Chapter 9

Natural Language Processing with TensorFlow: Sentiment Analysis

In the previous chapters, the discussion focused on computer vision tasks such as image classification and image segmentation, where convolutional neural networks like Inception Net and DeepLab v3 were used as backbones for solving visually grounded problems. This chapter shifts attention to another dominant data modality: text. Text data is pervasive, particularly on the web, making natural language processing (NLP) a critical area in modern machine learning. NLP enables machines to extract meaning from textual data and power applications such as sentiment analysis, language modeling, and machine translation.

NLP encompasses a wide range of tasks, from simple text normalization operations to complex semantic understanding problems. Common NLP tasks include stop word removal, lemmatization, part-of-speech tagging, named entity recognition, language modeling, sentiment analysis, and machine translation. Each of these tasks contributes to transforming raw text into a structured and meaningful representation that can be processed by machine learning models.

In this chapter, the focus is on **sentiment analysis**, specifically classifying video game reviews as positive or negative. The chapter walks through the complete workflow, starting from data acquisition and preprocessing, moving through data pipeline construction, and finally training and evaluating a recurrent neural network model based on Long Short-Term Memory (LSTM) units. A key challenge addressed throughout the chapter is **class imbalance**, which is common in real-world textual data.

9.1 What the Text? Exploring and Processing Text

The goal of this section is to understand and preprocess textual data before feeding it into a deep learning model. The use case involves building a sentiment analyzer for an online video game store, where textual reviews are considered more informative than numerical star ratings. The chosen dataset consists of Amazon video game reviews, each containing review text, star ratings, and metadata such as whether the reviewer is a verified buyer.

The first step involves downloading and extracting the dataset, which is stored as a compressed JSON file. Each line in the JSON file represents a single review as a collection of key-value pairs, including the review text and rating.

Listing 9.1 Downloading the Amazon review data set

```
import os
import requests
import gzip
import shutil

# Retrieve the data
if not os.path.exists(os.path.join('data','Video_Games_5.json.gz')):
    url =
➥ "http://deepyeti.ucsd.edu/jianmo/amazon/categoryFilesSmall/Video_Games_
➥ 5.json.gz"
    # Get the file from web
    r = requests.get(url)

    if not os.path.exists('data'):
        os.mkdir('data')

    # Write to a file
    with open(os.path.join('data','Video_Games_5.json.gz'), 'wb') as f:
        f.write(r.content)
else:
    print("The tar file already exists.")

if not os.path.exists(os.path.join('data', 'Video_Games_5.json')):
    with gzip.open(os.path.join('data','Video_Games_5.json.gz'), 'rb') as f_in:
        with open(os.path.join('data','Video_Games_5.json'), 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
else:
    print("The extracted data already exists")
```

If the gzip file has not been downloaded, download it and save it to the disk.

If the gzip file is located in the local disk, don't download it.

If the gzip file exists but has not been extracted, extract it.

After loading the JSON data into a pandas DataFrame using pd.read_json() with orient='records', only the relevant columns—such as the review text, star rating, and verification status—are retained. Records with missing or empty review text are removed to ensure data quality. Additionally, only reviews from verified buyers are kept to preserve the integrity and reliability of the dataset.

Table 9.1 Sample data from the Amazon review data set

	overall	verified	reviewTime	reviewText
0	5	True	10 17, 2015	This game is a bit hard to get the hang of, bu...
1	4	False	07 27, 2015	I played it a while but it was alright. The st...
2	3	True	02 23, 2015	ok game.
3	2	True	02 20, 2015	found the game a bit too complicated, not what...
4	5	True	12 25, 2014	great game, I love it and have played it since...

An exploratory analysis reveals a significant imbalance in the number of reviews across different star ratings, with 5-star reviews dominating the dataset. To simplify the classification task and reduce complexity, the problem is reframed as a binary classification task. Reviews with ratings of 4 or 5 stars are labeled as **positive**, while reviews with ratings of 3 stars or below are labeled

as **negative**. This transformation highlights a strong class imbalance, with approximately 83% positive samples and 17% negative samples.

To avoid unintended ordering effects, the dataset is randomly shuffled. The review texts and labels are then separated into individual variables to facilitate further processing.

Text Cleaning and Preprocessing

Raw text data is inherently noisy due to inconsistencies in spelling, grammar, punctuation, and formatting. To address this, a structured preprocessing pipeline is introduced. The preprocessing steps include converting text to lowercase, handling shortened word forms, tokenizing text into words, removing punctuation and numbers, filtering out stop words, and lemmatizing words to their base forms.

Several resources from the Natural Language Toolkit (NLTK) library are required to perform these steps, including tools for tokenization, stop word removal, part-of-speech tagging, and lemmatization. These resources are downloaded explicitly to ensure reproducibility.

