand how this is calculated in the context of machine translation.

Let's consider an example to learn the calculations of the BLEU score. Say we have two candidate sentences (that is, a sentence predicted by our MT system) and a reference sentence (that is, the corresponding actual translation) for some given source sentence:

- Reference 1: *The cat sat on the mat*
- Candidate 1: *The cat is on the mat*

To see how good the translation is, we can use one measure, **precision**. Precision is a measure of how many words in the candidate are actually present in the reference. In general, if you consider a classification problem with two classes (denoted by negative and positive), precision is given by the following formula:

$$\text{Precision} = \frac{\text{number of samples correctly classified as positive}}{\text{all the samples classified as positive}}$$

Let's now calculate the precision for candidate 1:

$$\text{Precision} = \# \text{ of times each word of candidate appeared in reference} / \# \text{ of words in candidate}$$

Mathematically, this can be given by the following formula:

$$\text{Precision} = \frac{\sum_{\text{unigram} \in \text{Candidate}} \text{IsFoundInRef}(\text{unigram})}{|\text{Candidate}|}$$

$$\text{Precision for candidate 1} = 5/6$$

This is also known as 1-gram precision since we consider a single word at a time.

Now let's introduce a new candidate:

- Candidate 2: *The the cat cat cat*

It is not hard for a human to see that candidate 1 is far better than candidate 2. Let's calculate the precision:

$$\text{Precision for candidate 2} = 6/6 = 1$$

As we can see, the precision score disagrees with the judgment we made. Therefore, precision alone cannot be trusted to be a good measure of the quality of a translation.

Modified precision

To address the precision limitation, we can use a modified 1-gram precision. The modified precision clips the number of occurrences of each unique word in the candidate by the number of times that word appeared in the reference:

$$p_1 = \frac{\sum_{\text{unigram} \in \{\text{Candidate}\}} \text{Min}(\text{Occurrences}(\text{unigram}), \text{unigram}_{\max})}{|\text{Candidate}|}$$

Therefore, for candidates 1 and 2, the modified precision would be as follows:

$$\text{Mod-1-gram-Precision Candidate 1} = (1 + 1 + 1 + 1 + 1) / 6 = 5/6$$

$$\text{Mod-1-gram-Precision Candidate 2} = (2 + 1) / 6 = 3/6$$

We can already see that this is a good modification as the precision of candidate 2 is reduced. This can be extended to any n-gram by considering n words at a time instead of a single word.

Brevity penalty

Precision naturally prefers small sentences. This raises a question in evaluation, as the MT system might generate small sentences for longer references and still have higher precision. Therefore, a **brevity penalty** is introduced to avoid this. The brevity penalty is calculated by the following:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

Here, c is the candidate sentence length and r is the reference sentence length. In our example, we calculate it as shown here:

- BP for candidate 1 = $e^{(1-(6/6))}=e^0=1$
- BP for candidate 2 = $e^{(1-(6/6))}=e^0=1$

The final BLEU score

Next, to calculate the BLEU score, we first calculate several different modified n-gram precisions for a bunch of different $n=1,2,\dots,N$ values. We will then calculate the weighted geometric mean of the n-gram precisions:

$$BLEU = BP \times \exp\left(\sum_{i=1}^N w_n p_n\right)$$

Here, w_n is the weight for the modified n-gram precision p_n . By default, equal weights are used for all n-gram values. In conclusion, BLEU calculates a modified n-gram precision and penalizes the modified-n-gram precision with a brevity penalty. The modified n-gram precision avoids potential high precision values given to meaningless sentences (for example, candidate 2).

Visualizing Attention patterns

Remember that we specifically defined a model called `attention_visualizer` to generate attention matrices? With the model trained, we can now look at these attention patterns by feeding data to the model. Here's how the model was defined:

```
attention_visualizer = tf.keras.models.Model(inputs=[encoder.inputs,
decoder_input], outputs=[attn_weights, decoder_out])
```

We'll also define a function to get the processed attention matrix along with label data that we can use directly for visualization purposes:

```
def get_attention_matrix_for_sampled_data(attention_model, target_lookup_layer, test_xy, n_samples=5):  
  
    test_x, test_y = test_xy  
  
    rand_ids = np.random.randint(0, len(test_xy[0]),  
                                size=(n_samples,))  
    results = []  
  
    for rid in rand_ids:  
        en_input = test_x[rid:rid+1]  
        de_input = test_y[rid:rid+1,:-1]  
  
        attn_weights, predictions = attention_model.predict([en_input,  
                                                              de_input])  
        predicted_word_ids = np.argmax(predictions, axis=-1).ravel()  
        predicted_words = [target_lookup_layer.get_vocabulary()[wid]  
                           for wid in predicted_word_ids]  
  
        clean_en_input = []  
        en_start_i = 0  
        for i, w in enumerate(en_input.ravel()):  
            if w=='<pad>':  
                en_start_i = i+1  
                continue  
  
            clean_en_input.append(w)  
            if w=='</s>': break  
  
        clean_predicted_words = []  
        for w in predicted_words:  
            clean_predicted_words.append(w)
```

```
if w=='</s>': break

results.append(
{
    "attention_weights": attn_weights[
        0,:len(clean_predicted_words),en_start_i:en_start_
        i+len(clean_en_input)
    ],
    "input_words": clean_en_input,
    "predicted_words": clean_predicted_words
}
)

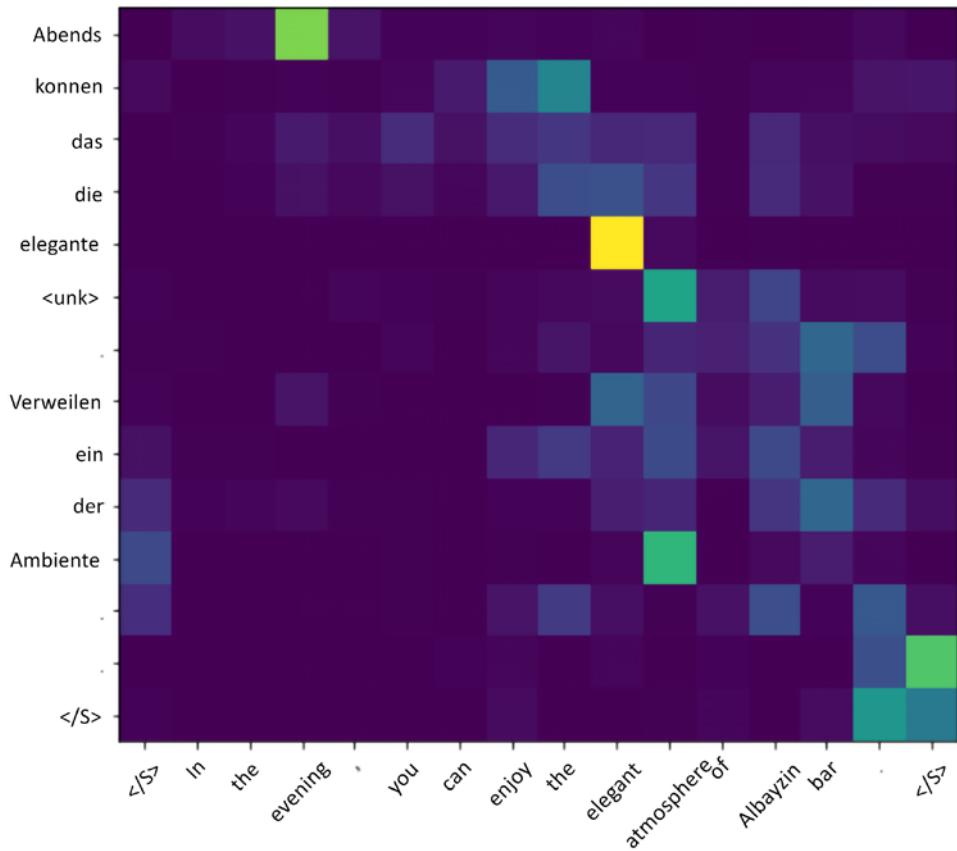
return results
```

This function does the following:

- Randomly samples n_samples indices from the test data.
- For each random index:
 - Gets the inputs of the data point at that index (en_input and de_input)
 - Gets the predicted words by feeding en_input and de_input to the attention_visualizer (stored in predicted_words)
 - Cleans en_input by removing any uninformative tokens (e.g. <pad>) and assigns to clean_en_input
 - Cleans predicted_words by removing tokens after the </s> token (stored in clean_predicted_words)
 - Gets the attention weights only corresponding to the words left in the clean inputs and predicted words from attn_weights
 - Appends the clean_en_input, clean_predicted_words, and attention weights matrix to results

The results contain all the information we need to visualize attention patterns. You can see the actual code used to create the following visualizations in the notebook Ch09-Seq2seq-Models/ch09_seq2seq.ipynb.

Let's take a few samples from our test dataset and visualize attention patterns exhibited by the model (*Figure 9.16*):



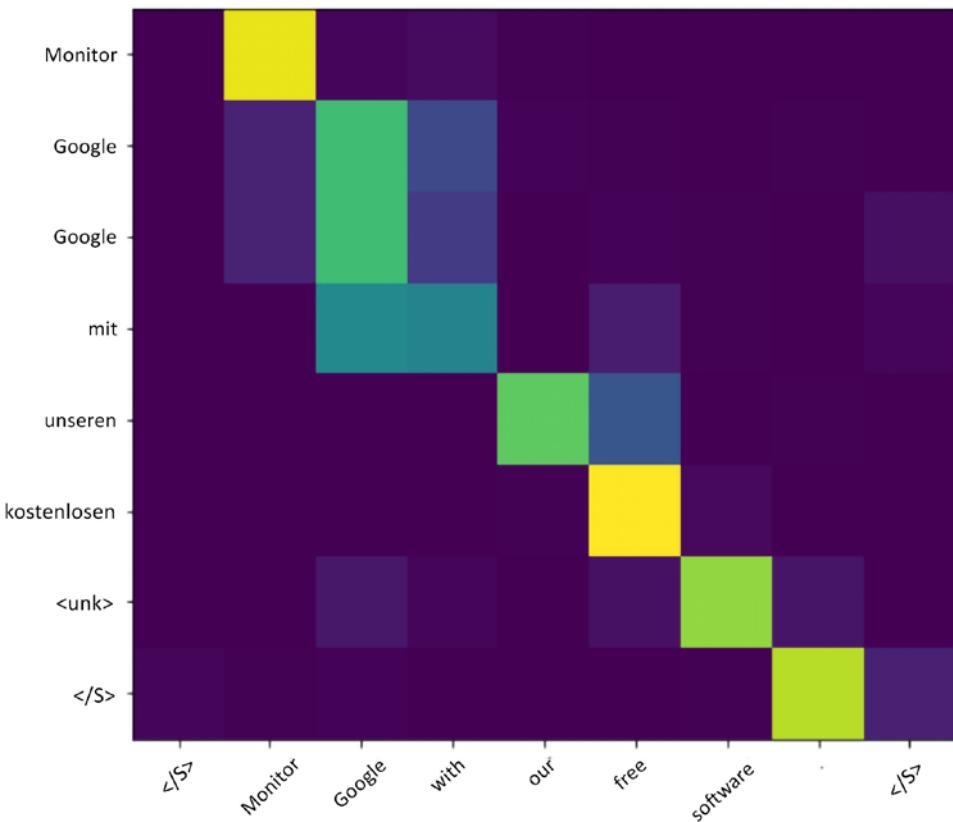


Figure 9.16: Visualizing attention patterns for a few test inputs

Overall, we'd like to see a heat map that has a roughly diagonal activation of energy. This is because both languages have a similar construct in terms of the direction of the language. And we can clearly see that in both examples.

Looking at specific words in the first example, you can see the model focuses heavily on *evening* to predict *Abends*, *atmosphere* to predict *Ambiente*, and so on. In the second, you see that the model is focusing on the word *free* to predict *kostenlosen*, which is German for *free*.

Next, we discuss how to infer translations from the trained model.

Inference with NMT

Inferencing is slightly different from the training process for NMT (Figure 9.17). As we do not have a target sentence at the inference time, we need a way to trigger the decoder at the end of the encoding phase. It's not difficult as we have already done the groundwork for this in the data we have. We simply kick off the decoder by using $\langle s \rangle$ as the first input to the decoder. Then we recursively call the decoder using the predicted word as the input for the next timestep. We continue this way until the model:

- Outputs $\langle /s \rangle$ as the predicted token or
- Reaches a pre-defined sentence length

To do this, we have to define a new model using the existing weights of the training model. This is because our trained model is designed to consume a sequence of decoder inputs at once. We need a mechanism to recursively call the decoder. Here's how we can define the inference model:

- Define an encoder model that outputs the encoder's hidden state sequence and the last encoder state.
- Define a new decoder that takes a decoder input having a time dimension of 1 and a new input, to which we will input the previous hidden state value of the decoder (initialized with the encoder's last state).

With that, we can start feeding data to generate predictions as follows:

- Preprocess x_s as in data processing
- Feed x_s into GRU_{enc} and calculate the encoder's state sequence and the last state h conditioned on x_s
- Initialize textGRU_{dec} with h
- For the initial prediction step, predict \hat{y}_t^2 by conditioning the prediction on $\hat{Y}_t^1 = \langle s \rangle$ as the first word and h

- For subsequent time steps, while $\hat{Y}_T^i \neq </s>$ and predictions haven't reached a pre-defined length threshold, predict \hat{Y}_T^{m+1} by conditioning the prediction on $\{\hat{Y}_T^m, \hat{Y}_T^{m-1}, \dots, < s >\}$ and h

This produces the translation given an input sequence of text:

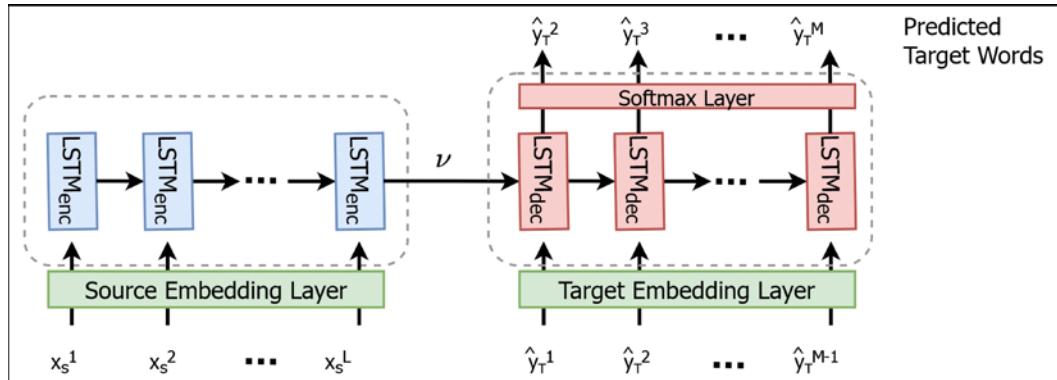


Figure 9.17: Inferring from an NMT

The actual code can be found in the notebook Ch09-Seq2seq-Models/ch09_seq2seq.ipynb. We will leave it for the reader to study the code and understand the implementation. We conclude our discussion about machine translation here. Now, let's briefly examine another application of sequence-to-sequence learning.

Other applications of Seq2Seq models – chatbots

One other popular application of sequence-to-sequence models is in creating chatbots. A chatbot is a computer program that is able to have a realistic conversation with a human. Such applications are very useful for companies with a huge customer base. Responding to customers asking basic questions for which answers are obvious accounts for a significant portion of customer support requests. A chatbot can serve customers with basic concerns when it is able to find an answer. Also, if the chatbot is unable to answer a question, the request gets redirected to a human operator. Chatbots can save a lot of the time that human operators spend answering basic concerns and let them attend to more difficult tasks.

Training a chatbot

So, how can we use a sequence-to-sequence model to train a chatbot? The answer is quite straightforward as we have already learned about the machine translation model. The only difference would be how the source and target sentence pairs are formed.

In the NMT system, the sentence pairs consist of a source sentence and the corresponding translation in a target language for that sentence. However, in training a chatbot, the data is extracted from the dialogue between two people. The source sentences would be the sentences/phrases uttered by person A, and the target sentences would be the replies to person A made by person B. One dataset that can be used for this purpose consists of movie dialogues between people and is found at https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html.

Here are links to several other datasets for training conversational chatbots:

- Reddit comments dataset: https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/
- Maluuba dialogue dataset: <https://datasets.maluuba.com/Frames>
- Ubuntu dialogue corpus: <http://dataset.cs.mcgill.ca/ubuntu-corpus-1.0/>
- NIPS conversational intelligence challenge: <http://convai.io/>
- Microsoft Research social media text corpus: <https://tinyurl.com/y7ha9rc5>

Figure 9.18 shows the similarity of a chatbot system to an NMT system. For example, we train a chatbot with a dataset consisting of dialogues between two people. The encoder takes in the sentences/phrases spoken by one person, where the decoder is trained to predict the other person's response. After training in such a way, we can use the chatbot to provide a response to a given question:

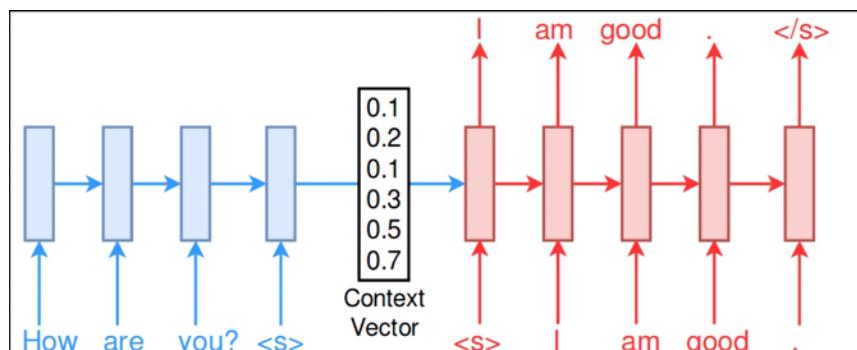


Figure 9.18: Illustration of a chatbot

Evaluating chatbots – the Turing test

After building a chatbot, one way to evaluate its effectiveness is using the Turing test. The Turing test was invented by Alan Turing in the 1950s as a way of measuring the intelligence of a machine. The experiment settings are well suited for evaluating chatbots. The experiment is set up as follows.

There are three parties involved: an evaluator (that is, a human) (A), another human (B), and a machine (C). The three of them sit in three different rooms so that none of them can see the others. The only communication medium is text, which is typed into a computer by one party, and the receiver sees the text on a computer on their side. The evaluator communicates with both the human and the machine. And at the end of the conversation, the evaluator is to distinguish the machine from the human. If the evaluator cannot make the distinction, the machine is said to have passed the Turing test. This setup is illustrated in *Figure 9.19*:

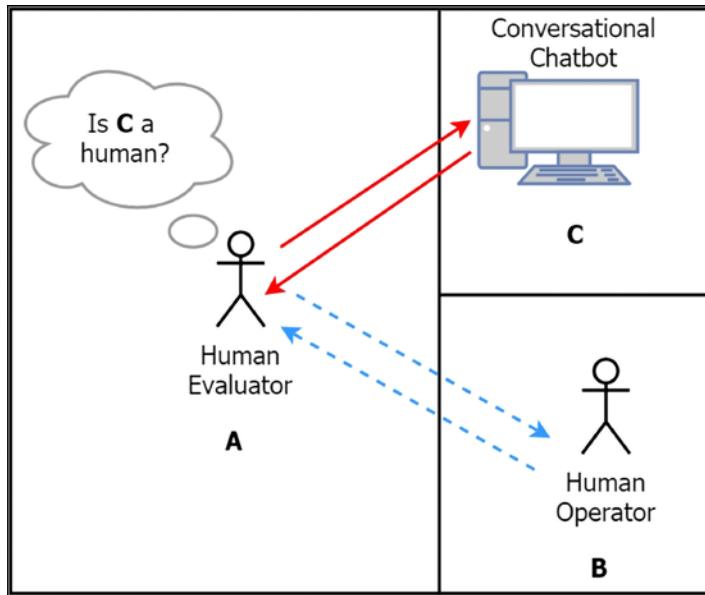


Figure 9.19: The Turing test

This concludes the section on other applications of Seq2Seq models. We briefly discussed the application of creating chatbots, which is a popular use for sequential models.

Summary

In this chapter, we talked in detail about NMT systems. Machine translation is the task of translating a given text corpus from a source language to a target language. First, we talked about the history of machine translation briefly to build a sense of appreciation for what has gone into machine translation for it to become what it is today. We saw that today, the highest-performing machine translation systems are actually NMT systems. Next, we solved the NMT task of generating English to German translations. We talked about the dataset preprocessing that needs to be done, and extracting important statistics about the data (e.g. sequence lengths). We then talked about the fundamental concept of these systems and decomposed the model into the embedding layer, the encoder, the context vector, and the decoder. We also introduced techniques like teacher forcing and Bahdanau attention, which are aimed at improving model performance. Then we discussed how training and inference work in NMT systems. We also discussed a new metric called BLEU and how it is used to measure performance on sequence-to-sequence problems like machine translation.

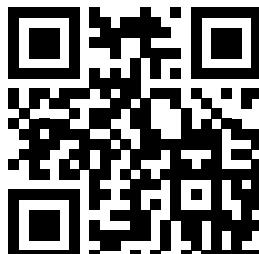
Finally, we briefly talked about another popular application of sequence-to-sequence learning: chatbots. Chatbots are machine learning applications that are able to have realistic conversations with a human and even answer questions. We saw that NMT systems and chatbots work similarly, and only the training data is different. We also discussed the Turing test, which is a qualitative test that can be used to evaluate chatbots.

In the next chapter, we will look at a new type of model that came out in 2016 and is leading both the NLP and computer vision worlds: the Transformer.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside
more than 1000 members at: <https://packt.link/nlp>



10

Transformers

Transformer models changed the playing field for most machine learning problems that involve sequential data. They have advanced the state of the art by a significant margin compared to the previous leaders, RNN-based models. One of the primary reasons that the Transformer model is so performant is that it has access to the whole sequence of items (e.g. sequence of tokens), as opposed to RNN-based models, which look at one item at a time. The term Transformer has come up several times in our conversations as a method that has outperformed other sequential models such as LSTMs and GRUs. Now, we will learn more about Transformer models.

In this chapter, we will first learn about the Transformer model in detail. Then we will discuss the details of a specific model from the Transformer family known as **Bidirectional Encoder Representations from Transformers (BERT)**. We will see how we can use this model to complete a question-answering task.

Specifically, we will cover the following main topics:

- Transformer architecture
- Understanding BERT
- Use case: Using BERT to answer questions

Transformer architecture

A Transformer is a type of Seq2Seq model (discussed in the previous chapter). Transformer models can work with both image and text data. The Transformer model takes in a sequence of inputs and maps that to a sequence of outputs.

The Transformer model was initially proposed in the paper *Attention is all you need* by Vaswani et al. (<https://arxiv.org/pdf/1706.03762.pdf>). Just like a Seq2Seq model, the Transformer consists of an encoder and a decoder (Figure 10.1):

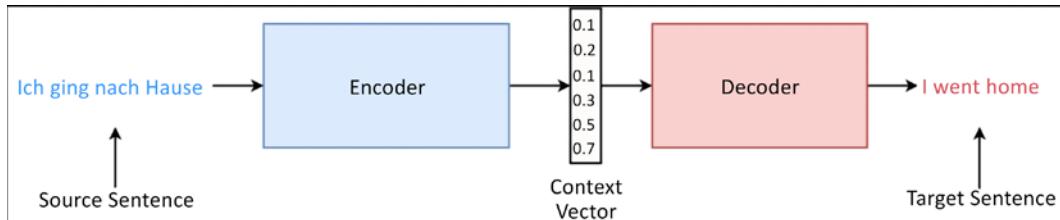


Figure 10.1: The encoder-decoder architecture

Let's understand how the Transformer model works using the previously studied Machine Translation task. The encoder takes in a sequence of source language tokens and produces a sequence of interim outputs. Then the decoder takes in a sequence of target language tokens and predicts the next token for each time step (the teacher forcing technique). Both the encoder and the decoder use attention mechanisms to improve performance. For example, the decoder uses attention to inspect all the past encoder states and previous decoder inputs. The attention mechanism is conceptually similar to Bahdanau attention, which we discussed in the last chapter.

The encoder and the decoder

Now let's discuss in detail what the encoder and the decoder consist of. They have more or less the same architecture with a few differences. Both the encoder and the decoder are designed to consume a sequence of input items at a time. But their goals during the task differ; the encoder produces a latent representation with the inputs, whereas the decoder produces a target output with the inputs and the encoder's outputs. To perform these computations, these inputs are propagated through several stacked layers. Each layer within these models takes in a sequence of elements and outputs another sequence of elements. Each layer is also made from several sub-layers that encapsulate different computations performed on a sequence of input tokens to produce a sequence of outputs.

A layer found in the Transformer mainly comprises the following two sub-layers:

- A self-attention layer
- A fully connected layer

The self-attention layer produces its output using matrix multiplications and activation functions (this is similar to a fully connected layer, which we will discuss in a minute). The self-attention layer takes in a sequence of inputs and produces a sequence of outputs. However, a special characteristic of the self-attention layer is that, when producing an output at each time step, it has access to all the other inputs in that sequence (*Figure 10.2*). This makes learning and remembering long sequences of inputs trivial for this layer. For comparison, RNNs struggle to remember long sequences of inputs as they need to go through each input sequentially. Additionally, by design, the self-attention layer can select and combine different inputs at each time step based on the task it's solving. This makes Transformers very powerful in sequential learning tasks.

Let's discuss why it's important to selectively combine different input elements this way. In an NLP context, the self-attention layer enables the model to peek at other words while processing a certain word. This means that while the encoder is processing the word *it* in the sentence *I kicked the ball and it disappeared*, the model can attend to the word *ball*. By doing this, the Transformer can learn dependencies and disambiguate words, which leads to better language understanding.

We can even understand how self-attention helps us to solve a task conveniently through a real-world example. Assume you are playing a game with two other people: person A and person B. Person A holds a question written on a board, and you need to answer that question. Say person A reveals one word of the question at a time, and after the last word of the question is revealed, you answer it. For long and complex questions, this would be challenging as you cannot physically see the complete question and would have to heavily rely on memory. This is what it feels like to perform computations without self-attention for a Transformer. On the other hand, say person B reveals the full question on the board in one go instead of word by word. Now it is much easier to answer the question as you can see the complete question at once. If the question is a complex question requiring a complex answer, you can look at different parts of the question as you are providing various sections of the full answer. This is what the self-attention layer enables.

The self-attention layer is followed by a fully connected layer. A fully connected layer has all the input nodes connected to all the output nodes, optionally followed by a non-linear activation function. It takes the output elements produced by the self-attention sub-layer and produces a hidden representation for each output element. Unlike the self-attention layer, the fully connected layer treats individual sequence items independently, performing computations on them in an element-wise fashion.

