Next Word Prediction using LSTM with TensorFlow

Table of Contents

Libraries

1. **`numpy`:** Provides support for large multi-dimensional arrays and matrices, essential for numerical operations.
2. **`tensorflow`:** The core library for building machine learning models. Here, TensorFlow's `Keras` API is used to simplify the process of building deep learning models.
3. **`Tokenizer`:** This converts text data into a sequence of integers where each integer represents a unique word.
4. **`pad_sequences`:** Padding all the sequences with zeros if necessary ensures that they are of the same length.
5. **`Sequential`:** The model architecture that allows stacking layers sequentially (like embedding, LSTM, and Dense layers).
6. **`Embedding Layer`:** Converts words into dense vectors of fixed size (word embeddings).
7. **`LSTM Layer`:** The core layer for processing sequences and capturing context in the data.
8. **`Dense Layer`:** A fully connected layer at the end is used for classification.

Reading the Dataset

1. Open the dataset using the path in read mode and store it in a variable.
2. `encoding='utf-8'`: This ensures that the file is read using the **UTF-8 character encoding**. UTF-8 is a widely used encoding that can handle a large range of characters, including special symbols or non-ASCII characters, without causing errors.

Preprocessing the Text

1. Tokenizer Initialization: `Tokenizer()` is a tool provided by `Keras` that transforms a corpus of text into a sequence of integers. Each unique word in the text will be assigned a unique integer. This is crucial for preparing text data to be processed by a machine learning model, especially neural networks like LSTM, which expect numerical input rather than raw text.

2. `Fitting` the Tokenizer: This step "fits" the tokenizer on our text data, meaning it learns the vocabulary (unique words) and assigns each word an integer index. This word-index mapping will be used later to convert sequences of words into sequences of integers.

3. Assigning `total_words`: After fitting the tokenizer on the text, `tokenizer.word_index` contains a dictionary where the keys are words, and the values are their corresponding unique integer indices.
    a. The `+1` is added because Keras uses zero-based indexing internally, so to accommodate for padding and other sequences, we add 1 to account for the maximum index.
    b. For instance, if our vocabulary contains 5000 unique words, `total_words` will be 5001. This will be used when building the embedding layer and for one-hot encoding during prediction.

    `total_words` is used because this value will be used to define the input size for the embedding layer in the neural network, as well as for output layers (predicting the next word).

Building Sequences and Tokenize Them

1. Declare `input_sequences`: The goal is to create sequences that the LSTM model can use for training. Each sequence will represent part of a sentence or phrase, including an increasing number of words. These sequences are also called **n-grams**, where "n" represents the number of words in each sequence.

2. Split the Text by Newlines: This splits the `text` into individual lines, using the newline character (`'\n'`) as the delimiter. Each `line` will represent a new sentence or paragraph from the text. This allows processing of the text line by line (i.e., sentence by sentence). Each line is considered separately to avoid combining words from different sentences, which could confuse the model.

3. Tokenizing Each Line: This line converts each sentence (or line of text) into a list of tokens (integers) using the tokenizer we created earlier.

4. Create N-gram Sequences: This loop generates **n-gram sequences** from each tokenized sentence and appends them to `input_sequences.`
    a. Suppose the tokenized sentence is `[1, 2, 3, 4, 5]` (corresponding to `"Sherlock Holmes is a detective"`).
        i. `[1, 2]` – corresponding to "Sherlock Holmes"

ii. `[1, 2, 3]` – corresponding to "Sherlock Holmes is"
iii. `[1, 2, 3, 4]` – corresponding to "Sherlock Holmes is a"
iv. `[1, 2, 3, 4, 5]` – corresponding to "Sherlock Holmes is a detective"

b. These n-grams will be used to train the LSTM model to predict the next word in a sequence.
c. For example, by providing the sequence `[1, 2, 3]` ("Sherlock Holmes is"), the model can learn to predict that the next word is `4` ("a").

