Chapter 10

Natural Language Processing with TensorFlow: Language Modeling

This chapter introduces **language modeling**, a foundational task in natural language processing that focuses on predicting the next token in a sequence given all previous tokens. Building on the sentiment analysis task from the previous chapter, the discussion shifts from classification to **generative modeling**, where the objective is to learn the statistical structure of language itself. Language modeling underpins many state-of-the-art NLP systems, including Transformer-based architectures such as BERT, and provides essential linguistic knowledge related to syntax, semantics, and word dependencies.

Language modeling is formally defined as computing the probability of the next word $w_n$given a sequence of previous words $w_1, w_2, \ldots, w_{n-1}$. In practice, directly conditioning on the full history is computationally infeasible for long texts, so the **Markov assumption** is applied, allowing the model to consider only a fixed-length context. This approximation enables efficient training while still capturing meaningful sequential dependencies.

10.1 Processing the Data

The chapter begins by focusing on **data preparation**, which is critical for training an effective language model. The dataset used is the **bAbI children's stories corpus**, which contains short narrative texts suitable for sequence modeling. The goal is to generate training examples where the input is a sequence of tokens and the target is the same sequence shifted by one position, allowing the model to learn next-token prediction.

10.1.1 What Is Language Modeling?

Language modeling is described as a probabilistic task that maximizes the likelihood of observing the next word given the previous sequence. During training, the model parameters are optimized to maximize this conditional probability. To make the task tractable, the Markov property is introduced, which limits the conditioning context to a fixed window of recent tokens.

10.1.2 Downloading and Exploring the Data

The dataset is loaded from disk and parsed into individual stories, with each story represented as a single string.

Listing 10.2 Reading the stories in Python

```
import os
import requests
import tarfile

import shutil
```



```
# Retrieve the data
if not os.path.exists(os.path.join('data', 'lm','CBTest.tgz')):
    url = "http://www.thespermwhale.com/jaseweston/babi/CBTest.tgz"
    # Get the file from web
    r = requests.get(url)

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

    # Write to a file
    with open(os.path.join('data', 'lm', 'CBTest.tgz'), 'wb') as f:
        f.write(r.content)

else:
    print("The tar file already exists.")

if not os.path.exists(os.path.join('data', 'lm', 'CBTest')):
    # Write to a file
    tarf = tarfile.open(os.path.join("data","lm","CBTest.tgz"))
    tarf.extractall(os.path.join("data","lm"))
else:
    print("The extracted data already exists")
```

If the tgz file containing data has not been downloaded, download the data.

Write the downloaded data to the disk.

If the tgz file is available but has not been extracted, extract it to the given directory.

Basic exploratory analysis is performed to inspect the number of stories in the training, validation, and test sets, as well as to examine sample text. A vocabulary analysis reveals that even after filtering rare words, the vocabulary size exceeds 14,000 unique tokens. This large vocabulary poses both **memory and computational challenges**, especially because language models require a softmax layer whose dimensionality scales with vocabulary size.

Vocabulary Size and Softmax Limitations

The chapter explains that the computational bottleneck arises from the softmax layer, which requires matrix multiplication and normalization across the entire vocabulary. As vocabulary size grows, both training time and memory consumption increase substantially. To mitigate this issue, alternative techniques such as **noise contrastive estimation (NCE)** and **hierarchical softmax** are introduced conceptually.

A hierarchical softmax structure is illustrated to show how vocabulary words can be organized as leaves of a binary tree, reducing inference complexity from $O(n)$to $O(\log\ n)$.