They introduce non-linear transformations while making the model deeper, thus allowing the model to perform better:

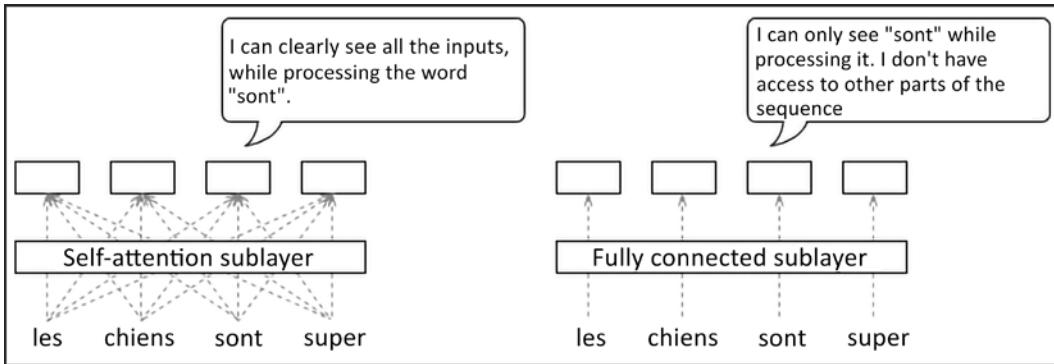


Figure 10.2: The difference between the self-attention sub-layer and the fully connected sub-layer. The self-attention sub-layer looks at all the inputs in the sequence, whereas the fully-connected sub-layer only looks at the input that is processed.

Now that we understand the basic building blocks of a Transformer layer, let's look at the encoder and the decoder separately. Before diving in, let's establish some basics. The encoder takes in an input sequence and the decoder takes in an input sequence as well (a different sequence to the encoder input). Then the decoder produces an output sequence. Let's call a single item in these sequences a *token*.

The encoder consists of a stack of layers, where each layer consists of two sub-layers:

- A self-attention layer – Generates a latent representation for each encoder input token in the sequence. For each input token, this layer looks at the whole sequence and selects other tokens in the sequence that enrich the semantics of the generated hidden output for that token (that is, ‘attended’ representation).
- A fully-connected layer – Generates an element-wise deeper hidden representation of the attended representation

The decoder layer consists of three sub-layers:

- A masked self-attention layer – For each decoder input, a token looks at all the tokens to the left of it. The decoder needs to mask words to the right to prevent the model from seeing words in the future. Having access to successive words during prediction can make the prediction task trivial for the decoder.

- An attention layer – For each input token in the decoder, it looks at both the encoder’s outputs and the decoder’s masked attended output to generate a semantically rich hidden output. Since this layer is not only focused on decoder inputs, we’ll call this an attention layer.
- A fully-connected layer – Generates an element-wise hidden representation of the attended representation of the decoder.

This is shown in *Figure 10.3*:

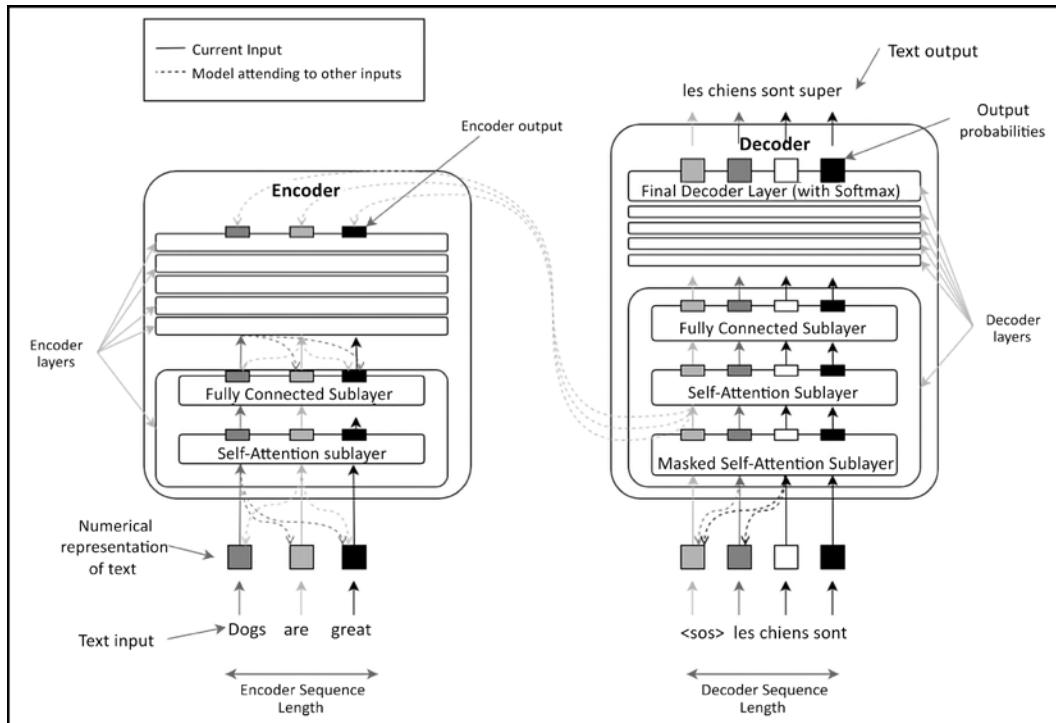


Figure 10.3: How a Transformer model is used to translate an English sentence to French. The diagram shows various layers in the encoder and the decoder, and various connections formed within the encoder, within the decoder, and between the encoder and the decoder. The squares represent the inputs and outputs of the models. The rectangular shaded boxes represent interim outputs of the sub-layers. The <sos> token represents the beginning of the decoder’s input.

Next, let’s learn about the computational mechanics of the self-attention layer.

Computing the output of the self-attention layer

There is no doubt that the self-attention layer is at the center of the Transformer. The computations that govern the self-attention mechanism can be difficult to understand. Therefore, this section is dedicated to understanding the self-attention technique in detail. There are three key concepts to understand: query, key, and value. The query and the key are used to generate an affinity matrix. For the decoder's attention layer, the affinity matrix's position i, j represents how similar the encoder state (key) i is to the decoder input j (query). Then, we create a weighted average of encoder states (value) for each position, where the weights are given by the affinity matrix.

To reinforce our understanding, let's imagine a scenario where the decoder is generating a self-attention output. Say we have an English to French machine translation task. Take the example sentence *Dogs are great*, which becomes *Les chiens sont super* in French. Say we are at time step 2, trying to produce the word *chiens*. Let's represent each word with a single floating point number (such as a simplified embedding representation of words):

Dogs -> 0.8

are -> 0.3

great -> -0.2

chiens -> 0.5

Now let's compute the affinity matrix (to be specific, the affinity vector since we are only considering a single decoder input). The query would be 0.5 and the key (i.e. the encoder state sequence) would be [0.8, 0.3, -0.2]. If we take the dot product, we have:

[0.4, 0.15, -0.1]

Let's understand what this affinity matrix is saying. With respect to the word *chiens*, the word *Dogs* has the highest similarity, and the word *are* also has a positive similarity (since *chiens* is plural, carrying a reference to the word *are* in English). However, the word *great* has a negative similarity to the word *chiens*. Then, we can compute the final attended output for that time step as:

$$[0.4 * 0.8, 0.15 * 0.3, -0.1 * -0.2] = [0.32 + 0.45 + 0.02] = 0.385$$

We have ended up with a final output somewhere in the middle of matching words from the English language, where the word *great* has the highest distance. This example was presented to show how the query, key, and value come into play to compute the final attended output.

Now let's look at the actual computation that transpires in the layer. To compute the query, key, and value, we use a linear projection of the actual inputs provided using weight matrices. The three weight matrices are:

- Query weights matrix (W_q)
- Key weights matrix (W_k)
- Value weights matrix (W_v)

Each of these weight matrices produces three outputs for a given token (at position i) in a given input sequence by multiplying with the weight matrix as follows:

$$Q_i = W_q q_i, K_i = W_k k_i, \text{ and } V_i = W_v v_i$$

Q, K , and V are $[B, T, d]$ sized tensors, where B is the batch size, T is the number of time steps, and d is a hyperparameter that defines the dimensionality of the latent representation. These are then used to compute the affinity matrix, as follows:

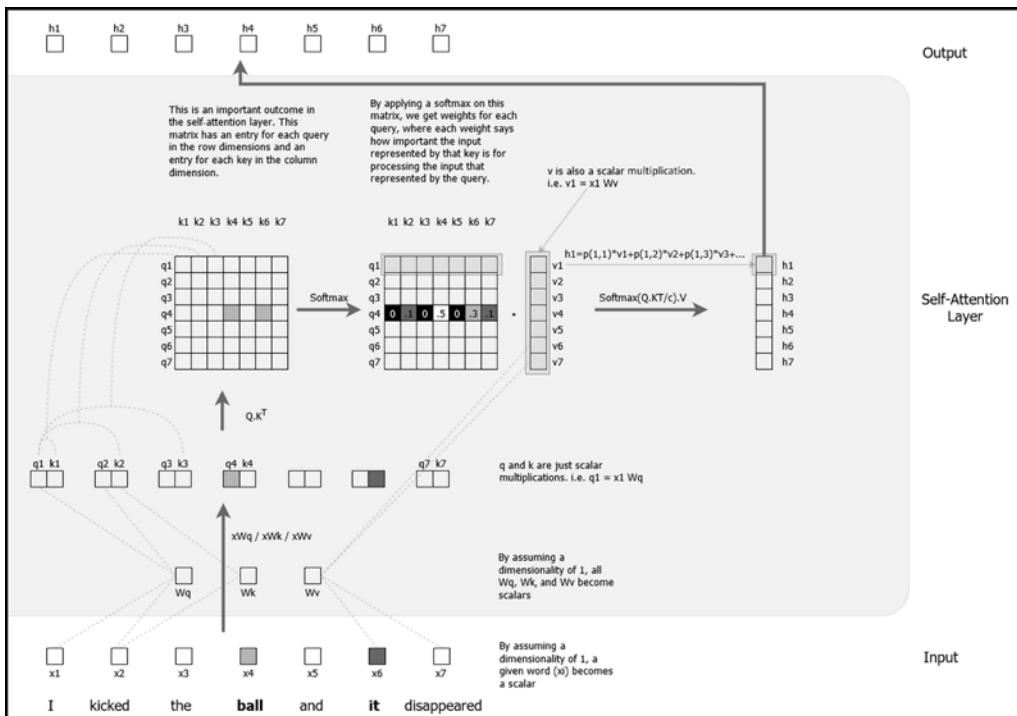


Figure 10.4: The computations in the self-attention layer. The self-attention layer starts with an input sequence and computes sequences of query, key, and value vectors. Then the queries and keys are converted to a probability matrix, which is used to compute a weighted sum of values

The affinity matrix P is computed as follows:

$$P = \text{softmax} \left(\frac{Q \cdot K^T}{\sqrt{d_k}} \right)$$

Then the final attended output of the self-attention layer is computed as follows:

$$h = P \cdot V = \text{softmax} \left(\frac{Q \cdot K^T}{\sqrt{d_k}} \right) V$$

Here, Q represents the queries tensor, K represents the keys tensor, and V represents the values tensor. This is what makes Transformer models so powerful; unlike LSTM models, Transformer models aggregate all tokens in a sequence to a single matrix multiplication, making these models highly parallelizable. *Figure 10.4* also depicts the computations that take place within the self-attention layer.

Embedding layers in the Transformer

Word embeddings provide a semantic-preserving representation of words based on the context in which words are used. In other words, if two words are used in the same context, they will have similar word vectors. For example, the words *cat* and *dog* will have similar representations, whereas *cat* and *volcano* will have vastly different representations.

Word vectors were initially introduced in the paper titled *Efficient Estimation of Word Representations in Vector Space* by Mikolov et al. (<https://arxiv.org/pdf/1301.3781.pdf>). It came in two variants: skip-gram and continuous bag-of-words. Embeddings work by first defining a large matrix of size $V \times E$, where V is the size of the vocabulary and E is the size of the embeddings. E is a user-defined hyperparameter; a larger E typically leads to more powerful word embeddings. In practice, you do not need to increase the size of embeddings beyond 300.

Motivated by the original word vector algorithms, modern deep learning models use embedding layers to represent words/tokens. The following general approach (along with pre-training later to fine-tune these embeddings) is taken to incorporate word embeddings into a machine learning model:

- Define a randomly initialized word embedding matrix (or pre-trained embeddings, available to download for free)
- Define the model (randomly initialized) that uses word embeddings as the inputs and produces an output (for example, sentiment, or a language translation)

- Train the whole model (embeddings and the model) end-to-end on the task

The same technique is used in Transformer models. However, in Transformer models, there are two different embeddings:

- Token embeddings (provide a unique representation for each token seen by the model in an input sequence)
- Positional embeddings (provide a unique representation for each position in the input sequence)

The token embeddings have a unique embedding vector for each token (such as character, word, and sub-word), depending on the model's tokenizing mechanism.

The positional embeddings are used to signal the model where a token is appearing. The primary purpose of the positional embeddings server is to inform the Transformer model where a word is appearing. This is because, unlike LSTMs/GRUs, Transformer models don't have a notion of sequence, as it processes the whole text in one go. Furthermore, a change to the position of a word can alter the meaning of a sentence/or a word. For example:

Ralph loves his tennis ball. It likes to chase the ball

Ralph loves his tennis ball. Ralph likes to chase it

In the sentences above, the word *it* refers to different things and the position of the word *it* can be used as a cue to identify this difference. The original Transformer paper uses the following equations to generate positional embeddings:

$$PE(pos, 2i) = \sin(pos/10000^{2i/d})$$

$$PE(pos, 2i + 1) = \cos(pos/10000^{2i/d})$$

where *pos* denotes the position in the sequence and *i* denotes the *ith* feature dimension ($0 \leq i < d$). Even-numbered features use a sine function and odd numbered features use a cosine function. *Figure 10.5* presents how positional embeddings change as the time step and the feature position change. It can be seen that feature positions with higher indices have lower-frequency sinusoidal waves. It is not entirely clear how the authors came up with the exact equation.

However, they do mention that they did not see a significant performance difference between the above equation and letting the model learn positional embeddings jointly during the training.

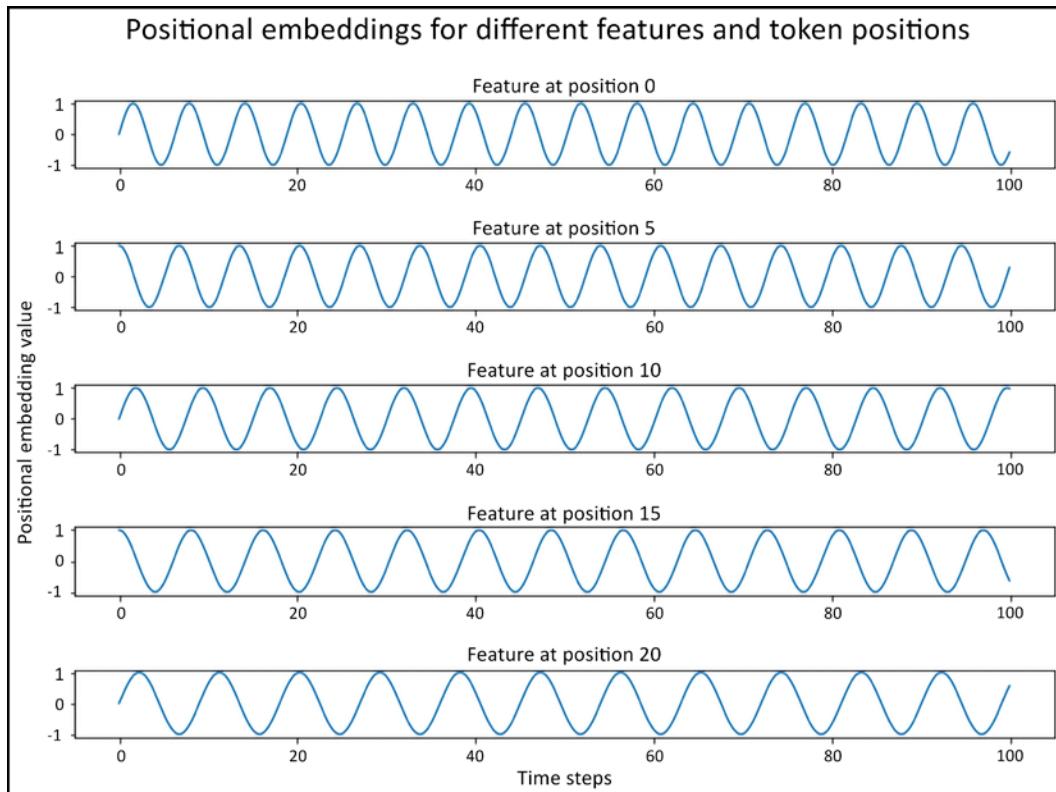


Figure 10.5: How positional embeddings change with the time step and the feature position. Even-numbered feature positions use the sine function and odd-numbered positions use the cosine function. Additionally, the frequency of the signals decreases as the feature position increases

It is important to note that both token and positional embeddings will have the same dimensionality d , making it possible to perform element-wise addition. Finally, as the input to the model, the token embeddings and the positional embeddings are summed to form a single hybrid embedding vector (*Figure 10.6*):

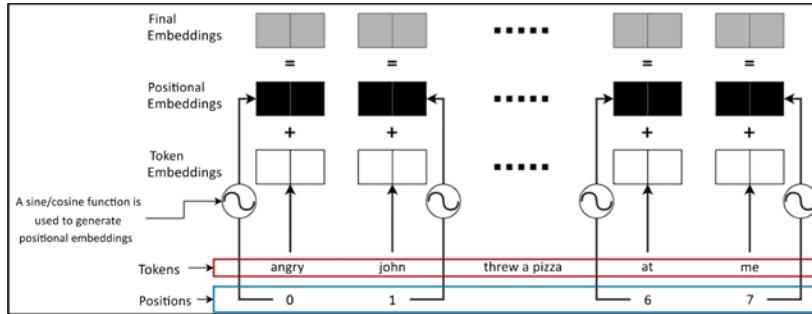


Figure 10.6: The embeddings generated in a Transformer model and how the final embeddings are computed

Let's now discuss two optimization techniques used in each layer of the Transformer: residual connections and layer normalizations.

Residuals and normalization

Another important characteristic of the Transformer models is the existence of the residual connections and the normalization layers in between the individual layers of the Transformer model.

Residual connections are formed by adding a given layer's output to the output of one or more layers ahead. This in turn forms shortcut connections through the model and provides a stronger gradient flow by reducing the changes of the phenomenon known as vanishing gradients (Figure 10.7). The vanishing gradients problem causes the gradients in the layers closest to the inputs to be very small so that the training in those layers is hindered. The residual connections for deep learning models were popularized by the paper “Deep Residual Learning for Image Recognition” by Kaiming He et.al. (<https://arxiv.org/pdf/1512.03385.pdf>)

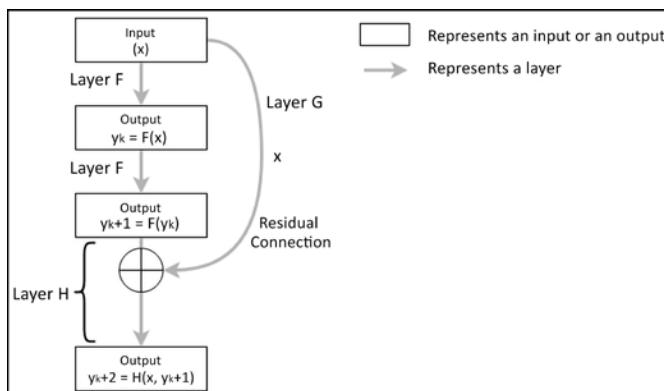


Figure 10.7: How residual connections work

In Transformer models, in each layer, residual connections are created the following way:

- Input to the self-attention sub-layer is added to the output of the self-attention sub-layer.
- Input to the fully-connected sub-layer is added to the output of the fully-connected sub-layer.

Next, the output reinforced by residual connections goes through a layer normalization layer. Layer normalization, similar to batch normalization, is a way to reduce the covariate shift in neural networks, allowing them to be trained faster and achieve better performance. Covariate shift refers to changes in the distribution of neural network activations (caused by changes in the data distribution), which transpires as the model goes through model training. These changes in the distribution damage consistency during model training and negatively impact the model. It was introduced in the paper *Layer Normalization* by Ba et al. (<https://arxiv.org/pdf/1607.06450.pdf>).

Batch normalization computes the mean and variance of activations as an average over the samples in the batch, causing its performance to rely on mini-batches used to train the model.

However, layer normalization computes the mean and variance (that is, the normalization terms) of the activations in such a way that the normalization terms are the same for every hidden unit. In other words, layer normalization has a single mean and a variance value for all the hidden units in a layer. This is in contrast to batch normalization, which maintains individual mean and variance values for each hidden unit in a layer. Moreover, unlike batch normalization, layer normalization does not average over the samples in the batch; instead, it leaves the averaging out and has different normalization terms for different inputs. By having a mean and variance per-sample, layer normalization gets rid of the dependency on the mini-batch size. For more details about this method, please refer to the original paper by Ba et al.

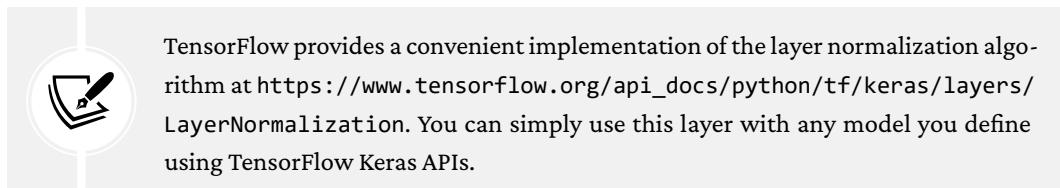


Figure 10.8 depicts how residual connections and layer normalization are used in Transformer models:

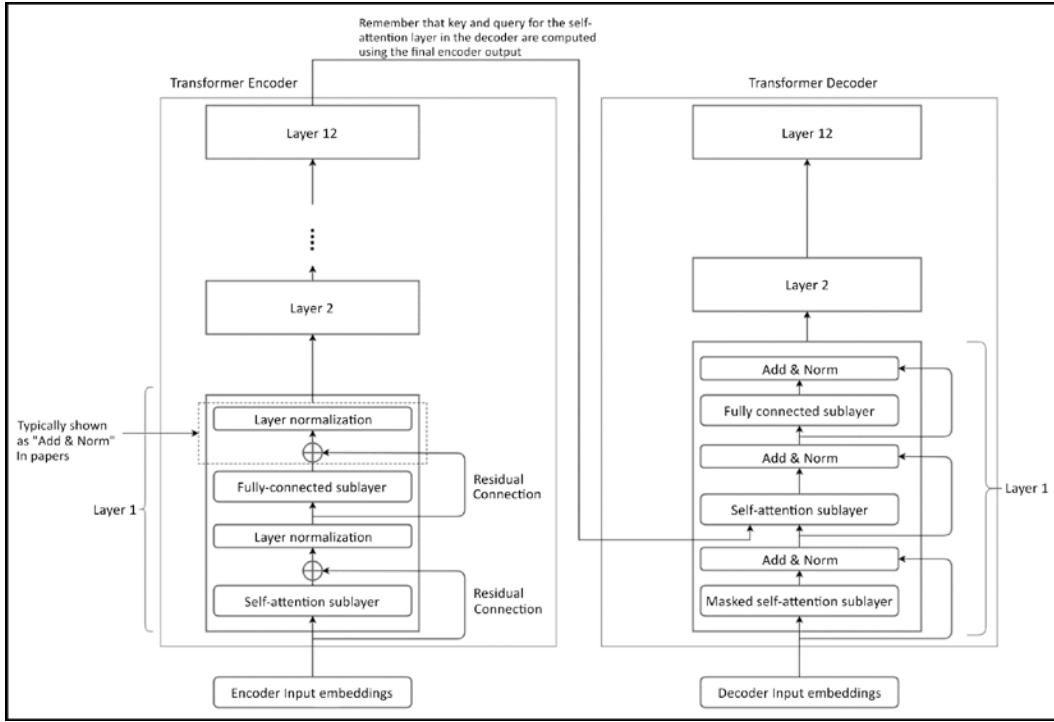


Figure 10.8: How residual connections and layer normalization layers are used in the transformer model

With that, we end our discussion on the components of the Transformer model. We have discussed all the bells and whistles of the Transformer model. The Transformer model is an encoder-decoder based model. Both the encoder and the decoder have the same structure, apart from a few small differences. The Transformer uses self-attention, a powerful parallelizable attention mechanism to attend to other inputs at every time step. The Transformer also uses several embedding layers, such as token embeddings and positional embeddings, to inject information about tokens and their positioning. The Transformer also uses residual connections and layer normalization to improve the performance of the model.

Next, we will discuss a specific Transformer model known as BERT, which we'll be using to solve a question-answering problem.

Understanding BERT

BERT (Bidirectional Encoder Representation from Transformers) is a Transformer model among a plethora of Transformer models that have come to light over the past few years.

BERT was introduced in the paper *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* by Delvin et al. (<https://arxiv.org/pdf/1810.04805.pdf>). The Transformer models are divided into two main factions:

- Encoder-based models
- Decoder-based (autoregressive) models

In other words, either the encoder or the decoder part of the Transformer provides the foundation for these models, compared to using both the encoder and the decoder. The main difference between the two is how attention is used. Encoder-based models use bidirectional attention, whereas decoder-based models use autoregressive (that is, left to right) attention.

BERT is an encoder-based Transformer model. It takes an input sequence (a collection of tokens) and produces an encoded output sequence. *Figure 10.9* depicts the high-level architecture of BERT:

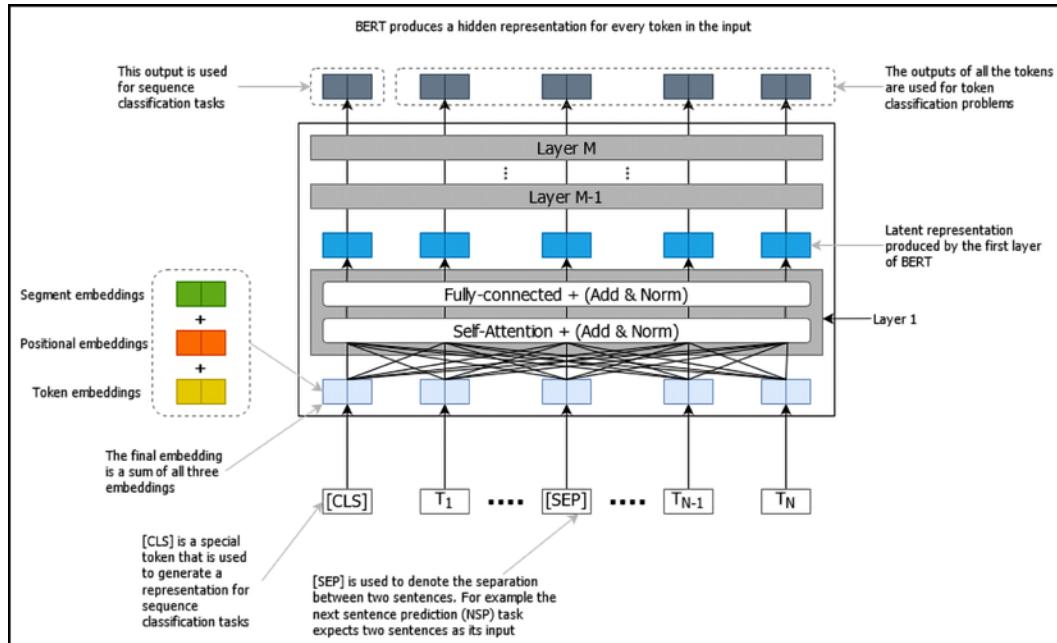


Figure 10.9: The high-level architecture of BERT. It takes a set of input tokens and produces a sequence of hidden representations generated using several hidden layers

Now let's discuss a few details pertinent to BERT, such as inputs consumed by BERT and the tasks it is designed to solve.

