|  |  |
| --- | --- |
| **Ex No: 8**  **Date: 25-08-2024** | **Lab 8 - RNN** |

**Objective:**

The objective of this lab is to build and train a character-level RNN to generate dinosaur names. This involves initializing the RNN parameters, performing forward and backward propagation with gradient clipping, and updating the model. The training process includes evaluating sample names every iterations to track and improve the model's performance.

**Descriptions:**

To create a character-level RNN that can generate dinosaur names. The process begins by setting up and initializing the RNN parameters and establishing a loss function to ensure smooth training. Dinosaur names are loaded, preprocessed, and shuffled to facilitate the iterative training of the model. During each training iteration, names are converted into character indices, and input and label sequences are created. The RNN uses forward propagation to calculate the loss and backward propagation to compute and clip gradients, preventing instability. Updated model parameters are used to refine the RNN's performance, with loss smoothing applied for stability. Every 2000 iterations, the model generates and prints dinosaur names to evaluate progress. The aim is to fine tune the RNN to produce convincing and imaginative dinosaur names.

**Code explanation:**

import numpy as np

from utils import \*

import random

import pprint

* numpy helps with array-based numerical computations (essential in machine learning for operations on matrices).
* from utils import \* imports all utility functions defined elsewhere simplifying main code.
* random is used for random operations, such as shuffling data or generating random names.
* pprint makes it easier to display complex structures like dictionaries in a clear, formatted way.

data = open('dinos.txt', 'r').read()

data = data.lower()

chars = list(set(data))

data\_size, vocab\_size = len(data), len(chars)

print('There are %d total characters and %d unique characters in your data.' % (data\_size, vocab\_size))

The code reads the entire content of the file `'dinos.txt'` and converts it to lowercase to ensure consistency. It then creates a set of unique characters from the text, removes duplicates, and converts this set into a list. The total number of characters in the dataset is stored as data\_size, and the number of unique characters is stored as vocab\_size. Finally, the code prints the total and unique character counts in the dataset. This preprocessing step is crucial for preparing the data for a character-level language model.

np.random.seed(3)

This ensures that the random numbers generated by NumPy are consistent each time the code is run. It's useful when debugging or comparing results across runs. Setting the seed ensures that the same set of random numbers is generated each time.

dWax = np.random.randn(5, 3) \* 10

dWaa = np.random.randn(5, 5) \* 10

dWya = np.random.randn(2, 5) \* 10

db = np.random.randn(5, 1) \* 10

dby = np.random.randn(2, 1) \* 10

* dWax: Gradient of the weights between input and hidden states (5x3 matrix).
* dWaa: Gradient of the weights between hidden states (5x5 matrix).
* dWya: Gradient of the weights between hidden and output states (2x5 matrix).
* db: Gradient of the bias for hidden states (5x1 vector).
* dby: Gradient of the bias for output states (2x1 vector).

The \* 10 multiplies the normally distributed values by 10 to simulate large gradients that could lead to exploding gradient problems.

gradients = {"dWax": dWax, "dWaa": dWaa, "dWya": dWya, "db": db, "dby": dby}

All the gradients are stored in a dictionary for easy access and manipulation. This structure allows us to pass and update all the gradients together in an organized way.

gradients = clip(gradients, mValue)

This calls the clip function, which applies gradient clipping to avoid the "exploding gradient" problem. The gradients are clipped to a range between -mValue and mValue . This helps stabilize the training process by ensuring that the gradient values don’t become too large, which can cause the model to diverge.

def clip(gradients, maxValue):

for gradient in [gradients['dWaa'], gradients['dWax'], gradients['dWya'], gradients['db'], gradients['dby']]:

np.clip(gradient, -maxValue, maxValue, out=gradient)

return gradients

Clips each gradient matrix or vector element-wise. Any value greater than maxValue is set to maxValue, and any value less than -maxValue is set to -maxValue. This operation is done "in-place" using the out=gradient parameter, meaning the original gradient is directly modified without creating a new object

x = np.zeros((vocab\_size, 1))

a\_prev = np.zeros((n\_a, 1))

