```
Epoch 98/100
120/120 [==============================] - 0s 128us/
sample - loss: 0.3522 - accuracy: 0.6500
Epoch 99/100
120/120 [==============================] - 0s 187us/
sample - loss: 0.3522 - accuracy: 0.6500
Epoch 100/100
120/120 [==============================] - 0s 167us/
sample - loss: 0.3521 - accuracy: 0.6500
```

Figure 6.1 displays a scatter plot of points based on the test values and the predictions for those test values.

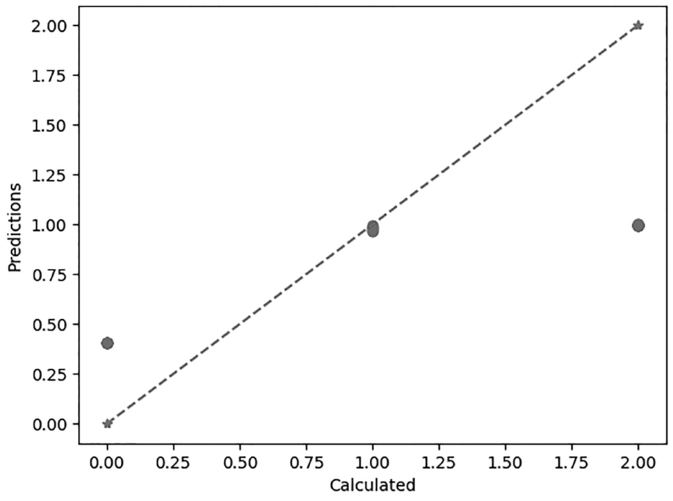

FIGURE 6.1: A scatter plot and a best-fitting line.

The accuracy is admittedly poor (abysmal?), and yet it's quite possible that you will encounter this type of situation. Experiment with a different number of hidden layers and replace the final hidden layer with a `Dense` layer that specifies a `softmax` activation function—or some other activation function—to see if this change improves the accuracy.

6.18 Summary

This chapter started with an explanation of classification and classifiers, followed by a brief explanation of commonly used classifiers in machine learning.

Next you learned about activation functions, why they are important in neural networks, and also how they are used in neural networks. Then you saw a list of the TensorFlow/`Keras` APIs for various activation functions, followed by a description of some of their merits.

You also learned about logistic regression that involves the sigmoid activation function, followed by a `Keras`-based code sample involving logistic regression.

CHAPTER 7

NATURAL LANGUAGE PROCESSING AND REINFORCEMENT LEARNING

- Working with Natural Language Processing (NLP)
- Popular NLP Algorithms
- What Are Word Embeddings?
- ELMo, ULMFit, OpenAI, BERT, and ERNIE 2.0 (optional)
- What is Translatotron?
- Deep Learning and NLP (optional)
- NLU versus NLG (optional)
- What is Reinforcement Learning (RL)?
- From NFAs to MDPs
- The Epsilon-Greedy Algorithm
- The Bellman Equation
- RL Toolkits and Frameworks
- Deep Reinforcement Learning (optional)
- Summary

This chapter provides a casual introduction you to natural language processing (NLP) and reinforcement learning (RL). Both topics can easily fill entire books, often involving complex topics, which means that this chapter provides a limited introduction to these topics. If you want to acquire a thorough grasp of BERT (discussed briefly later in the chapter), you need to learn about "attention" and the transformer architecture. Similarly, if you want to acquire a solid understanding of deep RL, then you need to understand deep learning architectures. After you finish reading the cursory introduction to NLP and RL in this chapter, you can find additional online information about the facets of NLP or RL that interest you.

The first section discusses NLP, along with some code samples in Keras. This section also discusses natural language understanding (NLU) and natural language generation (NLG).

The second section introduces RL, along with a description of the types of tasks that are well-suited to RL. You will learn about the `nchain` task and the epsilon-greedy algorithm that can solve problems that you cannot solve using a "pure" greedy algorithm. In this section you will also learn about the Bellman equation, which is a cornerstone of RLRL.

The third section discusses the TF-Agents toolkit from Google, deep RLRL (deep learning combined with RL), and the Google Dopamine toolkit.