Input processing for BERT

When BERT takes an input, it inserts some special tokens into the input. First, at the beginning, it inserts a [CLS] (an abbreviated form of the term classification) token that is used to generate the final hidden representation for certain types of tasks (such as sequence classification). It represents the output after attending to all the tokens in the sequence. Next, it also inserts a [SEP] (meaning ‘separation’) token depending on the type of input. The [SEP] token marks the end and beginning of different sequences in the input. For example, in question-answering, the model takes a question and a context (such as a paragraph) that may have the answer as an input, and [SEP] is used in between the question and the context. Additionally, we have the [PAD] token, which can be used to pad short sequences to a required length.

The [CLS] token is appended to any input sequence fed to BERT. This denotes the beginning of the input. It also forms the basis for the input fed into the classification head used on top of BERT to solve your NLP task. As you know, BERT produces a hidden representation for each input token in the sequence. As a convention, the hidden representation corresponding to the [CLS] token is used as the input to the classification model that sits on top of BERT.

Next, the final embedding of the tokens is generated using three different embedding spaces. The token embedding has a unique vector for each token in the vocabulary. The positional embeddings encode the position of each token, as discussed earlier. Finally, the segment embedding provides a distinct representation for each sub-component in the input, when the input consists of multiple components. For example, in question-answering, the question will have a unique vector as its segment embedding vector and the context will have a different embedding vector. This is done by having n embedding vectors for the n different components in the input sequence. Depending on the component index specified for each token in the input, the corresponding segment embedding vector is retrieved. n needs to be specified in advance.

Tasks solved by BERT

The task-specific NLP tasks solved by BERT can be classified into four different categories. These are motivated by the tasks found in the **General Language Understanding Evaluation (GLUE)** benchmark task suite (<https://gluebenchmark.com>):

- Sequence classification – Here, a single input sequence is given and the model is asked to predict a label for the whole sequence (for example, sentiment analysis or spam identification).

- Token classification – Here, a single input sequence is given and the model is asked to predict a label for each token in the sequence (for example, named entity recognition or part-of-speech tagging).
- Question-answering – Here, the input consists of two sequences: a question and a context. The question and the context are separated by a [SEP] token. The model is trained to predict the starting and ending indices of the span of tokens belonging to the answer.
- Multiple choice – Here, the input consists of multiple sequences; a question followed by multiple candidates that may or may not be the answer to the question. These multiple sequences are separated by the token [SEP] and provided as a single input sequence to the model. The model is trained to predict the correct answer (that is, the class label) for that question.

Figure 10.10 depicts how BERT is used to solve these different tasks:

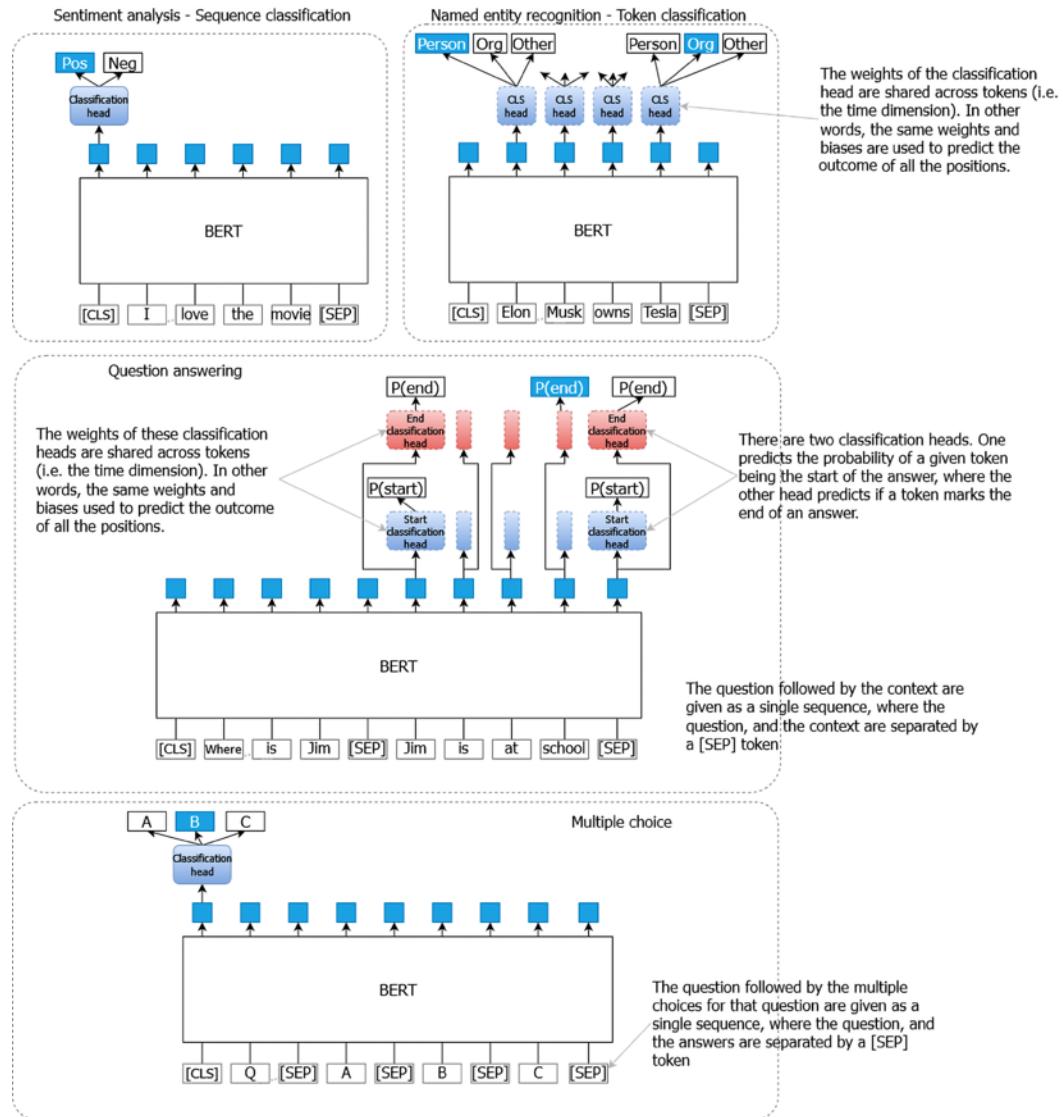


Figure 10.10: How BERT is used for different NLP tasks

BERT is designed in such a way that it can be used to complete these tasks without any modifications to the base model.

In tasks that involve multiple sequences (such as multiple-choice questions), you need the model to tell different inputs belonging to different segments apart (that is, which tokens are the question and which tokens are the context in a question-answering task). In order to make that distinction, the [SEP] token is used. A [SEP] token is inserted between the different sequences. For example, if you are solving a question-answering problem, you might have the following input:

Question: What color is the ball?

Paragraph: Tippy is a dog. She loves to play with her red ball.

Then the input to BERT might look like this:

[CLS] *What color is the ball* [SEP] *Tippy is a dog She loves to play with her red ball* [SEP]

Now that we have discussed all the elements of BERT so we can use it successfully to solve a downstream NLP task, let's reiterate the key points about BERT:

- BERT is an encoder-based Transformer
- BERT outputs a hidden representation for every token in the input sequence
- BERT has three embedding spaces: token embedding, positional embedding, and segment embedding
- BERT uses a special token [CLS] to denote the beginning of an input and is used as the input to a downstream classification model
- BERT is designed to solve four types of NLP tasks: sequence classification, token classification, free-text question-answering, and multiple-choice question-answering
- BERT uses the special token [SEP] to separate between sequence A and sequence B

The power within BERT doesn't just lie within its structure. BERT is pre-trained on a large corpus of text using a few different pre-training techniques. In other words, BERT already comes with a solid understanding of the language, making downstream NLP tasks easier to solve. Next, let's discuss how BERT is pre-trained.

How BERT is pre-trained

The real value of BERT comes from the fact that it has been pre-trained on a large corpus of data in a self-supervised fashion. In the pre-training stage, BERT is trained on two different tasks:

- **Masked language modeling** (sometimes abbreviated as **MLM**)
- **Next sentence prediction** (sometimes abbreviated as **NSP**)

Let's now discuss the details of the above two tasks and how they provide language understanding for BERT.

Masked Language Modeling (MLM)

The MLM task is inspired by the Cloze task, or the Cloze test, where a student is given a sentence with one or more blanks and is asked to fill the blanks. Similarly, given a text corpus, words are masked from sentences and then the model is asked to predict the masked tokens. For example, the sentence:

I went to the bakery to buy bread

might become:

I went to the [MASK] to buy bread

BERT uses a special token, **[MASK]**, to represent masked words. Then the target for the model will be the word *bakery*. But this introduces a practical issue to the model. The special **[MASK]** token does not appear in the actual text. This means that the text the model will see during the finetuning phase (that is, when training on a classification problem) will be different to what it will see during pre-training. This is sometimes referred to as the **pre-training-finetuning discrepancy**. Therefore, the authors of BERT suggest the following approach to cope with the issue. When masking a word, do one of the following:

- Use the **[MASK]** token as it is (with 80% probability)
- Use a random word (with 10% probability)
- Use the true word (with 10% probability)

In other words, instead of always seeing **[MASK]**, the model will see actual words on certain occasions, alleviating the discrepancy.

Next Sentence Prediction (NSP)

In the NSP task, the model is given a pair of sentences, A and B (in that order), and is asked to predict whether the B is the next sentence after A. This can be done by fitting a binary classifier onto BERT and training the whole model from end to end on selected pairs of sentences.

Generating pairs of sentences as inputs for the model is not hard and can be done in an unsupervised manner:

- A sample with the label TRUE is generated by picking two sentences that are adjacent to each other
- A sample with the label FALSE is generated by picking two sentences randomly that are not adjacent to each other

Following this approach, a labeled dataset is generated for the next sentence prediction task. Then BERT, along with the binary classifier, is trained from end to end using the labeled dataset to solve a downstream task. To see this in action, we'll be using Hugging Face's `transformers` library.

Use case: Using BERT to answer questions

Now let's learn how to implement BERT, train it on a question-answer dataset, and ask the model to answer a given question.

Introduction to the Hugging Face `transformers` library

We will use the `transformers` library built by Hugging Face. The `transformers` library is a high-level API that is built on top of TensorFlow, PyTorch, and JAX. It provides easy access to pre-trained Transformer models that can be downloaded and fine-tuned with ease. You can find models in the Hugging Face's model registry at <https://huggingface.co/models>. You can filter models by task, examine the underlying deep learning frameworks, and more.

The `transformers` library was designed with the aim of providing a very low barrier for entry to using complex Transformer models. For this reason, there's only a handful of concepts that you need to learn in order to hit the ground running with the library. Three important classes are required to load and use a model successfully:

- Model class (such as `TFBertModel`) – Contains the trained weights of the model in the form of `tf.keras.models.Model` or the PyTorch equivalent.
- Configuration (such as `BertConfig`) – Stores various parameters and hyperparameters needed to load the model. If you're using the pre-trained model as is, you don't need to explicitly define its configuration.
- Tokenizer (such as `BertTokenizerFast`) – Contains the vocabulary and token-to-ID mapping needed to tokenize the words for the model.

All of these classes can be used with two straightforward functions:

- `from_pretrained()` – Provides a way to instantiate a model/configuration/tokenizer available from the model repository or locally
- `save_pretrained()` – Provides a way to save the model/configuration/tokenizer so that it can be reloaded later



TensorFlow hosts a variety of Transformer models (released by both TensorFlow and third parties) in TensorFlow Hub (at <https://tfhub.dev/>). If you would like to know how to use TensorFlow Hub and the raw TensorFlow API to implement a model such as BERT, please visit https://www.tensorflow.org/text/tutorials/classify_text_with_bert.

We will soon see how these classes and functions are used in an actual use case. It is also important to note the side-effects of having such an easy-to-grasp interface for using models. Due to serving the very specific purpose of providing a way to use Transformer models built with TensorFlow, PyTorch, or Jax, you don't have the modularity or flexibility found in TensorFlow, for example. In other words, you cannot use the `transformers` library in the same way you would use TensorFlow to build a `tf.keras.models.Model` using `tf.keras.layers.Layer` objects.

Exploring the data

The dataset we are going to use for this task is a popular question-answering dataset called SQuAD. Each datapoint consists of four items:

- A question
- A context that may contain the answer to the question
- The start index of the answer
- The answer

We can download the dataset using Hugging Face's `datasets` library and call the `load_dataset()` function with the "squad" argument:

```
from datasets import load_dataset
dataset = load_dataset("squad")
```

Now let's print some examples using:

```
for q, a in zip(dataset["train"]["question"][:5], dataset["train"]
["answers"][:5]):
    print(f"{q} -> {a}")
```

which will output:

```
To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France? ->
{'text': ['Saint Bernadette Soubirous'], 'answer_start': [515]}

What is in front of the Notre Dame Main Building? -> {'text': ['a copper
statue of Christ'], 'answer_start': [188]}

The Basilica of the Sacred heart at Notre Dame is beside to which
structure? -> {'text': ['the Main Building'], 'answer_start': [279]}

What is the Grotto at Notre Dame? -> {'text': ['a Marian place of prayer
and reflection'], 'answer_start': [381]}

What sits on top of the Main Building at Notre Dame? -> {'text': ['a
golden statue of the Virgin Mary'], 'answer_start': [92]}
```

Here, `answer_start` indicates the character index at which this answer starts in the context provided. With a good understanding of what's available in the dataset, we'll perform a simple processing step. When training the model, we will be asking the model to predict the start and end indices of the answer. In its original form, only the `answer_start` is present. We will need to manually add `answer_end` to our dataset. The following function does this. Furthermore, it does a few sanitary checks on the dataset:

```
def compute_end_index(answers, contexts):
    """ Add end index to answers """

    fixed_answers = []
    for answer, context in zip(answers, contexts):
        gold_text = answer['text'][0]
        answer['text'] = gold_text
        start_idx = answer['answer_start'][0]
        answer['answer_start'] = start_idx

        # Make sure the starting index is valid and there is an answer
        assert start_idx >= 0 and len(gold_text.strip()) > 0:

        end_idx = start_idx + len(gold_text)
        answer['answer_end'] = end_idx

        # Make sure the corresponding context matches the actual answer
        assert context[start_idx:end_idx] == gold_text
```

```
    fixed_answers.append(answer)

    return fixed_answers, contexts

train_questions = dataset["train"]["question"]
print("Training data corrections")
train_answers, train_contexts = compute_end_index(
    dataset["train"]["answers"], dataset["train"]["context"]
)
test_questions = dataset["validation"]["question"]
print("\nValidation data correction")
test_answers, test_contexts = compute_end_index(
    dataset["validation"]["answers"], dataset["validation"]["context"]
)
```

Next, we will download a pre-trained BERT model from the Hugging Face repository and learn about the model in depth.

Implementing BERT

To use a pre-trained Transformer model from the Hugging Face repository, we need three components:

- **Tokenizer** – Responsible for splitting a long bit of text (such as a sentence) into smaller tokens
- **config** – Contains the configuration of the model
- **Model** – Takes in the tokens, looks up the embeddings, and produces the final output(s) using the provided inputs

We can ignore the `config` as we are using the pre-trained model as is. However, to paint a full picture, we will use the configuration nevertheless.

Implementing and using the Tokenizer

First, we will look at how to download the Tokenizer. You can download the Tokenizer using the `transformers` library. Simply call the `from_pretrained()` function provided by the `PreTrainedTokenizerFast` base class:

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

We will be using a Tokenizer called `bert-base-uncased`. It is the Tokenizer developed for the BERT base model and is uncased (that is, there's no distinction between uppercase and lowercase characters). Next, let's see the Tokenizer in action:

```
context = "This is the context"
question = "This is the question"

token_ids = tokenizer(
    text=context, text_pair=question,
    padding=False, return_tensors='tf'
)
print(token_ids)
```

Let's understand the arguments we've provided to the tokenizer's call:

- `text` – A single or batch of text sequences to be encoded by the tokenizer. Each text sequence is a string.
- `text_pair` – An optional single or batch of text sequences to be encoded by the tokenizer. It's useful in situations where the model takes a multi-part input (such as a question and a context in question-answering).
- `padding` – Indicates the padding strategy. If set to `True`, it will be padded to the maximum sequence length in the dataset. If set to `max_length`, it will be padded to the length specified by the `max_length` argument. If set to `False`, no padding will be done.
- `return_tensors` – An argument that defines the type of tensors returned. It could be either `pt` (PyTorch) or `tf` (TensorFlow). Since we want TensorFlow tensors, we define it as '`tf`'.

This prints:

```
{
    'input_ids': <tf.Tensor: shape=(1, 11), dtype=int32, numpy=array([[101, 2023, 2003, 1996, 6123, 102, 2023, 2003, 1996, 3160, 102]])>,
    'token_type_ids': <tf.Tensor: shape=(1, 11), dtype=int32,
    numpy=array([[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]])>,
    'attention_mask': <tf.Tensor: shape=(1, 11), dtype=int32,
    numpy=array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])>
}
```

This outputs a `transformers.tokenization_utils_base.BatchEncoding` object, which is essentially a dictionary. It has three keys and tensors as values:

- `input_ids` – Provides the IDs of the tokens found in the text sequences. Additionally, it introduces the [CLS] token ID at the beginning of the sequence and two instances of the [SEP] token ID, one between the question and context, and the other one at the end.
- `token_type_ids` – This is the segment ID we use for the segment embedding.
- `attention_mask` – The attention mask represents the words that are allowed to be attended to during the forward pass. Since BERT is an encoder model, any token can pay attention to any other token. The only exception is the padded tokens that will be ignored during the attention mechanism.

We could also convert these token IDs to actual tokens to know what they represent. To do that, we use the `convert_ids_to_tokens()` function:

```
print(tokenizer.convert_ids_to_tokens(token_ids['input_ids'].numpy()[0]))
```

This will print:

```
['[CLS]', 'this', 'is', 'the', 'context', '[SEP]', 'this', 'is', 'the', 'question', '[SEP]']
```

You can see how the tokenizer inserts special tokens like [CLS] and [SEP] into the text sequence. With the functionality of the tokenizer understood, let's use it to encode the train and test datasets:

```
# Encode train data
train_encodings = tokenizer(train_contexts, train_questions,
truncation=True, padding=True, return_tensors='tf')

# Encode test data
test_encodings = tokenizer(test_contexts, test_questions, truncation=True,
padding=True, return_tensors='tf')
```

You can check the size of the train encodings by running:

```
print("train_encodings.shape: {}".format(train_encodings["input_ids"].shape))
```

which will give:

```
train_encodings.shape: (87599, 512)
```

The maximum sequence length in our dataset is 512. Therefore, we see that the maximum length of the sequences is 512. Once we tokenize our data, we need to perform one more data processing step. Our `answer_start` and `answer_end` indices are character-based. However, since we are working with tokens, we need to convert our character-based indices to token-based indices. We will define a function for that:

```
def replace_char_with_token_indices(encodings, answers):
    start_positions = []
    end_positions = []
    n_updates = 0

    # Go through all the answers
    for i in range(len(answers)):
        # Get the token position for both start end char positions
        start_positions.append(encodings.char_to_token(i,
                                                        answers[i]['answer_start']))
        end_positions.append(encodings.char_to_token(i,
                                                        answers[i]['answer_end'] - 1))

        if start_positions[-1] is None or end_positions[-1] is None:
            n_updates += 1

        # if start position is None, the answer passage has been truncated
        # In the guide, https://huggingface.co/transformers/custom_
        # datasets.html#qa-squad they set it to model_max_length, but
        # this will result in NaN losses as the last available label is
        # model_max_length-1 (zero-indexed)
        if start_positions[-1] is None:
            start_positions[-1] = tokenizer.model_max_length - 1

        if end_positions[-1] is None:
            end_positions[-1] = tokenizer.model_max_length - 1

    print("{} / {} had answers truncated".format(n_updates,
                                                len(answers)))
    encodings.update({'start_positions': start_positions,
```

```
'end_positions': end_positions})
```

This function takes in a set of `BatchEncodings` called `encodings` generated by the tokenizer and a set of answers (a list of dictionaries). Then it updates the provided encodings with two new keys: `start_positions` and `end_positions`. These keys respectively hold the token-based indices denoting the start and end of the answer. If the answer is not found, we set the start and end indices to the last token. To convert our existing character-based indices to token-based indices, we use a function called `char_to_token()` provided by the `BatchEncodings` class. It takes a character index as the input and provides the corresponding token index as the output. With the function defined, let's call it on our training and testing data:

```
replace_char_with_token_indices(train_encodings, train_answers)  
replace_char_with_token_indices(test_encodings, test_answers)
```

With the clean data, we will now define a TensorFlow dataset. Note that this function modifies the encodings in place.

Defining a TensorFlow dataset

Next, let's implement a TensorFlow dataset to generate the data for the model. Our data will consist of two tuples: one containing inputs and the other containing the targets. The input tuple contains:

- Input token IDs – A batch of padded token IDs of size [batch size, sequence length]
- Attention mask – A batch of attention masks of size [batch size, sequence length]

The output tuple contains:

- Start index of the answer – A batch of start indices of the answer
- End index of the answer – A batch of end indices of the answer

We will first define a generator that generates the data in this format:

```
def data_gen(input_ids, attention_mask, start_positions, end_positions):  
    """ Generator for data """  
    for inps, attn, start_pos, end_pos in zip(input_ids,  
                                              attention_mask, start_positions, end_positions):  
        yield (inps, attn), (start_pos, end_pos)
```

Since we have already processed the data, it's a matter of reorganizing the already existing data to return using the code above.

Next, we will define a partial function that we can simply call without passing any arguments:

```
# Define the generator as a callable
train_data_gen = partial(data_gen,
    input_ids=train_encodings['input_ids'], attention_mask=train_
    encodings['attention_mask'],
    start_positions=train_encodings['start_positions'],
    end_positions=train_encodings['end_positions']
)
```

This function is then passed to the `tf.data.Dataset.from_generator()` function:

```
# Define the dataset
train_dataset = tf.data.Dataset.from_generator(
    train_data_gen, output_types=(('int32', 'int32'), ('int32', 'int32'))
)
```

We then shuffle the data in our training dataset. When shuffling a TensorFlow dataset we need to provide a buffer size. The buffer size defines how many samples are chosen to shuffle. Here we set that to 1,000 samples:

```
# Shuffling the data
train_dataset = train_dataset.shuffle(1000)
print('\tDone')
```

Next, we split our dataset into two: a training set and a validation dataset. We will use the first 10,000 samples as the validation set. The rest of the data is used as the training set. Both datasets will be batched using a batch size of 4:

```
# Valid set is taken as the first 10000 samples in the shuffled set
valid_dataset = train_dataset.take(10000)
valid_dataset = valid_dataset.batch(4)
```

```
# Rest is kept as the training data
train_dataset = train_dataset.skip(10000)
train_dataset = train_dataset.batch(4)
```

Finally, we follow the same procedure to create the test dataset:

```
# Creating test data
print("Creating test data")

# Define the generator as a callable
test_data_gen = partial(data_gen,
    input_ids=test_encodings['input_ids'],
    attention_mask=test_encodings['attention_mask'],
    start_positions=test_encodings['start_positions'],
    end_positions=test_encodings['end_positions']
)
test_dataset = tf.data.Dataset.from_generator(
    test_data_gen, output_types=((('int32', 'int32'), ('int32',
    'int32'))
)
test_dataset = test_dataset.batch(8)
```

Now let's see how BERT's architecture can be used to answer questions.

BERT for answering questions

There are a few modifications introduced on top of the pre-trained BERT model to leverage it for question-answering. First, the model takes in a question, followed by a context. As we discussed before, the context may or may not contain the answer to the question. The input has the format [CLS] <question tokens> [SEP] <context tokens> [SEP]. Then, for each token position of the context, we have two classification heads predicting a probability. One head predicts the probability of each context token being the start of the answer, whereas the other one predicts the probability of each context token being the end of the answer.

Once we figure out the start and end indices of the answer, we can simply extract the answer from the context using those indices.

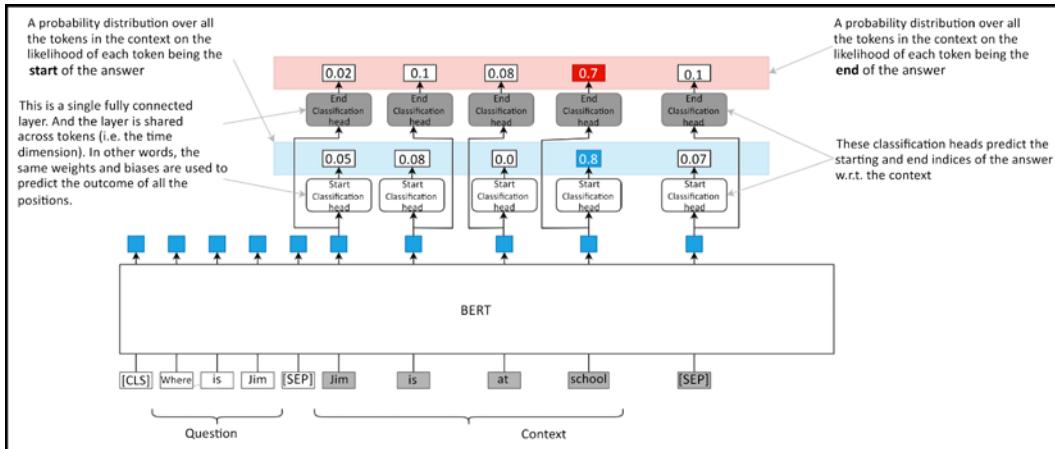


Figure 10.11: Using BERT for question-answering. The model takes in a question followed by a context. The model has two heads: one to predict the probability of each token in the context being the start of the answer and another to predict the end of the answer for each context token.

Defining the config and the model

In Hugging Face, you have several variants of each Transformer model. These variants are based on different tasks solved by these models. For example, for BERT we have:

- `TFBertForPretraining` – The pre-trained model without a task-specific head
- `TFBertForSequenceClassification` – Used for classifying a sequence of text
- `TFBertForTokenClassification` – Used for classifying each token in the sequence of text
- `TFBertForMultipleChoice` – Used for answering multiple-choice questions
- `TFBertForQuestionAnswering` – Used for extracting answers to a question from a given context
- `TFBertForMaskedLM` – Used for pre-training BERT on the masked language modeling task
- `TFBertForNextSentencePrediction` – Used for pre-training BERT to predict the next sentence

Here, we are interested in `TFBertForQuestionAnswering`. Let's import this class along with the `BertConfig` class, which we will extract important hyperparameters from:

```
from transformers import BertConfig, TFBertForQuestionAnswering
```

To get the pre-trained config, we call the `from_pretrained()` function of `BertConfig` with the model we're interested in. Here, we'll use the `bert-base-uncased` model:

```
config = BertConfig.from_pretrained("bert-base-uncased", return_dict=False)
```

You can print the `config` and see what's in there:

```
BertConfig {  
    "architectures": [  
        "BertForMaskedLM"  
    ],  
    "attention_probs_dropout_prob": 0.1,  
    "classifier_dropout": null,  
    "gradient_checkpointing": false,  
    "hidden_act": "gelu",  
    "hidden_dropout_prob": 0.1,  
    "hidden_size": 768,  
    "initializer_range": 0.02,  
    "intermediate_size": 3072,  
    "layer_norm_eps": 1e-12,  
    "max_position_embeddings": 512,  
    "model_type": "bert",  
    "num_attention_heads": 12,  
    "num_hidden_layers": 12,  
    "pad_token_id": 0,  
    "position_embedding_type": "absolute",  
    "return_dict": false,  
    "transformers_version": "4.15.0",  
    "type_vocab_size": 2,  
    "use_cache": true,  
    "vocab_size": 30522  
}
```

Finally, we get the model by calling the same function `from_pretrained()` from the `TFBertForQuestionAnswering` class and pass the `config` we just obtained:

```
model = TFBertForQuestionAnswering.from_pretrained("bert-base-uncased",  
    config=config)
```

When you run this, you will get a warning saying:

```
All model checkpoint layers were used when initializing
TFBertForQuestionAnswering.

Some layers of TFBertForQuestionAnswering were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized: ['qa_
outputs']

You should probably TRAIN this model on a down-stream task to be able to
use it for predictions and inference.
```

This is expected and totally fine. It's saying that there are some layers that have not been initialized from the pre-trained model; the output heads of the model need to be introduced as new layers, thus they are not pre-initialized.

