Chapter 10

Natural Language Processing with TensorFlow: Language Modeling

Language modeling is one of the foundational tasks in natural language processing. It aims to predict the next word in a sequence, a principle that underlies state-of-the-art models such as BERT. Although the task itself may appear limited, it enables models to acquire knowledge of semantics, grammar, and dependency structures that are crucial for solving downstream problems like information retrieval, question answering, and machine translation.

The chapter begins by outlining the preprocessing steps necessary for building a language model. To manage vocabulary size, the text is represented using n-grams rather than entire words or characters. For example, bigram representation reduces complexity by splitting text into overlapping two-character units. Using a TensorFlow data pipeline, inputs and targets are created by shifting sequences, preparing the data for modeling.

In a nutshell, language modeling is represented by this formula:

$$P(wn|w1, w2, ..., wn-1)$$
$$argmaxW\ P(wn|w1, w2, ..., wn-1)$$
$$P(wn|w1, w2, ..., wn-1) \approx P(wn|wk, wk+1, ..., wn-1) \text{ for some } k$$

Next, the gated recurrent unit (GRU) is introduced as the core model architecture. Compared with the LSTM, the GRU is simpler, using only one hidden state and two gates (update and reset) yet performing comparably well. Figure 1 presents the internal computations of a GRU cell, while Figure 2 illustrates how the return_state and return_sequences parameters affect outputs. The final model includes an embedding layer, a GRU with 1,024 units, a dense hidden layer with 512 ReLU units, and a softmax output layer over the vocabulary.