* x: This is a one-hot vector initialized as zeros. It represents the current input to the RNN, where the first character has not yet been generated. vocab\_size is the number of unique characters.
* a\_prev: This stores the previous hidden state, initialized to zero, as we haven't processed any characters yet. n\_a is the dimensionality of the hidden state.

while (idx != newline\_character and counter != 50):

The loop continues generating characters until a newline character (\n) is sampled or a maximum of 50 characters is generated. This prevents infinite loops in case of improper sampling or model issues.

a = np.tanh(np.dot(Wax, x) + np.dot(Waa, a\_prev) + b)

z = np.dot(Wya, a) + by

y = softmax(z)

* a: This computes the hidden state at the current timestep using the input x, the previous hidden state a\_prev, and the model parameters Wax, Waa, and b.
* z: This computes the pre-softmax activations using the hidden state a and model parameters Wya and by.
* y: The softmax function turns z into probabilities for each character in the vocabulary, representing the likelihood of each character being the next in the sequence.

idx = np.random.choice(list(range(vocab\_size)), p=y.ravel())

This line samples a character index (idx) based on the probabilities in y. np.random.choice selects the index according to the distribution provided by y.

x = np.zeros((vocab\_size, 1))

x[idx] = 1

a\_prev = a

* After sampling, x is updated to represent the new one-hot encoded vector corresponding to the sampled character.
* a\_prev is updated to store the current hidden state, so it can be used in the next time step for generating the next character.

if (counter == 50):

indices.append(char\_to\_ix['\n'])

If the loop reaches the maximum character limit (50), a newline character is appended to the indices, marking the end of the sequence.

Optimize function

def optimize(X, Y, a\_prev, parameters, learning\_rate = 0.01):

# Forward propagate through time (calculates loss and stores intermediate values in cache)

loss, cache = rnn\_forward(X, Y, a\_prev, parameters)

# Backpropagate through time (calculates the gradients using the cache from forward prop)

gradients, a = rnn\_backward(X, Y, parameters, cache)

# Clip your gradients between -5 and 5 (to prevent exploding gradients)

gradients = clip(gradients, 5)

# Update parameters (use gradient descent to adjust the parameters)

parameters = update\_parameters(parameters, gradients, learning\_rate)

return loss, gradients, a[len(X)-1]

In the forward propagation step, the inputs X, labels Y, and previous hidden state a\_prev are fed into the RNN to compute the loss and store intermediate values such as activations and hidden states in a cache. This cache is essential for the backward propagation step, where gradients of the loss function with respect to the model's weights and biases are computed. To address the issue of exploding gradients, these gradients are clipped to a range between -5 and 5. After clipping, the model parameters (weights and biases) are updated using gradient descent with a learning rate of 0.01, ensuring controlled adjustments. The function returns the current loss, the clipped gradients, and the last hidden state, which will serve as the starting point for the next sequence.

**Forward Propagation**: loss, cache = rnn\_forward(X, Y, a\_prev, parameters)

1. This line computes the loss and collects intermediate values (cached values) used in the next step of the training process.
2. The rnn\_forward function is responsible for taking the input sequence X, the target sequence Y, the previous hidden state a\_prev, and the model parameters (parameters). It processes this data through the RNN to produce the output predictions and compute the loss based on how well the predictions match the targets.
3. Outputs:

loss: This is a measure of how well the model's predictions align with the actual target values. It's used to evaluate the performance of the model.

cache: This contains intermediate values computed during the forward pass, such as activations and hidden states. These values are needed for the backward propagation step to compute gradients.