After that, we will define a function that will wrap the returned model as a `tf.keras.models.Model` object. We need to perform this step because if we try to use the model as it is, TensorFlow returns the following error:

```
TypeError: The two structures don't have the same sequence type.
Input structure has type <class 'tuple'>, while shallow structure has
type
<class 'transformers.modeling_tf_outputs.
TFQuestionAnsweringModelOutput'>.
```

Therefore, we will define two input layers: one takes in the input token IDs and the other takes the attention mask and passes it to the model. Finally, we get the output of the model. We then define a `tf.keras.models.Model` using these inputs and output:

```
def tf_wrap_model(model):
    """ Wraps the huggingface's model with in the Keras Functional API """

    # Define inputs
    input_ids = tf.keras.layers.Input([None,], dtype=tf.int32,
                                    name="input_ids")
    attention_mask = tf.keras.layers.Input([None,], dtype=tf.int32,
                                         name="attention_mask")

    # Define the output (TFQuestionAnsweringModelOutput)
    out = model([input_ids, attention_mask])
```

```
# Get the correct attributes in the produced object to generate an
# output tuple
wrap_model = tf.keras.models.Model([input_ids, attention_mask],
outputs=(out.start_logits, out.end_logits))

return wrap_model
```

As we learned when studying the structure of the model, the question-answering BERT has two heads: one to predict the starting index of the answer and the other to predict the end. Therefore, we have to optimize two losses coming from the two heads. This means we need to add the two losses to get the final loss. When we have a multi-output model such as this, we can pass multiple loss functions aimed at each output head. Here, we define a single loss function. This means the same loss will be used across both heads and will be summed to generate the final loss:

```
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
acc = tf.keras.metrics.SparseCategoricalAccuracy()
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-5)

model_v2 = tf_wrap_model(model)
model_v2.compile(optimizer=optimizer, loss=loss, metrics=[acc])
```

We will now see how we can train and evaluate our model on the question-answering task.

Training and evaluating the model

We already have the data prepared and the model defined. Training the model is quite easy, and is just a one-liner:

```
model_v2.fit(
    train_dataset,
    validation_data=valid_dataset,
    epochs=3
)
```

You should see an output as follows:

```
Epoch 1/2
19400/19400 [=====] - 7175s 369ms/step
```

```

- loss: 2.7193 - tf_bert_for_question_answering_loss: 1.4153 - tf_
bert_for_question_answering_1_loss: 1.3040 - tf_bert_for_question_
answering_sparse_categorical_accuracy: 0.5975 - tf_bert_for_question_
answering_1_sparse_categorical_accuracy: 0.6376 - val_loss: 2.1615
- val_tf_bert_for_question_answering_loss: 1.0898 - val_tf_bert_for_
question_answering_1_loss: 1.0717 - val_tf_bert_for_question_answering_
sparse_categorical_accuracy: 0.7120 - val_tf_bert_for_question_
answering_1_sparse_categorical_accuracy: 0.7350
Epoch 2/2
19400/19400 [=====] - 7192s 370ms/step
- loss: 1.6691 - tf_bert_for_question_answering_loss: 0.8865 - tf_
bert_for_question_answering_1_loss: 0.7826 - tf_bert_for_question_
answering_sparse_categorical_accuracy: 0.7245 - tf_bert_for_question_
answering_1_sparse_categorical_accuracy: 0.7646 - val_loss: 2.1836
- val_tf_bert_for_question_answering_loss: 1.0988 - val_tf_bert_for_
question_answering_1_loss: 1.0847 - val_tf_bert_for_question_answering_
sparse_categorical_accuracy: 0.7289 - val_tf_bert_for_question_
answering_1_sparse_categorical_accuracy: 0.7504
It took 14366.591783046722 seconds to complete the training

```

You should see the accuracy on the validation set reaching an accuracy between ~73 and 75%. This is quite high, given we only trained the model for two epochs. This performance can be attributed to the high level of language understanding the pre-trained model already had when we downloaded it. Let's evaluate the model on our test data:

```
model_v2.evaluate(test_dataset)
```

It should output the following:

```

1322/1322 [=====] - 345s 261ms/step - loss: 2.2205
- tf_bert_for_question_answering_loss: 1.1325 - tf_bert_for_question_
answering_1_loss: 1.0881 - tf_bert_for_question_answering_sparse_
categorical_accuracy: 0.6968 - tf_bert_for_question_answering_1_sparse_
categorical_accuracy: 0.7250

```

We see that it performs comparably well on the test dataset as well. Finally, we can save the model. We will save the `TFBertForQuestionAnswering` component of the model. We'll also save the tokenizer:

```

import os

# Create folders

```

```
if not os.path.exists('models'):
    os.makedirs('models')
if not os.path.exists('tokenizers'):
    os.makedirs('tokenizers')

# Save the model
model_v2.get_layer("tf_bert_for_question_answering").save_pretrained(os.
path.join('models', 'bert_qa'))

# Save the tokenizer
tokenizer.save_pretrained(os.path.join('tokenizers', 'bert_qa'))
```

We have trained our model and evaluated it to ensure the model performs well. Once we confirmed that the model is performing well, we finally saved it for future use. Next, let's discuss how we can use this model to generate answers for a given question.

Answering questions with Bert

Let's now write a simple script to generate answers to questions from the trained model. First, let's define a sample question to generate an answer for. We'll also store the inputs and the ground truth answer to compare:

```
i = 5

# Define sample question
sample_q = test_questions[i]
# Define sample context
sample_c = test_contexts[i]
# Define sample answer
sample_a = test_answers[i]
```

Next, we'll define the inputs to the model. The input to the model needs to have a batch dimension. Therefore we use the `[i:i+1]` syntax to make sure the batch dimension is not flattened:

```
# Get the input in the format BERT accepts
sample_input = (test_encodings["input_ids"][i:i+1],
test_encodings["attention_mask"][i:i+1])
```

Let's now define a simple function called `ask_bert` to find an answer from the context for a given question. This function takes in an input, a tokenizer, and a model.

Then it generates the token IDs from the tokenizer, passes them to the model, outputs the start and end indices for the answer, and finally extracts the corresponding answer from the text of the context:

```
def ask_bert(sample_input, tokenizer, model):
    """ This function takes an input, a tokenizer, a model and returns the
    predicton """
    out = model.predict(sample_input)
    pred_ans_start = tf.argmax(out[0][0])
    pred_ans_end = tf.argmax(out[1][0])
    print("{}-{} token ids contain the answer".format(pred_ans_start,
    pred_ans_end))
    ans_tokens = sample_input[0][0][pred_ans_start:pred_ans_end+1]

    return " ".join(tokenizer.convert_ids_to_tokens(ans_tokens))
```

Let's execute the following lines to print the answer given by our model:

```
print("Question")
print("\t", sample_q, "\n")
print("Context")
print("\t", sample_c, "\n")
print("Answer (char indexed)")
print("\t", sample_a, "\n")
print('*50, '\n')

sample_pred_ans = ask_bert(sample_input, tokenizer, model_v2)

print("Answer (predicted)")
print(sample_pred_ans)
print('*50, '\n')
```

which will print:

```
Question
What was the theme of Super Bowl 50?

Context
Super Bowl 50 was an American football game to determine the champion
of the National Football League (NFL) for the 2015 season. The American
```

```
Football Conference (AFC) champion Denver Broncos defeated the National Football Conference (NFC) champion Carolina Panthers 24–10 to earn their third Super Bowl title. The game was played on February 7, 2016, at Levi's Stadium in the San Francisco Bay Area at Santa Clara, California. As this was the 50th Super Bowl, the league emphasized the "golden anniversary" with various gold-themed initiatives, as well as temporarily suspending the tradition of naming each Super Bowl game with Roman numerals (under which the game would have been known as "Super Bowl L"), so that the logo could prominently feature the Arabic numerals 50.
```

```
Answer (char indexed)
```

```
{'answer_start': 487, 'text': '"golden anniversary"', 'answer_end': 507}
```

```
=====
```

```
98-99 token ids contain the answer
```

```
Answer (predicted)
```

```
golden anniversary
```

```
=====
```

We can see that BERT has answered the question correctly. We have learned a lot about Transformers in general as well as BERT-specific architecture. We then used this knowledge to adapt BERT to solve a question-answering problem. Here we end our discussion about Transformers and BERT.

Summary

In this chapter, we talked about Transformer models. First, we looked at the Transformer at a microscopic level to understand the inner workings of the model. We saw that Transformers use self-attention, a powerful technique to attend to other inputs in the text sequences while processing one input. We also saw that Transformers use positional embeddings to inform the model about the relative position of tokens in addition to token embeddings. We also discussed that Transformers leverage residual connections (that is, shortcut connections) and layer normalization in order to improve model training.

We then discussed BERT, an encoder-based Transformer model. We looked at the format of the data accepted by BERT and the special tokens it uses in the input. Next, we discussed four different types of task BERT can solve: sequence classification, token classification, multiple-choice, and question-answering.

Finally, we looked at how BERT is pre-trained on a large corpus of text.

After that, we started on a use case: answering questions with BERT. To implement the solution, we used the `transformers` library by Hugging Face. It's an extremely useful high-level library built on top of deep learning frameworks such as TensorFlow, PyTorch, and Jax. The `transformers` library is specifically designed for quickly loading and using pre-trained Transformer models. In this use case, we first processed the data and created a `tf.data.Dataset` to stream the data in batches. Then we trained the model on that data and evaluated it on a test set. Finally, we used the model to infer answers to a sample question given to the model.

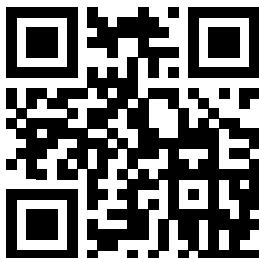
In the next chapter, we will learn a bit more about Transformers and how they can be used in a more complicated task that involves both images and text: image caption generation.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside

more than 1000 members at: <https://packt.link/nlp>



11

Image Captioning with Transformers

Transformer models changed the playing field for many NLP problems. They have redefined the state of the art by a significant margin, compared to the previous leaders: RNN-based models. We have already studied Transformers and understood what makes them tick. Transformers have access to the whole sequence of items (e.g. a sequence of tokens), as opposed to RNN-based models that look at one item at a time, making them well-suited for sequential problems. Following their success in the field of NLP, researchers have successfully used Transformers to solve computer vision problems. Here we will learn how to use Transformers to solve a multi-modal problem involving both images and text: image captioning.

Automated image captioning, or image annotation, has a wide variety of applications. One of the most prominent applications is image retrieval in search engines. Automated image captioning can be used to retrieve all the images belonging to a certain class (for example, a cat) as per the user's request. Another application can be in social media where, when an image is uploaded by a user, the image is automatically captioned so that the user can either refine the generated caption or post it as it is.

In this chapter, we will learn to caption images using machine learning, where a model is trained to generate a sequence of tokens (i.e. a caption) when given an image. We will first understand how Transformer models are used in computer vision, and then extend our understanding to solve the problem of generating captions for images. For generating captions for images, we will use a popular dataset for image captioning tasks known as **Microsoft Common Objects in Context (MS-COCO)**.

Solving this will require two Transformer models: one to generate an image representation and the other to generate the relevant caption. Once the image representation is generated, it will be fed as one of the inputs to the text-based Transformer model. The text-based Transformer model will be trained to predict the next token in the caption given the current caption, at a given time step.

We will generate three datasets: training, validation, and testing datasets. We use the training dataset to train the model and the validation set to monitor performance during training. Finally we use the test dataset to generate captions for a set of unseen images.

Looking at the image caption generation pipeline at a very high level, we have two main components:

1. A pretrained Vision Transformer model that takes in an image and produces a 1D hidden representation of the image
2. A text-based Transformer decoder model that can decode the hidden image representation to a series of token IDs

We will use a pretrained Transformer model to generate image representations. Known as the Vision Transformer (ViT), it has been trained on the ImageNet dataset and has delivered great performance on the ImageNet classification task.

Specifically, this chapter will cover the following main topics:

- Getting to know the data
- Downloading the data
- Processing and tokenizing data
- Defining a `tf.data.Dataset`
- The machine learning pipeline for image caption generation
- Implementing the model with TensorFlow
- Training the model
- Evaluating the results quantitatively
- Evaluating the model
- Captions generated for test images

Getting to know the data

Let's first understand the data we are working with both directly and indirectly. There are two datasets we will rely on:

- The ILSVRC ImageNet dataset (<http://image-net.org/download>)
- The MS-COCO dataset (<http://cocodataset.org/#download>)

We will not engage the first dataset directly, but it is essential for caption learning. This dataset contains images and their respective class labels (for example, cat, dog, and car). We will use a CNN that is already trained on this dataset, so we do not have to download and train on this dataset from scratch. Next we will use the MS-COCO dataset, which contains images and their respective captions. We will directly learn from this dataset by mapping the image to a fixed-size feature vector, using the Vision Transformer, and then map this vector to the corresponding caption using a text-based Transformer (we will discuss this process in detail later).

ILSVRC ImageNet dataset

ImageNet is an image dataset that contains a large set of images (~1 million) and their respective labels. These images belong to 1,000 different categories. This dataset is very expressive and contains almost all the objects found in the images we want to generate captions for. *Figure 11.1* shows some of the classes available in the ImageNet dataset:



Figure 11.1: A small sample of the ImageNet dataset

ImageNet is a good dataset to train on, in order to obtain image encodings that are required for caption generation. We say we use this dataset indirectly because we will use a pretrained Transformer that is trained on this dataset. Therefore, we will not be downloading, nor training the model on this dataset, by ourselves.

The MS-COCO dataset

Now we will move on to the dataset that we will actually be using, which is called **MS-COCO** (short for **M**icrosoft - **C**ommon **O**bjects in **C**ontext). We will use the training dataset from the year 2014 and the validation set from 2017. We use datasets belonging to different times to avoid using large datasets for this exercise. As described earlier, this dataset consists of images and their respective descriptions. The dataset is quite large (for example, the training dataset consists of ~120,000 samples and can measure over 15 GB). Datasets are updated every year, and a competition is then held to recognize the team that achieves state-of-the-art performance. Using the full dataset is important when the objective is to achieve state-of-the-art performance. However, in our case, we want to learn a reasonable model that is able to suggest what is in an image generally. Therefore, we will use a smaller dataset (~40,000 images and ~200,000 captions) to train our model. *Figure 11.2* includes some of the samples available:



- A woman stands in the dining area at the table.
- A room with chairs, a table, and a woman in it.
- A woman standing in a kitchen by a window
- A person standing at a table in a room.
- A living area with a television and a table



- A big burly grizzly bear is show with grass in the background.
- The large brown bear has a black nose.
- Closeup of a brown bear sitting in a grassy area.
- A large bear that is sitting on grass.
- A close up picture of a brown bear's face.



- Bedroom scene with a bookcase, blue comforter and window.
- A bedroom with a bookshelf full of books.
- This room has a bed with blue sheets and a large bookcase
- A bed and a mirror in a small room.
- a bed room with a neatly made bed a window and a book shelf



- A stop sign is mounted upside-down on its post.
- A stop sign that is hanging upside down.
- An upside down stop sign by the road.
- a stop sign put upside down on a metal pole
- A stop sign installed upside down on a street corner



- Three teddy bears, each a different color, snuggling together.
- Three stuffed animals are sitting on a bed.
- three teddy bears giving each other a hug
- A group of three stuffed animal teddy bears.
- Three stuffed bears hugging and sitting on a blue pillow

Figure 11.2: A small sample of the MS-COCO dataset

For learning with and testing our end-to-end image caption generation model, we will use the 2017 validation dataset, provided on the official MS-COCO dataset website.

Note

In practice, you should use separate datasets for testing and validation, to avoid data leakage during testing. Using the same data for validation and testing can lead the model to incorrectly represent its generalizability to the real world.

In *Figure 11.3*, we can see some of the images found in the validation set. These are some hand-picked examples from the validation set representing a variety of different objects and scenes:



Figure 11.3: Unseen images that we will use to test the image caption generation capability of our algorithm

Downloading the data

The MS-COCO dataset we will be using is quite large. Therefore, we will manually download these datasets. To do that, follow the instructions below:

1. Create a folder called `data` in the `Ch11-Image-Caption-Generation` folder
2. Download the 2014 Train images set (<http://images.cocodataset.org/zips/train2014.zip>) containing 83K images (`train2014.zip`)
3. Download the 2017 Val images set (<http://images.cocodataset.org/zips/val2017.zip>) containing 5K images (`val2017.zip`)

4. Download the annotation sets for 2014 (`annotations_trainval2014.zip`) (http://images.cocodataset.org/annotations/annotations_trainval2014.zip) and 2017 (`annotations_trainval2017.zip`) (http://images.cocodataset.org/annotations/annotations_trainval2017.zip)
5. Copy the downloaded zip files to the `Ch11-Image-Caption-Generation/data` folder
6. Extract the zip files using the **Extract to** option so that it unzips the content within a sub-folder

Once you complete the above steps, you should have the following subfolders:

- `data/train2014` – Contains the training images
- `data/annotations_trainval2014` – Contains the captions of the training images
- `data/val2017` – Contains the validation images
- `data/annotations_trainval2017` – Contains the captions of the validation images

Processing and tokenizing data

With the data downloaded and placed in the correct folders, let's define the directories containing the required data:

```
trainval_image_dir = os.path.join('data', 'train2014', 'train2014')
trainval_captions_dir = os.path.join('data', 'annotations_trainval2014',
'annotations')
test_image_dir = os.path.join('data', 'val2017', 'val2017')
test_captions_dir = os.path.join('data', 'annotations_trainval2017',
'annotations')

trainval_captions_filepath = os.path.join(trainval_captions_dir,
'captions_train2014.json')
test_captions_filepath = os.path.join(test_captions_dir, 'captions_'
val2017.json')
```

Here we have defined the directories containing training and testing images as well as the file paths of the JSON files that contain the captions of the training and testing images.

Preprocessing data

As the next step, let's split the training set in to train and validation sets. We will use 80% of the original set as training data and 20% as the validation data (randomly chosen):

```
all_filepaths = np.array([os.path.join(trainval_image_dir, f) for f in
os.listdir(trainval_image_dir)])
rand_indices = np.arange(len(all_filepaths))
np.random.shuffle(rand_indices)

split = int(len(all_filepaths)*0.8)

train_filepaths, valid_filepaths = all_filepaths[rand_indices[:split]],
all_filepaths[rand_indices[split:]]
```

We can print the dataset sizes and see what we ended up with:

```
print(f"Train dataset size: {len(train_filepaths)}")
print(f"Valid dataset size: {len(valid_filepaths)}")
```

This will print:

```
Train dataset size: 66226
Valid dataset size: 16557
```

Now let's read the captions and create a pandas DataFrame using them. Our DataFrame will have four important columns:

- `image_id` – Identifies an image (used to generate the file path)
- `image_filepath` – File location of the image identified by `image_id`
- `caption` – Original caption
- `preprocessed_caption` – Caption after some simple preprocessing

First we will load the data in the JSON file and get the data into a DataFrame:

```
with open(trainval_captions_filepath, 'r') as f:
    trainval_data = json.load(f)

trainval_captions_df = pd.json_normalize(trainval_data, "annotations")
```

The data we're looking for in the file is found under a key called "annotations". Under "annotations" we have a list of dictionaries each having the `image_id`, `id`, and `caption`. The function `pd.json_normalize()` takes in the loaded data and converts that to a `pd.DataFrame`.

We then create the column called `image_filepath` by prefixing the root directory path to the `image_id` and appending the extension `.jpg`.

We will only keep the data points where the `image_filepath` values are in the training images we stored in `train_filepaths`:

```
trainval_captions_df["image_filepath"] = trainval_captions_df["image_id"].apply(  
    lambda x: os.path.join(trainval_image_dir,  
        'COCO_train2014_'+format(x, '012d')+'.jpg')  
)  
train_captions_df = trainval_captions_df[trainval_captions_df["image_  
filepath"].isin(train_filepaths)]
```

We now define a function called `preprocess_captions()` that processes the original caption:

```
def preprocess_captions(image_captions_df):  
    """ Preprocessing the captions """  
  
    image_captions_df["preprocessed_caption"] = "[START] " +  
    image_captions_df["caption"].str.lower().str.replace('[^\w\s]', '')  
    + " [END]"  
    return image_captions_df
```

In the above code, we:

- Added two special tokens, `[START]` and `[END]`, to denote the start and the end of each caption respectively
- Converted the captions to lowercase
- Removed everything that is not a word, character, or space

We then call this function on the training dataset:

```
train_captions_df = preprocess_captions(train_captions_df)
```

We then follow a similar process for both validation and test data:

```
valid_captions_df = trainval_captions_df[  
    trainval_captions_df[  
        "image_filepath"  
    ].isin(valid_filepaths)  
]  
valid_captions_df = preprocess_captions(valid_captions_df)  
  
with open(test_captions_filepath, 'r') as f:  
    test_data = json.load(f)
```

```

test_captions_df = pd.json_normalize(test_data, "annotations")
test_captions_df["image_filepath"] = test_captions_df["image_id"].apply(
    lambda x: os.path.join(test_image_dir, format(x, '012d')+'.jpg'))
)
test_captions_df = preprocess_captions(test_captions_df)

```

Let's check the data in `training_captions_df` (*Figure 11.4*):

	image_id	id	caption	image_filepath	preprocessed_caption
0	318556	48	A very clean and well decorated empty bathroom	data\train2014\train2014\COCO_train2014_000000...	[START] a very clean and well decorated empty ...
1	116100	67	A panoramic view of a kitchen and all of its a...	data\train2014\train2014\COCO_train2014_000000...	[START] a panoramic view of a kitchen and all ...
2	318556	126	A blue and white bathroom with butterfly theme...	data\train2014\train2014\COCO_train2014_000000...	[START] a blue and white bathroom with butter...
3	116100	148	A panoramic photo of a kitchen and dining room	data\train2014\train2014\COCO_train2014_000000...	[START] a panoramic photo of a kitchen and din...
4	379340	173	A graffiti-ed stop sign across the street from...	data\train2014\train2014\COCO_train2014_000000...	[START] a graffitied stop sign across the stre...

Figure 11.4: Data contained in training_captions_df

This data shows important information such as where the image is located in the file structure, the original caption, and the preprocessed caption.

Let's also analyze some statistics about the images. We will take a small sample of the first 1,000 images from the training dataset and look at image sizes:

```

n_samples = 1000

train_image_stats_df = train_captions_df.loc[:n_samples, "image_"
                                             "filepath"].apply(lambda x: Image.open(x).size)
train_image_stats_df = pd.DataFrame(train_image_stats_df.tolist(),
                                     index=train_image_stats_df.index)
train_image_stats_df.describe()

```

This will produce *Figure 11.5*:

	0	1
count	808.000000	808.000000
mean	571.698020	499.007426
std	95.377281	100.333224
min	332.000000	182.000000
25%	480.000000	427.000000
50%	640.000000	480.000000
75%	640.000000	640.000000
max	640.000000	640.000000

Figure 11.5: Statistics about the size of the images in the training dataset

We can see that most images have a resolution of 640x640. We will later need to resize images to 224x224 to match the model's input requirements. We'll also look at our vocabulary size:

```
train_vocabulary = train_captions_df["preprocessed_caption"].str.split(" ")
    .explode().value_counts()
print(len(train_vocabulary[train_vocabulary>=25]))
```

This prints:

```
3629
```

This tells us that 3,629 words occur at least 25 times in our train dataset. We use this as our vocabulary size.

Tokenizing data

Since we are developing Transformer models, we need a robust tokenizer similar to the ones used by popular models like BERT. Hugging Face's `tokenizers` library provides us with a range of tokenizers that are easy to use. Let's understand how we can use one of these tokenizers for our purpose. You can import it using:

```
from tokenizers import BertWordPieceTokenizer
```

Next, let's define the `BertWordPieceTokenizer`. We will pass the following arguments when doing so:

- `unk_token` – Defines a token to be used for out-of-vocabulary words
- `clean_text` – Whether to perform simple preprocessing steps to clean text
- `lowercase` – Whether to lowercase the text

These arguments can be seen in the following:

```
# Initialize an empty BERT tokenizer
tokenizer = BertWordPieceTokenizer(
    unk_token="[UNK]",
    clean_text=False,
    lowercase=False,
)
```

With the tokenizer defined, we can call the `train_from_iterator()` function to train the tokenizer on our dataset:

```
tokenizer.train_from_iterator(  
    train_captions_df["preprocessed_caption"].tolist(),  
    vocab_size=4000,  
    special_tokens=["[PAD]", "[UNK]", "[START]", "[END]"]  
)
```

The `train_from_iterator()` function takes in several arguments:

- `iterator` – An iterable that produces a string (containing the caption) as one item.
- `vocab_size` – Size of the vocabulary.
- `special_tokens` – Special tokens that will be used in our data. Specifically we use `[PAD]` (to denote padding), `[UNK]` (to denote OOV tokens), `[START]` (to denote the start), and `[END]` (to denote the end). These tokens will get assigned lower IDs starting from 0.

Once the tokenizer is trained, we can use it to convert strings of text to sequences of tokens. Let's convert a few example sentences to sequences of tokens using the trained tokenizer:

```
# Encoding a sentence  
example_captions = valid_captions_df["preprocessed_caption"].iloc[:10].  
tolist()  
example_tokenized_captions = tokenizer.encode_batch(example_captions)  
  
for caption, tokenized_cap in zip(example_captions, example_tokenized_  
captions):  
    print(f"{caption} -> {tokenized_cap.tokens}")
```

This will print:

```
[START] an empty kitchen with white and black appliances [END] ->  
['[START]', 'an', 'empty', 'kitchen', 'with', 'white', 'and', 'black',  
'appliances', '[END]']  
[START] a white square kitchen with tile floor that needs repairs [END]  
-> ['[START]', 'a', 'white', 'square', 'kitchen', 'with', 'tile', 'floor',  
'that', 'need', '##s', 'rep', '##air', '##s', '[END]']  
[START] a few people sit on a dim transportation system [END] ->  
['[START]', 'a', 'few', 'people', 'sit', 'on', 'a', 'dim', 'transport',  
'##ation', 'system', '[END]']  
[START] a person protected from the rain by their umbrella walks down the  
road [END] -> ['[START]', 'a', 'person', 'prote', '##cted', 'from',
```

```
'the', 'rain', 'by', 'their', 'umbrella', 'walks', 'down', 'the', 'road',
'[END']']
[START] a white kitchen in a home with the light on [END] -> ['[START]',
'a', 'white', 'kitchen', 'in', 'a', 'home', 'with', 'the', 'light', 'on',
'[END']']
```

You can see how the tokenizer has learned its own vocabulary and is tokenizing string sentences. The words that contain ## in front mean they must be combined with the previous token (without spaces) to get the final result. For example, the final string from the tokens 'image', 'cap' and '###tion' is 'image caption'. Let's see which IDs the special tokens we defined are mapped to:

```
vocab = tokenizer.get_vocab()

for token in ["[UNK]", "[PAD]", "[START]", "[END]"]:
    print(f"{token} -> {vocab[token]}")
```

This will output:

```
[UNK] -> 1
[PAD] -> 0
[START] -> 2
[END] -> 3
```

Now let's look at how we can define a TensorFlow data pipeline using the processed data.