7.1 Working with NLP

This section highlights some concepts in NLP, and depending on your background, you might need to perform an online search to learn more about some of the concepts (try Wikipedia). Although the concepts are treated in a very superficial manner, you will know what to pursue in order to further your study of NLP.

NLP is currently the focus of significant interest in the machine learning (ML) community. Some of the use cases for NLP are listed here:

- chatbots
- search (text and audio)
- text classification
- sentiment analysis
- recommendation systems
- question answering
- speech recognition
- NLU
- NLG

You encounter many of these use cases in every day life: when you visit Web pages, or perform an online search for books, or recommendations regarding movies.

7.1.1 NLP Techniques

The earliest approach for solving NLP tasks involved rule-based approaches, which dominated the industry for decades. Examples of techniques using rule-based approaches include regular expressions (RegExs) and context-free grammars (CFGs). RegExs are sometimes used in order to remove HTML tags from text that has been "scraped" from a Web page, or unwanted special characters from a document.

The second approach involved training a ML model with some data that is based on some user-defined features. This technique requires a considerable amount of feature engineering (a nontrivial task), and includes analyzing the text to remove undesired and superfluous content (including "stop" words), as well as transforming the words (e.g., converting uppercase to lowercase).

The most recent approach involves deep learning, whereby a neural network learns the features instead of relying on humans to perform feature engineering. One of the key ideas involves "mapping" words to numbers, which enables us to map sentences to vectors of numbers. After transforming documents to vectors, we can perform a myriad of operations on those vectors. For example, we can use the notion of vector spaces to define vector space models, where the distance between two vectors can be measured by the angle between them (related to cosine similarity). If two vectors are "close" to each other, then it's likelier that the corresponding sentences are similar in meaning. Their similarity is based on the *distributional hypothesis*, which asserts that words in the same contexts tend to have similar meanings.

A nice article that discusses vector representations of words, along with links to code samples, is here:

https://www.tensorflow.org/tutorials/representation/word2vec

7.1.2 The Transformer Architecture and NLP

In 2017, Google introduced the `Transformer` neural network architecture, which is based on a "self-attention" mechanism that is well-suited for language understanding.

Google showed that the `Transformer` outperforms earlier benchmarks for both `RNN`s and `CNN`s involving the translation of academic English to German as well as English to French. Moreover, the `Transformer` required less computation to train, and also improved the training time by as much as an order of magnitude.

The `Transformer` can process the sentence "I arrived at the bank after crossing the river" and correctly determine that the word "bank" refers to the shore of a river and not a financial institution. The `Transformer` makes this determination in a single step by making the association between "bank" and "river." As another example, the `Transformer` can determine the different meanings of “it” in these two sentences:

“The horse did not cross the street because it was too tired.”

“The horse did not cross the street because it was too narrow.”

The `Transformer` computes the next representation for a given word by comparing the word to every other word in the sentence, which results in an "attention score" for the words in the sentence. The `Transformer` uses these scores to determine the extent to which other words will contribute to the next representation of a given word.

The result of these comparisons is an attention score for every other word in the sentence. As a result, "river" received a high attention score when computing a new representation for "bank."

Although `LSTM`s and bidirectional `LSTM`s are heavily utilized in `NLP` tasks, the `Transformer` has gained a lot of traction in the AI community, not only for translation between languages, but also the fact that for some tasks it can outperform both `RNN`s and `CNN`s. The `Transformer` architecture requires much less computation time in order to train a model, which explain why some people believe that the `Transformer` has already begun to supplant `RNN`s and `LSTM`s.

The following link contains a TF 2 code sample of a `Transformer` neural network that you can launch in Google Colaboratory:

https://www.tensorflow.org/alpha/tutorials/text/transformer

Another interesting and recent architecture is called “attention augmented convolutional networks,” which is a combination of `CNN`s with self-attention. This combination achieves better accuracy than “pure” `CNN`s, and you can find more details in this paper: *https://arxiv.org/abs/1904.09925*

7.1.3 Transformer-XL Architecture

The `Transformer-XL` combines a `Transformer` architecture with two techniques called recurrence mechanism and relative positional encoding to obtain better results than a `Transformer`. `Transformer-XL` works with word-level and character-level language modeling.

