**Next Word Prediction with Deep Learning in NLP**

Identifying the most likely word to follow a given string of words is the basic goal of the Natural Language Processing (NLP) task of “next word prediction.” This predictive skill is essential in various applications, including text auto-completion, speech recognition, and machine translation. Deep learning approaches have transformed NLP by attaining remarkable success in various language-related tasks, such as next-word prediction.

Deep learning models are excellent at identifying complex dependencies and patterns in sequential data, which makes them suitable for challenges requiring the prediction of the next word. These models may successfully describe the context and make precise predictions by utilizing recurrent neural networks (RNNs) and their variations, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU).

There are many benefits to being able to correctly predict the following word in a given situation. Offering relevant and coherent word suggestions improves the user experience in auto-completion systems.

**Model Architecture**

The model architecture is a critical component in building an effective next-word prediction system using deep learning in NLP. One common approach is to utilize recurrent neural networks (RNNs) or their variants, such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU). These architectures are specifically designed to capture sequential dependencies in the input text, enabling accurate predictions of the next word.

**Recurrent Neural Networks (RNNs):** Recurrent Neural Networks are a class of neural networks that can process sequential data by maintaining hidden states that capture the context and information from previous inputs. RNNs have loops within their architecture that allow them to store and propagate information through time.

**Long Short-Term Memory (LSTM):** LSTM is a specialized variant of RNNs that overcomes the limitations of traditional RNNs, such as the vanishing gradient problem. LSTM introduces a memory cell that allows the network to selectively store and retrieve information over long sequences, making it particularly effective in modeling long-term dependencies.

**Model Architecture for Next Word Prediction**

The model architecture for next-word prediction typically consists of the following components:

**Embedding Layer:** The word embeddings, often referred to as distributed representations of words, are learned by the embedding layer. It captures semantic and contextual information by mapping every word in the lexicon to a dense vector representation. The model can be trained to change these embeddings as trainable parameters.

**Recurrent Layers:** The recurrent layers, like LSTM or GRU, process the input word embeddings in sequential order and keep hidden states that record the sequential data. The model can learn the contextual connections between words and their placements in the sequence thanks to these layers.

**Dense Layers:** One or more dense layers are then added after the recurrent layers to convert the learned features into the appropriate output format. When predicting the next word, the dense layers translate the hidden representations to a probability distribution across the vocabulary, indicating the likelihood that each word will be the following word.

**Training and Optimization**

The model is trained using a large corpus of text data, where the input sequences are paired with their corresponding target word. The training process involves optimizing the model’s parameters by minimizing a suitable loss function, such as categorical cross-entropy. The optimization is typically performed using an optimization algorithm like Adam or Stochastic Gradient Descent (SGD).

**Inference and Prediction**

The model can be used to predict the next word once it has been trained. The trained model receives an input of a list of words, processes it through the learned architecture, and outputs a probability distribution across the vocabulary. The anticipated next word is then chosen as the one with the highest likelihood.

We can create reliable and accurate next-word prediction models for NLP by combining the strength of recurrent neural networks, such as LSTM or GRU, with the right training and optimization methods. The model design successfully captures the text’s sequential dependencies, enabling the system to produce fluent and contextually relevant predictions.

**Import libraries**

The necessary libraries are imported. TensorFlow is imported as ‘tf’ to utilize its functionalities for deep learning. The Sequential model from Keras is imported to build a sequential neural network, and specific layers such as Embedding, LSTM, and Dense are imported. Numpy is imported as np for generating arrays and regex as a re for data processing pattern recognition.

*import tensorflow as tf*

*from tensorflow.keras.models import Sequential*

*from tensorflow.keras.layers import Embedding, LSTM, Dense*

*from tensorflow.keras.preprocessing.text import Tokenizer*

*from tensorflow.keras.preprocessing.sequence import pad\_sequences*

*import numpy as np*

*import regex as re*

**Understanding and preprocessing the dataset**

The Section carries out a number of operations to prepare text data for a language model’s training. It begins by reading a file and breaking up its text into separate sentences. The Keras Tokenizer class, which learns the vocabulary from the input sentences, tokenizes the text data once it has been parsed.

The tokenized data is then used to produce n-gram sequences, each of which contains a range of tokens from the start to the current index. Input\_sequences contains a list of these sequences. The sequences are padding with zeros in order to guarantee uniform length. The input sequences are divided into predictors (X) and labels (Y), where X contains all elements other than the last token of each sequence and Y only the final token. Finally, the target data y is converted to one-hot encoding, ready for training a language model.