Defining a `tf.data.Dataset`

Now let's look at how we can create a `tf.data.Dataset` using the data. We will first write a few helper functions. Namely, we'll define:

- `parse_image()` to load and process an image from a `filepath`
- `generate_tokenizer()` to generate a tokenizer trained on the data passed to the function

First let's discuss the `parse_image()` function. It takes three arguments:

- `filepath` – Location of the image
- `resize_height` – Height to resize the image to
- `resize_width` – Width to resize the image to

The function is defined as follows:

```
def parse_image(filepath, resize_height, resize_width):
```

```
""" Reading an image from a given filepath """

# Reading the image
image = tf.io.read_file(filepath)
# Decode the JPEG, make sure there are 3 channels in the output
image = tf.io.decode_jpeg(image, channels=3)
image = tf.image.convert_image_dtype(image, tf.float32)
# Resize the image to 224x224
image = tf.image.resize(image, [resize_height, resize_width])

# Bring pixel values to [-1, 1]
image = image*2.0 - 1.0

return image
```

We are mostly relying on `tf.image` functions to load and process the image. This function specifically:

- Reads the image from the `filepath`
- Decodes the bytes in the JPEG image to a `uint8` tensor and converts to a `float32 dtype` tensor.

By the end of these steps, we'll have an image whose pixel values are between 0 and 1. Next, we:

- Resize the image to a given height and width
- Finally normalize the image so that the pixel values are between -1 and 1 (as required by the ViT model we'll be using)

With that we define the second helper function. This function encapsulates the functionality of the `BertWordPieceTokenizer` we have discussed previously:

```
def generate_tokenizer(captions_df, n_vocab):
    """ Generate the tokenizer with given captions """

    # Define the tokenizer
    tokenizer = BertWordPieceTokenizer(
        unk_token="[UNK]",
        clean_text=False,
        lowercase=False,
    )
```

```

# Train the tokenizer
tokenizer.train_from_iterator(
    captions_df["preprocessed_caption"].tolist(),
    vocab_size=n_vocab,
    special_tokens=["[PAD]", "[UNK]", "[START]", "[END]"]
)

return tokenizer

```

With that we can define our main data function to generate the TensorFlow data pipeline:

```

def generate_tf_dataset(
    image_captions_df, tokenizer=None, n_vocab=5000, pad_length=33, batch_
size=32, training=False
):
    """ Generate the tf.data.Dataset"""

    # If the tokenizer is not available, create one
    if not tokenizer:
        tokenizer = generate_tokenizer(image_captions_df, n_vocab)

    # Get the caption IDs using the tokenizer
    image_captions_df["caption_token_ids"] = [enc.ids for enc in
    tokenizer.encode_batch(image_captions_df["preprocessed_caption"])]

    vocab = tokenizer.get_vocab()

    # Add the padding to short sentences and truncate long ones
    image_captions_df["caption_token_ids"] =
    image_captions_df["caption_token_ids"].apply(
        lambda x: x+[vocab["[PAD]"]]*(pad_length - len(x) + 2) if
        pad_length + 2 >= len(x) else x[:pad_length + 1] + [x[-1]])
    )

    # Create a dataset with images and captions
    dataset = tf.data.Dataset.from_tensor_slices({
        "image_filepath": image_captions_df["image_filepath"],
        "caption_token_ids":
            np.array(image_captions_df["caption_token_ids"].tolist())
    })

```

```
})

# Each sample in our dataset consists of (image, caption token
# IDs, position IDs), (caption token IDs offset by 1)
dataset = dataset.map(
    lambda x: (
        (parse_image(x["image_filepath"], 224, 224),
         x["caption_token_ids"][:-1], tf.range(pad_length+1,
                                               dtype='float32')), x["caption_token_ids"]
    )
)

# Shuffle and batch data in the training mode
if training:
    dataset = dataset.shuffle(buffer_size=batch_size*10)

dataset = dataset.batch(batch_size)

return dataset, tokenizer
```

This function takes the following arguments:

- `image_captions_df` – A pandas DataFrame containing image file paths and processed captions
- `tokenizer` – An optional tokenizer that will be used to tokenize the captions
- `n_vocab` – The vocabulary size
- `pad_length` – The length to pad captions
- `batch_size` – Batch size to batch the data
- `training` – Whether the data pipeline should be run in training mode or not. In training mode, we shuffle data whereas otherwise, we do not

First this function generates a tokenizer if a new tokenizer has not been passed. Next we create a column called “`caption_token_ids`” in our DataFrame, which is created by calling the `encode_batch()` function of the tokenizer on the `preprocessed_caption` column. We then perform padding on the `caption_token_ids` column. We add the [PAD] token ID if a caption is shorter than `pad_length`, or truncate it if it’s longer. We then create a `tf.data.Dataset` using the `from_tensor_slices()` function.

Each sample in this dataset will be a dictionary with the key `image_filepath` and `caption_token_ids` and values containing corresponding values. Once we do this, we have the ingredients to get the actual data. We will call the `tf.data.Dataset.map()` function to:

- Call `parse_image()` on each `image_filepath` to produce the actual image
- Return all caption token IDs, except the last, as inputs
- A range from 0 to the number of tokens, representing position of each input token ID (used to get positional embeddings for the Transformer)
- Return all caption token IDs as the targets

Let's understand what the inputs and outputs are going to look like for an example. Say you have the caption *a brown bear*. Here's how the inputs and outputs going to look for our Transformer decoder (*Figure 11.6*):

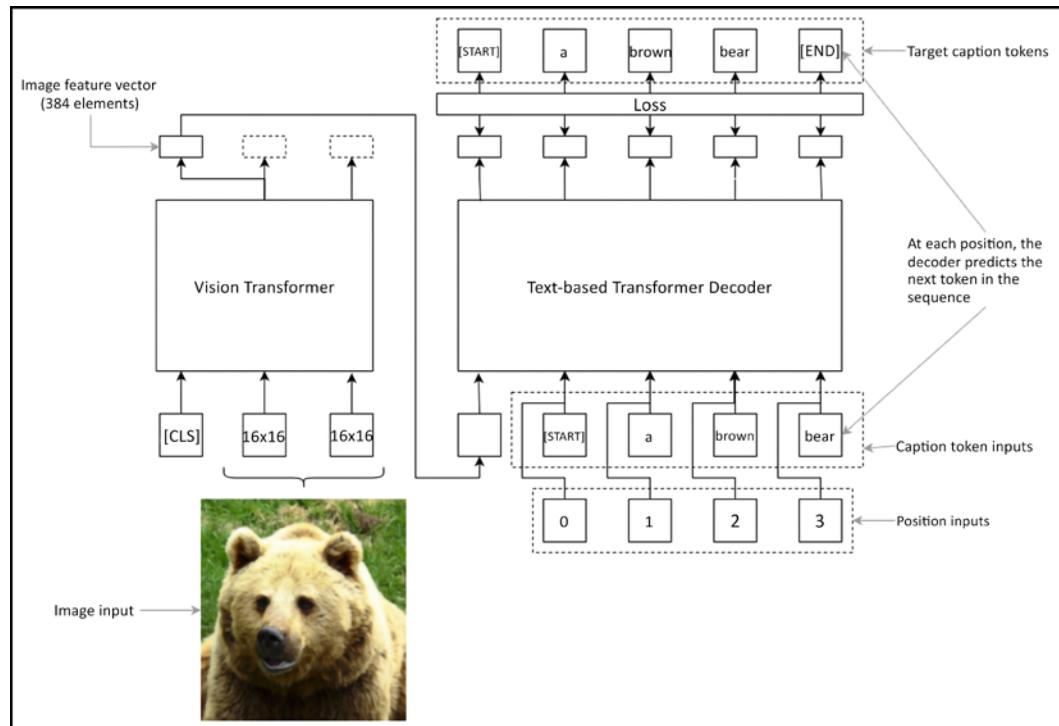


Figure 11.6: How inputs and targets are organized for the model

Finally, if in training mode, we shuffle the dataset using a `buffer_size` of 10 times the batch size. Then we batch the dataset using the `batch_size` provided when calling the function. Let's call this function on our training dataset to see what we get:

```
n_vocab=4000
batch_size=2
sample_dataset, sample_tokenizer = generate_tf_dataset(train_captions_df,
n_vocab=n_vocab, pad_length=10, batch_size=batch_size, training=True)
for i in sample_dataset.take(1):
    print(i)
```

Which will output:

```
(  
  (  
      <tf.Tensor: shape=(2, 224, 224, 3), dtype=float32, numpy=  
      array([[[[-0.2051357 , -0.22082198, -0.31493968],  
              [-0.2015593 , -0.21724558, -0.31136328],  
              [-0.17017174, -0.18585801, -0.2799757 ],  
              ...,  
              [-0.29620153, -0.437378 , -0.6155298 ],  
              [-0.28843057, -0.41392076, -0.6178423 ],  
              [-0.29654706, -0.43772352, -0.62483776]],  
  
                  [[-0.8097613 , -0.6725868 , -0.55734015],  
                  [-0.7580646 , -0.6420185 , -0.55782473],  
                  [-0.77606916, -0.67418844, -0.5419755 ],  
                  ...,  
                  [-0.6400192 , -0.4753132 , -0.24786222],  
                  [-0.70908225, -0.5426947 , -0.31580424],  
                  [-0.7206869 , -0.5324516 , -0.3128438 ]]]], dtype=float32)>,  
  
      <tf.Tensor: shape=(2, 11), dtype=int32, numpy=  
      array([[ 2,   24,   356,  114,   488,  1171,  1037,  2820,   566,  
        445,  116],  
             [ 2,   24, 1357, 2142,   63, 1473,   495,   282,   116,   24,  
            301]]>,  
  
      <tf.Tensor: shape=(2, 11), dtype=float32, numpy=  
      array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.],  
             [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]])]
```

```

        dtype=float32)>
),
<tf.Tensor: shape=(2, 12), dtype=int32, numpy=
array([[ 2,   24,  356,  114,  488, 1171, 1037, 2820,  566,  445,
116,
       3],
       [ 2,   24, 1357, 2142,   63, 1473,  495,  282,  116,   24,  301,
       3]])>
)

```

Here we can see the inputs and outputs organized into a nested tuple. It has the format ((image, input caption token IDs, position IDs), target caption token IDs). For example, we have produced a data pipeline with a batch size of 2, a pad length of 10, and a vocabulary size of 4,000. We can see the image batch has the shape [2, 224, 224, 3], input caption token IDs and the position IDs have the shape [2, 11], and finally, the target caption token IDs are of shape [2, 12]. It is important to note that we use an additional buffer for padding length to incorporate the [START] and [END] tags. Therefore, the resulting tensors use a caption length of 12 (i.e. 10+2). The most important thing to note here is the length of the input and target captions. Input captions have one item less than the target captions as shown by the lengths. This is because, the first item in our input captions would be the image feature vector. This brings the length of input tokens to be equal to the length of target tokens.

With the data pipeline out of the way, we will discuss the mechanics of the model we'll be using.

The machine learning pipeline for image caption generation

Here we will look at the image caption generation pipeline at a very high level and then discuss it piece by piece until we have the full model. The image caption generation framework consists of two main components:

- A pretrained Vision Transformer model to produce an image representation
- A text-based decoder model that can decode the image representation to a series of token IDs. This uses a text tokenizer to convert tokens to token IDs and vice versa

Though the Transformer models were initially used for text-based NLP problems, they have outgrown the domain of text data and have been used in other areas such as image data and audio data.

Here we will be using one Transformer model that can process image data and another that can process text data.

Vision Transformer (ViT)

First, let's look at the Transformer generating the encoded vector representations of images. We will be using a pretrained **Vision Transformer (ViT)** to achieve this. This model has been trained on the ImageNet dataset we discussed above. Let's understand the architecture of this model.

Originally, the ViT was proposed in the paper *An Image is Worth 16X16 Words: Transformers for Image Recognition at Scale* by Dosovitskiy et al (<https://arxiv.org/pdf/2010.11929.pdf>). This can be considered the first substantial step toward adapting Transformers for computer vision problems. This model is called the Vision Transformer model.

The idea is to decompose an image into small patches of 16x16 and consider each as a separate token. Each image patch is flattened to a 1D vector and their position is encoded by a positional encoding mechanism similar to the original Transformer. But images are 2D structures; is it enough to have 1D positional information, and not 2D positional information? The authors argue that a 1D positional encoding was adequate and 2D positional encoding did not provide a significant boost. Once the image is broken into patches of 16x16 and flattened, each image can be presented as a sequence of tokens, just like a textual input sequence (*Figure 11.7*).

Then the model is pretrained in a self-supervised fashion, using a vision dataset called JFT-300M (<https://paperswithcode.com/dataset/jft-300m>). The paper proposes an elegant way to train the ViT in a semi-supervised fashion using image data. Similar to how NLP problems represent a unit of text as a token, a token is a patch of an image (i.e. a sequence of continuous values where values are normalized pixels). Then the ViT is pretrained to predict the mean 3-bit RGB color of a given image patch. Each channel (i.e. red, green, and blue) is represented with 3 bits (each bit having a value of 0 or 1), which gives 512 possibilities or classes. In other words, for a given image, patches (similar to how tokens are treated in NLP) are masked randomly (using the same approach as BERT), and the model is asked to predict the mean 3-bit RGB color of that image patch.

After pretraining, the model can be fine-tuned for a task-specific problem by fitting a classification or a regression head on top of the ViT, just like BERT. The ViT also has the [CLS] token at the beginning of the sequence, which will be used as the input representation for downstream vision models that are plugged on top of the ViT.

Figure 11.7 illustrates the mechanics of the ViT:

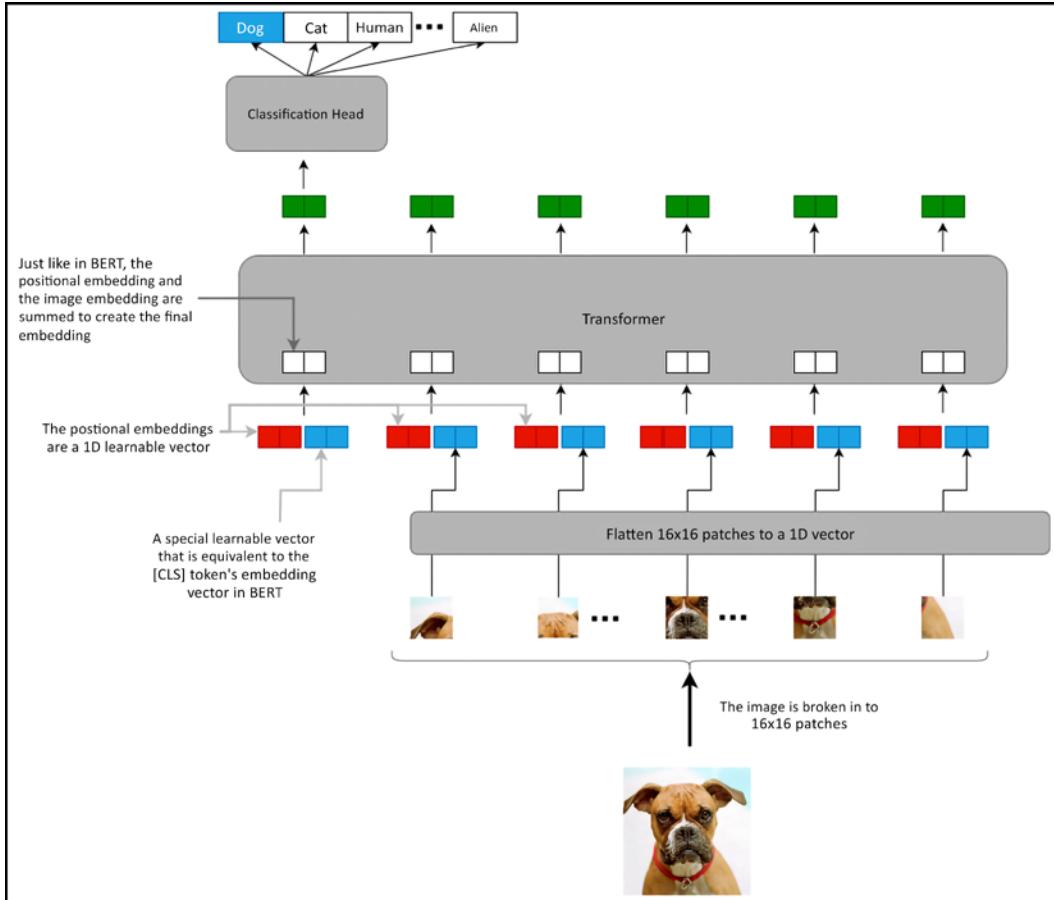


Figure 11.7: The Vision Transformer model

The model we'll be using here originated from the paper *How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers* by Steiner et al (<https://arxiv.org/pdf/2106.10270.pdf>). It proposes several variants of the ViT model. Specifically, we will use the ViT-S/16 architecture. ViT-S is the second smallest ViT model with 12 layers and a hidden output dimensionality of 384; in total it has 22.2M parameters. The number 16 here means that the model is trained on image patches of 16x16. The model has been fine-tuned using the ImageNet dataset we discussed earlier. We will use the feature extractor part of the model for the purpose of image captioning.

Text-based decoder Transformer

The text-based decoder's primary purpose is to predict the next token in the sequence given the previous tokens. This decoder is mostly similar to the BERT we used in the previous chapter. Let's refresh our memory on what the Transformer model is composed of. The Transformer consists of several stacked layers. Each layer has:

- A self-attention layer – Generates a hidden representation for each token position by taking in the input token and attending to the tokens at other positions in the sequence
- A fully connected subnetwork – Generates an element-wise non-linear hidden representation by propagating the self-attention layer's output through two fully connected layers

In addition to these, the network uses residual connections and layer normalization techniques to enhance performance. When speaking of inputs, the model uses two types of input embeddings to inform the model:

- Token embeddings – Each token is represented with an embedding vector that is jointly trained with the model
- Position embeddings – Each token position is represented by an ID and a corresponding embedding for that position

Compared to the BERT we used in the previous chapter, a key difference in our model is how we use the self-attention mechanism. When using BERT, the self-attention layer was able to pay attention in a bidirectional manner (i.e. pay attention to tokens on both sides of the current input). However, in the decoder-based model, it can only pay attention to the tokens to the left of the current token. In other words, the attention mechanism only has access to inputs seen up to the current input.

Putting everything together

Let's now learn how to put the two models together. We will use the following procedure to train the model end to end:

1. We generate the image encoding via the ViT model. It generates a single representation of 384 items for an image.
2. This representation, along with all the caption tokens except the last, goes into the decoder as the inputs.

- Given the current input token, the decoder predicts the next token. At the end of this process we will have the full image caption.

Another alternative for connecting the ViT and the text decoder models is by providing direct access to the ViT's full sequence of encoder outputs as a part of the attention mechanism of the decoder. In this work, not to overcomplicate our discussion, we only use a single output from the ViT model as an input to the decoder.

Implementing the model with TensorFlow

We will now implement the model we just studied. First let's import a few things:

```
import tensorflow_hub as hub
import tensorflow as tf
import tensorflow.keras.backend as K
```

Implementing the ViT model

Next, we are going to download the pretrained ViT model from TensorFlow Hub. We will be using a model submitted by Sayak Paul. The model is available at https://tfhub.dev/sayakpaul/vit_s16_fe/1. You can see other Vision Transformer models available at https://tfhub.dev/sayakpaul/collections/vision_transformer/1.

```
image_encoder = hub.KerasLayer("https://tfhub.dev/sayakpaul/vit_s16_fe/1",
trainable=False)
```

We then define an input layer to input images and pass that to the `image_encoder` to get the final feature vector for that image:

```
image_input = tf.keras.layers.Input(shape=(224, 224, 3))
image_features = image_encoder(image_input)
```

You can look at the size of the final image representation by running:

```
print(f"Final representation shape: {image_features.shape}")
```

which will output:

```
Final representation shape: (None, 384)
```

Next we will look at the details of how to implement the text-based Transformer model, which will take in the image representation to generate the image caption.

Implementing the text-based decoder

Here we will implement a Transformer decoder model from the ground up. This is different from how we used Transformer models before, where we downloaded a pretrained model and used them.

Before we implement the model itself, we are going to implement two custom Keras layers: one for the self-attention mechanism and the other one to capture the functionality of a single layer in the Transformer model. Let's start with the self-attention layer.

Defining the self-attention layer

Here we define the self-attention layer using the Keras subclassing API:

```
class SelfAttentionLayer(tf.keras.layers.Layer):
    """ Defines the computations in the self attention layer """

    def __init__(self, d):
        super(SelfAttentionLayer, self).__init__()
        # Feature dimensionality of the output
        self.d = d

    def build(self, input_shape):
        # Query weight matrix
        self.Wq = self.add_weight(
            shape=(input_shape[-1], self.d),
            initializer='glorot_uniform',
            trainable=True, dtype='float32'
        )
        # Key weight matrix
        self.Wk = self.add_weight(
            shape=(input_shape[-1], self.d),
            initializer='glorot_uniform',
            trainable=True, dtype='float32'
        )
        # Value weight matrix
        self.Wv = self.add_weight(
            shape=(input_shape[-1], self.d),
            initializer='glorot_uniform',
```

```

        trainable=True, dtype='float32'
    )

def call(self, q_x, k_x, v_x, mask=None):

    q = tf.matmul(q_x, self.Wq) #[None, t, d]
    k = tf.matmul(k_x, self.Wk) #[None, t, d]
    v = tf.matmul(v_x, self.Wv) #[None, t, d]

    # Computing the final output
    h = tf.keras.layers.Attention(causal=True)([
        q, #q
        v, #v
        k, #k
    ], mask=[None, mask])
    # [None, t, t] . [None, t, d] => [None, t, d]

    return h

```

Here we have to populate the logic for three functions:

- `__init__()` and `__build__()` – Define various hyperparameters and layer initialization specific logic
- `call()` – Computations that need to happen when the layer is called

You can see that we define the dimensionality of the attention output, d , as an argument to the `__init__()` method. Next in the `__build__()` method, we define three weight matrices, W_q , W_k , and W_v . If you remember our discussion from the previous chapter, these represent the weights of the query, key, and value respectively.

Finally, in the `call` method we have the logic. It takes four inputs: query, key, value inputs, and an optional mask for values. We then compute the latent q , k , and v by multiplying with the corresponding weight matrices W_q , W_k , and W_v . To compute attention, we will be using the out-of-the-box layer `tf.keras.layers.Attention`. We used a similar layer to compute the Bahdanau attention mechanism in *Chapter 9, Sequence-to-Sequence Learning – Neural Machine Translation*. The `tf.keras.layers.Attention()` layer has several arguments. One that we care about here is setting `causal=True`.

By doing this, we are instructing the layer to mask the tokens to the right of the current token. This essentially prevents the decoder from leaking information about future tokens. Next, the layer takes in the following arguments during the call:

- `inputs` – A list of inputs containing the query, value, and key in that order
- `mask` – A list of two items containing the masks for the query and value

Finally it returns the output of the attention layer `h`. Next, we will implement the computations of a Transformer layer.

Defining the Transformer layer

With the self-attention layer, let's capture the computations of a single Transformer layer in the following class. It uses self-attention, fully connected layers, and other optimization techniques to compute the output:

```
class TransformerDecoderLayer(tf.keras.layers.Layer):  
    """ The Decoder layer """  
  
    def __init__(self, d, n_heads):  
        super(TransformerDecoderLayer, self).__init__()  
        # Feature dimensionality  
        self.d = d  
  
        # Dimensionality of a head  
        self.d_head = int(d/n_heads)  
  
        # Number of heads  
        self.n_heads = n_heads  
  
        # Actual attention heads  
        self.attn_heads = [SelfAttentionLayer(self.d_head) for i in  
                          range(self.n_heads)]  
  
        # Fully connected Layers  
        self.fc1_layer = tf.keras.layers.Dense(512, activation='relu')  
        self.fc2_layer = tf.keras.layers.Dense(d)  
  
        self.add_layer = tf.keras.layers.Add()
```

```
        self.norm1_layer = tf.keras.layers.LayerNormalization()
        self.norm2_layer = tf.keras.layers.LayerNormalization()

    def _compute_multihead_output(self, x):
        """ Computing the multi head attention output"""
        outputs = [head(x, x, x) for head in self.attn_heads]
        outputs = tf.concat(outputs, axis=-1)
        return outputs

    def call(self, x):

        # Multi head attention layer output
        h1 = self._compute_multihead_output(x)

        h1_add = self.add_layer([x, h1])
        h1_norm = self.norm1_layer(h1_add)

        # Fully connected outputs
        h2_1 = self.fc1_layer(h1_norm)
        h2_2 = self.fc2_layer(h2_1)

        h2_add = self.add_layer([h1, h2_2])
        h2_norm = self.norm2_layer(h2_add)

    return h2_norm
```

The TransformerDecoderLayer performs the following steps:

1. Using the given input, the layer computes a multi-head attention output. A multi-head attention output is generated by computing attention outputs with several smaller heads and concatenating those outputs to a single output ($h1$).
2. Next we add the original input x to $h1$ to form a residual connection, ($h1_add$).
3. This is followed by a layer normalization step that normalizes ($h1_norm$).
4. $h1_norm$ goes through a fully connected layer to produce $h2_1$.

5. $h2_1$ goes through another fully connected layer to produce $h2_2$.
6. Then we create another residual connection by adding $h1$ and $h2_2$ to produce $h2_add$.
7. Finally we perform layer normalization to produce $h2_norm$, which is the final output of this custom layer.

Defining the full decoder

With all the utility layers implemented, we can implement the text decoder. We will define two input layers. The first takes in a sequence of tokens as the input and the second takes in a sequence of positions (0-index based) to denote the position of each token. You can see that both layers are defined such that they can take in an arbitrary length sequence as an input. This will serve an important purpose as we will see later during inference:

```
caption_input = tf.keras.layers.Input(shape=(None,))
position_input = tf.keras.layers.Input(shape=(None,))
```

Next we define the embeddings. Our embedding vectors will have a length of 384 to match the ViT model's output dimensionality. We defined two embedding layers: the token embedding layer and the positional embedding layer:

```
d_model = 384

# Token embeddings
input_embedding = tf.keras.layers.Embedding(len(tokenizer.get_vocab()),
d_model, mask_zero=True)
```

The token embedding layer works just as we have seen several times. It produces an embedding vector for each token in the sequence. We mask inputs with ID 0 as they represent the padded tokens. Next let's understand how we can implement the positional embeddings:

```
position_embedding = tf.keras.layers.Lambda(
    lambda x: tf.where(
        tf.math.mod(tf.repeat(tf.expand_dims(x, axis=-1), d_model,
        axis=-1), 2)==0,
        tf.math.sin(
            tf.expand_dims(x, axis=-1) /
            10000**tf.reshape(tf.range(d_model,
            dtype='float32'), [1,1, -1])/d_model)
    ),
```

```

        tf.math.cos(
            tf.expand_dims(x, axis=-1) /
            10000** (2*tf.reshape(tf.range(d_model,
            dtype='float32'), [1,1, -1])/d_model)
        )
    )
)

```

We have already discussed how positional embeddings are calculated. The original Transformer paper uses the following equations to generate positional embeddings:

$$PE(pos, 2i) = \sin(pos/10000^{2i/d})$$

$$PE(pos, 2i + 1) = \cos(pos/10000^{2i/d})$$

Here pos denotes the position in the sequence and i denotes the i^{th} feature dimension ($0 < i < d_{\text{model}}$). Even-numbered features use a sine function, where odd-numbered features use a cosine function. Computing this as a layer requires some effort. Let's slowly break down the logic. First we compute the following two tensors (let's refer to them with x and y for ease):

```

x = PE(pos, i) = sin(pos/10000** (2i/d))
y = PE(pos, i) = cos(pos/10000** (2i/d))

```

We use the `tf.where(cond, x, y)` function to select values element-wise from x and y using a Boolean matrix $cond$ of the same size. For a given position, if $cond$ is `True`, select x , and if $cond$ is `False`, select y . Here we use the condition as `pos%2 == 0`, which provides `True` for even positions and `False` for odd positions.