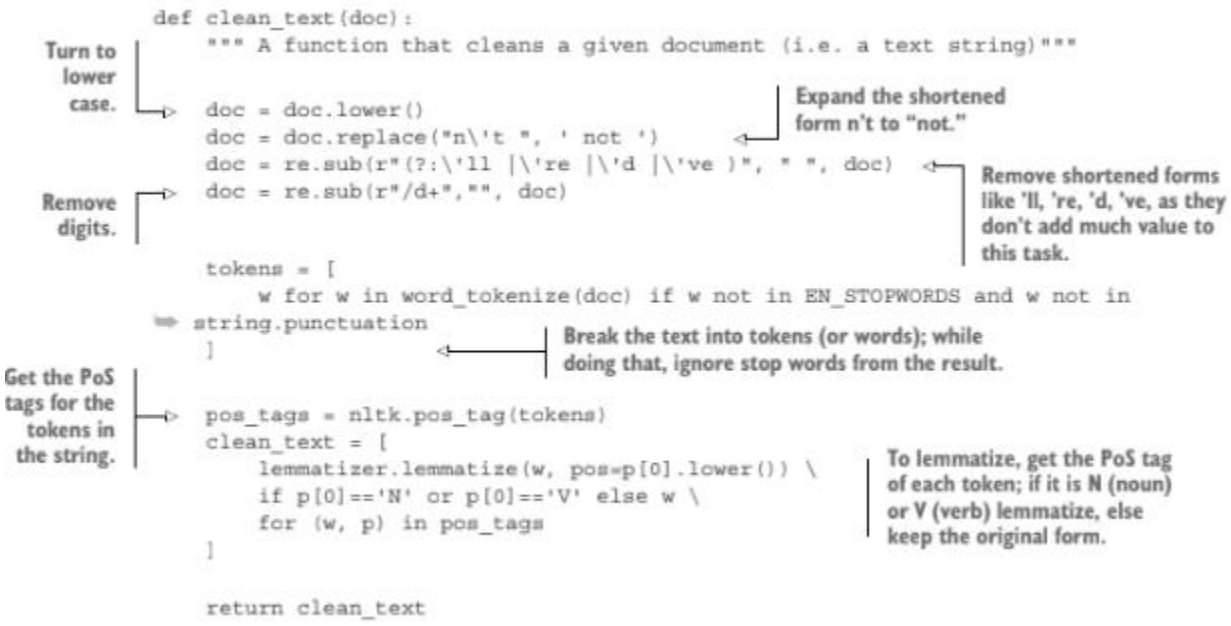
Lowercasing is performed to ensure that words with different casing are treated identically. Shortened negations such as “n’t” are expanded to “not” using regular expressions, as negation plays a critical role in sentiment interpretation. Other contractions such as ’ll, ’re, ’d, and ’ve are removed, as they contribute little semantic value for sentiment analysis.

Numeric characters, punctuation, and most stop words are removed next. However, unlike typical NLP pipelines, the words **“no”** and **“not”** are deliberately retained, as they can significantly alter sentiment polarity. Tokenization is performed using NLTK’s word_tokenize() function, which converts raw text into a sequence of word tokens.

Lemmatization is then applied to reduce words to their base forms, using the WordNet lemmatizer. This process relies on part-of-speech (PoS) tags to ensure accurate lemmatization, as the transformation rules differ for nouns and verbs. Only nouns and verbs are lemmatized to balance linguistic accuracy with computational efficiency.

The entire preprocessing workflow is encapsulated into a reusable function:

Listing 9.2 Preprocessing logic for reviews in the dataset



Applying this function to the dataset is computationally expensive, taking close to an hour to process all reviews. To avoid repeating this step, the processed inputs and corresponding labels are serialized and saved to disk.

Table 9.2 Original text versus preprocessed text

Original text	Clean text (tokenized)
Worked perfectly on Wii and GameCube. No issues with compatibility or loss of memory.	['work', 'perfectly', 'wii', 'gamecube', 'no', 'issue', 'compatibility', 'loss', 'memory']
Loved the game, and the other collectibles that came with it are well made. The mask is big, and it almost fits my face, so that was impressive.	['loved', 'game', 'collectible', 'come', 'well', 'make', 'mask', 'big', 'almost', 'fit', 'face', 'impressive']
It's an okay game. To be honest, I am very bad at these types of games and to me it's very difficult! I am always dying, which depresses me. Maybe if I had more skill I would enjoy this game more!	["'s", 'okay', 'game', 'honest', 'bad', 'type', 'game', '--', "'s", 'difficult', 'always', 'die', 'depresses', 'maybe', 'skill', 'would', 'enjoy', 'game']
Excellent product as described.	['excellent', 'product', 'describe']
The level of detail is great; you can feel the love for cars in this game.	['level', 'detail', 'great', 'feel', 'love', 'car', 'game']
I can't play this game.	['not', 'play', 'game']

At the end of this section, the dataset consists of clean, lemmatized token sequences paired with binary sentiment labels. This processed data forms the foundation for the TensorFlow data pipeline and LSTM-based sentiment classification model introduced in the subsequent sections.