Use text file : pizza.txt

*def file\_to\_sentence\_list(file\_path):*

*with open(file\_path, 'r') as file:*

*text = file.read()*

*# Splitting the text into sentences using*

*# delimiters like '.', '?', and '!'*

*sentences = [sentence.strip() for sentence in re.split(*

*r'(?<=[.!?])\s+', text) if sentence.strip()]*

*return sentences*

*file\_path = 'pizza.txt'*

*text\_data = file\_to\_sentence\_list(file\_path)*

*# Tokenize the text data*

*tokenizer = Tokenizer()*

*tokenizer.fit\_on\_texts(text\_data)*

*total\_words = len(tokenizer.word\_index) + 1*

*# Create input sequences*

*input\_sequences = []*

*for line in text\_data:*

*token\_list = tokenizer.texts\_to\_sequences([line])[0]*

*for i in range(1, len(token\_list)):*

*n\_gram\_sequence = token\_list[:i+1]*

*input\_sequences.append(n\_gram\_sequence)*

*# Pad sequences and split into predictors and label*

*max\_sequence\_len = max([len(seq) for seq in input\_sequences])*

*input\_sequences = np.array(pad\_sequences(*

*input\_sequences, maxlen=max\_sequence\_len, padding='pre'))*

*X, y = input\_sequences[:, :-1], input\_sequences[:, -1]*

*# Convert target data to one-hot encoding*

*y = tf.keras.utils.to\_categorical(y, num\_classes=total\_words)*

**Defining the Model**

The input sequences are mapped to dense vectors of fixed size in the first layer, which is an embedding layer. It requires three arguments: input\_length (the length of the input sequences less one because we are predicting the next word), 10 (the dimensionality of the embedding space), and total\_words (the total number of unique words in the vocabulary).

An LSTM (Long Short-Term Memory) layer with 128 units makes up the second layer. Recurrent neural networks (RNNs) of the LSTM variety may recognize long-term dependencies in sequential input.

A Dense layer with total\_words units and softmax activation make up the third layer. The output probabilities for each word in the vocabulary are generated by this layer. The categorical cross-entropy loss function used in the model is appropriate for multi-class classification applications. Adam is the chosen optimizer, and accuracy is the evaluation metric.

*# Define the model*

*model = Sequential()*

*model.add(Embedding(total\_words, 10,*

*input\_length=max\_sequence\_len-1))*

*model.add(LSTM(128))*

*model.add(Dense(total\_words, activation='softmax'))*

*model.compile(loss='categorical\_crossentropy',*

*optimizer='adam', metrics=['accuracy'])*

Train the Model

The fit method of the model object is used in this code to train the defined model. As parameters, the input data X and the target data Y are given. The number of iterations that the entire dataset will undergo during training is indicated by the epochs parameter, which is set to 500. The model develops the ability to anticipate the following word in a sequence depending on the input data during training. The progress bar and training data for each epoch are shown when the verbose parameter is set to 1, which is the default value.

The model is trained for 500 epochs by running this code, and the weights of the model’s layers are adjusted iteratively to reduce the defined loss function and increase precision in predicting the next word in the input sequences.

*# Train the model*

*model.fit(X, y, epochs=500, verbose=1)*

Output:

Epoch 496/500

51/51 [==============================] - 1s 20ms/step - loss: 0.0679 - accuracy: 0.9668

Epoch 497/500

51/51 [==============================] - 1s 20ms/step - loss: 0.0671 - accuracy: 0.9687

Epoch 498/500

51/51 [==============================] - 1s 24ms/step - loss: 0.0676 - accuracy: 0.9687

Epoch 499/500

51/51 [==============================] - 1s 20ms/step - loss: 0.0679 - accuracy: 0.9681

Epoch 500/500

51/51 [==============================] - 1s 21ms/step - loss: 0.0673 - accuracy: 0.9705

**Predicting the next word**

The variable seed\_text contains the initial input text from which we want to generate the next word predictions. The variable next\_words indicates the number of words to be predicted. A loop is then executed next\_words times. Inside the loop, the seed\_text is tokenized using the tokenizer’s texts\_to\_sequences method. The token list is padded to match the expected input length of the model’s input sequences.

The model’s prediction method is called on the padded token list to obtain the predicted probabilities for each word in the vocabulary. The argmax function is used to determine the index of the word with the highest probability.

The predicted word is obtained by converting the index to the corresponding word using the tokenizer’s index\_word dictionary. The predicted word is then appended to the seed\_text. This process is repeated for the desired number of next\_words predictions. Finally, the generated sequence of words is printed as “Next predicted words: [seed\_text]”.

*# Generate next word predictions*

*seed\_text = "Pizza have different "*

*next\_words = 5*

*for \_ in range(next\_words):*

*token\_list = tokenizer.texts\_to\_sequences([seed\_text])[0]*

*token\_list = pad\_sequences(*

*[token\_list], maxlen=max\_sequence\_len-1, padding='pre')*

*predicted\_probs = model.predict(token\_list)*

*predicted\_word = tokenizer.index\_word[np.argmax(predicted\_probs)]*

*seed\_text += " " + predicted\_word*

*print("Next predicted words:", seed\_text)*

*Output:*

*1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 191ms/step*

*1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step*

*1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step*

*1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step*

*1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step*

*Next predicted words: Pizza have different become a symbol of comfort*

**Conclusion**

Deep learning in NLP has several useful applications, including next-word prediction. We can efficiently capture the sequential dependencies in text data and produce precise predictions by applying models like LSTM or GRU. Next-word prediction models keep getting better thanks to deep learning developments and the accessibility of big text corpora, which enhance user experience and enable a variety of NLP applications.