In order to make sure we produce tensors with correct shapes, we utilize the broadcasting capabilities of TensorFlow.

Let's understand a little bit how broadcasting has helped. Take the computation:

```

tf.math.sin(
    tf.expand_dims(x, axis=-1) /
    10000** (2*tf.reshape(tf.range(d_model,
    dtype='float32'), [1,1, -1])/d_model)
)

```

Here we need a [batch size, time steps, d_model]-sized output. `tf.expand_dims(x, axis=1)` produces a [batch size, time steps, 1]-sized output. `10000**((2*tf.reshape(tf.range(d_model, dtype='float32'),[1,1, -1])/d_model))` produces a [1, 1, d_model]-sized output. Dividing the first output by the second provides us with a tensor of size [batch size, time steps, d_model]. This is because the broadcasting capability of TensorFlow allows it to perform operations between arbitrary-sized dimensions and dimensions of size 1. You can imagine TensorFlow copying the dimension of size 1, n many times to perform an operation with an n-sized dimension. But in reality it does this more efficiently.

Once the token and positional embeddings are computed. We add them element-wise to get the final embeddings:

```
embed_out = input_embedding(caption_input) + position_embedding(position_input)
```

If you remember, the first input to the decoder is the image feature vector followed by caption tokens. Therefore, we need to concatenate `image_features` (produced by the ViT) with the `embed_out` to get the full sequence of inputs:

```
image_caption_embed_out = tf.keras.layers.concatenate([tf.expand_dims(image_features, axis=1), embed_out])
```

Then we define four Transformer decoder layers and compute the hidden output of those layers:

```
out = image_caption_embed_out
for l in range(4):
    out = TransformerDecoderLayer(d_model, 64)(out)
```

We use a Dense layer having `n_vocab` output nodes and a *softmax* activation to compute the final output:

```
final_out = tf.keras.layers.Dense(n_vocab, activation='softmax')(out)
```

Finally, we define the full model. It takes in:

- `image_input` – A batch of images of size 224x224x3
 - `caption_input` – The token IDs of the caption (except the last token)
 - `position_input` – A batch of position IDs representing each token position

And gives final_out as the output:

```
full_model = tf.keras.models.Model(inputs=[image_input, caption_input, position_input], outputs=final_out)
```

```
full_model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics='accuracy')

full_model.summary()
```

Now we have defined the full model (*Figure 11.8*):

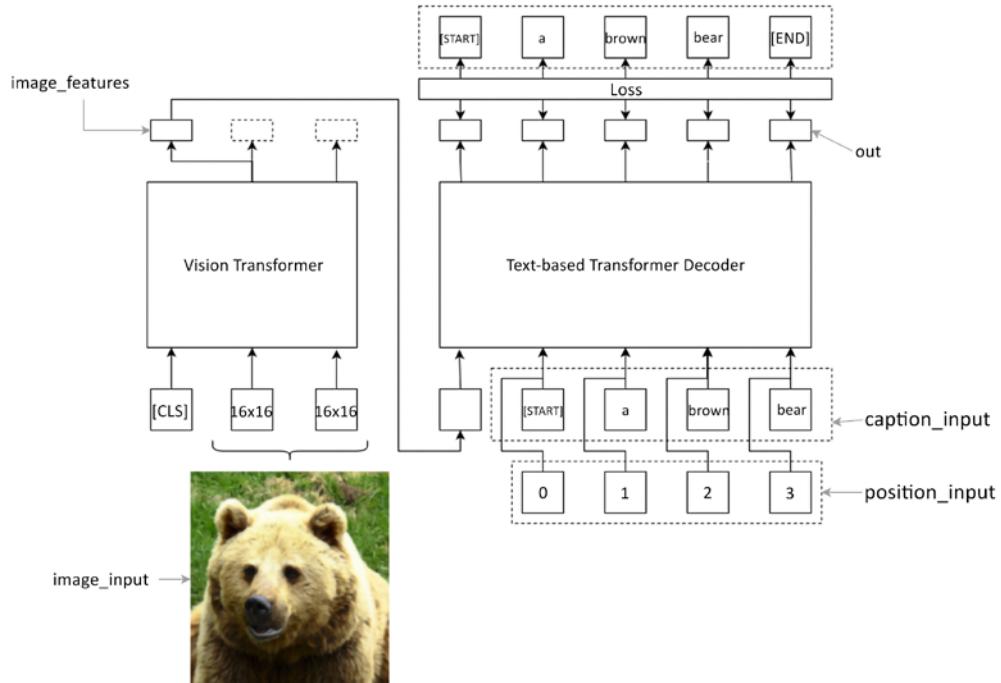


Figure 11.8: Code references overlaid on the illustration of the full model

Training the model

Now that the data pipeline and the model are defined, training it is quite easy. First let's define a few parameters:

```
n_vocab = 4000
batch_size=96

train_fraction = 0.6
valid_fraction = 0.2
```

We use a vocabulary size of 4,000 and a batch size of 96. To speed up the training we'll only use 60% of training data and 20% of validation data. However, you could increase these to get better results. Then we get the tokenizer trained on the full training dataset:

```
tokenizer = generate_tokenizer(  
    train_captions_df, n_vocab=n_vocab  
)
```

Next we define the BLEU metric. This is the same BLEU computation from *Chapter 9, Sequence-to-Sequence Learning – Neural Machine Translation*, with some minor differences. Therefore, we will not repeat the discussion here.

```
bleu_metric = BLEUMetric(tokenizer=tokenizer)
```

Sample the smaller set of validation data outside the training loop to keep the set constant:

```
sampled_validation_captions_df = valid_captions_df.sample(frac=valid_fraction)
```

Next we train the model for 5 epochs:

```
for e in range(5):  
    print(f"Epoch: {e+1}")  
  
    train_dataset, _ = generate_tf_dataset(  
        train_captions_df.sample(frac=train_fraction),  
        tokenizer=tokenizer, n_vocab=n_vocab, batch_size=batch_size,  
        training=True  
    )  
    valid_dataset, _ = generate_tf_dataset(  
        sampled_validation_captions_df, tokenizer=tokenizer,  
        n_vocab=n_vocab, batch_size=batch_size, training=False  
    )  
  
    full_model.fit(  
        train_dataset,  
        epochs=1  
    )  
  
    valid_loss, valid_accuracy, valid_bleu = [], [], []  
    for vi, v_batch in enumerate(valid_dataset):
```

```

        print(f"{vi+1} batches processed", end='\r')
        loss, accuracy = full_model.test_on_batch(v_batch[0],
                                                v_batch[1])
        batch_predicted = full_model(v_batch[0])
        bleu_score =
            bleu_metric.calculate_bleu_from_predictions(v_batch[1],
            batch_predicted)
        valid_loss.append(loss)
        valid_accuracy.append(accuracy)
        valid_bleu.append(bleu_score)

    print(
        f"\nvalid_loss: {np.mean(valid_loss)} - valid_accuracy:
        {np.mean(valid_accuracy)} - valid_bleu: {np.mean(valid_bleu)}"
    )
)

```

In each iteration, we generate a `train_dataset` and a `valid_dataset`. Note that the training set is sampled randomly in each epoch, resulting in different data points, while the validation set is constant. Also note that we are passing the previously generated tokenizer as an argument to the data pipeline function. We call the `full_model.fit()` function with the train dataset to train it for a single epoch within the loop. Finally we iterate through the batches of the validation dataset and compute loss, accuracy, and BLEU values for each batch. Then we print out the mean value of those batch metrics. The output looks like below:

```

Epoch: 1
2071/2071 [=====] - 1945s 903ms/step - loss:
1.3344 - accuracy: 0.7625
173 batches processed
valid_loss: 1.1388846477332142 - valid_accuracy: 0.7819634135058849 -
valid_bleu: 0.09385878526196685
Epoch: 2
2071/2071 [=====] - 1854s 894ms/step - loss:
1.0860 - accuracy: 0.7878
173 batches processed
valid_loss: 1.090059520192229 - valid_accuracy: 0.7879036186058397 -
valid_bleu: 0.10231472779803133
Epoch: 3
2071/2071 [=====] - 1855s 895ms/step - loss:

```

```
1.0610 - accuracy: 0.7897
173 batches processed
valid_loss: 1.0627685799075 - valid_accuracy: 0.7899546606003205 - valid_
bleu: 0.10398145099074609
Epoch: 4
2071/2071 [=====] - 1937s 935ms/step - loss:
1.0479 - accuracy: 0.7910
173 batches processed
valid_loss: 1.0817485169179177 - valid_accuracy: 0.7879597275932401 -
valid_bleu: 0.10308500219058511
Epoch: 5
2071/2071 [=====] - 1864s 899ms/step - loss:
1.0244 - accuracy: 0.7937
173 batches processed
valid_loss: 1.0498641329693656 - valid_accuracy: 0.79208166544148 - valid_
bleu: 0.10667336005789202
```

Let's go through the results. We can see that the training loss and validation losses have more or less gone down consistently. We have a training and validation accuracy of ~80%. Finally, the `valid_bleu` score is around 0.10. You can see the state of the art for a few models here: <https://paperswithcode.com/sota/image-captioning-on-coco>. You can see that a BLEU-4 score of 39 has been reached by the UNIMO model. It is important to note that, in reality, our BLEU score is higher than what's reported here. This is because each image has multiple captions. And when computing the BLEU score with multiple references, you compute BLEU for each and take the max. We have only considered one caption per image when computing the BLEU score. Additionally, our model was far less complicated and trained on a small fraction of the data available. If you would like to increase model performance, you can expose the full training set, and experiment with larger ViT models and data augmentation techniques to improve performance.

Next let's discuss some of the different metrics used to measure the quality of sequences in the context of image captioning.

Evaluating the results quantitatively

There are many different techniques for evaluating the quality and the relevancy of the captions generated. We will briefly discuss several such metrics we can use to evaluate the captions. We will discuss four metrics: BLEU, ROGUE, METEOR, and CIDEr.

All these measures share a key objective, to measure the adequacy (the meaning of the generated text) and fluency (the grammatical correctness of text) of the generated text. To calculate all these measures, we will use a candidate sentence and a reference sentence, where a candidate sentence is the sentence/phrase predicted by our algorithm and the reference sentence is the true sentence/phrase we want to compare with.

BLEU

Bilingual Evaluation Understudy (BLEU) was proposed by Papineni and others in *BLEU: A Method for Automatic Evaluation of Machine Translation, Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July (2002): 311-318*. It measures the n-gram similarity between reference and candidate phrases, in a position-independent manner. This means that a given n-gram from the candidate is present anywhere in the reference sentence and is considered to be a match. BLEU calculates the n-gram similarity in terms of precision. BLEU comes in several variations (BLEU-1, BLEU-2, BLEU-3, and so on), denoting the value of n in the n-gram.

$$\text{BLEU}(\text{candidate}, \text{ref}) = \frac{\sum_{\forall n\text{-gram in candidate}} \text{Count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram in candidate}} \text{Count}(n\text{-gram})} \times \text{BP}$$

Here, $\text{Count}(n\text{-gram})$ is the number of total occurrences of a given n-gram in the candidate sentence. $\text{Count}_{\text{clip}}(n\text{-gram})$ is a measure that calculates $\text{Count}(n\text{-gram})$ for a given n-gram and clips that value by a maximum value. The maximum value for an n-gram is calculated as the number of occurrences of that n-gram in the reference sentence. For example, consider these two sentences:

- Candidate: **the the the the the the**
- Reference: **the cat sat on the mat**

$$\text{Count}("the") = 7$$

$$\text{Count}_{\text{clip}}("the") = 2$$

Note that the entity, $\frac{\sum_{\forall n\text{-gram in candidate}} \text{Count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram in candidate}} \text{Count}(n\text{-gram})}$, is a form of precision. In fact, it is called the modified n-gram precision. When multiple references are present, the BLEU is considered to be the maximum:

$$\text{BLEU} = \max(\text{BLEU}(\text{candidate}, \text{ref}_i))$$

However, the modified n-gram precision tends to be higher for smaller candidate phrases because this entity is divided by the number of n-grams in the candidate phrase. This means that this measure will incline the model to produce shorter phrases. To avoid this, a penalty term, BP , is added to the preceding term that penalizes short candidate phrases as well. BLEU possesses several limitations such as BLEU ignores synonyms when calculating the score and does not consider recall, which is also an important metric to measure accuracy. Furthermore, BLEU appears to be a poor choice for certain languages. However, this is a simple metric that has been found to correlate well with human judgment as well in most situations.

ROUGE

Recall-Oriented Understudy for Gisting Evaluations (ROUGE), proposed by Chin-Yew Lin in *ROUGE: A Package for Automatic Evaluation of Summaries, Proceedings of the Workshop on Text Summarization Branches Out (2004)*, can be identified as a variant of BLEU, and uses recall as the basic performance metric. The ROUGE metric looks like the following:

$$\text{ROUGE} - N = \frac{\text{Count}_{\text{match}}}{\text{Count}_{\text{ref}}}$$

Here, $\text{Count}_{\text{match}}$ is the number of n-grams from candidates that were present in the reference, and $\text{Count}_{\text{ref}}$ is the total n-grams present in the reference. If there exist multiple references, $\text{ROUGE}-N$ is calculated as follows:

$$\text{ROUGE} - N = \max(\text{ROUGE} - N(\text{ref}_i, \text{candidate}))$$

Here, ref_i is a single reference from the pool of available references. There are numerous variants of the ROUGE measure that introduce various improvements to the standard ROUGE metric. ROUGE-L computes the score based on the longest common subsequence found between the candidate and reference sentence pairs. Note that the longest common subsequence does not need to be continuous in this case. Next, ROUGE-W calculates the score based on the longest common subsequence, which is penalized by the amount of fragmentation present within the subsequence. ROUGE also suffers from limitations such as not considering precision in the calculations of the score.

METEOR

Metric for Evaluation of Translation with Explicit Ordering (METEOR), proposed by Michael Denkowski and Alon Lavie in *Meteor Universal: Language Specific Translation Evaluation for Any Target Language, Proceedings of the Ninth Workshop on Statistical Machine Translation (2014): 376-380*, is a more advanced evaluation metric that performs alignments for a candidate and a reference sentence. METEOR is different from BLEU and ROUGE in the sense that METEOR takes the position of words into account. When computing similarities between a candidate sentence and a reference sentence, the following cases are considered as matches:

- **Exact:** The word from the candidate exactly matches the word from the reference sentence
- **Stem:** A stemmed word (for example, *walk* of the word *walked*) matches the word from the reference sentence
- **Synonym:** The word from a candidate sentence is a synonym for the word from the reference sentence

To calculate the METEOR score, the matches between a reference sentence and a candidate sentence can be shown, as in *Figure 11.10*, with the help of a table. Then, precision (P) and recall (R) values are calculated based on the number of matches, present in the candidate and reference sentences. Finally, the harmonic mean of P and R is used to compute the METEOR score:

$$F_{mean} = \frac{P \cdot R}{\alpha P + (1 - \alpha)R} (1 - \gamma \times frag^\beta)$$

Here, α , β , and γ are tunable parameters, and $frag$ penalizes fragmented matches, in order to prefer candidate sentences that have fewer gaps in matches as well as those that closely follow the order of words of the reference sentence. The $frag$ is calculated by looking at the number of crosses in the final unigram mapping (*Figure 11.9*):

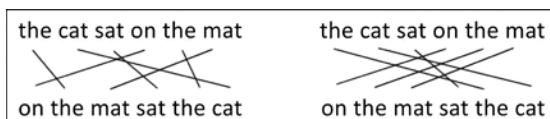


Figure 11.9: Different possible alignments for two strings

For example, we can see that the left side has 7 crosses, whereas the right side has 10 crosses, which means the right-side alignment will be more penalized than the left side.

		Reference Sentence									
		Bob decided to flee when he observed Tom as Bob owed money									
Candidate Sentence	Bob	●									
	run			(○)							
	away										
	when				●						
	he					●					
	observes						(○)				
	Tom							●			
	as								●		
	Bob									●	
	was										
	in										
	debt									(○)	

Figure 11.10: The METEOR word matching table

You can see that we denoted matches between the candidate sentence and the reference sentence in circles and ovals. For example, we denote exact matches with a solid black circle, synonyms with a dashed hollow circle, and stemmed matches with dotted circles.

METEOR is computationally more complex, but has often been found to correlate with human judgment more than BLEU, suggesting that METEOR is a better evaluation metric than BLEU.

CIDEr

Consensus-based Image Description Evaluation (CIDEr), proposed by Ramakrishna Vedantam and others in *CIDEr: Consensus-based Image Description Evaluation, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015*, is another measure that evaluates the consensus of a candidate sentence to a given set of reference statements. CIDEr is defined to measure the grammaticality, saliency, and accuracy (that is, precision and recall) of a candidate sentence.

First, CIDEr weighs each n-gram found in both the candidate and reference sentences by means of TF-IDF, so that more common n-grams (for example, the words *a* and *the*) will have a smaller weight, whereas rare words will have a higher weight. Finally, CIDEr is calculated as the cosine similarity between the vectors formed by TF-IDF-weighted n-grams found in the candidate sentence and the reference sentence:

$$CIDEr(cand, ref) = \frac{1}{m} \sum_j \frac{TF - IDF_{vec}(cand).TF - IDF_{vec}(ref_j)}{\|TF - IDF_{vec}(cand)\| \|TF - IDF_{vec}(ref_j)\|}$$

Here, *cand* is the candidate sentence, *ref* is the set of reference sentences, *ref_j* is the *j*th sentence of *ref*, and *m* is the number of reference sentences for a given candidate. Most importantly, *TF – IDF_{vec}(cand)* is the TF-IDF values calculated for all the n-grams in the candidate sentence and formed as a vector. *TF – IDF_{vec}(ref_j)* is the same vector for the reference sentence, *ref_j*. $\|\cdot\|$ denotes the magnitude of the vector.

Overall, it should be noted that there is no clear-cut winner that is able to perform well across all the different tasks that are found in NLP. These metrics are significantly task-dependent and should be carefully chosen depending on the task. Here we'll be using the BLEU score for our model.

Evaluating the model

With the model trained, let's test the model on our unseen test dataset. Testing logic is almost identical to the validation logic we discussed earlier during model training. Therefore we will not repeat our discussion here.

```
bleu_metric = BLEUMetric(tokenizer=tokenizer)

test_dataset, _ = generate_tf_dataset(
    test_captions_df, tokenizer=tokenizer, n_vocab=n_vocab, batch_size=batch_size, training=False
)

test_loss, test_accuracy, test_bleu = [], [], []
for ti, t_batch in enumerate(test_dataset):
    print(f"{ti+1} batches processed", end='\r')
    loss, accuracy = full_model.test_on_batch(t_batch[0], t_batch[1])
    batch_predicted = full_model.predict_on_batch(t_batch[0])
    bleu_score = bleu_metric.calculate_bleu_from_predictions(t_batch[1],
    batch_predicted)
```

```
    test_loss.append(loss)
    test_accuracy.append(accuracy)
    test_bleu.append(bleu_score)

    print(
        f"\ntest_loss: {np.mean(test_loss)} - test_accuracy: {np.mean(test_
accuracy)} - test_bleu: {np.mean(test_bleu)}"
    )
```

This will output:

```
261 batches processed
test_loss: 1.057080413646625 - test_accuracy: 0.7914185857407434 - test_
bleu: 0.10505496256163914
```

Great, we can see the model is showing a similar performance to what it did on the validation data. This means our model has not overfitted data, and should perform reasonably well in the real world. Let's now generate captions for a few sample images.

Captions generated for test images

With the help of metrics such as accuracy and BLEU, we have ensured our model is performing well. But, one of the most important tasks a trained model has to perform is generating outputs for new data. We will learn how we can use our model to generate actual captions. Let's first understand how we can generate captions at a conceptual level. It's quite straightforward to generate the image representation using an image. The tricky part is adapting the text decoder to generate captions. As you can imagine, the decoder inference needs to work in a different setting than the training. This is because at inference we don't have caption tokens to input to the model.

The way we predict with our model is by starting with the image and a starting caption that has the single token [START]. We feed these two inputs to the model to generate the next token. We then combine the new token with the current input and predict the next token. We keep going this way until we reach a certain number of steps or the model outputs [END] (*Figure 11.11*). If you remember, we developed the model in a way that it can accept an arbitrary length token sequence. This is extremely helpful during inference as at each time step, the length of the sequence increases.

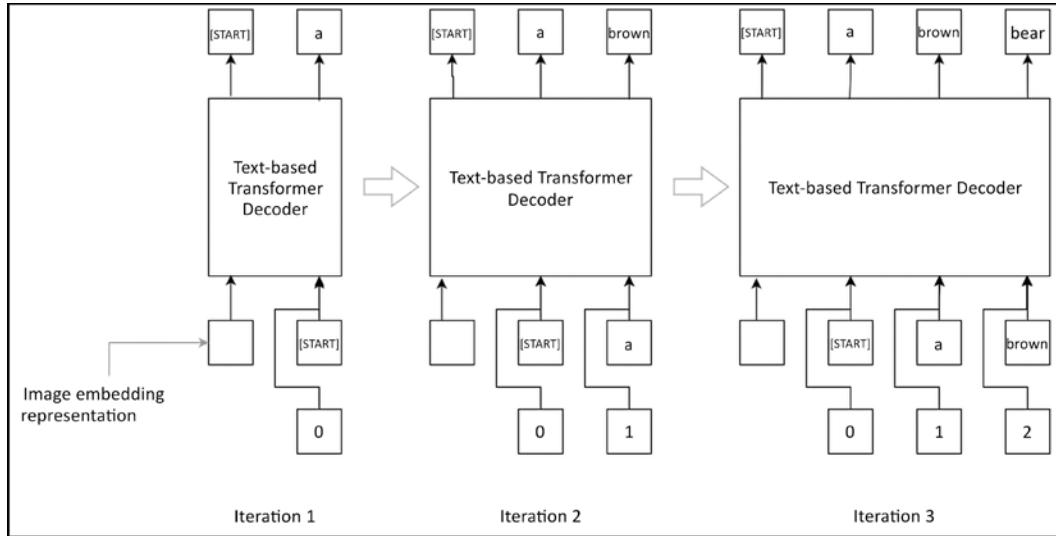


Figure 11.11: How the decoder of the trained model is used to generate a new caption for a given image

We will choose a small dataset of 10 samples from the test dataset and generate captions:

```
n_samples = 10
test_dataset, _ = generate_tf_dataset(
    test_captions_df.sample(n=n_samples), tokenizer=tokenizer,
    n_vocab=n_vocab, batch_size=n_samples, training=False
)
```

Next let's define a function called `generate_caption()`. This function takes in:

- `model` – The trained model
- `image_input` – A batch of input images
- `tokenizer` – Trained tokenizer
- `n_samples` – Number of samples in the batch

As we can see in the following:

```
def generate_caption(model, image_input, tokenizer, n_samples):
    # 2 -> [START]
    batch_tokens = np.repeat(np.array([[2]]), n_samples, axis=0)
```

```
for i in range(30):
    if np.all(batch_tokens[:, -1] == 3):
        break

    position_input = tf.repeat(tf.reshape(tf.range(i+1), [1, -1]), n_samples, axis=0)
    probs = full_model((image_input, batch_tokens, position_input)).numpy()
    batch_tokens = np.argmax(probs, axis=-1)

predicted_text = []
for sample_tokens in batch_tokens:
    sample_predicted_token_ids = sample_tokens.ravel()
    sample_predicted_tokens = []
    for wid in sample_predicted_token_ids:
        sample_predicted_tokens.append(tokenizer.id_to_token(wid))
    if wid == 3:
        break
    sample_predicted_text = " ".join([tok for tok in sample_predicted_tokens])
    sample_predicted_text = sample_predicted_text.replace(" ##", "")
    predicted_text.append(sample_predicted_text)

return predicted_text
```

This function starts with a single caption token ID. The ID 2 maps to the token [START]. We predict for 30 steps or if the last token is [END] (mapped to token ID 3). We generate position inputs for the batch of data by creating a range sequence from 0 to i and repeating that n_sample times across the batch dimension. We then predict the token probabilities by feeding the inputs to the model.

We can now use this function to generate captions:

```
for batch in test_dataset.take(1):
    (batch_image_input, _, _), batch_true_caption = batch

batch_predicted_text = generate_caption(full_model, batch_image_input,
tokenizer, n_samples)
```

Let's now visualize the captions side by side with the image inputs. Additionally, we'll show the ground truth captions:

```
fig, axes = plt.subplots(n_samples, 2, figsize=(8,30))

for i,(sample_image_input, sample_true_caption, sample_predicated_caption)
in enumerate(zip(batch_image_input, batch_true_caption, batch_predicted_
text)):

    sample_true_caption_tokens  = [tokenizer.id_to_token(wid) for wid in
sample_true_caption.numpy().ravel()]

    sample_true_text = []
    for tok in sample_true_caption_tokens:
        sample_true_text.append(tok)
        if tok == '[END]':
            break

    sample_true_text = " ".join(sample_true_text).replace(" ##", "")

    axes[i][0].imshow(((sample_image_input.numpy()+1.0)/2.0))
    axes[i][0].axis('off')

    true_annotation = f"TRUE: {sample_true_text}"
    predicted_annotation = f"PRED: {sample_predicated_caption}"
    axes[i][1].text(0, 0.75, true_annotation, fontsize=18)
    axes[i][1].text(0, 0.25, predicted_annotation, fontsize=18)
    axes[i][1].axis('off')
```

You will get a plot similar to the following. The images *sampled* will be randomly sampled every time it runs. The results of this run can be seen in *Figure 11.12*:

	TRUE: [START] cars parked along a street with a street light that is green [END] PRED: [START] a car stopped on a city street with cars [END]
	TRUE: [START] several containers of food and beverages on a wooden table outdoors [END] PRED: [START] a table and a plates on a table [END]
	TRUE: [START] a few street lights on the side of the road [END] PRED: [START] a traffic light on a pole in the middle of a city [END]
	TRUE: [START] a person riding the horse jumping over a wooden obstacle [END] PRED: [START] a person on a horse in a field [END]
	TRUE: [START] a blue cake features a small island on top and sharks swimming on the bottom of the cake [END] PRED: [START] a cake with a blue and white frosting on it [END]
	TRUE: [START] a plastic dish with the food sectioned off [END] PRED: [START] a tray of food and vegetables and a sandwich [END]
	TRUE: [START] three images of the same man hitting a tennis ball [END] PRED: [START] a man standing on a tennis court holding a racquet [END]
	TRUE: [START] two people who are swimming in the water on surfboards [END] PRED: [START] a dog in a body of water in the water [END]
	TRUE: [START] a person on a surf board riding a wave [END] PRED: [START] a man on a surfboard in the ocean [END]
	TRUE: [START] an individual is taken in this very picture [END] PRED: [START] a group of people standing in a terminal [END]

Figure 11.12: Captions generated on a sample of test data

We can see that our model does a good job of generating captions. In general, we can see that the model can identify objects and activities portrayed in the images. It is also important to remember that each of our images has multiple captions associated with it. Therefore, the predicted captions do not necessarily need to match the ground truth caption shown in the image.