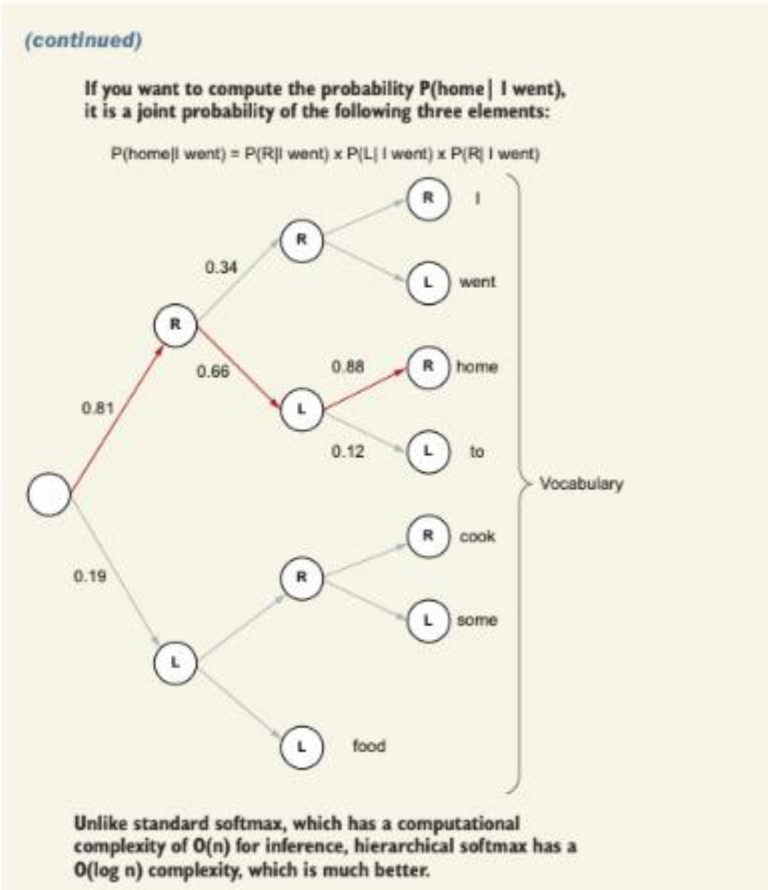

Figure Hierarchical softmax representation of the final layer

---

10.1.3 Too Large a Vocabulary? N-Grams to the Rescue

To further reduce vocabulary size, the chapter adopts an **n-gram representation**, specifically **character-level bigrams**. Instead of modeling full words, text is decomposed into fixed-length character sequences. This dramatically reduces vocabulary size while maintaining enough structure to generate readable text.

The process of generating n-grams is demonstrated through examples, and a helper function is introduced to extract non-overlapping n-grams from text.

After converting the dataset to bigrams and reanalyzing token frequencies, the vocabulary size drops from approximately 15,000 words to fewer than 1,000 bigrams, making the modeling task significantly more manageable.

10.1.4 Tokenizing Text

The bigram sequences are then converted into numerical form using TensorFlow's **Tokenizer**. The tokenizer is fitted only on the training data to avoid data leakage and is configured with an out-of-vocabulary token to handle rare bigrams. Training, validation, and test datasets are all transformed into sequences of integer IDs, enabling them to be processed by neural networks.

Sample outputs are shown to illustrate how raw text is converted into bigrams and then mapped to numerical IDs.

10.1.5 Defining a tf.data Pipeline

The chapter then presents a sophisticated **tf.data pipeline** that transforms variable-length bigram sequences into fixed-length training samples suitable for batching. The pipeline uses **ragged tensors**, windowing, flattening, shuffling, batching, and prefetching to efficiently generate input–target pairs.

The pipeline creates sliding windows of length $n\_seq + 1$, where the first $n_seq$tokens serve as inputs and the remaining tokens act as targets shifted by one position.