**Backward Propagation**: gradients, a = rnn\_backward(X, Y, parameters, cache)

1. This line calculates the gradients of the loss function with respect to the model's parameters and the hidden states. These gradients are crucial for updating the model parameters to minimize the loss.
2. The rnn\_backward function takes the same input and target sequences X and Y, the model parameters, and the cache from the forward propagation step. It uses this information to compute the gradients for each parameter (weights and biases) by propagating the error backward through the network.
3. **Outputs**:

gradients: This dictionary contains the derivatives of the loss with respect to each parameter in the model. These gradients indicate how much each parameter should be adjusted to reduce the loss.

a: This includes the hidden states of the RNN, which are used in the subsequent time steps or iterations.

Model function

def model(data, ix\_to\_char, char\_to\_ix, num\_iterations = 35000, n\_a = 50, dino\_names = 7, vocab\_size = 27, verbose = False):

# Retrieve n\_x and n\_y from vocab\_size

n\_x, n\_y = vocab\_size, vocab\_size

# Initialize parameters

parameters = initialize\_parameters(n\_a, n\_x, n\_y)

# Initialize loss (this is required because we want to smooth our loss)

loss = get\_initial\_loss(vocab\_size, dino\_names)

# Build list of all dinosaur names (training examples).

with open("dinos.txt") as f:

examples = f.readlines()

examples = [x.lower().strip() for x in examples]

# Shuffle list of all dinosaur names

np.random.seed(0)

np.random.shuffle(examples)

# Initialize the hidden state of your LSTM

a\_prev = np.zeros((n\_a, 1))

# Optimization loop

for j in range(num\_iterations):

### START CODE HERE ###

# Set the index `idx` (see instructions above)

idx = j % len(examples)

# Set the input X (see instructions above)

single\_example = examples[idx]

single\_example\_chars = [c for c in single\_example]

single\_example\_ix = [char\_to\_ix[c] for c in single\_example\_chars]

X = [None] + single\_example\_ix

# Set the labels Y (see instructions above)

ix\_newline = char\_to\_ix['\n']

Y = X[1:] + [ix\_newline]

# Perform one optimization step: Forward-prop -> Backward-prop -> Clip -> Update parameters

# Choose a learning rate of 0.01

curr\_loss, gradients, a\_prev = optimize(X, Y, a\_prev, parameters)

### END CODE HERE ###

# debug statements to aid in correctly forming X, Y

if verbose and j in [0, len(examples) -1, len(examples)]:

print("j = " , j, "idx = ", idx,)

if verbose and j in [0]:

print("single\_example =", single\_example)

print("single\_example\_chars", single\_example\_chars)

print("single\_example\_ix", single\_example\_ix)

print(" X = ", X, "\n", "Y = ", Y, "\n")

# Use a latency trick to keep the loss smooth. It happens here to accelerate the training.

loss = smooth(loss, curr\_loss)

# Every 2000 Iteration, generate "n" characters thanks to sample() to check if the model is learning properly

if j % 2000 == 0:

print('Iteration: %d, Loss: %f' % (j, loss) + '\n')

# The number of dinosaur names to print

seed = 0

for name in range(dino\_names):

# Sample indices and print them

sampled\_indices = sample(parameters, char\_to\_ix, seed)

print\_sample(sampled\_indices, ix\_to\_char)

seed += 1 # To get the same result (for grading purposes), increment the seed by one.

print('\n')