The `Transformer-XL` and `Transformer` both process the first segment of tokens, and the former also keeps the outputs of the hidden layers. Consequently, each hidden layer receives two inputs from the previous hidden layer, and then concatenates them to provide additional information to the neural network.

According to the following article, `Transformer-XL` significantly outperforms `Transformer`, and its dependency is 80% longer than "vanilla" RNNs:

https://hub.packtpub.com/transformer-xl-a-google-architecture-with-80-longer-dependency-than-rnns/

7.1.4 Reformer Architecture

Recently the `Reformer` architecture was released, which uses two techniques to improve the efficiency (i.e., lower memory and faster performance on long sequences) of the `Transformer` architecture. As a result, the `Reformer` architecture also has lower complexity than the `Transformer`. More details regarding the `Reformer` are here:

https://openreview.net/pdf?id=rkgNKkHtvB

Some Reformer-related code is here: *https://pastebin.com/62r5FuEW*

7.1.5 NLP and Deep Learning

The `NLP` models that use deep learning can comprise `CNNs`, `RNNs`, `LSTMs`, and bi-directional `LSTMs`. For example, Google released `BERT` in 2018, which is an extremely powerful framework for `NLP`. `BERT` is quite sophisticated, and involves bidirectional transformers and so-called "attention" (discussed briefly later in this chapter).

Deep learning for NLP often yields higher accuracy than other techniques, but keep in mind that sometimes it's not as fast as rule-based and classical ML methods. In case you're interested, a code sample that uses TensorFlow and `RNNs` for text classification is here:

https://www.tensorflow.org/alpha/tutorials/text/text_classification_rnn

A code sample that uses TensorFlow and RNNs for text generation is here:

https://www.tensorflow.org/alpha/tutorials/text/text_generation

7.1.6 Data Preprocessing Tasks in NLP

There are some common preprocessing tasks that are performed on documents, as listed here:

- [1] lowercasing
- [1] noise removal
- [2] normalization
- [3] text enrichment
- [3] stopword removal
- [3] stemming
- [3] lemmatization

The preceding tasks can be classified as follows:

- [1]: mandatory tasks
- [2]: recommended tasks
- [3]: task dependent

In brief, preprocessing tasks involve at least the removal of redundant words ("a," "the," and so forth), removing the endings of words ("running," "runs," and "ran" are treated the same as "run"), and converting text from uppercase to lowercase.

7.2 Popular NLP Algorithms

Some of the popular NLP algorithms appear in the following list, and in some cases they are the foundation for more sophisticated NLP toolkits:

- BoW: Bag of Words
- n-grams and skip-grams
- TF-IDF: basic algorithm in extracting keywords
- Word2Vector (Google): O/S project to describe text

- GloVe (Stanford NLP Group)
- LDA: text classification
- CF (collaborative filtering): an algorithm in news recommend system (Google News and Yahoo News)

The topics in the first half of the preceding list are discussed briefly in subsequent sections.

7.2.1 What is an n-gram?

An n-gram is a technique for creating a vocabulary that is based on adjacent words that are grouped together. This technique retains some word positions (unlike BoW). You need to specify the value of "n" that in turn specifies the size of the group.

The idea is simple: for each word in a sentence, construct a vocabulary term that contains the n words on the left side of the given word and n words that are on the right side of the given word. As a simple example, "This is a sentence" has the following 2-grams:

```
(this, is), (is, a), (a, sentence)
```

As another example, we can use the same sentence "This is a sentence" to determine its 3-grams:

```
(this, is, a), (is, a, sentence)
```

The notion of n-grams is surprisingly powerful, and it's used heavily in popular open source toolkits such as `ELMo` and `BERT` when they pre-train their models.