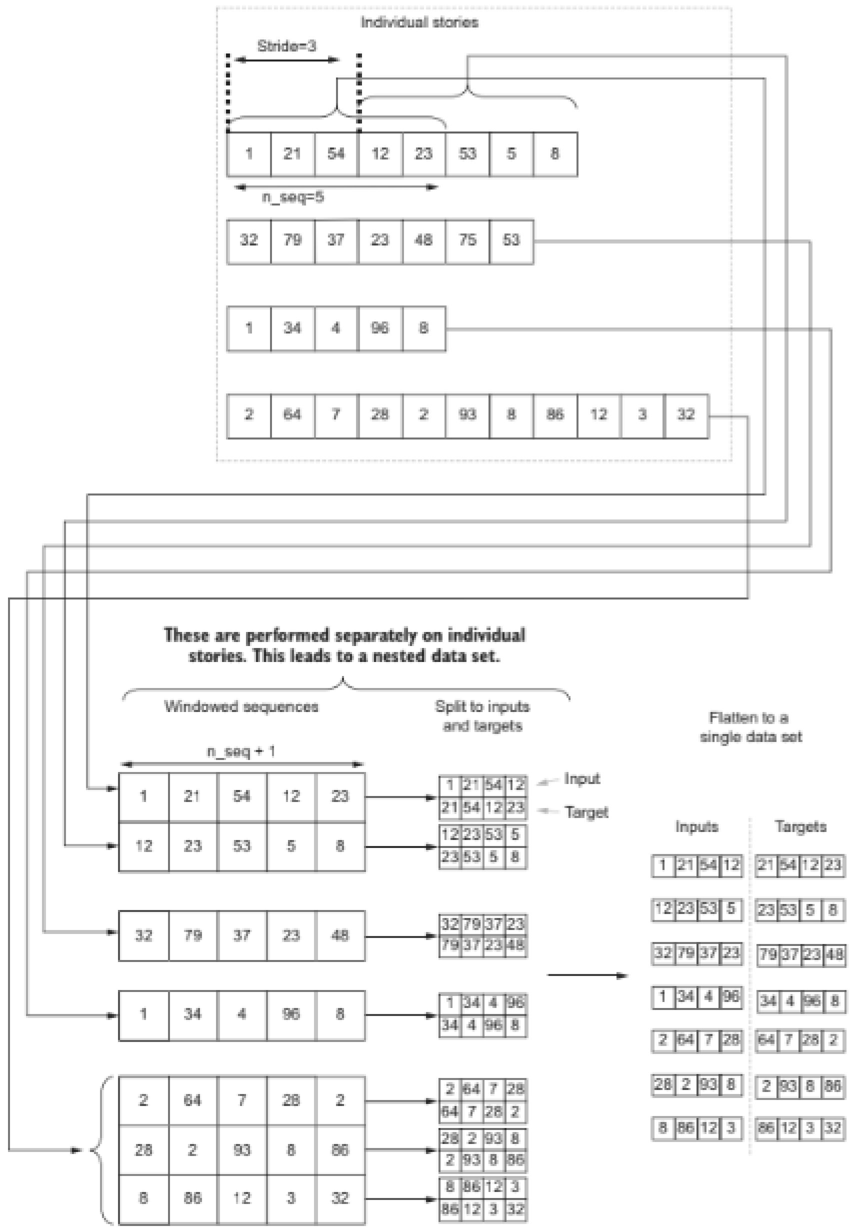

Figure 10.1 High-level steps of the TensorFlow data pipeline

A complex combination of window() and flat_map() operations is used to remove nested dataset structures and produce a flat stream of training samples.

Listing 10.3 The tf.data pipeline from free text sequences

```
def get_tf_pipeline(data_seq, n_seq, batch_size=64, shift=1, shuffle=True):
    """ Define a tf.data pipeline that takes a set of sequences of text and
    convert them to fixed length sequences for the model """
```



```
text_ds = tf.data.Dataset.from_tensor_slices(tf.ragged.constant(data_seq))

if shuffle:
    text_ds = text_ds.shuffle(buffer_size=len(data_seq)//2)

text_ds = text_ds.flat_map(
    lambda x: tf.data.Dataset.from_tensor_slices(
        x
    ).window(
        n_seq+1, shift=shift
    ).flat_map(
        lambda window: window.batch(n_seq+1, drop_remainder=True)
    )
)

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

text_ds = text_ds.batch(batch_size)

text_ds = tf.data.Dataset.zip(
    text_ds.map(lambda x: (x[:,:-1], x[:, 1:]))
).prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

return text_ds
```

Define a tf.dataset from a ragged tensor created from data_seq.

If shuffle is set, shuffle the data (shuffle story order).

Shuffle the data (shuffle the order of the windows generated).

Batch the data.

Split each sequence into an input and a target and enable pre-fetching.

By the end of this section, the data pipeline produces batches of input–target tensor pairs that can be fed directly into a neural language model.

10.2 GRUs in Wonderland: Generating Text with Deep Learning

With the data pipeline in place, the chapter introduces **gated recurrent units (GRUs)** as the core modeling architecture. GRUs are presented as a simplified alternative to LSTMs that maintain comparable performance while being faster to train.

The behavior of LSTMs is briefly reviewed to highlight their use of cell state, hidden state, and gating mechanisms.