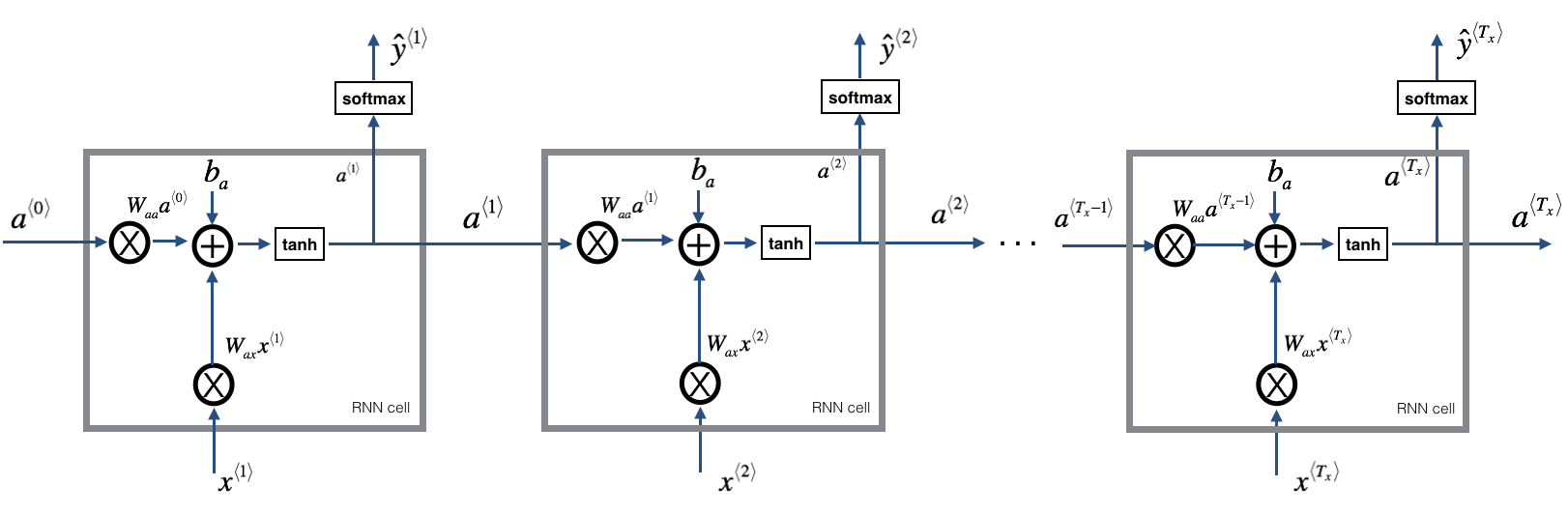
return parameters

The process begins by setting both input and output sizes to the vocabulary size of 27 and initializing the RNN parameters. The loss is then initialized to help smooth out fluctuations during training. Dinosaur names are loaded from "dinos.txt," converted to lowercase, and stripped of extra spaces to prepare the dataset. The hidden state, a\_prev, is initialized to zeros before entering the main training loop.

During each iteration of the loop, the model processes dinosaur names in shuffled order, using idx = j % len(examples) to wrap the index and ensure continuous training with the names. Input sequences X are created by prepending None to the list of character indices, and labels Y are generated by shifting X and appending a newline character. The optimize function is then called to perform forward propagation, backpropagation, and parameter updates.

To ensure stable training, the loss is smoothed using a moving average. Every 2000 iterations, the model generates and prints new dinosaur names to see progress and assess the quality of the generated names.

**Model:**

****

**Building the parts of algorithm**

1. Define Model Structure: Set up the RNN with appropriate input and output dimensions, such as the vocabulary size for character-level tasks.
2. Initialize Parameters: Initialize weights and biases for the RNN layers, including matrices for input-to-hidden and hidden-to-hidden transitions, and biases.
3. Prepare Data: Load and preprocess training data (e.g., dinosaur names), converting text into integer indices and creating input-output pairs for the RNN.
4. Training Loop:

* Forward Propagation: Compute the loss by passing input data through the RNN and using the cached values for backpropagation.
* Backward Propagation: Calculate gradients of the loss with respect to the parameters and clip them to prevent explosion.
* Update Parameters: Adjust the weights and biases using gradient descent based on the computed gradients.

5) Check Progress: Every few iterations, generate sample outputs to assess the quality of text generation and adjust hyperparameters as needed.

**GitHub Link:**

**https://github.com/amruthaa-m/DL-Lab1/tree/main/Unit-3/Lab8**