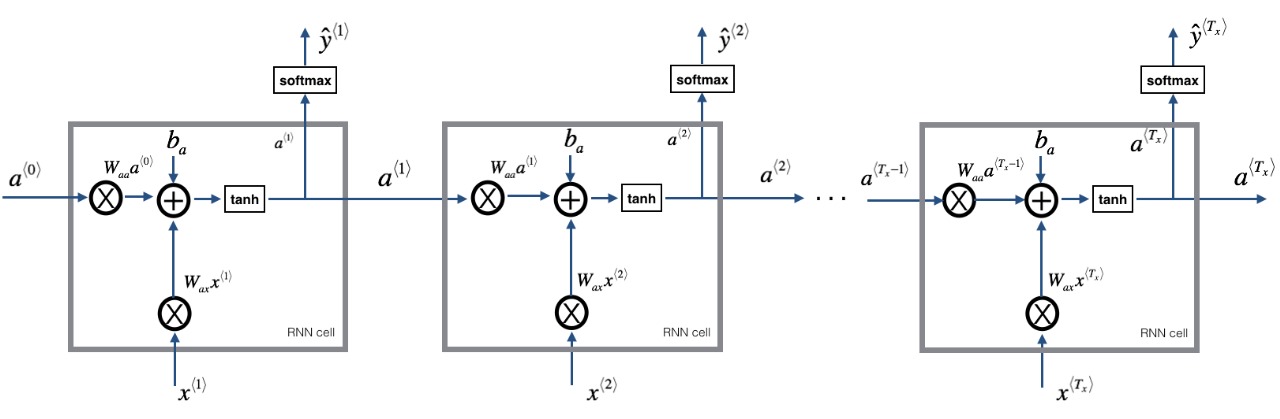
|  |  |
| --- | --- |
| Ex No: 8  Date: 25/09/2024 | **RNN Implementation for Name Generation** |

**Objective:**

The main objective is to implement a Recurrent Neural Network (RNN) to generate human-readable names based on character-level modeling. The model learns the structure of names from a dataset of known names and generates new names by predicting the next character iteratively.



**Code Explanation for RNN:**

**Dataset Preprocessing**: The dataset consists of character-level data representing names. Each name is broken down into individual characters, and a mapping of characters to indices (char\_to\_ix) and indices to characters (ix\_to\_char) is created. This enables the model to work with sequential data.

**RNN Model Architecture**:

1. **Input Embedding**: Converts characters to their corresponding one-hot vectors.
2. **Recurrent Hidden Layers**: The hidden state is updated at each time step by combining the previous state and the input character using learned weight matrices.
3. **Softmax Output Layer**: After processing the sequence, the final output layer predicts the next character in the sequence using softmax activation, producing a probability distribution over all possible characters.

1. **Gradient Clipping Function**:

* def clip(gradients, maxValue): Defines the clip function that takes two arguments: gradients (a dictionary of gradient values) and maxValue (the threshold for clipping).
* Docstring: Explains the purpose of the function and the arguments it accepts.
* Extracts the individual gradient matrices (dWaa, dWax, dWya, db, dby) from the gradients dictionary.
* Loops over each gradient and clips its values using np.clip(). If any value exceeds maxValue, it's set to maxValue. If it's below -maxValue, it's set to -maxValue.
* Reconstructs the gradients dictionary with the clipped gradients.
* Returns the clipped gradients.

2. **Sampling Function**:

* def sample(parameters, char\_to\_ix, seed): Defines the sample function, which generates a sequence of characters from the RNN model's learned parameters.
* Retrieves the weight matrices and bias terms from the parameters dictionary.
* vocab\_size: Defines the size of the vocabulary based on the shape of the by matrix.
* n\_a: Sets the size of the hidden layer from the Waa matrix.
* x: Initializes the input as a one-hot vector of size vocab\_size.
* a\_prev: Initializes the hidden state as a zero vector.
* indices: An empty list to store the sampled character indices.
* idx **= -1**: Initializes idx to -1 to start sampling.
* np.tanh(): Applies the tanh activation function to compute the new hidden state.
* y = softmax(z): Converts the logits z into a probability distribution using the softmax function.
* np.random.choice(): Samples a character index idx based on the probability distribution y.
* indices.append(idx): Adds the sampled index to the list indices.
* x[idx] = 1: Sets the corresponding one-hot vector for the sampled character.
* a\_prev = a: Updates the hidden state for the next step.
* indices.append(char\_to\_ix['\n']): If the loop runs for 50 steps without reaching the newline character, it appends a newline.
* return indices: Returns the list of sampled character indices.