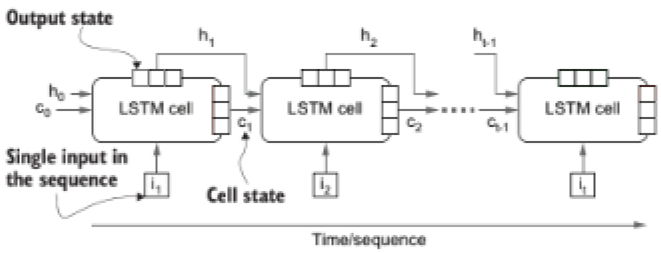

Figure 10.2 Overview of LSTM sequence processing

GRUs simplify this structure by using a single hidden state and two gates: the **update gate** and the **reset gate**. These gates control how much past information is retained and how much new input influences the current state.

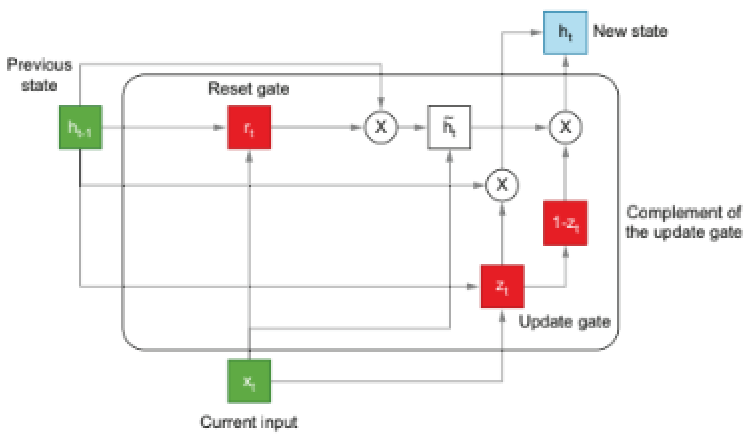

Figure 10.3 Computations inside a GRU cell

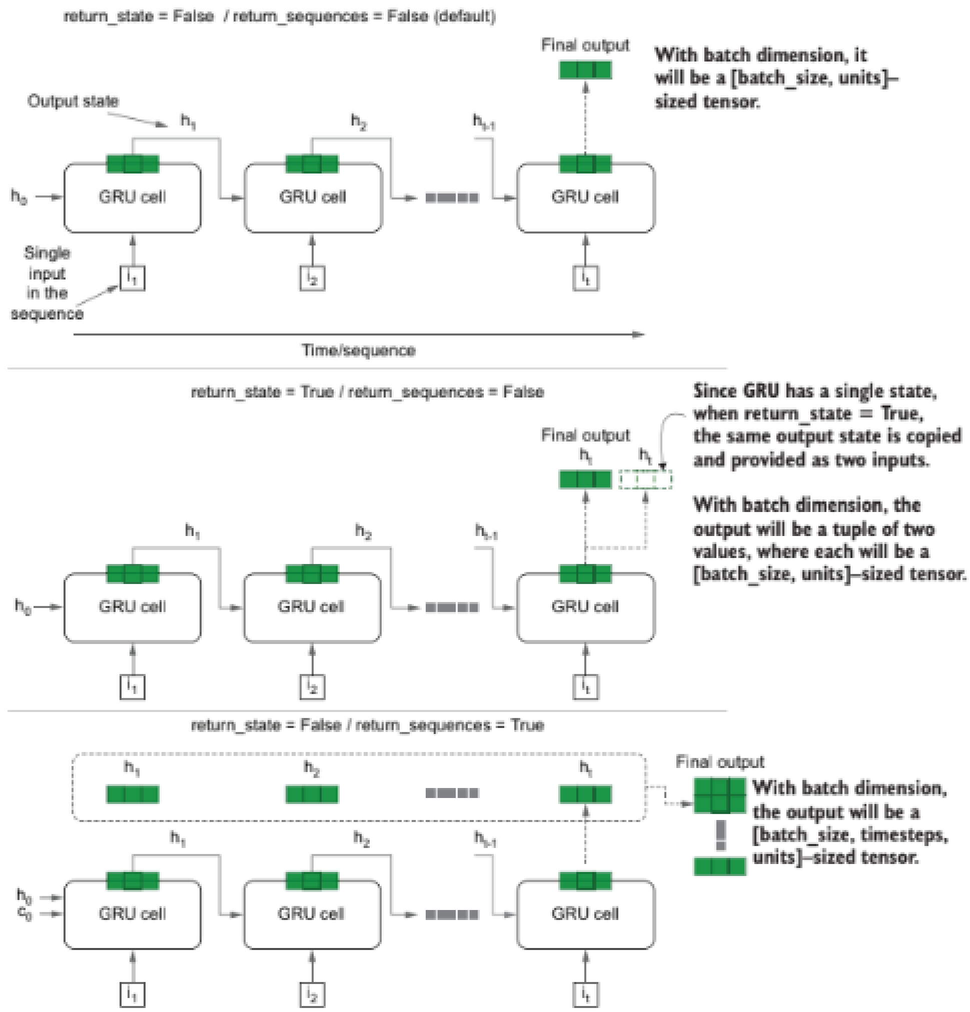

Figure 10.4 Changes in results depending on the return_state and return_sequences arguments of the GRU cell