9.2 Getting Text Ready for the Model

After cleaning, further processing is required to convert text into **numerical representations** suitable for machine learning models.

9.2.1 Splitting Training, Validation, and Test Data

To avoid **data leakage**, the dataset is split before any analysis:

- Balanced **validation** and **test** sets
- Remaining data used for **training**

Sampling is performed separately for positive and negative classes to ensure balanced evaluation datasets.

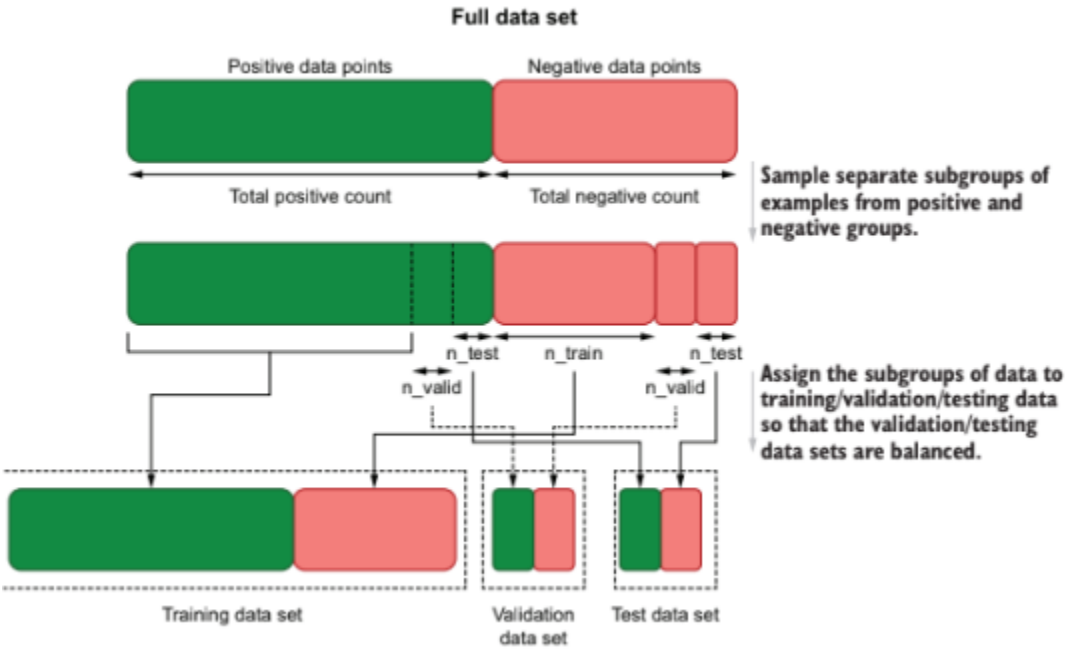


Figure 9.1 – The process for splitting training/validation/test data

The full splitting logic is wrapped into a reusable function.

Listing 9.3 – Splitting training, validation, and testing datasets

```
def train_valid_test_split(inputs, labels, train_fraction=0.8):
    """ Splits a given dataset into three sets; training, validation and test """

    neg_indices = pd.Series(labels.loc[(labels==0)].index)
    pos_indices = pd.Series(labels.loc[(labels==1)].index)

    n_valid = int(min([len(neg_indices), len(pos_indices)])
       * ((1-train_fraction)/2.0))
    n_test = n_valid

    neg_test_inds = neg_indices.sample(n=n_test, random_state=random_seed)
    neg_valid_inds = neg_indices.loc[~neg_indices.isin(
       neg_test_inds)].sample(n=n_test, random_state=random_seed)
```

Separate indices of negative and positive data points.

Compute the valid and test data set sizes (for minority class).

Get the indices of the minority class that goes to the test set.

