USN NUMBER: 1RVU22CSE128
NAME: Reethu RG Thota

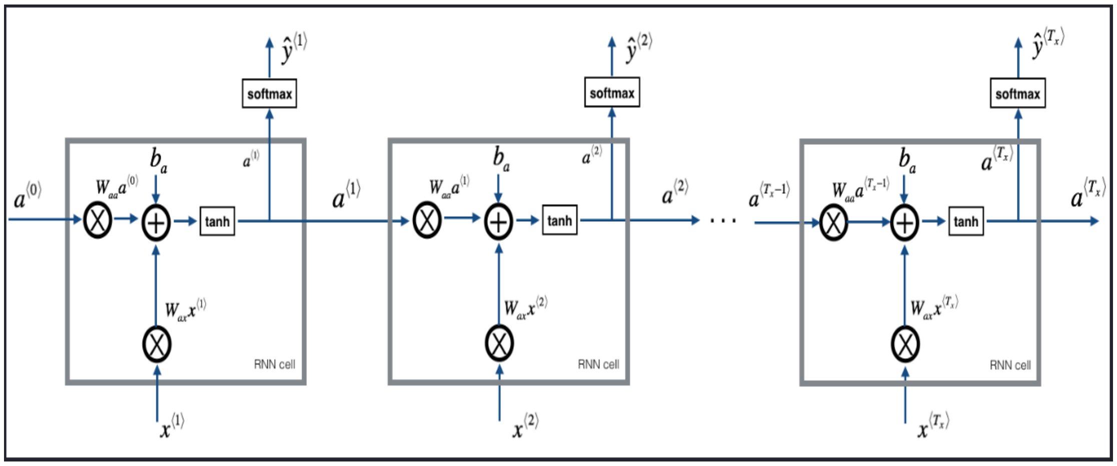
Ex No: 8 **Date:** 25/09/24	**Character level language model - Name generation**

Objective

The objective of this experiment is to build and train a character-level Recurrent Neural Network (RNN) to generate names by predicting each character in a sequence. Specifically, the goal is to generate dinosaur names by learning from a dataset of existing names and producing new ones. This project demonstrates how neural networks can be used to model sequences and generate realistic text by understanding patterns in character-level data.

Descriptions

A character-level language model is a type of sequence model that predicts the next character in a sequence of characters. Unlike word-level models, which operate on whole words, a character-level model works on individual characters. This allows it to generate completely new words, which is especially useful for tasks like name generation where individual characters must be predicted in order.



In this experiment, we implement a Recurrent Neural Network (RNN) for name generation. The RNN is trained on a dataset of dinosaur names, learning to predict each character based on the previous ones. By using stochastic gradient descent (SGD), we optimize the network's parameters, allowing it to generate new names character by character. The key steps involved include forward propagation (to calculate the loss), backward propagation (to compute gradients), gradient clipping (to avoid exploding gradients), and parameter updates.We monitor the model's performance by sampling new names from it at intervals during the training process. As the model improves, the generated names should start resembling real dinosaur names.

USN NUMBER: 1RVU22CSE128
NAME: Reethu RG Thota

Dataset

The dataset used in this experiment consists of dinosaur names. Each name serves as an individual training example. The names are converted into lists of characters, and the RNN is tasked with predicting the next character in the sequence. The characters include all letters in the names and a special newline character `\n`, which represents the end of a name.

Model

The model consists of the following parts:

1. **Forward Propagation**:
    a. The input character sequence is passed through the RNN cell. The RNN uses its weights to combine the input and the previous hidden state, updating the hidden state and generating predictions for the next character.
    b. The cross-entropy loss is calculated based on the predicted characters and the actual next characters in the sequence.
    c. The forward propagation returns the loss and a cache of values needed for backpropagation.
2. **Backward Propagation**:
    a. In this step, the loss is propagated backward through time (backpropagation through time, or BPTT). Gradients of the loss with respect to the RNN's parameters (weights and biases) are computed.
    b. These gradients indicate how to adjust the model's parameters to reduce the loss in the next iteration.
3. **Gradient Clipping**:
    a. Gradient clipping is performed to avoid the problem of exploding gradients, which can destabilize the training process. The gradients are clipped within a certain range (typically -5 to 5) before the parameters are updated.
4. **Parameter Update**:
    a. Using the computed gradients, the parameters (weights and biases) are updated through gradient descent. The learning rate controls the size of the update steps.
5. **Optimization**:
    a. The `optimize` function combines all these steps. For each training example, the model performs forward propagation, computes the gradients via backpropagation, clips the gradients, and updates the parameters.

Building the parts of algorithm

1. **Initialization** - The first step is initializing the model parameters. These include the weight matrices (`Wax`, `Waa`, `Wya`) and bias vectors (`b`, `by`). The weight matrices are initialized randomly, and the biases are initialized to zeros.