---

Defining the Language Model

The final language model consists of an **embedding layer**, a **GRU layer with 1,024 units**, and two fully connected layers, the final one using a softmax activation over the vocabulary.

Listing 10.4 Implementing the language model

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Embedding(
        input_dim=n_vocab+1, output_dim=512,input_shape=(None,)
    ),
    tf.keras.layers.GRU(1024, return_state=False, return_sequences=True),

    tf.keras.layers.Dense(512, activation='relu'),

    tf.keras.layers.Dense(n_vocab, name='final_out'),
    tf.keras.layers.Activation(activation='softmax')
])
```

Define an embedding layer to learn word vectors of the bigrams.

Define an LSTM layer.

Define a Dense layer.

Define a final Dense layer and softmax activation.

The GRU layer is configured to return sequences so that predictions are generated at every time step. Dense layers operate on three-dimensional tensors, applying the same transformation across all time steps.

10.3 Measuring the Quality of Generated Text

Traditional accuracy metrics are shown to be inadequate for language modeling, as multiple predictions may be equally plausible. Instead, the chapter introduces **perplexity**, a metric derived from information theory that measures how surprised the model is by the next token.

Perplexity is defined as the exponential of the entropy of the model's predicted probability distribution. Lower perplexity values indicate better language modeling performance.

A custom TensorFlow metric is implemented by exponentiating the categorical cross-entropy loss.

Listing 10.5 Implementation of the perplexity metric

```
import tensorflow.keras.backend as K

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):

      loss_ = self.cross_entropy(real, pred)

      mean_loss = K.mean(loss_, axis=-1)
      perplexity = K.exp(mean_loss)

      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)
```

Define a function to compute perplexity given real and predicted targets.

Compute the categorical cross-entropy loss.

Compute the mean of the loss.

Compute the exponential of the mean loss (perplexity).

10.4 Training and Evaluating the Language Model

The model is trained using the previously defined data pipeline, with callbacks for learning rate scheduling, early stopping, and logging. Only a subset of the training stories is used to reduce training time.

After training, the model achieves a validation perplexity of approximately 9.5, indicating reasonable predictive confidence. Evaluation on the test set confirms similar performance, demonstrating good generalization. The trained model is then saved for later use.

10.5 Generating New Text: Greedy Decoding

The chapter then explains how to generate text using the trained model. Since training and inference differ fundamentally, a separate **inference model** is constructed using the Keras functional API. This model predicts one token at a time while maintaining and updating the GRU state.

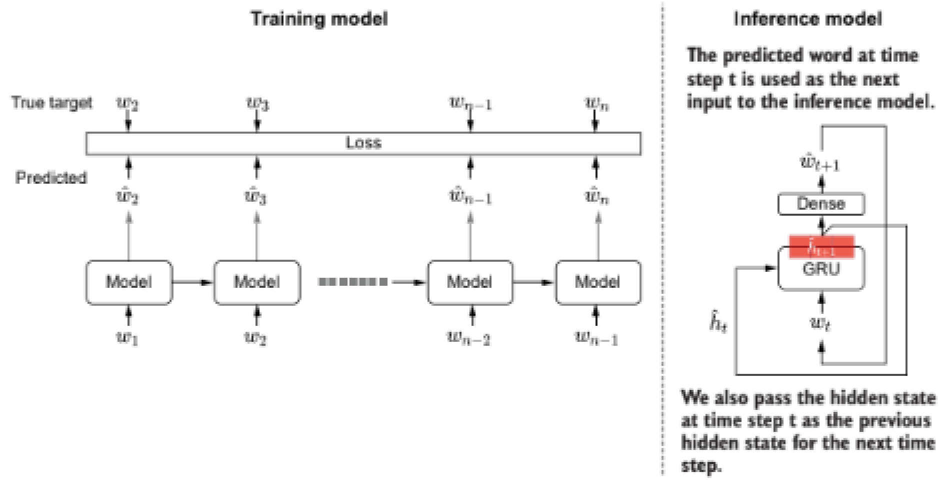

Figure 10.5 Comparison between the language model at training time and the inference/decoding phrase. In the inference phase, we predict one time step at a time. In each time step, we get the predicted word as the input and the new hidden state as the previous hidden state for the next time step.

Weights from the trained model are copied into the inference model, and text generation proceeds recursively, starting from an initial seed sequence.

Listing 10.6 Inference model definition

Define an input that can take an arbitrarily long sequence of word IDs.

Define another input that will feed in the previous state.

Define an embedding layer.

Get the embedding vectors from the input word ID.