Get the indices of the minority class that goes to the validation set.


```
    neg_train_inds = neg_indices.loc[~neg_indices.isin(
        neg_test_inds.tolist()+neg_valid_inds.tolist())]

    pos_test_inds = pos_indices.sample(n=n_test)
    pos_valid_inds = pos_indices.loc[
        ~pos_indices.isin(pos_test_inds)].sample(n=n_test)
    pos_train_inds = pos_indices.loc[
        ~pos_indices.isin(pos_test_inds.tolist()+pos_valid_inds.tolist())
    ]

    tr_x = inputs.loc[neg_train_inds.tolist() +
    pos_train_inds.tolist()].sample(frac=1.0, random_state=random_seed)
    tr_y = labels.loc[neg_train_inds.tolist() +
    pos_train_inds.tolist()].sample(frac=1.0, random_state=random_seed)
    v_x = inputs.loc[neg_valid_inds.tolist() +
    pos_valid_inds.tolist()].sample(frac=1.0, random_state=random_seed)
    v_y = labels.loc[neg_valid_inds.tolist() +
    pos_valid_inds.tolist()].sample(frac=1.0, random_state=random_seed)
    ts_x = inputs.loc[neg_test_inds.tolist() +
    pos_test_inds.tolist()].sample(frac=1.0, random_state=random_seed)
    ts_y = labels.loc[neg_test_inds.tolist() +
    pos_test_inds.tolist()].sample(frac=1.0, random_state=random_seed)

    print('Training data: {}'.format(len(tr_x)))
    print('Validation data: {}'.format(len(v_x)))
    print('Test data: {}'.format(len(ts_x)))

    return (tr_x, tr_y), (v_x, v_y), (ts_x, ts_y)
```

The rest of the indices in the minority class belong to the training set.

Compute the majority class indices for the test/ validation/ train sets

Get the training/ valid/test data sets using the indices created.

9.2.2 Analyzing the Vocabulary

Vocabulary size is treated as a **model hyperparameter**. Word frequencies are computed using only the training set to prevent leakage.

Analysis reveals that most words appear very infrequently. A threshold of **25 occurrences** is selected, resulting in a vocabulary size of approximately **11,800 words**.

9.2.3 Analyzing Sequence Length

Each review is a variable-length word sequence. Sequence length statistics are analyzed to guide bucketing decisions.

By excluding extreme outliers (top and bottom 10%), three buckets are defined:

- **Short**: [0, 5)
- **Medium**: [5, 15)
- **Long**: [15, ∞)

These boundaries later drive the TensorFlow bucketing strategy.

9.2.4 Text to Numbers with Keras Tokenizer

Text is converted to numerical IDs using the **Keras Tokenizer**, which:

- Builds a word-to-ID dictionary
- Handles out-of-vocabulary words
- Converts tokenized text into sequences of integers

The tokenizer is fit only on training data, then applied to validation and test sets.

This completes the transformation from raw text to numerical sequences.

9.3 Defining an End-to-End NLP Pipeline with TensorFlow

A **tf.data pipeline** is constructed to efficiently feed variable-length sequences into the model.

Key steps include:

- Combining labels and sequences
- Creating a tf.RaggedTensor to support variable-length data

- Truncating sequences to a maximum length
- Filtering empty reviews
- Bucketing sequences by length
- Padding sequences within each bucket
- Shuffling and batching
- Separating inputs and labels

Bucketing ensures that sequences of similar lengths are grouped together, minimizing unnecessary padding.

Listing 9.4 – The tf.data pipeline

```
def get_tf_pipeline(
    text_seq, labels, batch_size=64, bucket_boundaries=[5,15],
➥ max_length=50, shuffle=False
):
    """ Define a data pipeline that converts sequences to batches of data """

    data_seq = [[b]+a for a,b in zip(text_seq, labels) ]

    tf_data = tf.ragged.constant(data_seq)[:,:max_length]
```

Concatenate the label and the input sequence so that we don't mess up the order when we shuffle.

Define the variable sequence data set as a ragged tensor.


```
    text_ds = tf.data.Dataset.from_tensor_slices(tf_data)

    bucket_fn = tf.data.experimental.bucket_by_sequence_length(
        lambda x: tf.cast(tf.shape(x)[0],'int32'),
        bucket_boundaries=bucket_boundaries,
        bucket_batch_sizes=[batch_size,batch_size,batch_size],
        padded_shapes=None,
        padding_values=0,
        pad_to_bucket_boundary=False
    )

    text_ds = text_ds.map(lambda x: x).apply(bucket_fn)

    if shuffle:
        text_ds = text_ds.shuffle(buffer_size=10*batch_size)

    text_ds = text_ds.map(lambda x: (x[:,1:], x[:,0]))

    return text_ds
```

Create a data set out of the ragged tensor.

Bucket the data (assign each sequence to a bucket depending on the length).

For example, for bucket boundaries [5, 15], you get buckets [0, 5], [5, 15], [15,inf].

Apply bucketing.

Shuffle the data.

Split the data to inputs and labels.

This pipeline converts raw numerical sequences into batches ready for model training.

9.4 Happy Reviews Mean Happy Customers: Sentiment Analysis

With the data pipeline in place, the chapter introduces **recurrent neural networks**, focusing on **Long Short-Term Memory (LSTM)** models for sentiment classification.