The idea behind using n-gram sequences is to give the LSTM model partial sentences and train it to predict the next word in the sequence. The model sees several examples of these sequences and learns to generalize patterns in the language.
By training on n-gram sequences of increasing lengths (starting with 2-word sequences and building up to longer ones), the model can learn the relationships between words in various contexts. This process teaches the model to understand word order and how words in a sentence relate to one another, which is crucial for next-word prediction.

Padding the Sequences

1. Finding the Maximum Sentence Length: This line calculates the **maximum length** of any sequence (n-gram) in `input_sequences.`

   The LSTM model expects input sequences to have the same length. Since the sequences we generated (n-grams) have different lengths, we need to standardize them by either **padding** or **truncating**. To ensure consistency, we find the length of the longest sequence and then **pad** all other sequences to match this length (which we'll do in the next step).

2. Padding the Input Sequences: Then the `input_sequences` are **padded** to make them the same length (the length of the longest sequence found in the previous step). It then converts the padded sequences into a NumPy array.

   LSTM models require input sequences to have the same length because they process data in batches, and all sequences in a batch must have the same shape. If some sequences are shorter, we need to add zeros (padding) to make them as long as the longest sequence. The padding ensures that all input sequences have consistent dimensions, making them suitable for input into the model.

3. After padding, the padded sequences are converted into a NumPy array using `np.array()`. This is because deep learning models usually work with NumPy arrays or tensors for efficient numerical computation.

4. This step prepares the `input_sequences` to be fed into the LSTM model by ensuring they all have the same length and are represented as a NumPy array.

Splitting Features and Labels

1. Creating the Features (X) and Labels (y):
   a. **X** represents the input features (n-gram sequences except for the last word).

i. This selects all the columns except the last one for each row (sequence). Essentially, it grabs all the tokens in each sequence except the last token.
ii. The notation `[:, :-1]` means: for every row (`:`), take all columns **except the last one** (`:-1`).
iii. **Purpose:** This is done because the model needs to predict the next word, so the input will be the sequence of words up to the second-to-last word.

b. `y` represents the target labels (the last word in each sequence).
i. This selects the last token (word) from each sequence.
ii. The notation `[:, -1]` means: for every row (`:`), take only the last column (`-1`).
iii. **Purpose:** This is the word that the model is trying to predict (the next word after the sequence of tokens in `X`).

2. One-Hot Encoding the Labels: The target labels `y` (which are integers representing the tokenized words) are converted into **one-hot encoded** vectors using TensorFlow's `to_categorical` function.
a. tf.keras.utils.to_categorical(y, num_classes=total_words):
i. `y` is a 1D array where each element represents a word's token (its integer representation in the vocabulary).
ii. `to_categorical` converts these integers into a one-hot encoded format, which is a vector where the index corresponding to the word's token is set to 1, and all other indices are 0.
iii. The parameter `num_classes=total_words` specifies the total number of unique words (or tokens) in the vocabulary. This determines the length of the one-hot encoded vectors.

In classification problems, deep learning models typically expect the labels (target outputs) to be one-hot encoded. Instead of predicting a single integer (the tokenized word), the model will predict a probability distribution over all possible words in the vocabulary. The one-hot encoding ensures that the model learns to output a vector of probabilities.

One-hot encoding allows the model to treat the prediction as a **classification task** where it classifies the next word as one of the possible `total_words` in the vocabulary. During training, the model will output a probability distribution over the vocabulary, and it will be trained to maximize the probability of the correct next word.

Building the Model

1. Defining the Sequential Model: `Sequential()` creates a linear stack of layers in Keras, which means that each layer is added sequentially on top of the previous one.
In this case, we are creating a simple stack of layers, which is well-suited for a feed-forward neural network. The layers we add later will be placed in the order in which the model processes data: input to output.