$$r_t = \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r)$$

$$z_t = \sigma(W_{zh}h_{t-1} + W_{zx}x_t + b_z)$$

$$\tilde{h}_t = \tanh(W_h(rh_{t-1}) + W_x x_t + b)$$

$$h_t = (z_{th}h_{t-1} + (1 - z_t)\tilde{h}_t$$

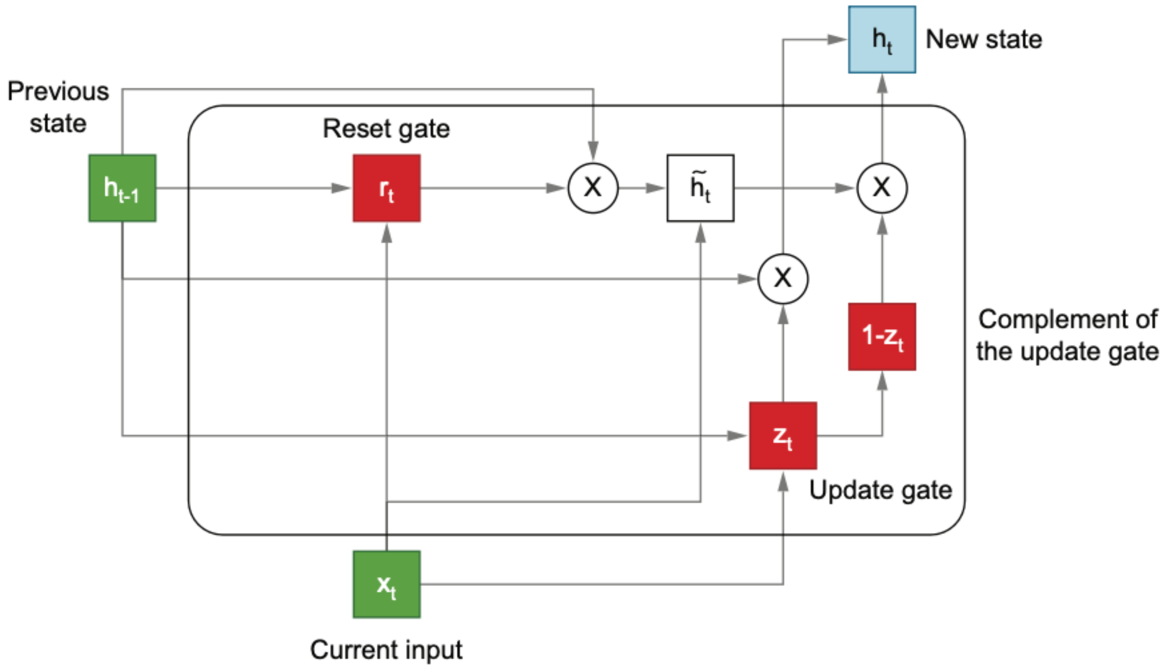

Figure 1 Overview of the computations transpiring in a GRU cell

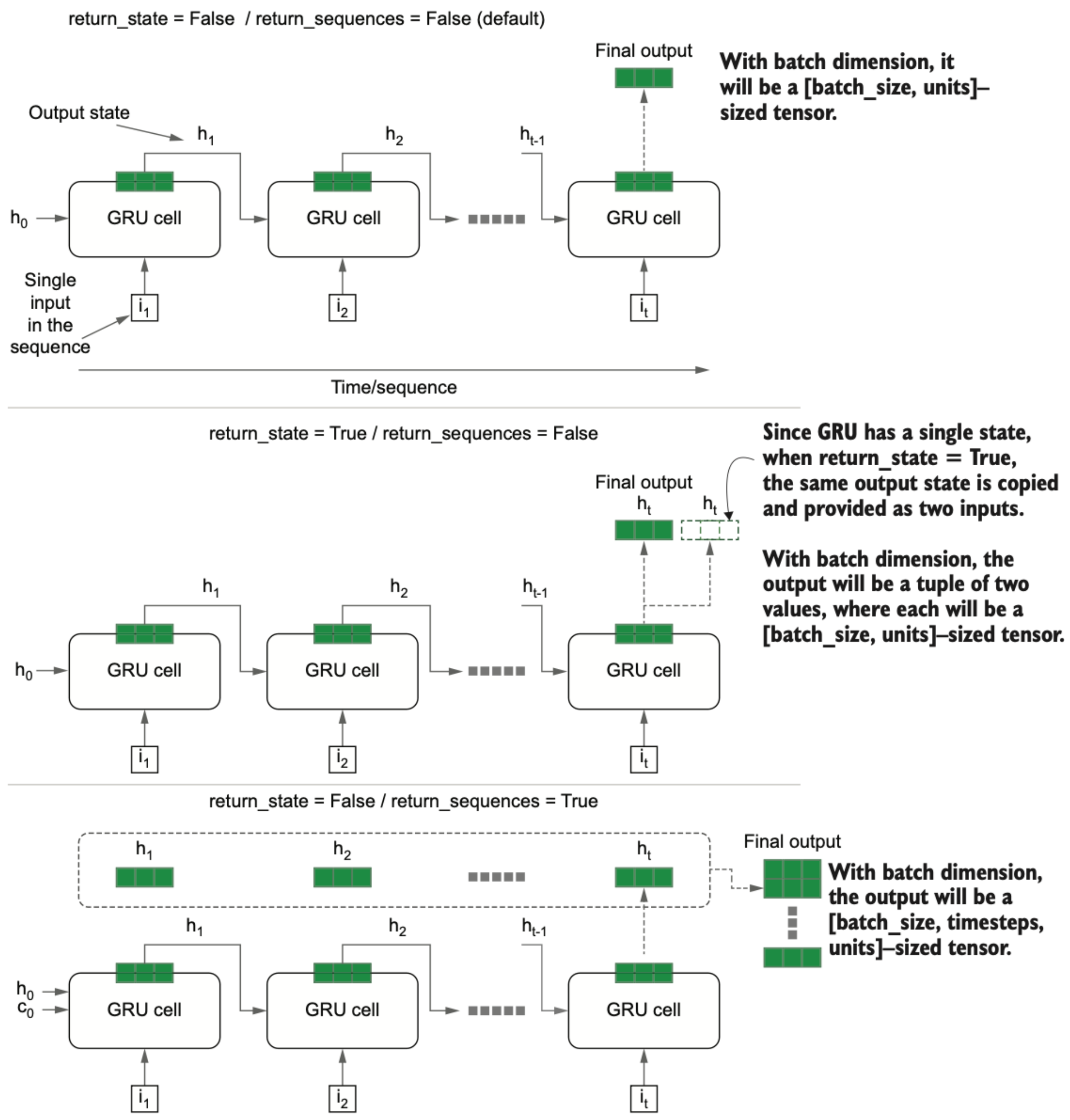
return_state = False / return_sequences = False (default)
Final output
With batch dimension, it will be a [batch_size, units]–sized tensor.
Output state
h1
h2
ht-1
h0
GRU cell
GRU cell
GRU cell
Single input in the sequence
i1
i2
it
Time/sequence
return_state = True / return_sequences = False
Since GRU has a single state, when return_state = True, the same output state is copied and provided as two inputs.
Final output
ht
ht
With batch dimension, the output will be a tuple of two values, where each will be a [batch_size, units]–sized tensor.
return_state = False / return_sequences = True
Final output
With batch dimension, the output will be a [batch_size, timesteps, units]–sized tensor.
h0
c0