3. **Optimize Function:**

* def optimize(X, Y, a\_prev, parameters, learning\_rate=0.01): Defines the optimize function that performs one step of optimization (forward and backward propagation) to update the parameters of the RNN. It takes the input sequence X, the target sequence Y, the previous hidden state a\_prev, the model's parameters, and the learning rate as inputs.
* The docstring explains the function’s arguments and return values.
* rnn\_forward(X, Y, a\_prev, parameters): Performs forward propagation through the RNN to compute the loss and caches intermediate values for use in backpropagation.
* **loss, cache**: loss stores the cross-entropy loss, and cache stores intermediate values needed for backpropagation.
* rnn\_backward(X, Y, parameters, cache): Performs backpropagation through time (BPTT) to compute the gradients of the loss with respect to the parameters. It uses the cache from forward propagation.
* gradients: A dictionary containing the gradients for each parameter.
* **a**: The updated hidden states at each time step.
* clip(gradients, 5): Clips the gradients to avoid exploding gradients. The values of the gradients are clipped between -5 and 5 to stabilize training.
* update\_parameters(parameters, gradients, learning\_rate): Updates the RNN parameters using gradient descent. The parameters are adjusted in the opposite direction of the gradients with a step size defined by the learning rate.
* return: Returns the loss, gradients, and the last hidden state a[len(X)-1] for use in the next time step during training.

4. **Model Function**:

* Defines the main training function for the RNN model. It trains the model for a specified number of iterations and generates dinosaur names during the training process.
* num\_iterations: Number of iterations to train the model.
* n\_a: Number of units in the RNN’s hidden layer.
* dino\_names: Number of dinosaur names to generate and print at each checkpoint.
* vocab\_size: The size of the character vocabulary.
* n\_x, n\_y = vocab\_size, vocab\_size: Defines the input and output dimensions of the RNN, which are both equal to the vocabulary size (since we are predicting characters).
* initialize\_parameters(n\_a, n\_x, n\_y): Initializes the RNN's weight matrices and biases.
* get\_initial\_loss(vocab\_size, dino\_names): Initializes the loss for tracking model performance.
* with open("dinos.txt")...: Loads the dinosaur name dataset from the text file.
* examples = [x.lower().strip() for x in examples]: Converts the dataset to lowercase and strips any whitespace characters.
* np.random.shuffle(examples): Shuffles the order of the examples to randomize the training data.
* a\_prev: Initializes the hidden state of the RNN to zeros. This will be updated after every sequence of characters is processed.
* for j in range(num\_iterations):: Loops over the total number of iterations to train the model.
* single\_example = examples[idx]: Selects the current training example from the shuffled examples list.
* single\_example\_chars: Converts the dinosaur name into a list of characters.
* single\_example\_ix: Converts the characters into their corresponding indices using char\_to\_ix.
* X = [None] + single\_example\_ix: Prepares the input sequence for the RNN, with the first element set to None (to represent the start of the sequence).
* ix\_newline = char\_to\_ix['\n']: Retrieves the index for the newline character, used as the end-of-sequence marker.
* Y = X[1:] + [ix\_newline]: Creates the target sequence by shifting the input sequence to the left and adding a newline character at the end.
* optimize(X, Y, a\_prev, parameters): Performs one step of optimization by calculating the loss, gradients, and updating the model parameters.
* smooth(loss, curr\_loss): Applies a smoothing function to keep the loss curve stable over time, making it easier to observe the general trend.
* if j % 2000 == 0:: Every 2000 iterations, prints the current iteration and loss.
* sample(parameters, char\_to\_ix, seed): Samples a sequence of characters (i.e., a dinosaur name) based on the current parameters of the RNN.
* print\_sample(sampled\_indices, ix\_to\_char): Converts the sampled indices back into characters and prints the generated name.
* seed: A variable to ensure the randomness in sampling is controlled for consistency.
* return parameters: Returns the final learned parameters after the model finishes training.

5. **Training Model:**

* Calls the model function to train the RNN and generate dinosaur names periodically. The verbose=True argument enables detailed output during training.

**Results**

The trained Recurrent Neural Network (RNN) successfully generates plausible dinosaur names from the character-level model. As the model progresses through the iterations, the generated names become more coherent and resemble real names found in the dataset. Initially, the output is random, but as the training continues, the model learns patterns in character sequences, forming recognizable names with proper structure. The loss decreases steadily, indicating that the model improves its ability to predict the next character over time. Although some generated names may not always be realistic, the RNN's ability to capture linguistic patterns demonstrates its effectiveness in generating text.

**Result Analysis:**

The RNN model shows a clear progression in learning to generate realistic names as the training continues. At the start, the generated names are completely random, with a high loss of over 23, but by iteration 2000, the model starts forming shorter and partially recognizable names. By iteration 8000, more coherent and valid names like Joshua and Viveda appear, indicating significant improvement. As the training progresses to iteration 10000 and beyond, the model consistently generates realistic names such as Matthew, Isaac, and Vivaan, with the loss decreasing steadily below 6. The final iterations demonstrate the model's ability to generate high-quality names with minimal loss, showcasing its effective learning of patterns and structure in character sequences.

**Summary:**

This lab implements a character-level RNN to generate new dinosaur names based on a dataset of known names. It uses recurrent connections to learn temporal dependencies between characters in sequences. The model is trained using gradient-based optimization, and the parameters are updated to minimize cross-entropy loss. Through iterative training, the RNN learns to produce names that mimic real-world names, illustrating the power of sequence modeling for text generation tasks. This exercise highlights the importance of recurrent architectures in handling sequential data and showcases the potential of deep learning for creative generative tasks like name generation.

**GitHub Link:**

https://github.com/ManeshaMadhu/DeepLearning-5thsem-/blob/main/Lab%208/RNN%20lab.ipynb