2. Embedding Layer: The `Embedding` layer converts integer-encoded words into dense vectors of fixed size, allowing the model to work with word embeddings.
   The parameters are as follows:
   a. **`total_words`**: This is the size of the vocabulary (i.e., the number of unique words in the tokenizer's word index + 1). It tells the `Embedding` layer how many unique integer tokens need to be embedded.
   b. **`100`**: This is the dimensionality of the embedding space. Each word will be represented as a vector of 100 dimensions.
   c. **`input_length=max_sequence_len-1`**: This specifies the length of the input sequences to the model. The input to the `Embedding` layer will be sequences of length `max_sequence_len - 1`, which corresponds to the length of the n-grams created earlier.

   Instead of passing one-hot encoded vectors (which are very sparse), we pass a dense vector representation (embedding) for each word. The model learns the best way to represent each word in a dense space during training. The dense vectors capture semantic similarities between words, which helps the model generalize better when predicting the next word. The word embeddings are learned jointly with the rest of the model, allowing the network to improve them based on the task at hand (next-word prediction).

   Embeddings help the model learn relationships between words. Semantically similar words (like "dog" and "puppy") are often mapped to nearby points in the embedding space.

3. LSTM Layer: This adds a Long Short-Term Memory (LSTM) layer to the model with 150 units (neurons).
   LSTM (Long Short-Term Memory) is a type of Recurrent Neural Network (RNN) architecture that is particularly good at capturing long-term dependencies in sequential data. In the task of next-word prediction, the context from previous words is crucial for predicting the next word. LSTM units are designed to remember information from previous steps and use that to influence the current step. LSTMs avoid the "vanishing gradient problem" faced by regular RNNs, allowing them to retain information over longer sequences of text.

   What Happens Inside the LSTM Layer?
   a. The LSTM layer processes the embedded word sequences sequentially (word-by-word).
   b. The LSTM units have internal memory cells that decide what information to retain, what to forget, and what to output at each time step.
   c. The 150 units define how many "memory cells" the LSTM has, and thus how much capacity it has to remember past information.

4. Dense (Fully Connected) Layer: This adds a fully connected (Dense) output layer with `total_words` neurons (one for each word in the vocabulary) and a `softmax` activation function.
   Dense Layer: Each neuron in the dense layer is connected to every neuron in the previous LSTM layer. In this case, the dense layer has `total_words` neurons,

corresponding to the size of the vocabulary. Each neuron will output the probability of a particular word being the next word in the sequence.

activation='softmax':

a. `softmax` is used for multi-class classification problems. It outputs a probability distribution over all the words in the vocabulary (i.e., the sum of the output probabilities will be 1).
b. The model will assign a probability to each possible next word, and the word with the highest probability will be predicted as the next word.

The LSTM layer generates a high-dimensional representation of the input sequence, which is then passed to the Dense layer. The Dense layer reduces this representation down to `total_words` possible outcomes, each corresponding to a word in the vocabulary. The `softmax` function turns these raw scores into probabilities, allowing us to interpret the model's predictions as the likelihood of each word being the next word.

Train the Model

Train the model using the sequences and corresponding next words (y). This process can take some time depending on the dataset size.

1. Compiling the Model: **`model.compile()`** is where we specify the loss function, the optimizer, and the metrics we want to track during training.
    a. `categorical_crossentropy` is the loss function commonly used for multi-class classification problems. It measures how far the predicted probability distribution is from the true distribution (where the actual next word is the true class). Since this is a multi-class classification problem (predicting the next word from a large vocabulary), `categorical_crossentropy` is appropriate as it compares the predicted word probabilities with the actual word (one-hot encoded).
    b. `adam` (Adaptive Moment Estimation) is an optimization algorithm that combines the benefits of two popular optimizers: Gradient Descent with Momentum and RMSProp. It adjusts the learning rate for each parameter dynamically. Adam is commonly used for its robustness and efficiency. It tends to perform well in many tasks, especially for models with large data and parameters, like LSTMs. It helps in faster convergence and avoids local minima effectively.
    c. Metrics: `['accuracy']`: Here, we are specifying that we want the model to track accuracy during training. This metric will show how often the predicted word matches the actual word. Monitoring the accuracy provides insight into how well the model is learning to predict the next word correctly. It helps us track improvements over each epoch.