7.2.2 What is a skip-gram?

Given a word in a sentence, a skip gram creates a vocabulary term by constructing a list that contains the n words on both sides of a given word, followed by the word itself. For example, consider the following sentence:

```
the quick brown fox jumped over the lazy dog
```

A skip-gram of size 1 yields the following vocabulary terms:

```
([the,brown], quick), ([quick,fox], brown),
([brown,jumped], fox),...
```

A skip-gram of size 2 yields the following vocabulary terms:

```
([the,quick,fox,jumped], brown),
([quick,brown,jumped,over], fox), ([brown,fox,over,the],
jumped),...
```

More details regarding skip-grams are discussed here:

https://www.tensorflow.org/tutorials/representation/word2vec#the_skip-gram_model

7.2.3 What is BoW?

BoW (Bag of Words) assigns a numeric value to each word in a sentence and treats those words as a set (or bag). Hence, BoW does not keep track of adjacent words, so it's a very simple algorithm.

Listing 7.1 displays the contents of the Python script `bow_to_vector.py` that illustrates how to use the BoW algorithm.

Listing 7.1: bow_to_vector.py

```
VOCAB = ['dog', 'cheese', 'cat', 'mouse']
TEXT1 = 'the mouse ate the cheese'
TEXT2 = 'the horse ate the hay'

 def to_bow(text):
  words = text.split(" ")
  return [1 if w in words else 0 for w in VOCAB]

print("VOCAB: ",VOCAB)
print("TEXT1:",TEXT1)
print("BOW1: ",to_bow(TEXT1))  # [0, 1, 0, 1]
print("")

print("TEXT2:",TEXT2)
print("BOW2: ",to_bow(TEXT2))  # [0, 0, 0, 0]
```

Listing 7.1 initializes a list `VOCAB` and two text strings `TEXT1` and `TEXT2`. The next portion of Listing 7.1 defines the Python function `to_bow()` that returns an array containing 0s and 1s: if a word in the current sentence appears in the vocabulary, then a 1 is returned (otherwise a 0 is returned). The last portion of Listing 7.1 invokes the Python function with two different sentences. The output from launching the code in Listing 7.1 is here:

```
('VOCAB: ', ['dog', 'cheese', 'cat', 'mouse'])
('TEXT1:', 'the mouse ate the cheese')
('BOW1: ', [0, 1, 0, 1])

('TEXT2:', 'the horse ate the hay')
('BOW2: ', [0, 0, 0, 0])
```

7.2.4 What is Term Frequency?

Term frequency is the number of times that a word appears in a document, which can vary among different documents. Consider the following simple example that consists of two "documents" `Doc1` and `Doc2`:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

The term frequency for the word "is" and the word "short" is given here:

```
tf(is) = 1/5 for doc1
tf(is) = 0 for doc2
tf(short) = 1/5 for doc1
tf(short) = 1/4 for doc2
```

The preceding values will be used in the calculation of `tf-idf` that is explained in a later section.

7.2.5 What is Inverse Document Frequency (idf)?

Given a set of N documents and given a word in a document, let's define `dc` and `idf` of each word as follows:

```
dc = # of documents containing a given word
idf = log(N/dc)
```

Now let's use the same two documents Doc1 and Doc2 from a previous section:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

The calculations of the `idf` value for the word "is" and the word "short" are shown here:

```
idf(is) = log(2/1) = log(2)
idf(short) = log(2/2) = 0
```

The following link provides more detailed information about inverse document frequency: https://en.wikipedia.org/wiki/Tf–idf#Example_of_tf–idf

7.2.6 What is `tf-idf`?

The term `tf-idf` is an abbreviation for "term frequency, inverse document frequency," and it's the product of the `tf` value and the `idf` value of a word, as shown here:

```
tf-idf = tf * idf
```

A high frequency word has a higher `tf` value but a lower `idf` value. In general, "rare" words are more relevant than "popular" ones, so they help to extract "relevance." For example, suppose you have a collection of 10 documents (real documents, not the toy documents we used earlier). The word “the” occurs frequently in English sentences, but it does not provide any indication of the topics in any of the documents. On the other hand, if you determine that the word “universe” appears multiple times in a single document, this information can provide some indication of the theme of that document, and with the help of NLP techniques, assist in determining the topic (or topics) in that document.

7.3 What are Word Embeddings?

An *embedding* is a fixed-length vector to encode and represent an entity (document, sentence, word, graph). Each word is represented by a real-valued vector, which can result in hundreds of dimensions. Furthermore, such an encoding can result in sparse vectors: one example is one-hot encoding, where one position has the value 1 and all other positions have the value 0.