9.4.1 LSTM Networks

LSTMs are well suited for sequential data due to their ability to maintain both **short-term** and **long-term memory**.

Sequential input data is represented as a **3D tensor**:

- Batch dimension
- Time (sequence) dimension
- Feature dimension

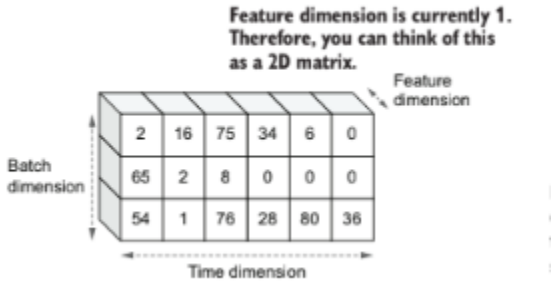


Figure 9.2 3D view of sequential data. Typically, sequential data is found with three dimensions; batch size, sequence/time, and feature.

An LSTM processes input step by step, maintaining:

- A **cell state** (long-term memory)
- An **output state** (short-term memory)

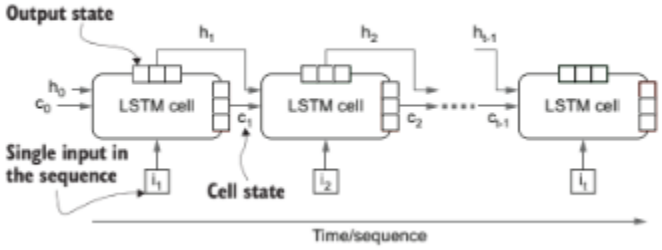


Figure 9.3 High-level longitudinal view of an LSTM cell. At a given time step t, the LSTM cell takes in two previous states (ht–1 and ct–1), along with the input, and produces two states (ht and ct).

The gating mechanism (input, forget, and output gates) controls information flow, enabling the model to capture complex linguistic dependencies such as negation and coreference.

Mathematical equations governing gate operations are presented for completeness.

Finally, the Keras LSTM layer is introduced, with configuration options such as units, return_state, and return_sequences.

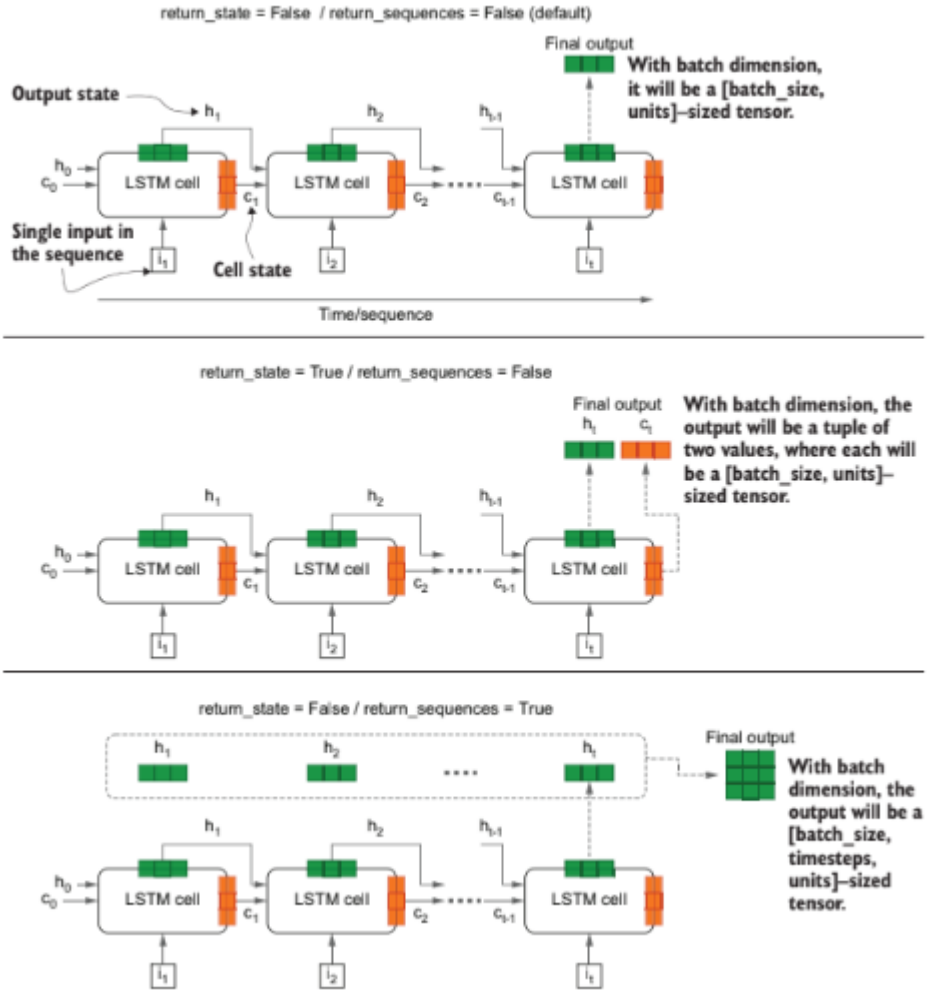


Figure 9.4 The changes to the output of the LSTM layer resulted in changes to the return_state and return_sequences arguments.

