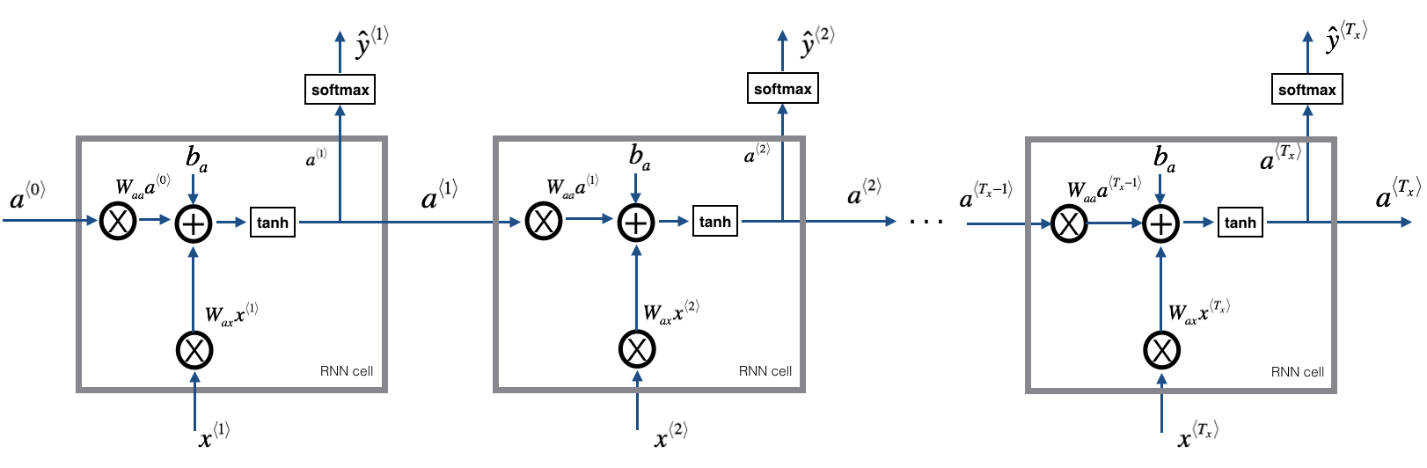
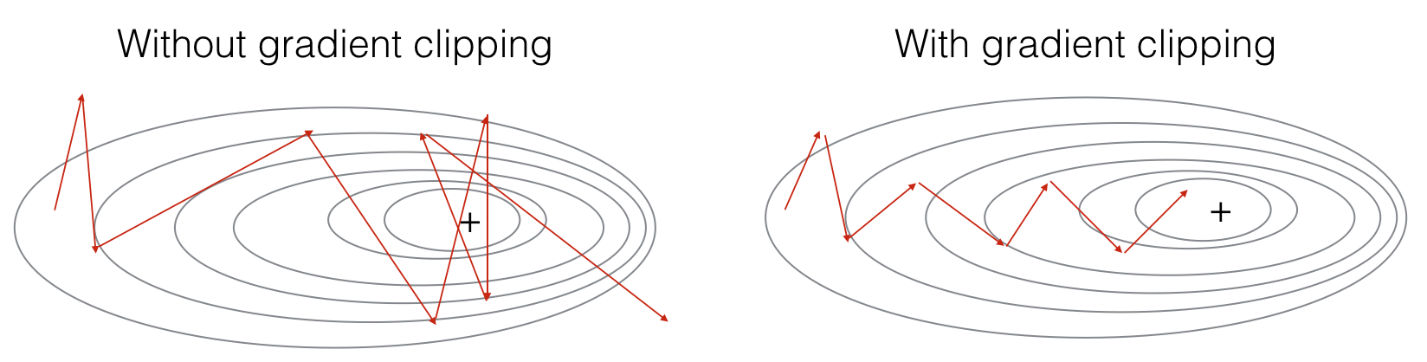
|  |  |
| --- | --- |
| **Ex No: 8**  **Date: 25/09/2024** | **Lab Record: Recurrent Neural Networks (RNNs)** |

**Objective:**

The objective of this lab is to implement and understand a character-level language model using a Recurrent Neural Network (RNN). Specifically, the goal is to generate sequences of characters, such as dinosaur names, based on a dataset of existing names. The model will be trained using stochastic gradient descent, with gradient clipping to prevent exploding gradients.