Three popular word embedding algorithms are Word2vec, GloVe, and FastText. Keep in mind that these three algorithms involve unsupervised approaches. They are also based on the distributional hypothesis: words in the same contexts tend to have similar meanings: *https://aclweb.org/aclwiki/Distributional_Hypothesis*.

A good article regarding Word2Vec in TensorFlow is here:

https://towardsdatascience.com/learn-word2vec-by-implementing-it-in-tensorflow-45641adaf2ac

This article is useful if you want to see Word2Vec with FastText in gensim:

https://towardsdatascience.com/word-embedding-with-word2vec-and-fast-text-a209c1d3e12c

Another good article, and this one pertains to the skip-gram model:

https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b

A useful article that describes how FastText works “under the hood”:

https://towardsdatascience.com/fasttext-under-the-hood-11efc57b2b3

Along with the preceding popular algorithms there are also some popular embedding models, some of which are listed here:

- baseline averaged sentence embeddings
- Doc2Vec
- neural-net language models
- skip-thought vectors
- quick-thought vectors
- inferSent
- universal sentence encoder

Perform an online search for more information about the preceding embedding models.

7.4 ELMo, ULMFit, OpenAI, BERT, and ERNIE 2.0

During 2018 there were some significant advances in NLP-related research, resulting in the following toolkits and frameworks:

- `ELMo:      released in 02/2018`
- `ULMFit:    released in 05/2018`
- `OpenAI:    released in 06/2018`
- `BERT:      released in 10/2018`
- `MT-DNN:    released in 01/2019`
- `ERNIE 2.0: released in 08/2019`

`ELMo` is an acronym for "embeddings from language models," which provides deep contextualized word representations and state-of-the-art contextual word vectors, resulting in noticeable improvements in word embeddings.

Jeremy Howard and Sebastian Ruder created universal language model fine-tuning (ULMFit), which is a transfer learning method that can be applied to any task in NLP. ULMFit significantly outperforms the state-of-the-art on six text classification tasks, reducing the error by 18–24% on the majority of datasets.

Furthermore, with only 100 labeled examples, it matches the performance of training from scratch on 100x more data. ULMFit is downloadable from GitHub:

https://github.com/jannenev/ulmfit-language-model

OpenAI developed GPT-2 (a successor to GPT), which is a model that was trained to predict the next word in 40GB of Internet text. OpenAI chose not to release the trained model due to concerns regarding malicious applications of their technology.

GPT-2 is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million Web pages (curated by humans), with an emphasis on diversity of content. GPT-2 is trained to predict the next word, given all of the previous words within some text. The diversity of the dataset causes this goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.

`BERT` is an acronym for "bidirectional encoder representations from transformers." `BERT` can pass this simple English test (i.e., `BERT` can determine the correct choice among multiple choices):

```
On stage, a woman takes a seat at the piano. She:
a) sits on a bench as her sister plays with the doll.
b) smiles with someone as the music plays.
c) is in the crowd, watching the dancers.
d) nervously sets her fingers on the keys.
```

Details of `BERT` and this English test are here:

https://www.lyrn.ai/2018/11/07/explained-bert-state-of-the-art-language-model-for-nlp/

The `BERT` (TensorFlow) source code is available here on GitHub:

https://github.com/google-research/bert

https://github.com/hanxiao/bert-as-service

Another interesting development is MT-DNN from Microsoft, which asserts that MT-DNN can outperform Google `BERT`:

https://medium.com/syncedreview/microsofts-new-mt-dnn-outperforms-google-bert-b5fa15b1a03e

A Jupyter notebook with `BERT` is available, and you need the following in order to run the notebook in Google Colaboratory:

- a GCP (Google Compute Engine) account
- a GCS (Google Cloud Storage) bucket

Here is the link to the notebook in Google Colaboratory:

https://colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab/bert_finetuning_with_cloud_tpus.ipynb

In March, 2019 Baidu open sourced ERNIE 1.0 (Enhanced Representation through kNowledge IntEgration) that (according to Baidu) outperformed BERT in tasks involving Chinese language understanding. In August, 2019 Baidu open sourced ERNIE 2.0, which is downloadable here:

https://github.com/PaddlePaddle/ERNIE/

An article with additional information about ERNIE 2.0 (including its architecture) is here:

https://hub.packtpub.com/baidu-open-sources-ernie-2-0-a-continual-pre-training-nlp-model-that-outperforms-bert-and-xlnet-on-16-nlp-tasks/

7.5 What is Translatotron?

Translatotron is an end-to-end speech-to-speech translation model (from Google) whose output retains the original speaker's voice; moreover it's trained with less data.