9.4.2 Defining the Final Model

In this section, the final sentiment analysis model is defined using the **Keras Sequential API**. The model is designed to process variable-length text sequences produced by the tf.data pipeline and output a binary sentiment prediction. The overall architecture combines masking, sequence encoding, recurrent modeling, and dense classification layers.

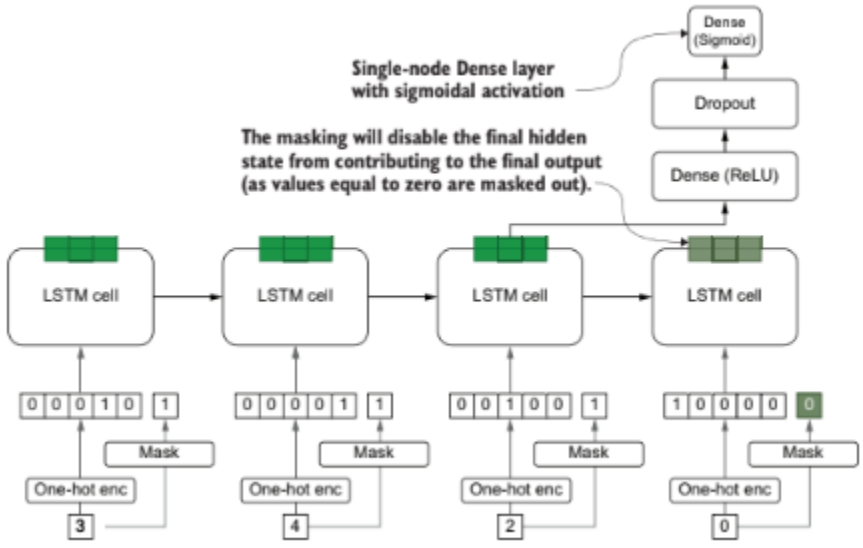


Figure 9.5 illustrates the complete high-level architecture of the sentiment analyzer.

The model begins with a **Lambda reshaping layer**, which transforms the two-dimensional [None, None] tensor produced by the data pipeline into a three-dimensional [None, None, 1] tensor. This reshaping is necessary because subsequent layers, such as masking and sequence-processing layers, expect a feature dimension.

Next, a **Masking layer** is introduced. Padding is required when batching variable-length text sequences, but padded values (typically zeros) do not carry semantic meaning. The masking layer ensures that these padded values do not influence model computations, particularly within the LSTM. The mask generated at this stage is automatically propagated to downstream layers that support masking.

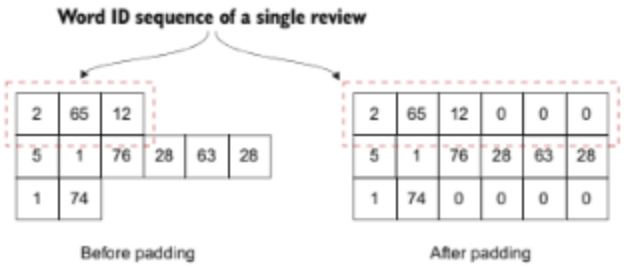


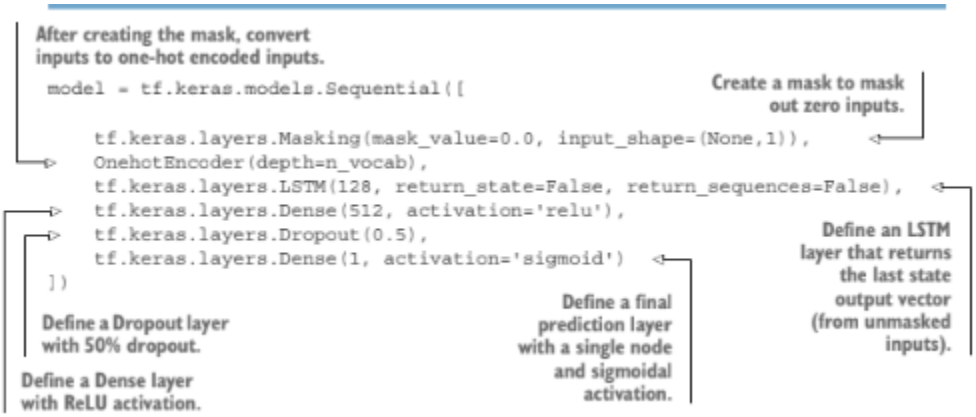
Figure 9.6 visualizes how text sequences look before and after padding, highlighting why masking is necessary.