2. Fitting the Model (Training the Model): **`model.fit()`** is where we train our model on the data, iterating over it multiple times (epochs) and adjusting the model weights based on the loss.
    a. Input `X` and Target `y`:

    i. **X**: These are our input sequences (n-grams, minus the last word).
    ii. **y**: These are the actual next words (the target values, which are one-hot encoded).
  b. `epochs=50`:
    i. We are specifying that the model should go through the entire dataset 50 times. Each epoch means one complete forward and backward pass through the training dataset.

    Why 50 epochs? For sequence prediction models like this one, it often takes many passes through the data to adjust the weights effectively and for the model to learn the patterns in the data.

3. `verbose=1`:
  a. This controls the amount of output we see during training. When set to `1`, we get a progress bar for each epoch, along with the loss and accuracy metrics.

4. In each epoch:
  a. The model processes a batch of input sequences (`X`), makes predictions for the next word, and calculates the loss (`categorical_crossentropy`) based on how far off the predictions are from the true values (`y`).
  b. The model then adjusts the weights in the LSTM and Dense layers based on the gradients computed by the optimizer (`adam`).
  c. After completing the epoch, the accuracy is computed, and we can see how well the model is doing at predicting the next word.

  The weights of the model are updated incrementally as it sees more and more batches of data, and with each epoch, the accuracy generally improves (if the model is learning effectively). The LSTM layer learns to recognize patterns in word sequences, and the Dense layer fine-tunes the output to match the correct next word. The model becomes better at generalizing the relationship between word sequences and the next word with each epoch, ideally leading to improved next-word predictions over time.

Next-Word Prediction based on a Seed Text

1. Initial Setup:
  a. **`seed_text`**: This is the initial input or starting sequence from which the model will begin predicting the next words.
  b. **`next_words = 7`**: This means we want to generate 7 new words to follow the seed text. The model will predict one word at a time, and then add it to the existing sentence before predicting the next one.

2. Loop for Predicting Multiple Words: The loop runs 7 times (or as many times as specified by `next_words`), predicting one word during each iteration and appending it to the `seed_text`.

3. Convert Seed Text to Tokens:

a. **`tokenizer.texts_to_sequences([seed_text])`**: The `Tokenizer` object (from earlier preprocessing) is used to convert the `seed_text` into a sequence of tokens (integers). Each word in the text is mapped to its respective integer from the tokenizer's word index.
b. **`[0]`**: Since `texts_to_sequences()` returns a list of lists, `[0]` extracts the inner list to get just the token sequence.

4. Padding the Sequence:
    a. **`pad_sequences()`**: This function pads the sequence with zeros at the beginning (since `padding='pre'`) to ensure that the sequence has the same length as the `max_sequence_len - 1`. This is necessary because the LSTM model was trained on sequences of this fixed length.
    b. `max_sequence_len-1:` The model expects input sequences of this length, as it was trained with sequences of `max_sequence_len` but excluding the final word (which is the prediction target).

5. Model Prediction:
    a. **`model.predict(token_list)`**: The model takes the padded sequence and predicts the probability distribution for the next word in the sequence.
    b. **`np.argmax()`**: This function finds the index of the highest probability from the model's output, which corresponds to the predicted next word. The output of the model is a probability distribution over the entire vocabulary (with size `total_words`), and `np.argmax()` selects the word with the highest probability.

6. Mapping the Predicted Token Back to a Word:
    a. **`tokenizer.word_index.items()`**: This gives a dictionary where each word is mapped to an index. This loop goes through the dictionary to find the word that corresponds to the predicted index.
    b. Once the word is found, it is stored in the `output_word` variable.

7. Updating the Seed Text: The predicted word is added to the `seed_text`. After each iteration, the sentence grows longer, and this updated `seed_text` is then used for predicting the next word.