Summary

In this chapter, we focused on a very interesting task that involves generating captions for given images. Our image-captioning model was one of the most complex models in this book, which included the following:

- A vision Transformer model that produces an image representation
- A text-based Transformer decoder

Before we began with the model, we analyzed our dataset to understand various characteristics such as image sizes and the vocabulary size. Then we understood how we can use a tokenizer to tokenize captions strings. We then used this knowledge to build a TensorFlow data pipeline.

We discussed each component in detail. The Vision Transformer (ViT) takes in an image and produces a hidden representation of that image. Specifically, the ViT breaks an image into a sequence of 16x16 patches of pixels. After that, it treats each patch as a token embedding to the Transformer (along with positional information) to produce a representation of each patch. It also incorporates the [CLS] token at the beginning to provide a holistic representation of the image.

Next the text decoder takes in the image representation along with caption tokens as inputs. The objective of the decoder becomes to predict the next token at each time step. We were able to reach a BLEU-4 score of just above 0.10 for the validation dataset.

Thereafter, we discussed several different metrics (BLEU, ROUGE, METEOR, and CIDEr), which we can use to quantitatively evaluate the generated captions, and we saw that as we ran our algorithm through the training data, the BLEU-4 score increased over time. Additionally, we visually inspected the generated captions and saw that our ML pipeline progressively gets better at captioning images.

Next, we evaluated our model on the test dataset and validated that it demonstrates similar performance on test data as expected. Finally, we learned how we can use the trained model to generate captions for unseen images.

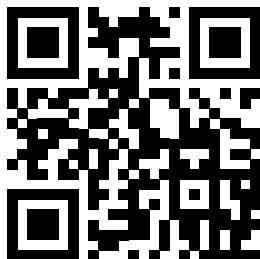
We have reached the end of the book. We have covered many different topics in natural language and discussed state-of-the-art models and techniques that help us to solve problems.

In the Appendix, we will discuss some mathematical concepts related to machine learning, followed by an explanation of how to use the visualization tool TensorBoard to visualize word vectors.

To access the code files for this book, visit our GitHub page at:

<https://packt.link/nlpgithub>

Join our Discord community to meet like-minded people and learn alongside
more than 1000 members at: <https://packt.link/nlp>



Appendix A: Mathematical Foundations and Advanced TensorFlow

Here we will discuss some concepts that will be useful for helping you to understand certain details provided in the chapters. First, we will discuss several mathematical data structures found throughout the book, followed by a description of the various operations performed on those data structures. After that, we will discuss the concept of probabilities. Probabilities play a vital role in machine learning, as they usually give insights into how uncertain a model is about its prediction. Finally, we will conclude this appendix with a guide on how to use TensorBoard as a visualization tool for word embeddings.

Basic data structures

Scalar

A scalar is a single number, unlike a matrix or a vector. For example, 1.3 is a scalar. A scalar can be mathematically denoted as follows: $n \in R$.

Here, R is the real number space.

Vectors

A vector is an array of numbers. Unlike a set, where there is no order to the elements, a vector has a certain order to the elements. An example vector is [1.0, 2.0, 1.4, 2.3]. Mathematically, it can be denoted as follows:

$$a = (a_0, a_1, \dots, a_{\{n-1\}})$$

$$a \in R^n$$

Here, R is the real number space and n is the number of elements in the vector.

Matrices

A matrix can be thought of as a two-dimensional arrangement of a collection of scalars. In other words, a matrix can be thought of as a vector of vectors. An example matrix is shown as follows:

$$A = \begin{bmatrix} 1 & 4 & 2 & 3 \\ 2 & 7 & 7 & 1 \\ 5 & 6 & 9 & 0 \end{bmatrix}$$

A more general matrix of size $m \times n$ can be mathematically defined like this:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

And:

$$A \in R^{m \times n}$$

Here, m is the number of rows of the matrix, n is the number of columns in the matrix, and R is the real number space.

Indexing of a matrix

We will be using zero-indexed notation (that is, indexes that start with 0).

To index a single element from a matrix at the $(i, j)^{\text{th}}$ position, we use the following notation:

$$A_{i,j} = a_{i,j}$$

Referring to the previously defined matrix, we get the following:

$$A = \begin{bmatrix} 1 & 4 & 2 & 3 \\ 2 & 7 & 7 & 1 \\ 5 & 6 & 9 & 0 \end{bmatrix}$$

We index an element from A like this:

$$A_{1,0} = 2$$

We denote a single row of any matrix A as shown here:

$$A_{i,:} = (a_{i,0}, a_{i,1}, \dots, a_{i,n-1})$$

For our example matrix, we can denote the second row (indexed as 1) of the matrix as shown here:

$$A_{1,:} = (2, 7, 7, 1)$$

We denote the slice starting from the $(i, k)^{\text{th}}$ index to the $(j, l)^{\text{th}}$ index of any matrix A as shown here:

$$A_{i:j+1, k:l+1} = \begin{bmatrix} a_{i,k} & \cdots & a_{i,l} \\ \vdots & \ddots & \vdots \\ a_{j,k} & \cdots & a_{j,l} \end{bmatrix}$$

In our example matrix, we can denote the slice from first row third column to second row fourth column as shown here:

$$A_{0:2, 2:4} = \begin{bmatrix} 2 & 3 \\ 7 & 1 \end{bmatrix}$$

Special types of matrices

Identity matrix

An identity matrix is a square matrix where values are equal to 1 on the diagonal of the matrix and 0 everywhere else. Mathematically, it can be shown as follows:

$$I_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

This would look like the following:

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Here, $I \in R^{n \times n}$.

The identity matrix gives the following nice property when multiplied with another matrix A :

$$AI = IA = A$$

Square diagonal matrix

A square diagonal matrix is a more general case of the identity matrix, where the values along the diagonal can take any value and the off-diagonal values are zeros:

$$A = \begin{bmatrix} a_{0,0} & 0 & \cdots & 0 \\ 0 & a_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} \end{bmatrix}$$

Tensors

An n -dimensional matrix is called a **tensor**. In other words, a matrix with an arbitrary number of dimensions is called a tensor. For example, a four-dimensional tensor can be denoted as shown here:

$$T \in R^{k \times l \times m \times n}$$

Here, R is the real number space.

Tensor/matrix operations

Transpose

Transpose is an important operation defined for matrices or tensors. For a matrix, the transpose is defined as follows:

$$(A^T)_{i,j} = A_{j,i}$$

Here, A^T denotes the transpose of A .

An example of the transpose operation can be illustrated as follows:

$$A = \begin{bmatrix} 1 & 4 & 2 & 3 \\ 2 & 7 & 7 & 1 \\ 5 & 6 & 9 & 0 \end{bmatrix}$$

After the transpose operation:

$$A^T = \begin{bmatrix} 1 & 2 & 5 \\ 4 & 7 & 6 \\ 2 & 7 & 9 \\ 3 & 1 & 0 \end{bmatrix}$$

For a tensor, transpose can be seen as permuting the dimensions order. For example, let's define a tensor S , as shown here:

$$S \in R^{d_1, d_2, d_3, d_4}$$

Now one transpose operation (out of many) can be defined as follows:

$$S^T \in R^{d_4, d_3, d_2, d_1}$$

Matrix multiplication

Matrix multiplication is another important operation that appears quite frequently in linear algebra.

Given the matrices $A \in R^{m \times n}$ and $B \in R^{n \times p}$, the multiplication of A and B is defined as follows:

$$C = AB$$

Here, $C \in R^{m \times p}$.

Consider this example:

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 8 & 5 & 2 \\ 9 & 6 & 3 \end{bmatrix}$$

This gives $C = AB$, and the value of C is as follows:

$$C = \begin{bmatrix} 26 & 17 & 8 \\ 77 & 50 & 23 \\ 128 & 83 & 38 \end{bmatrix}$$

Element-wise multiplication

Element-wise matrix multiplication (or the **Hadamard product**) is computed for two matrices that have the same shape. Given the matrices $A \in R^{m \times n}$ and $B \in R^{m \times n}$, the element-wise multiplication of A and B is defined as follows:

$$C = A \circ B$$

Here, $C \in R^{m \times n}$.

Consider this example:

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 6 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 2 \\ 1 & 3 \\ 3 & 5 \end{bmatrix}$$

This gives $C = A \circ B$, and the value of C is as follows:

$$C = \begin{bmatrix} 6 & 6 \\ 1 & 6 \\ 18 & 5 \end{bmatrix}$$

Inverse

The inverse of the matrix A is denoted by A^{-1} , where it satisfies the following condition:

$$A^{-1}A = I$$

Inverse is very useful if we are trying to solve a system of linear equations. Consider this example:

$$Ax = b$$

We can solve for x like this:

$$A^{-1}(Ax) = A^{-1}b$$

This can be written as $(A^{-1}A)x = A^{-1}b$, using the associative law – that is, $A(BC) = (AB)C$.

Next, we will get, where I is the identity matrix.

Lastly, $x = A^{-1}b$ because $Ix = x$.

For example, polynomial regression, one of the regression techniques, uses a linear system of equations to solve the regression problem. Regression is similar to classification, but instead of outputting a class, regression models output a continuous value. Let's look at an example problem: given the number of bedrooms in a house, we'll calculate the real-estate value of the house. Formally, a polynomial regression problem can be written as follows:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m + \varepsilon_i \quad (i = 1, 2, \dots, n)$$

Here, (x_i, y_i) is the i^{th} data input, where x_i is the input, y_i is the label, and ε is the noise in data. In our example, x is the number of bedrooms and y is the price of the house. This can be written as a system of linear equations as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_3 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

However, A^{-1} does not exist for all A . There are certain conditions that need to be satisfied in order for the inverse to exist for a matrix. For example, to define the inverse, A needs to be a square matrix (that is, $R^{m \times m}$). Even when the inverse exists, we cannot always find it in the closed form; sometimes it can only be approximated with finite-precision computers. If the inverse exists, there are several algorithms for finding it, which we will discuss next.

Note



When it is said that A needs to be a square matrix for the inverse to exist, we refer to the standard inversion. There exist variants of the inverse operation (for example, the **Moore-Penrose inverse**, also known as pseudoinverse) that can perform matrix inversion on general $m \times n$ matrices.

Finding the matrix inverse – Singular Value Decomposition (SVD)

Let's now see how we can use SVD to find the inverse of a matrix A . SVD factorizes A into three different matrices, as shown here:

$$A = UDV^T$$

Here the columns of U are known as left singular vectors, columns of V are known as right singular vectors, and diagonal values of D (a diagonal matrix) are known as singular values. Left singular vectors are the eigenvectors of AA^T and the right singular vectors are the eigenvectors of A^TA . Finally, the singular values are the square roots of the eigenvalues of AA^T and A^TA . The eigenvector v and its corresponding eigenvalue λ of the square matrix A satisfy the following condition:

$$Av = \lambda v$$

Then, if the SVD exists, the inverse of A is given by this:

$$A^{-1} = VD^{-1}U^T$$

Since D is diagonal, D^{-1} is simply the element-wise reciprocal of the nonzero elements of D . SVD is an important matrix factorization technique that appears on many occasions in machine learning. For example, SVD is used for calculating **Principal Component Analysis (PCA)**, which is a popular dimensionality reduction technique for data (a purpose similar to that of t-SNE, which we saw in *Chapter 4, Advanced Word Vector Algorithms*). Another, more NLP-oriented application of SVD is document ranking. That is, when you want to get the most relevant documents (and rank them by relevance to some term, for example, *football*), SVD can be used to achieve this. To learn more about SVD, you can consult this blog post, which provides a geometric intuition on SVD, as well as showing how it's applied in PCA: <https://gregorygundersen.com/blog/2018/12/10/svd/>.

Norms

A norm is used as a measure of the *size* of the vector (that is, of the values in the vector). The p^{th} norm is calculated and denoted as shown here:

$$\| A \|_p = \left(\sum_i |A_i|^p \right)^{1/p}$$

For example, the *L2* norm would be this:

$$\| A \|_2 = \sqrt{\sum_i |A_i|^2}$$

Determinant

The determinant of a square matrix is denoted by $\det(A)$. The determinant is very useful in many ways. For example, A is invertible if, and only if, the determinant is nonzero. The determinant is also interpreted as the product of all the eigenvalues of the matrix. The determinant of a 2×2 matrix A ,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

is denoted as

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

and computed as

$$\det(A) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

The following equation shows the calculations for the determinant of a 3×3 matrix:

$$\begin{aligned} \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} &= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= a(ei - fh) - b(di - fg) + c(dh - eg) \\ &= aei + bfg + cdh - ceg - bdi - afh \end{aligned}$$

Probability

Next, we will discuss the terminology related to probability theory. Probability theory is a vital part of machine learning, as modeling data with probabilistic models allows us to draw conclusions about how uncertain a model is about some predictions. Consider a use case of sentiment analysis. We want to output a prediction (positive/negative) for a given movie review. Though the model outputs some value between 0 and 1 (0 for negative and 1 for positive) for any sample we input, the model doesn't know how *uncertain* it is about its answer.

Let's understand how uncertainty helps us to make better predictions. For example, a deterministic model (i.e. a model that outputs an exact value instead of a distribution for the value) might incorrectly say the positivity of the review *I never lost interest* is 0.25 (that is, it's more likely to be a negative comment). However, a probabilistic model will give a mean value and a standard deviation for the prediction. For example, it will say, this prediction has a mean of 0.25 and a standard deviation of 0.5. With the second model, we know that the prediction is likely to be wrong due to the high standard deviation. However, in the deterministic model, we don't have this luxury. This property is especially valuable for critical machine systems (for example, a terrorism risk assessment model).

To develop such probabilistic machine learning models (for example, Bayesian logistic regression, Bayesian neural networks, or Gaussian processes), you should be familiar with basic probability theory. Therefore, we will provide some basic probability information here.

Random variables

A random variable is a variable that can take some value at random. Also, random variables are represented as x_1 , x_2 , and so on. Random variables can be of two types: discrete and continuous.

Discrete random variables

A discrete random variable is a variable that can take discrete random values. For example, trials of flipping a coin can be modeled as a random variable; that is, the side a coin lands on when you flip it is a discrete variable as the value can only be *heads* or *tails*. Alternatively, the value you get when you roll a die is discrete, as well, as the values can only come from the set $\{1, 2, 3, 4, 5, 6\}$.

Continuous random variables

A continuous random variable is a variable that can take any real value, that is, if x is a continuous random variable:

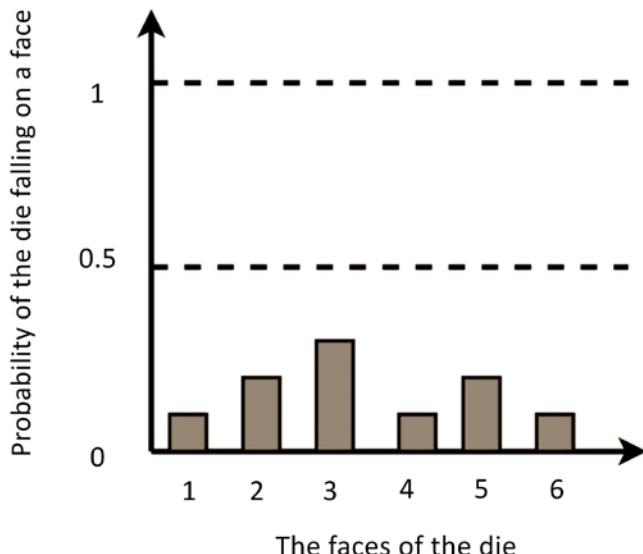
$$x \in R$$

Here, R is the real number space.

For example, the height of a person is a continuous random variable as it can take any real value.

The probability mass/density function

The **probability mass function (PMF)** or the **probability density function (PDF)** is a way of showing the probability distribution over different values a random variable can take. For discrete variables, a PMF is defined, and for continuous variables, a PDF is defined. *Figure A.1* shows an example PMF:



A.1: Probability mass function (PMF) discrete

The preceding PMF might be achieved by a *biased* die. In this graph, we can see that there is a high probability of getting a 3 with this die. Such a graph can be obtained by running a number of trials (say, 100) and then counting the number of times each face fell on top. Finally, you would divide each count by the number of trials to obtain the normalized probabilities. Note that all the probabilities should add up to 1, as shown here:

$$P(X \in \{1,2,3,4,5,6\}) = 1$$

The same concept is extended to a continuous random variable to obtain a PDF. Say that we are trying to model the probability of a certain height given a population. Unlike the discrete case, we do not have individual values to calculate the probability for, but rather a continuous spectrum of values (in the example, it extends from 0 to 2.4 m). If we are to draw a graph for this example like the one in *Figure A.1*, we need to think of it in terms of infinitesimally small bins. For example, we find out the probability density of a person's height being between 0.0 m-0.01 m, 0.01-0.02 m, ..., 1.8 m-1.81 m, ..., and so on. The probability density can be calculated using the following formula:

$$\text{probability density for } bin_i = \frac{\text{probability of person's height being in } bin_i}{bin_i \text{ size}}$$

Then, we will plot those bars close to each other to obtain a continuous curve, as shown in *Figure A.2*. Note that the probability density for a given bin can be greater than 1 (since it's density), but the area under the curve must be 1:

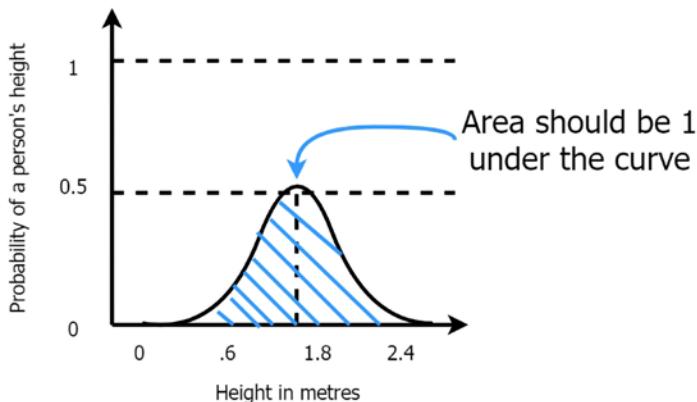


Figure A.2: Probability density function (PDF) continuous

The shape shown in *Figure A.2* is known as the normal (or Gaussian) distribution. It is also called the *bell curve*. We previously gave just an intuitive explanation of how to think about a continuous probability density function.

More formally, a continuous PDF of the normal distribution has an equation and is defined as follows. Let's assume that a continuous random variable X has a normal distribution with mean μ and standard deviation σ . The probability of $X = x$ for any value of x is given by this formula:

$$P(X = x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

You should get the area (which needs to be 1 for a valid PDF) if you integrate this quantity over all possible infinitesimally small dx values, as denoted by this formula:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

The integral of the normal for the arbitrary a, b values is given by the following formula:

$$\int_{-\infty}^{\infty} e^{-a(x+b)^2} dx = \sqrt{\frac{\pi}{a}}$$

Using this, we can get the integral of the normal distribution, where $a = 1/2\sigma^2$ and $b = -\mu$:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \frac{1}{\sqrt{2\pi\sigma^2}} \sqrt{\frac{\pi}{1/2\sigma^2}} = \frac{1}{\sqrt{2\pi\sigma^2}} \sqrt{2\pi\sigma^2} = 1$$

This gives the accumulation of all the probability values for all the values of x and gives you a value of 1.



You can find more information at <http://mathworld.wolfram.com/GaussianIntegral.html>, or for a less complex discussion, refer to https://en.wikipedia.org/wiki/Gaussian_integral.

Conditional probability

Conditional probability represents the probability of an event happening given the occurrence of another event. For example, given two random variables, X and Y , the conditional probability of $X = x$, given that $Y = y$, is denoted by this formula:

$$P(X = x | Y = y)$$

A real-world example of such a probability would be as follows:

$$P(\text{Bob going to school} = \text{Yes} \mid \text{It rains} = \text{Yes})$$

Joint probability

Given two random variables, X and Y , we will refer to the probability of $X = x$ together with $Y = y$ as the joint probability of $X = x$ and $Y = y$. This is denoted by the following formula:

$$P(X = x, Y = y) = P(X = x)P(Y = y \mid X = x)$$

If X and Y are mutually exclusive events, this expression reduces to this:

$$P(X = x, Y = y) = P(X = x)P(Y = y)$$

A real-world example of this is as follows:

$$P(\text{It Rains} = \text{yes}, \text{Play Golf} = \text{yes}) = P(\text{It Rains} = \text{Yes}) P(\text{Play Golf} = \text{yes} \mid \text{It Rains} = \text{Yes})$$

Marginal probability

A marginal probability distribution is the probability distribution of a subset of random variables, given the joint probability distribution of all variables. For example, consider that two random variables, X and Y , exist, and we already know $P(X = x, Y = y)$ and we want to calculate $P(x)$:

$$P(X = x) = \sum_{\forall y} P(X = x, Y = y')$$

Intuitively, we are taking the sum over all possible values of Y , effectively making the probability of $Y = 1$.

Bayes' rule

Bayes' rule gives us a way to calculate $P(Y = y \mid X = x)$ if we already know $P(X = x, Y = y)$, $P(X = x)$, and $P(Y = y)$. We can easily arrive at Bayes' rule as follows:

$$P(X = x, Y = y) = P(X = x)P(Y = y \mid X = x) = P(Y = y) P(X = x \mid Y = y)$$

Now let's take the middle and right parts:

$$P(X = x)P(Y = y \mid X = x) = P(Y = y) P(X = x \mid Y = y)$$

$$P(Y = y \mid X = x) = \frac{P(X = x \mid Y = y) P(Y = y)}{P(X = x)}$$

This is Bayes' rule. Let's put it simply, as shown here:

$$P(y | x) = \frac{P(x | y) P(y)}{P(x)}$$

Visualizing word embeddings with TensorBoard

When we wanted to visualize word embeddings in *Chapter 3, Word2vec – Learning Word Embeddings*, we manually implemented the visualization with the t-SNE algorithm. However, you also could use TensorBoard to visualize word embeddings. TensorBoard is a visualization tool provided with TensorFlow. You can use TensorBoard to visualize the TensorFlow variables in your program. This allows you to see how different variables behave over time (for example, model loss/accuracy), so you can identify potential issues in your model.

TensorBoard enables you to visualize scalar values (e.g. loss values over training iterations) and vectors as histograms (e.g. model's layer node activations). Apart from this, TensorBoard also allows you to visualize word embeddings. Therefore, it takes all the required code implementation away from you, if you need to analyze what the embeddings look like. Next, we will see how we can use TensorBoard to visualize word embeddings. The code for this exercise is provided in `tensorboard_word_embeddings.ipynb` in the Appendix folder.

Starting TensorBoard

First, we will list the steps for starting TensorBoard. TensorBoard acts as a service and runs on a specific port (by default, on 6006). To start TensorBoard, you will need to follow these steps:

1. Open up Command Prompt (Windows) or Terminal (Ubuntu/macOS).
2. Go into the project home directory.
3. If you are using the `python virtualenv`, activate the virtual environment where you have installed TensorFlow.
4. Make sure that you can see the TensorFlow library through Python. To do this, follow these steps:
 - a. Type in `python3`; you will get a `>>>` looking prompt
 - b. Try `import tensorflow as tf`
 - c. If you can run this successfully, you are fine
 - d. Exit the `python` prompt (that is, `>>>`) by typing `exit()`
5. Type in `tensorboard --logdir=models`:

- a. The `--logdir` option points to the directory where you will create data to visualize
 - b. Optionally, you can use `--port=<port_you_like>` to change the port TensorBoard runs on
6. You should now get the following message:
- `TensorBoard 1.6.0 at <url>:6006 (Press CTRL+C to quit)`
7. Enter the `<url>:6006` into the web browser. You should be able to see an orange dashboard at this point. You won't have anything to display because we haven't generated any data.

Saving word embeddings and visualizing via TensorBoard

First, we will download and load the 50-dimensional GloVe embeddings file (`glove.6B.zip`) from <https://nlp.stanford.edu/projects/glove/> and place it in the `Appendix` folder. We will load the first 50,000 word vectors in the file and later use these to initialize a TensorFlow variable. We will also record the word strings of each word, as we will later provide these as labels for each point to display on TensorBoard:

```
vocabulary_size = 50000

embedding_df = []
index = []
# Open the zip file
with zipfile.ZipFile('glove.6B.zip') as glovezip:
    # Read the file with 50 dimensional embeddings
    with glovezip.open('glove.6B.50d.txt') as glovefile:
        # Read Line by Line
        for li, line in enumerate(glovefile):
            # Print progress
            if (li+1)%10000==0: print('.',end='')

            # Get the word and the corresponding vector
            line_tokens = line.decode('utf-8').split(' ')
            word = line_tokens[0]
            vector = [float(v) for v in line_tokens[1:]]

            assert len(vector)==50
            index.append(word)
```

```

# Update the embedding matrix
embedding_df.append(np.array(vector))

# If the first 50000 words being read, finish
if li >= vocabulary_size-1:
    break

embedding_df = pd.DataFrame(embedding_df, index=index)

```

We have defined our embeddings as a pandas DataFrame. It has the vector values as columns and words as the index.