Speech-to-speech translation systems have been developed over the past several decades with the goal of helping people who speak different languages to communicate with each other. Such systems have three parts:

- automatic speech recognition to transcribe the source speech as text
- machine translation to translate the transcribed text into the target language
- text-to-speech synthesis (TTS) to generate speech in the target language from the translated text

The preceding approach has been successful in commercial products (including Google Translate). However, Translatatron does not require separate stages, resulting in the following advantages:

- faster inference speed
- avoids compounding errors between recognition and translation
- easier to retain the voice of the original speaker after translation
- better handling of untranslated words (names and proper nouns)

This concludes the portion of this chapter that pertains to NLP. Another area of great interest in the AI community is RL, which is introduced later in this chapter.

7.6 Deep Learning and NLP

In Chapter 4, you learned about CNNs and how they are well-suited for image classification tasks. You might be surprised to discover that CNNs also work with NLP tasks. However, you must first "map" each word in a dictionary (which can be a subset of the words in English or some other language) to numeric values and then construct a vector of numeric values from the words in a sentence. A document can be transformed into a set of numeric vectors (involving various techniques that are not discussed here) in order to create a dataset that's suitable for input to a `CNN`.

Another option involves the use of `RNN`s and `LSTM`s instead of `CNN`s for `NLP`-related tasks. In fact, a "bidirectional LSTM" is being used successfully in ELMo (Embeddings from Language Models), whereas `BERT` is based on a bi-directional transformer architecture. The Google AI team developed `BERT` (open sourced in 2018) and it's considered a breakthrough in its ability to solve `NLP` problems. The source code is here: *https://github.com/google-research/bert*

7.7 NLU versus NLG

NLU is an acronym for natural language understanding. NLU pertains to machine reading comprehension, and it's considered a difficult problem. At the same time, NLU is relevant to machine translation, question answering, and text categorization (among others). NLU attempts to discern the meaning of fragmented sentences and run-on sentences, after which some type of action can be performed (e.g., respond to voice queries).

NLG is an acronym for natural language generation, which involves generating documents. The Markov chain (discussed later in this chapter) was one of the first algorithms for NLG. Another technique involves RNNs (discussed in Chapter 5) that can retain some history of previous words, and the probability of the next word in a sequence is calculated. Recall that RNNs suffer from limited memory, which limits the length of the sentences that can be generated. A third technique involves LSTMs, which can maintain state for a long period of time, and also avoid the "exploding gradient" problem.

Recently (circa 2017) Google introduced the transformer architecture, which involves a stack of encoders for processing inputs and a set of decoders to produce generated sentences. A transformer-based architecture is more efficient than an LSTM because a transformer requires a small and fixed number of steps in order to apply the so-called "self-attention mechanism" in order to simulate the relationship among all the words in a sentence.

In fact, the transformer differs from previous models in one important way: it uses the representation of all words in context without compressing all the information into a single fixed-length representation. This technique enables a transformer to handle longer sentences without high computational costs.

The transformer architecture is the foundation for the GPT-2 language model (from OpenAI). The model learns to predict the next word in a sentence by focusing on words that were previously seen in the model and related to predicting the next word. In 2018, Google released the BERT architecture for NLP, which is based on transformers with a two-way encoder representation.

7.8 What is Reinforcement Learning (RL)?

RL is a subset of machine learning that attempts to find the maximum reward for a so-called "agent" that interacts with an "environment." RL is suitable for solving tasks that involve deferred rewards, especially when those rewards are greater than intermediate rewards.

In fact, RL can handle tasks that involve a combination of negative, zero, and positive rewards. For example, if you decide to leave your job in order to attend school on a full-time basis, you are spending money (a negative reward) with the believe that your investment of time and money will