Following masking, word IDs are transformed into one-hot encoded vectors using a **custom OnehotEncoder layer**. This layer converts integer-based word representations into high-dimensional sparse vectors, where the dimensionality equals the vocabulary size. Crucially, the layer explicitly propagates the mask to ensure consistent masking behavior through the network.

The encoded sequences are then passed into an **LSTM layer** with 128 units. For sentiment classification, only the **final output state** of the LSTM is required, so both return_sequences and return_state are set to False. This configuration ensures that the model captures the overall semantic meaning of each review rather than intermediate token-level representations.

The LSTM output is fed into a **Dense layer with 512 units and ReLU activation**, which learns higher-level abstractions useful for classification. A **Dropout layer** follows, randomly disabling 50% of activations during training to reduce overfitting. Finally, a **single-node Dense layer with sigmoid activation** produces a probability representing positive or negative sentiment.

Listing 9.5 Implementation of the full sentiment analysis model



The model is compiled using the **binary cross-entropy loss function**, which is appropriate for binary classification problems, along with the Adam optimizer and accuracy as the evaluation metric. The model summary confirms the dimensional transformations across layers and shows that the majority of trainable parameters originate from the LSTM and dense layers.

9.5 Training and Evaluating the Model

With the model defined, training begins by constructing **training and validation pipelines** using the previously defined tf.data pipeline. A batch size of 128 is used to balance computational efficiency and model convergence.

Due to the strong **class imbalance** in the dataset—where positive reviews significantly outnumber negative ones—a **class weighting strategy** is employed. Negative samples are assigned a higher loss weight so that the model pays greater attention to underrepresented negative sentiment during optimization.

The model is trained for 10 epochs using model.fit(), with class weights passed via the class_weight argument. Training is monitored using three callbacks: **CSV logging**, **learning rate scheduling**, and **early stopping**.

Listing 9.6 Training procedure for the sentiment analyzer

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

csv_logger = tf.keras.callbacks.CSVLogger(
       os.path.join('eval','1_sentiment_analysis.log'))   <- Log the performance metrics to a CSV file.

monitor_metric = 'val_loss'
mode = 'min'
print("Using metric={} and mode={} for EarlyStopping".format(monitor_metric,
     mode))

lr_callback = tf.keras.callbacks.ReduceLROnPlateau(      <- The learning rate reduction callback
    monitor=monitor_metric, factor=0.1, patience=3, mode=mode, min_lr=1e-8
)
```



```
es_callback = tf.keras.callbacks.EarlyStopping(
    monitor=monitor_metric, patience=6, mode=mode, restore_best_weights=False
)                                                        <- The early stopping callback