	0	1	2	3	4	5	6	7	8	9	...	40	41	42	43	44
the	0.418000	0.249680	-0.41242	0.121700	0.345270	-0.044457	-0.49688	-0.178620	-0.000680	-0.656800	...	-0.298710	-0.157490	-0.347580	-0.045637	-0.442510
.	0.013441	0.236820	-0.16899	0.409510	0.638120	0.477090	-0.42852	-0.556410	-0.364000	-0.239380	...	-0.080262	0.630030	0.321110	-0.467650	0.227860
.	0.151640	0.301770	-0.16763	0.176840	0.317190	0.339730	-0.43478	-0.310860	-0.449990	-0.294860	...	-0.000064	0.068987	0.087939	-0.102850	-0.139310
of	0.708530	0.570880	-0.47160	0.180480	0.544490	0.726030	0.18157	-0.523930	0.103810	-0.175660	...	-0.347270	0.284830	0.075693	-0.062178	-0.389880
to	0.680470	-0.039263	0.30196	-0.177920	0.429620	0.032246	-0.41376	0.132280	-0.298470	-0.085253	...	-0.094375	0.018324	0.210480	-0.030880	-0.197220
and	0.268180	0.143460	-0.27877	0.016257	0.113840	0.699230	-0.51332	-0.473680	-0.330750	-0.138340	...	-0.069043	0.368850	0.251680	-0.245170	0.253810
in	0.330420	0.249950	-0.60874	0.109230	0.036372	0.151000	-0.55083	-0.074239	-0.092307	-0.328210	...	-0.486090	-0.008027	0.031184	-0.365760	-0.426990
a	0.217050	0.465150	-0.46757	0.100820	1.013500	0.748450	-0.53104	-0.262560	0.168120	0.131820	...	0.138130	0.369730	-0.642890	0.024142	-0.039315
"	0.257690	0.456290	-0.76974	-0.376790	0.592720	-0.063527	0.20545	-0.573850	-0.290090	-0.136620	...	0.030498	-0.395430	-0.385150	-1.000200	0.087599
's	0.237270	0.404780	-0.20547	0.588050	0.655330	0.328670	-0.81984	-0.232360	0.274280	0.242650	...	-0.123420	0.659610	-0.518020	-0.829950	-0.082739

10 rows × 50 columns

Figure A.3: GloVe vectors presented as a pandas DataFrame

We will need to define TensorFlow-related variables and operations. Before doing this, we will create a directory called `embeddings`, which will be used to store the variables:

```

# Create a directory to save our model
log_dir = 'embeddings'
os.makedirs(log_dir, exist_ok=True)

```

Then, we will define a variable that will be initialized with the word embeddings we copied from the text file earlier:

```

# Save the weights we want to analyse as a variable.
embeddings = tf.Variable(embedding_df.values)
print(f"weights.shape: {embeddings.shape}")

# Create a checkpoint from embedding
checkpoint = tf.train.Checkpoint(embedding=embeddings)

```

```
checkpoint.save(os.path.join(log_dir, "embedding.ckpt"))
```

We also need to save a metadata file. A metadata file contains labels/images or other types of information associated with the word embeddings, so that when you hover over the embedding visualization, the corresponding points will show the word/label they represent. The metadata file should be of the .tsv (tab-separated values) format and should contain vocabulary_size rows in it, where each row contains a word in the order they appear in the embeddings matrix:

```
with open(os.path.join(log_dir, 'metadata.tsv'), 'w', encoding='utf-8') as f:  
    for w in embedding_df.index:  
        f.write(w+'\n')
```

Then, we will need to tell TensorFlow where it can find the metadata for the embedding data we saved to the disk. For this, we need to create a ProjectorConfig object, which maintains various configuration details about the embedding we want to display. The details stored in the ProjectorConfig folder will be saved to a file called projector_config.pbtxt in the models directory:

```
config = projector.ProjectorConfig()
```

Here, we will populate the required fields of the ProjectorConfig object we created. First, we will tell it the name of the variable we're interested in visualizing. Then, we will tell it where it can find the metadata corresponding to that variable:

```
config = projector.ProjectorConfig()  
  
# You can add multiple embeddings. Here we add only one.  
embedding_config = config.embeddings.add()  
embedding_config.tensor_name = "embedding/.ATTRIBUTES/VARIABLE_VALUE"  
# Link this tensor to its metadata file (e.g. labels).  
embedding_config.metadata_path = 'metadata.tsv'  
  
# TensorBoard will read this file during startup.  
projector.visualize_embeddings(log_dir, config)
```

Note that we are adding the suffix /.ATTRIBUTES/VARIABLE_VALUE to the name embedding. This is required for TensorBoard to find this tensor. TensorBoard will read the necessary files at startup:

```
projector.visualize_embeddings(log_dir, config)
```

Now if you load TensorBoard, you should see something similar to *Figure A.4*:

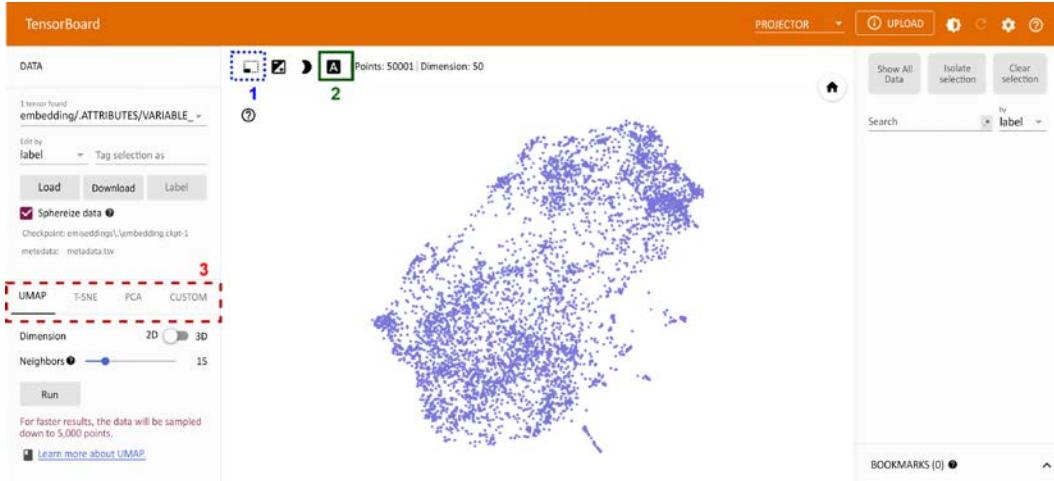


Figure A.4: TensorBoard view of the embeddings

When you hover over the displayed point cloud, it will show the label of the word you're currently hovering over, as we provided this information in the `metadata.tsv` file. Furthermore, you have several options. The first option (shown with a dotted line and marked as 1) will allow you to select a subset of the full embedding space. You can draw a bounding box over the area of the embedding space you're interested in, and it will look as shown in *Figure A.5*. I have selected the embeddings from the right side of the visualization. You can see the full list of selected words on the right:



Figure A.5: Selecting a subset of the embedding space

Another option you have is the ability to view words themselves, instead of dots. You can do this by selecting the second option in *Figure A.4* (shown inside a solid box and marked as 2). This would look as shown in *Figure A.6*. Additionally, you can pan/zoom/rotate the view to your liking. If you click on the help button (shown within a solid box and marked as 1 in *Figure A.6*), it will show you a guide for controlling the view:



Figure A.6: Embedding vectors displayed as words instead of dots

Finally, you can change the visualization algorithm from the panel on the left-hand side (shown with a dashed line and marked with 3 in *Figure A.4*).

Summary

Here we discussed some of the mathematical background as well as some implementations we did not cover in the other chapters. First, we discussed the mathematical notation for scalars, vectors, matrices, and tensors. Then, we discussed various operations performed on these data structures such as matrix multiplication and inversion. After that, we discussed various terminology that is useful for understanding probabilistic machine learning, such as probability density functions, joint probability, marginal probability, and Bayes' rule. Finally, we ended the appendix with a guide to visualizing word embeddings using TensorBoard, a visualization platform that comes with TensorFlow.



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

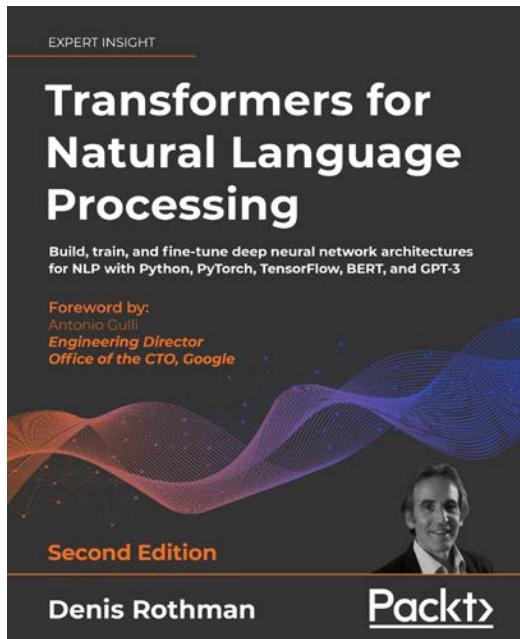
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



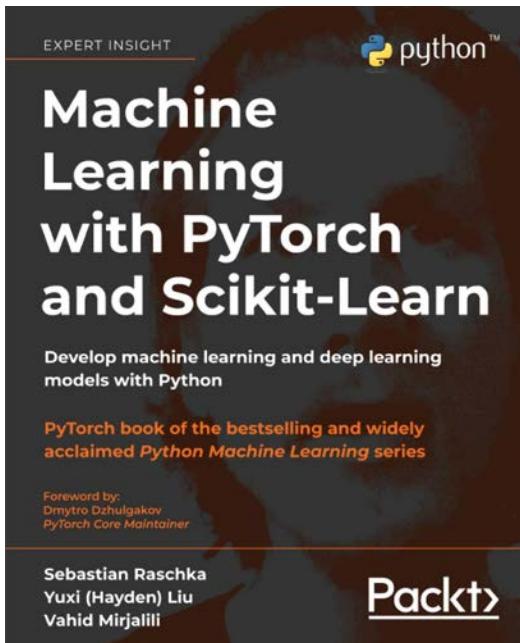
Transformers for Natural Language Processing, Second Edition

Denis Rothman

ISBN: 9781803247335

- Find out how ViT and CLIP label images (including blurry ones!) and create images from a sentence using DALL-E

- Discover new techniques to investigate complex language problems
- Compare and contrast the results of GPT-3 against T5, GPT-2, and BERT-based transformers
- Carry out sentiment analysis, text summarization, casual speech analysis, machine translations, and more using TensorFlow, PyTorch, and GPT-3
- Measure the productivity of key transformers to define their scope, potential, and limits in production



Machine Learning with PyTorch and Scikit-Learn

Sebastian Raschka

Yuxi (Hayden) Liu

Vahid Mirjalili

ISBN: 9781801819312

- Explore frameworks, models, and techniques for machines to ‘learn’ from data
- Use scikit-learn for machine learning and PyTorch for deep learning
- Train machine learning classifiers on images, text, and more
- Build and train neural networks, transformers, and boosting algorithms
- Discover best practices for evaluating and tuning models
- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Natural Language Processing with TensorFlow, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

1-gram precision 354

A

Adam optimizer 13

additive attention layer

reference link 344

Amazon Web Services (AWS)

URL 21

Anaconda

download link 18

installing 18

Application Programming Interface (API) 24

attention patterns

visualizing 355-360

AutoGraph 26

Automatic Language Processing Advisory Committee (ALPAC) 314

average pooling 157

B

backpropagation (BP) 197

avoiding, for RNNs 199

working 197, 198

Backpropagation Through Time

(BPTT) 197, 256

limitations 201, 202

RNNs, training 200

bag-of-words 6, 8

batch normalization 378

Bayes' rule 463, 464

beam 260

beam length 260

beam search 260, 301, 302

implementing 302-304

text, generating with 304, 305

used, for improving LSTMs 260, 261

bell curve 461

Bidirectional Encoder Representation from Transformers (BERT) 379, 380

answering questions, with 401-403

config, defining 396-399

data, exploring 387-389

implementing 389

input processing 381

model, defining 396-399

model, evaluating 399, 400

model, training 399, 400

tasks, solved 381-384

TensorFlow dataset, defining 393-395

Tokenizer, implementing and using 389-393

using to answer questions 386 395, 396

- bidirectional LSTMs (BiLSTMs)** 244, 263-265
bi-directional RNNs 240
Bilingual Evaluation Understudy (BLEU) 351, 353, 354, 438, 439
calculating 355
modified precision 354
brevity penalty 355
- C**
- Cafe Le TensorFlow 2** 33, 34
candidate value 249
captions
generating, for test images 443-448
Central Processing Units (CPUs) 13
character-based TextVectorization layer
defining 235
character embeddings
convolution, performing 237, 238
Named Entity Recognition (NER) with 231
char_vectorize_layer
inputs, processing 235, 236
chatbot 361
evaluating 363
training 362
cloud-based services
benefits 21
comparison operations 45, 46
computational graph 25
Compute Unified Device Architecture (CUDA) 24
concept 113
Conda environment
creating 19
conditional probability 462
CoNLLPP 206
- Consensus-based Image Description Evaluation (CIDEr)** 441
context vector (v) 323
Continuous Bag-of-Words (CBOW) algorithm 102, 103
data, generating for 103, 104
evaluating 106-108
implementing, in TensorFlow 104-106
training 106-108
continuous random variable 460
convolution
using, to generate token embeddings 231-234
Convolutional Neural Networks (CNNs) 13, 134, 148, 151
convolution operation 151
formation, summarizing 158
fully connected layers 158
fundamentals 148-150
pooling operation 155
power 151
used, for image classification on Fashion-MNIST with CNN 159
using, for sentence classification 168, 182-186
convolution operation 51-54
convolution operation, CNNs 151
standard convolution operation 152
stride parameter, using 152
transposed convolution 154, 155
with padding 153, 154
convolution window or filter 148
co-occurrence matrices 72
advantages 113

D

dataflow graph 25
data generators
 implementing, with TensorFlow 83-95
dataset, NMT system 324, 325
 sentences, padding 328-330
 sequence lengths, defining for two languages 327, 328
 special tokens, adding 326
 splitting, into testing set 326
 splitting, into training set 326
 splitting, into validation set 326
decaying the learning rate technique 186
declarative programming 38
decoder 321-324
decoder, stack of layers
 attention layer 371
 fully-connected layer 371
 masked self-attention layer 370
deconvolution 154
DeepL 318
deep learning, history of 11-13
 attention-based models 15
 current state 13, 14
 fully connected neural network (FCNN), working 16, 17
 non-sequential models 15
 sequential models 15
deep LSTMs
 used, for improving NMT performance 353
determinant 458, 459
directories
 defining, training and testing images 410
discrete random variable 460
distributed representation 67

document classification, with ELMo 135

 dataset 135-138

document embeddings

 generating 138-141

 used, for classifying documents 142-145

E

eager execution mode 29

element-wise matrix multiplication 455

Elman network 192

embedding layer 322

Embeddings from Language Models (ELMo) 126

 document classification 134

 downloading, from TensorFlow Hub 128, 129

 evaluating 126

 inputs, preparing from 129-132

 used, for generation of embeddings 132-134

encoder 321-323

encoder-decoder model

 decoder, defining 333-336

 defining 330

 encoder, defining 332, 333

 tokens, converting to IDs 331, 332

 visualizer model, defining 344, 345

encoder, stack of layers

 fully-connected layer 370

 self-attention layer 370

encoder states

 analyzing 336-338

 attention, computing 338-340

 attention, implementing 340-344

energy value 339

exploding gradient 202

F

FastText 133
reference link 133

feature engineering 6, 7

features map 152

feed-forward neural networks
issue with 193, 194

filter size 151

full decoder
defining 431-434

fully connected layer 370

fully connected neural network (FCNN)
working 16, 17

fully-connected subnetwork 425

Functional API 57, 58

G

Gated Recurrent Units (GRUs) 231, 244, 266-268, 296, 322
reviewing 296, 297

gather operation 49

gating mechanism, LSTMs
forget gate 247
input gate 246
output gate 247

Gaussian integral
reference link 462

General Language Understanding Evaluation (GLUE)
URL 381

Global Vectors (GloVe) 112-116
data, generating 120-122
evaluating 123-126
implementing 116-119

reference link 113
training 123-126

Google Cloud Platform (GCP)
URL 21

Google Colab
URL 21

Graphical Processing Units (GPUs) 13

greedy sampling 259
used, for improving LSTMs 259, 260

Group Method of Data Handling (GMDH) 12

GRU-based language model 297

H

hashing trick 171

Hidden Markov Model (HMM) 9

Hugging Face's model registry
reference link 386

Hugging Face transformers library 386, 387

hyperparameters
defining, for RNN 215

I

identity matrix 453

IDs
tokens, converting to 331, 332

ILSVRC ImageNet dataset 407
reference link 407

image caption generation
components 422
machine learning pipeline 422
metrics, evaluating 437, 438
text-based decoder Transformer 425
Vision Transformer (ViT) 423, 424

- image captioning model**
evaluating 442, 443
training 434-437
- Image Captioning on COCO**
reference link 437
- image classification, on Fashion-MNIST**
CNN, implementing 162-167
predictions produced with CNN,
analyzing 167, 168
- image classification, on Fashion-MNIST with CNN** 159
data, downloading and exploring 160-162
- ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** 151
- indexing of matrix** 452, 453
- inference model**
defining 288-292
- inputs** 38
defining, in TensorFlow 39
- insertion phase** 316
- inverse** 456, 457
- J**
- JFT-300M** 423
reference link 423
- joint probability** 463
- Jupyter Notebook**
accessing 19, 20
- K**
- Keras** 56
Functional API 57, 58
model building 57
neural network, implementing 61
Sequential API 57, 58
sub-classing API 57-59
- URL 57
- Keras API**
reference link 57
- Keras callbacks**
reference link 99
- Keras initializer**
reference link 28
- Keras layers API**
reference link 62
- L**
- lambda functions** 42
- language model**
compiling 286-288
implementing 282
LSTM-based model, defining 284-286
metrics, defining 286-288
text, generating with 292-295
TextVectorization layer, defining 283, 284
training 288
- Large Scale Visual Recognition Challenge (LSVRC)** 13
- Latent Semantic Analysis (LSA)** 113
- layer normalizations** 378
- learning algorithm** 6, 9
- LeNet** 163
- localist representation** 71
- Long Short-Term Memory (LSTM)** 14, 134, 231, 243-245
abstract view, of data flow 246
bidirectional LSTMs 244
cell operations 253, 254
example 246
functioning 246
gating mechanism 246
improving 305

- language modeling example 247-253
- softmax output layer 254, 255
- vanishing gradient problem,
resolving 256-259
- versus standard RNNs 255
- loss**
 - defining 56
- LSTM-based model**
 - defining 284-286
 - parameters 285
- LSTM components**
 - cell state 245
 - forget gate 245
 - hidden state 245
 - input gate 245
 - output gate 245
- LSTM performance improvement** 259
 - beam search 260, 261
 - bidirectional LSTMs (BiLSTM) 263-265
 - greedy sampling 259
 - word vectors, using 261, 262
- LSTMs, quality**
 - curse of dimensionality 306
 - text, generating with Word2vec 306-308
 - Word2vec, to rescue 306
- LSTMs, with peepholes** 298
 - code 299
 - reviewing 298
- LSTM variants** 265
 - Gated Recurrent Units (GRUs) 266-268
 - peephole connections 265
- M**
- machine learning pipeline**
 - for image caption generation 422
- machine translation (MT)** 4, 312, 313
 - history 313
- macro-averaged accuracy** 223
- many-to-many RNNs** 205
- many-to-one RNNs** 204
- map function** 42
- marginal probability distribution** 463
- Masked Language Modeling (MLM) task** 385
- mathematical operations, in TensorFlow** 46, 47
 - reference link 46
- matrices** 452
- matrix inverse**
 - finding, with SVD 457, 458
- matrix multiplication** 455
- metric averaging types**
 - macro 224
 - micro 224
 - weighted 224
- Metric for Evaluation of Translation with Explicit Ordering (METEOR)** 440, 441
- Microsoft Common Objects in Context (MS-COCO) dataset** 405, 408, 409
 - reference link 407
- MNIST dataset**
 - reference link 59
- Moore-Penrose inverse** 457
- multilayer perceptron** 16

N

- Named Entity Recognition (NER)** 3, 206
 - with character embeddings 231
 - with RNNs 206
 - with token embeddings 231
- Named Entity Recognition (NER) with RNN**
 - data 206-211
 - data, processing 212-215

- evaluation metrics 223-226
- hyperparameters, defining 215
- loss function 223-226
- model, defining 216, 219-223
- outputs, analyzing visually 229, 230
- Natural Language Processing (NLP) 2**
 - current state 13, 14
 - deep learning approach to 11
 - traditional approach to 5
- Natural Language Processing (NLP) tasks**
 - machine translation (MT) 4
 - Named Entity Recognition (NER) 3
 - Part-of-Speech (PoS) tagging 3
 - QA techniques 4
 - sentence/synopsis classification 3
 - taxonomy 4
 - text generation 3
 - tokenization 3
 - Word-Sense Disambiguation (WSD) 3
- Neocognitron 12**
- NER model**
 - character-based TextVectorization layer, defining 235
 - char_vectorize_layer inputs, processing 235, 236
 - convolution, performing on character embeddings 237, 238
 - hyperparameters, defining 235
 - implementing 234
 - input layer, defining 235
 - model evaluation 239
 - model training 239
 - token-based TextVectorization layer, defining 235
- Neural Machine Translation (NMT) 316-320**
 - inference with 360, 361
 - model, training 346-352
- neural network**
 - data, preparing 60, 61
 - implementing, with Keras 59-62
 - model, testing 63
 - model, training 62, 63
- neural network-related operations 49**
 - convolution operation 51-54
 - loss, defining 56
 - nonlinear activations 49, 50
 - pooling operation 54
- Next Sentence Prediction (NSP) task 385, 386**
- n-gram 8**
- NMT architecture 320-322**
 - context vector 323
 - decoder 323, 324
 - embedding layer 322
 - encoder 322, 323
- NMT performance**
 - improving, with deep LSTMs 353
- NMT system**
 - data, preparing 324
- node 30**
- normal/Gaussian distribution 461**
- norms 458**
- noun phrase (NP) 316**
- n-table 316**
- NumPy 25**
- NumPy arrays**
 - data, feeding as 39
- O**
- one-hot encoded representation 70**
- one-hot encoding 60**
- one-to-many RNN 203**

one-to-one RNNs 203

OpenAI-GPT-2 4

operations 38

defining, in TensorFlow 45

original word embeddings paper

using, two embedding layers 82

outputs 38

defining, in TensorFlow 45

P

padding 151, 153

Part-of-Speech (PoS) tagging 3

reference link 3

peephole connections 244, 265

perplexities 286

training and validation 300, 301

phrase-based translation 315

pooling operation 54

pooling operation, CNNs 155

average pooling 157, 158

max pooling 156

max pooling, with stride 156, 157

pooling over time 188

position embeddings 425

precision 353

preprocessing 7

pretrained GloVe word vectors

reference link 308

probability 459

conditional probability 462

joint probability 463

marginal probability 463

random variable 459

probability density function (PDF) 460-462

probability mass function (PMF) 460-462

pseudoinverse 457

Python

installing 18

Q

Question Answering (QA) techniques 4

R

random variables 459

continuous random variable 460

discrete random variables 460

Recall-Oriented Understudy for Gisting Evaluations (ROUGE) 439

Rectified Linear Units (ReLUs) 13

Recurrent Neural Network (RNN) 14, 191-193

modeling with 194-196

Named Entity Recognition (NER) with 206

performance, improving 239, 240

regularization techniques 240

technical description 196, 197

Recurrent Neural Network (RNN) applications 203

many-to-many RNNs 205

many-to-one RNNs 204

one-to-many RNN 203

one-to-one RNN 203

reordering phase 316

residual connections 377, 378

RNN layers 239

RNN on NER task

evaluating 226-229

training 226-229

r-table 316

rule-based translation 313-315

S

scalar 451

scatter operation 47, 48

scikit-learn

reference link 142

self-attention layer 369, 425

defining, with Keras subclassing API 427-429

output, computing 372-374

sentence classification

CNNs, using 168, 182-186

data, downloading 171

data, preparing 172-176

data, transforming for 169, 170

sentence classification CNN model 177

convolution operation 177-179

pooling operation 180-182

training 186-188

sentence/synopsis classification 3

sequence classification 381

sequence learning

masking 221

sequence-to-sequence (Seq2Seq) learning

application 361

Sequential API 57, 58

sequential models

improving 301, 302

single task 32

Singular Value Decomposition (SVD)

reference link 458

used, for finding inverse of matrix 457, 458

singular values 457

skip-gram algorithm 78

cross-entropy loss function 81

data preparation process 78

implementing, with TensorFlow 82

raw text to semi-structured text 78, 79

running, with TensorFlow 82

word embeddings, learning 79

skip-gram architecture

implementing, with TensorFlow 95-99

skip-gram model

evaluating 99-101

training 99-101

sklearn.metrics.average_precision_score

reference link 224

source language 312

source sentence

reversing 330

square diagonal matrix 454

standard inversion 457

standard LSTM 295

reviewing 295

Statistical Machine Translation (SMT) 313-316

stop words 86

stride 151-153

sub-classing API 58, 59

subsampling 93

subsampling operation 155

Swivel embeddings 134

reference link 134

symbols, in TensorFlow 2

reference link 24

syntax-based translation 316

T

t-Distributed Stochastic Neighbor Embedding (t-SNE) 69

teacher forcing 334

technical tools 18

description 18

tensor 454

TensorBoard

starting, steps 464, 465

used, for visualizing word embeddings 464

word embeddings, saving via 465-469

word embeddings, visualizing via 465-469

TensorFlow 24

architecture 31-33

Continuous Bag-of-Words (CBOW) algorithm, implementing 104-106

inputs, defining 39

operations, defining 45

outputs, defining 45

used, for implementing data generators 83-95

used, for implementing skip-gram algorithm 82

used, for implementing skip-gram architecture 95-99

used, for implementing text-based decoder 427

used, for implementing ViT model 426

used, for running skip-gram algorithm 82

variables, defining 43, 44

TensorFlow 1

tensors, defining 36

working 35-37

TensorFlow 1.x 27

limitations 38

TensorFlow 2

architecture 29-31

working with 24-29

TensorFlow GPU software prerequisites

reference link 19

TensorFlow Hub (TF Hub)

download link 127

Embeddings from Language Models (ELMo), downloading from 128, 129

reference link 387

TensorFlow installation

reference link 20

verifying 20

TensorFlow operations

comparison operations 45, 46

gather operation 49

mathematical operations 46, 47

reference link 45

scatter operations 47, 48

tensors 27

data, feeding as 39, 40

test sets

generating 274, 275

text-based decoder Transformer 425

implementing, with TensorFlow 427

text decoder model

connecting, with Vision Transformer (ViT) 426

text generation

with beam search 304, 305

with language model 292-295

with Word2vec 306-308

with words instead of n-grams 305

TextVectorization layer 217, 218

defining 283, 284

- tf.data API**
reference link 41
used, for building data pipeline 40-42
- tf.data Dataset**
defining, with data 416-422
- tf.data pipeline**
defining 276-282
- tf.gather**
reference link 226
- TF-IDF method** 71, 72
- tf.keras.layers.LayerNormalization**
reference link 378
- tf.random namespace**
reference link 102
- token-based TextVectorization layer**
defining 235
- token classification** 382
- token embeddings** 425
generating, with convolution 231-234
Named Entity Recognition (NER) with 231
- tokenization** 3
- Tokenizer** 79
building 176, 177
- tokens**
converting, to IDs 331, 332
- traditional approach, NLP** 5
drawbacks 10, 11
feature engineering 6
learning algorithm 6
prediction of test data 6
preprocessing 5
use case 7-10
- trained model**
using, to generate captions for unseen images 443-448
- training dataset**
generating 274, 275
preprocessing 410-414
- Transformer architecture** 367, 368
decoder 368
encoder 368
layer normalizations 377-379
layers, embedding 374-377
residual connections 377-379
self-attention layer, output computing 372-374
- Transformer layers**
defining 429-431
fully-connected subnetwork 425
self-attention layer 425
- Transformer models** 14, 134
data, tokenizing 414-416
- translation phase** 316
- transposed convolution** 151, 154
- transpose operation** 454, 455
- Truncated Backpropagation Through Time (TBPTT)** 200
- turing test** 363
- V**
- validation, data**
generating 274, 275
- vanishing gradient** 201
- variable initializers** 28
- variables** 38
defining, in TensorFlow 43, 44
- vector** 451
- velocity term** 201
- Virtual Assistants (VAs)** 2
- Vision Transformer (ViT)** 423, 424
connecting, with text decoder models 426

implementing, with TensorFlow 426
reference link 424, 426

vocabulary size

analyzing 275, 276

W**whitening** 60**Word2vec** 73

example, for learning word representations 74-78
reference link 308
text, generating with 306-308

word alignment problem 4**word embeddings**

saving, via TensorBoard 465-469
visualizing, via TensorBoard 465-469
visualizing, with TensorBoard 464

word representation

co-occurrence matrices 72
one-hot encoded representation 70
TF-IDF method 71, 72

word representation, learning

classical approaches 70

word representation/meaning 69

exercise 74-78

Word-Sense Disambiguation (WSD) 3**word vectors** 261

used, for improving LSTMs 261, 262

Z**Zipf's law** 90