**Description:**

The lab focuses on implementing key components of a character-level RNN for text generation. We will:

1. Load and preprocess the dataset.
2. Implement gradient clipping to manage exploding gradients.
3. Train the model using backpropagation through time.
4. Sample characters from the RNN to generate new names.

**Code Line by Line Explanation:**

**1. Importing Required Libraries**

- NumPy is used for array and matrix operations.

- utils contains helper functions like `rnn\_forward` and `rnn\_backward`.

- random helps in generating random numbers, which are used for shuffling.

- pprint pretty prints dictionaries for easier viewing.

**2. Dataset and Preprocessing**

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

chars = sorted(list(set(data)))

char\_to\_ix = { ch:i for i,ch in enumerate(chars) }

ix\_to\_char = { i:ch for i,ch in enumerate(chars) }

- Data loading: Reads dinosaur names from `dinos.txt` and converts all characters to lowercase.

- Unique characters: `chars` contains all unique characters in the dataset.

- Mappings: `char\_to\_ix` maps each character to an index, while `ix\_to\_char` reverses the mapping.

**3. Gradient Clipping**: Essential to prevent exploding gradients during training.

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

- Clipping: The function limits gradients to a specified range, ensuring no gradient exceeds `maxValue` or drops below `-maxValue`.

**4. Sampling Function:** Generates a sequence of characters from the RNN’s learned probability distribution.

def sample(parameters, char\_to\_ix, seed):

Waa, Wax, Wya, b, by = parameters['Waa'], parameters['Wax'], parameters['Wya'], parameters['b'], parameters['by']

vocab\_size = by.shape[0]

n\_a = Waa.shape[1]

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

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

indices = []

idx = -1

counter = 0

newline\_character = char\_to\_ix['\n']

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

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

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

y = softmax(z)

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

indices.append(idx)

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

x[idx] = 1

a\_prev = a

counter += 1

if (counter == 50):

indices.append(newline\_character)

return indices

- Input: Takes RNN parameters and a seed value for reproducibility.

- Process: The function runs a loop to sample characters from the learned probability distribution, stopping when a newline character is generated or after 50 characters.

- Output: Returns a list of indices corresponding to characters.

**5. Gradient Descent Step:** The `optimize` function performs one iteration of forward and backward propagation, clipping gradients, and updating parameters.

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

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

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

gradients = clip(gradients, 5)

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

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

- Forward Propagation: Computes the RNN output and loss.

- Backward Propagation: Calculates gradients using backpropagation through time.

- Clipping: Ensures the gradients do not explode.

- Parameter Update: Uses the gradients to update the RNN’s parameters via gradient descent.

**6. Model Training:** The core of the model training involves running several iterations, processing input sequences (dinosaur names), and generating predictions.

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

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

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

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

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

examples = f.readlines()

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

np.random.shuffle(examples)

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

for j in range(num\_iterations):

idx = j % len(examples)

single\_example = examples[idx]

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

X = [None] + single\_example\_ix

Y = X[1:] + [char\_to\_ix['\n']]

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

loss = smooth(loss, curr\_loss)

if j % 2000 == 0:

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

for name in range(dino\_names):

sampled\_indices = sample(parameters, char\_to\_ix, seed=0)

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

return parameters

- Inputs: The function takes the data, character mappings, number of iterations, hidden units (`n\_a`), and vocabulary size.

- Training loop:

- For each iteration, the RNN processes one name from the shuffled dataset.

- A single name is broken into input and target sequences.

- The `optimize()` function computes the loss and updates the model parameters.

- Loss Smoothing: Helps stabilize training.

- Checkpoint: Every 2000 iterations, the model generates samples of dinosaur names, providing insight into training progress.

**7. Sample Output:** As the model trains, the generated names become more coherent. Early iterations might produce random characters, but later stages should generate realistic names like:

This RNN-based character-level language model learns to generate dinosaur names by predicting sequences of characters. By leveraging gradient descent and sampling, the model iteratively improves, generating more plausible names over time. This approach can be adapted for other text generation tasks such as music, poetry, or even code completion.

**GitHub Link:** https://github.com/dyuthiramesh/Deep\_Learning\_Elective/blob/main/Sem5/Lab8/RNN\_lab.ipynb