USN NUMBER: 1RVU22CSE128
NAME: Reethu RG Thota

```
def initialize_parameters(n_a, n_x, n_y):
    """
    Initializes the parameters of the RNN.
    Arguments:
    n_a -- number of units in the RNN cell
    n_x -- number of input features (vocabulary size)
    n_y -- number of output features (vocabulary size)
    Returns:
    parameters -- python dictionary containing the initialized
parameters
    """
    np.random.seed(1)  # Set a seed for reproducibility
    Wax = np.random.randn(n_a, n_x) * 0.01  # Weight matrix for
input to hidden state
    Waa = np.random.randn(n_a, n_a) * 0.01  # Weight matrix for
hidden state to hidden state
    Wya = np.random.randn(n_y, n_a) * 0.01  # Weight matrix for
hidden state to output
    b = np.zeros((n_a, 1))  # Bias for hidden state
    by = np.zeros((n_y, 1))  # Bias for output
    parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "b": b,
"by": by}  # Create a dictionary of parameters
    return parameters
```

2. **Optimization Process**:
   - Each step of the training process involves feeding a name (or sequence of characters) into the RNN, performing forward and backward propagation, clipping the gradients, and updating the model parameters.
   - **Forward Propagation**: The input characters are fed through the RNN, and the hidden states are updated at each time step. The RNN produces predictions for the next character at each step.

```
def rnn_forward(X, Y, a_prev, parameters):
    """
    Performs the forward propagation through the RNN and
computes the cross-entropy loss.
   Arguments:
    X -- input sequence (list of integers)
    Y -- output sequence (list of integers)
    a_prev -- previous hidden state
    parameters -- dictionary containing the parameters of
the model
    Returns:
    loss -- value of the loss function (cross-entropy)
    cache -- values needed for backpropagation
    """
    # Initialize cache to store intermediate values
    cache = {}
    loss = 0
    a = a_prev  # Set initial hidden state
    for t in range(len(X)):  # Loop through each time step
        # Calculate the next hidden state and output
        a, yt_pred = rnn_step_forward(X[t], a, parameters)
        cache[t] = (a, yt_pred)  # Store the hidden state
and output in cache
        loss += -np.log(yt_pred[Y[t], 0])  # Cross-entropy
loss for the correct output
    loss = loss / len(X)  # Average loss over the sequence
length
    return loss, cache
```

- **Backward Propagation**: Gradients of the loss with respect to the model parameters are computed by propagating the loss backward through time.

```
def rnn_backward(X, Y, parameters, cache):
    """
    Performs the backward propagation through time to
compute the gradients.
    Arguments:
    X -- input sequence (list of integers)
    Y -- output sequence (list of integers)
    parameters -- dictionary containing the parameters of
the model
    cache -- cache of values from forward propagation
    Returns:
    gradients -- dictionary of gradients with respect to
parameters
    a -- the hidden states
    """
    gradients = {} # Initialize gradients
    for key in parameters.keys():
        gradients[key] = np.zeros_like(parameters[key])  #
Create zero matrices for each parameter gradient
    # Retrieve the last hidden state
    a = cache[len(X) - 1][0]  # Last hidden state from
cache
    for t in reversed(range(len(X))):  # Loop backward
through time
        # Compute the gradient for the loss w.r.t. output
        dy = cache[t][1]  # Predicted output at time t
        dy[Y[t]] -= 1  # Subtract one for the correct class
        # Accumulate the gradients
        gradients["dWya"] += np.dot(dy, cache[t][0].T)  #
Gradient w.r.t. output weights
        gradients["dby"] += dy  # Gradient w.r.t. output
bias
        da = np.dot(parameters["Wya"].T, dy) + da_next  #
Gradient of the loss w.r.t. hidden state
        # Call function to get gradients w.r.t. hidden
state parameters
        gradients = rnn_cell_backward(da, cache[t],
parameters, gradients)  # Backpropagation through cell
    return gradients, a
```

USN NUMBER: 1RVU22CSE128
NAME: Reethu RG Thota

- **Gradient Clipping**: The gradients are clipped to prevent the problem of exploding gradients.

```
def clip(gradients, maxValue):
    """
    Clips the gradients to prevent exploding gradients.
    Arguments:
    gradients -- dictionary of gradients
    maxValue -- maximum value for clipping
    Returns:
    gradients -- clipped gradients
    """
    for key in gradients.keys():
        np.clip(gradients[key], -maxValue, maxValue,
out=gradients[key])  # Clip each gradient
    return gradients
```

- **Updating Parameters**: The model parameters are updated using gradient descent, with the learning rate controlling the update step size.

```
def update_parameters(parameters, gradients,
learning_rate):
    """
    Updates parameters using the Gradient Descent Update
Rule.
    Arguments:
    parameters -- dictionary of model parameters
    gradients -- dictionary of gradients
    learning_rate -- learning rate for the model
    Returns:
    parameters -- updated parameters
    """
    for key in parameters.keys():
        parameters[key] -= learning_rate * gradients[key]
# Update each parameter using the gradient
    return parameters
```

3. **Training Loop** - The model is trained over multiple iterations (e.g., 35,000). During each iteration, a randomly shuffled dinosaur name is used as the input, and the RNN learns to predict the next character in the sequence. Every 2000 iterations, new names are sampled and printed to observe the model's progress.

4. **Sampling**- After training, the RNN can be used to generate new names. A sampling function is used to generate a sequence of characters, starting from a random seed and producing one character at a time. The function samples characters based on the probabilities predicted by the RNN.

GitHub Link

https://github.com/reethuthota/Deep_Learning/tree/main/Lab8