```
inp = tf.keras.layers.Input(shape=(None,))
inp_state = tf.keras.layers.Input(shape=(1024,))

emb_layer = tf.keras.layers.Embedding(
    input_dim=n_vocab+1, output_dim=512, input_shape=(None,)
)
emb_out = emb_layer(inp)
```



```
gru_layer = tf.keras.layers.GRU(
    1024, return_state=True, return_sequences=True
)
gru_out, gru_state = gru_layer(emb_out, initial_state=inp_state)

dense_layer = tf.keras.layers.Dense(512, activation='relu')
dense_out = dense_layer(gru_out)

final_layer = tf.keras.layers.Dense(n_vocab, name='final_out')
final_out = final_layer(dense_out)
softmax_out = tf.keras.layers.Activation(activation='softmax')(final_out)

infer_model = tf.keras.models.Model(
    inputs=[inp, inp_state], outputs=[softmax_out, gru_state]
)
```

Get the GRU output and the state from the model.

Compute the first fully connected layer output.

Define a final layer that is the same size as the vocabulary and get the final output of the model.

Define the final model that takes an input and a state vector as the inputs and produces the next word prediction and the new state vector as the outputs.

Define a GRU layer that returns both the output and the state. However, note that they will be the same for a GRU.

Listing 10.7 Recursive greedy decoding

```
for _ in range(500):

    out, state = infer_model.predict([x, state])

    out_argsort = np.argsort(out[0], axis=-1).ravel()
    wid = int(out_argsort[-1])
    word = tokenizer.index_word[wid]

    if word.endswith(' '):
        if np.random.normal()>0.5:
            width = 3
            i = np.random.choice(
                list(range(-width,0)),
                p=out_argsort[-width:]/out_argsort[-width:].sum()
            )
            wid = int(out_argsort[i])
            word = tokenizer.index_word[wid]
```

Get the next output and state.

Get the word ID and the word from out.

If the word ends with space, we introduce a bit of randomness to break repeating text.

Essentially pick one of the top three outputs for that timestep depending on their likelihood.


```
    text.append(word)

    x = np.array([[wid]])
```

Append the prediction cumulatively to text.

Recursively make the current prediction the next input.

While greedy decoding produces readable text, it often suffers from repetitive patterns and grammatical errors.

10.6 Beam Search: Enhancing Text Quality

To improve text generation quality, the chapter introduces **beam search**, which explores multiple candidate sequences simultaneously instead of selecting the most probable token at each step.

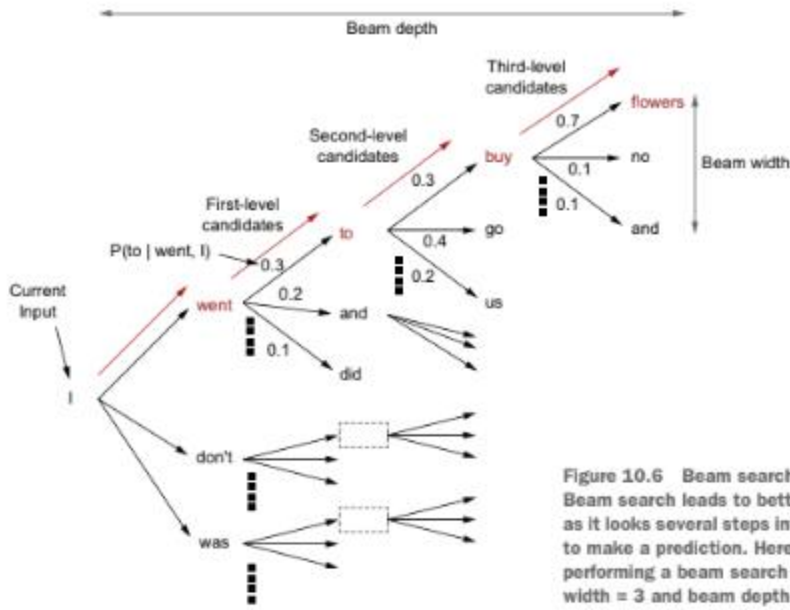

Figure 10.6 Beam search in action. Beam search leads to better solutions as it looks several steps into the future to make a prediction. Here, we are performing a beam search with beam width = 3 and beam depth = 5.

Figure 10.6 Beam search process illustration

Beam search evaluates candidate sequences based on joint probability, allowing the model to look several steps ahead before committing to a prediction.

// **Listing 10.8** Recursive beam search function

Define an outer wrapper for the computational function of beam search.

Define an inner function that is called recursively to find the beam paths.