model.fit(
    train_ds,                  <- Train the model.
    validation_data=valid_ds,
    epochs=10,
    class_weight={0:neg_weight, 1:1.0},
    callbacks=[es_callback, lr_callback, csv_logger])
```

Training results indicate that the model achieves over **80% validation accuracy** on a balanced validation dataset. To ensure generalization, the trained model is saved and then evaluated on a previously unseen **test dataset**. The test accuracy closely matches validation performance, confirming that the model generalizes well.

9.6 Injecting Semantics with Word Vectors

Although the one-hot–based model performs reasonably well, it suffers from several limitations, including high dimensionality, sparsity, and the inability to capture semantic relationships between words. To address these issues, the chapter introduces **word embeddings**.

Word embeddings provide **dense, low-dimensional representations** of words that encode semantic similarity. Unlike one-hot vectors, embeddings allow the model to understand that words such as *cat* and *dog* are more similar to each other than to unrelated words like *volcano*.

The theoretical motivation for word embeddings is rooted in contextual similarity, famously summarized by the principle that *a word is known by the company it keeps*. Embedding algorithms such as Skip-gram, CBOW, GloVe, and ELMo leverage word co-occurrence patterns to learn these representations.

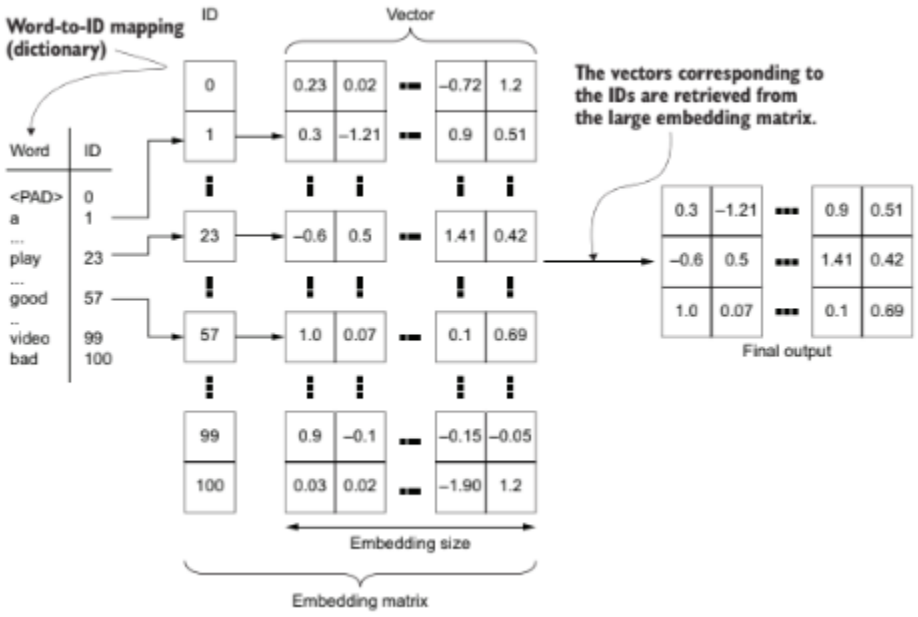
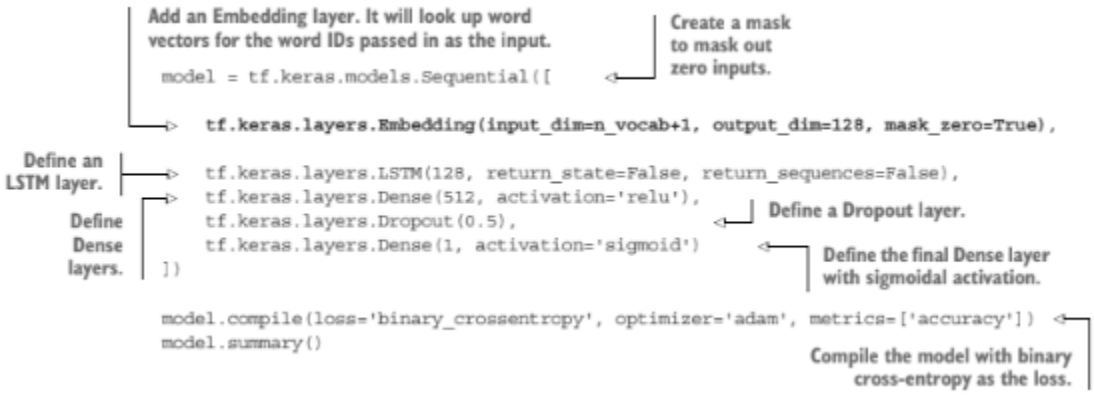


*Figure 9.7 An overview of how the embedding matrix is used to obtain word vectors. A lookup is performed using the input word IDs to fetch the vectors corresponding to those indices. The actual values of the vectors are learned during the model training***.**

9.6.2 Defining the Final Model with Word Embeddings

To enhance the sentiment analyzer, the one-hot encoding layer is replaced with a **trainable Embedding layer**. This layer introduces a (V + 1) × d embedding matrix, where V is the vocabulary size and d is the embedding dimension (set to 128). Padding-aware masking is handled directly within the embedding layer by enabling mask_zero=True.

This modification simplifies the model while significantly improving representational efficiency and semantic expressiveness. The rest of the architecture—LSTM, dense layers, dropout, and output layer—remains unchanged.



Listing 9.7 Implementing the sentiment analyzer with word embeddings

Special care is taken to ensure that **empty reviews** are removed during preprocessing. Feeding fully padded sequences into an LSTM can cause runtime errors, making this filtering step both a data-cleaning and model-stability requirement.

9.6.3 Training and Evaluating the Embedding-Based Model

Training and evaluation procedures remain identical to those used previously. The embedding-enhanced model converges faster and achieves **slightly higher validation and test accuracy** than the one-hot–based model.

Beyond accuracy metrics, the model's predictions are qualitatively evaluated by inspecting the **most positive and most negative reviews** in the test set. This manual inspection confirms that the model's decisions are semantically sensible, providing confidence in its real-world applicability.

Summary

This chapter demonstrated a complete **end-to-end NLP pipeline** for sentiment analysis using TensorFlow. It covered text preprocessing, numerical encoding, efficient data pipelines, LSTM-based sequence modeling, handling class imbalance, and improving semantic understanding through word embeddings. Together, these components illustrate how deep learning models can be effectively applied to real-world natural language tasks.