Figure 2 Changes in results depending on the return_state and return_sequence arguments of the GRU cell

To evaluate quality, the metric of perplexity is introduced. Rooted in information theory, perplexity measures how “surprised” a model is when predicting the target word given previous words. A low perplexity implies better predictive certainty. This is derived from entropy, with a custom implementation in TensorFlow.

$$CategoricalCrossEntropy = -\sum_{c=1}^{c} y_{i,c} \log(\gamma_{i,c})$$

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)
```

Training is carried out with sequences of 100 bigrams, sliding by 25, using a batch size of 128. Callbacks such as early stopping and learning rate reduction help stabilize training. Validation is monitored using perplexity, ensuring the model generalizes well.

Once trained, the model can generate new text. During inference, the process differs from training: predictions are made step by step, feeding the previous output back as input. Figure 3 contrasts the training and inference phases. An inference model is constructed with functional API inputs for both word IDs and hidden states, allowing recursive prediction. While this approach produces coherent text, it often suffers from grammatical inconsistencies and repetition.

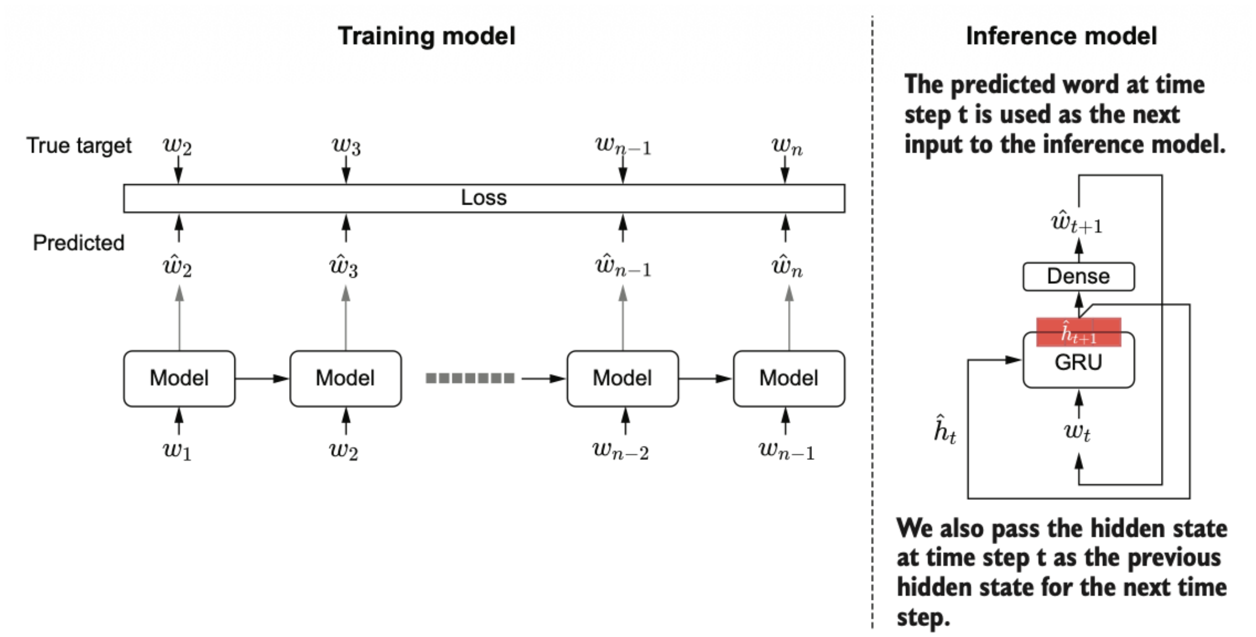

Figure 3 Comparison between the language model at training time and the inference/decoding phrase

To overcome these limitations, beam search is introduced. Instead of choosing only the top prediction, beam search expands multiple candidate sequences at each step, balancing breadth (beam width) and depth (beam depth). Figure 10.6 illustrates this branching search strategy. Beam search generates text with better grammar and fluency than greedy decoding. For example, generated passages resemble natural storytelling more closely, though occasional repetition remains.

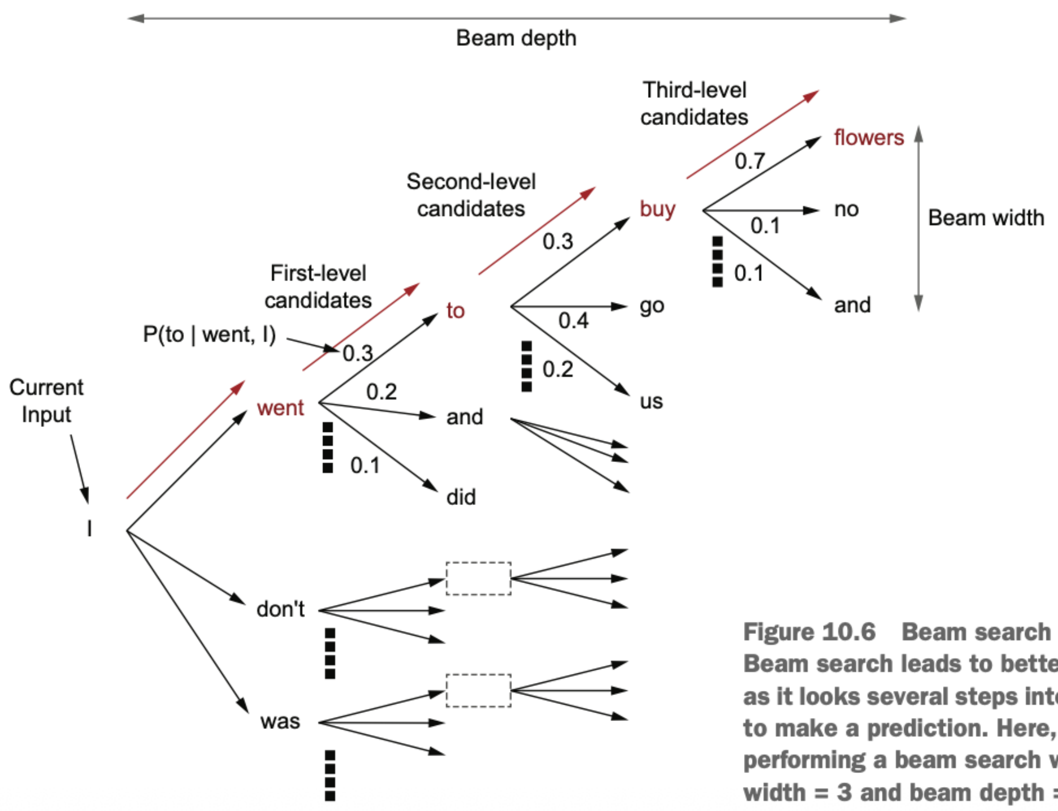

Figure 10.6 Beam search in action. Beam search leads to better solutions as it looks several steps into the future to make a prediction. Here, we are performing a beam search with beam width = 3 and beam depth = 5.

In conclusion, language modeling predicts the next word in a sequence, GRUs provide an efficient architecture for the task, perplexity measures performance, and beam search significantly improves generative quality over greedy decoding.