```
def beam_search(
    model, input_, state, beam_depth=5, beam_width=3, ignore_blank=True
):
    """ Defines an outer wrapper for the computational function of beam
➥ search """

    def recursive_fn(input_, state, 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), state, np.exp(log_prob)))
            return sequence, log_prob, state
        else:
            """ Recursive case: Keep computing the output using the
➥ previous outputs"""
            output, new_state = beam_one_step(model, input_, state)

            # Get the top beam_widht 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)
```

Define the base case for terminating the recursion.

Append the result we got at the termination so we can use it later.

During recursion, get the output word and the state by calling the model.

Get the top k candidates for that step.

For each candidate, compute the joint probability. We will do this in log space to have numerical stability.

10.6 Beam search: Enhancing the predictive power of sequential models **381**

```
                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([[wid]]), new_state, sequence, new_log_prob, i+1
                )
                sequence.pop()

    results = []
    sequence = []
    log_prob = 0.0
    recursive_fn(input_, state, sequence, log_prob, 0)

    results = sorted(results, key=lambda x: x[2], reverse=True)

    return results
```

Penalize joint probability whenever the same symbol repeats.

Append the current candidate to the sequence that maintains the current search path at the time.

Call the function recursively to find the next candidates.

Make a call to the recursive function to trigger the recursion.

Sort the results by log probability.

Listing 10.9 Beam search decoding implementation

```
text = get_ngrams(
    "CHAPTER I. Down the Rabbit-Hole Alice was beginning to get very tired
➥ of sitting by her sister on the bank ,".lower(),
    ngrams
)                                  Define a sequence of ngrams
                                   from an initial sequence of text.
                                                            Convert the
                                                            bigrams to
seq = tokenizer.texts_to_sequences([text])                  word IDs.

                                                                       Build up
state = np.zeros(shape=(1,1024))                                       model state
for c in seq[0]:                                                       using the
    out, state = infer_model.predict([np.array([[c]]), state           given string.

wid = int(np.argmax(out[0],axis=-1).ravel())   Get the predicted
word = tokenizer.index_word[wid]               word after processing
text.append(word)                              the sequence.

x = np.array([[wid]])              Predict for 100
                                   time steps.
for i in range(100):                                             Get the
                                                                 results from
    result = beam_search(infer_model, x, state, 7, 2)            beam search.

    n_probs = np.array([p for _,_,p in result[:10              Get one of the top 10 results
    p_j = np.random.choice(list(range(                         based on their likelihood.
       n_probs.size)), p=n_probs/n_probs.sum())
```



```
    best_beam_ids, state, _ = result[p_j]      Replace x and state with
    x = np.array([[best_beam_ids[-1]]])        the new values computed.

    text.extend([tokenizer.index_word[w] for w in best_beam_ids])

print('\n')
print('='*60)
print("Final text: ")
print(''.join(text))
```

Compared to greedy decoding, beam search produces text with better grammar, coherence, and reduced repetition. The chapter also briefly discusses **diverse beam search**, which encourages variety among generated sequences.

Chapter Summary

This chapter presented a complete **end-to-end language modeling workflow** using TensorFlow. It covered data preprocessing with n-grams, scalable tf.data pipelines, GRU-based sequence modeling, perplexity-based evaluation, and advanced text generation techniques such as greedy decoding and beam search. Language modeling is positioned as a cornerstone task in NLP, enabling models to learn deep linguistic structure and supporting a wide range of downstream applications.