|  |  |
| --- | --- |
| **Ex No: 9**  **Date: 25-09-2024** | **Character level language model - Name generation** |

**Objective:**

The objective of this lab is to build a character-level recurrent neural network (RNN) model to generate new names by learning patterns from a dataset of dinosaur names. The model will predict the next character in a sequence, allowing for the generation of realistic, new names by synthesizing learned patterns.

**Description:**

In this lab, we implement a character-level language model using an RNN to generate text sequences, specifically dinosaur names. By training the model on a small dataset of names, the RNN learns the structure and patterns in the dataset. During training, the model performs forward and backward propagation through time, computes the gradients, clips them to avoid the problem of exploding gradients, and updates the weights using gradient descent. We then sample the model to generate names at various stages of the training process to evaluate its performance.

**Model Architecture:**

1. **Input Layer:** Takes a sequence of integers representing characters (one-hot encoded) from the dataset.
2. **Recurrent Layer:** An RNN with hidden state a that takes the previous state and the current character to compute the next state. This helps the model learn character dependencies in sequences.
3. **Output Layer:** A softmax layer that generates a probability distribution over the possible next characters.
4. **Loss Function:** Cross-entropy loss between the predicted character and the actual character.
5. **Backpropagation through Time (BPTT):** Used to compute gradients of the loss with respect to the model parameters over time steps.
6. **Gradient Clipping:** Applied to prevent exploding gradients by ensuring gradients are within a certain range.
7. **Optimization:** Stochastic Gradient Descent (SGD) is used to update the model parameters after clipping the gradients.
8. **Sampling Mechanism:** The model samples characters iteratively to generate new names based on learned probabilities.

**Step-by-Step Explanation of Implementation:**

1. Loading and Preprocessing Data

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))

chars = sorted(chars)

print(chars)

* **data**: Loads the text file containing dinosaur names and converts all characters to lowercase to simplify processing (as case sensitivity is not required).
* **chars**: A list of all unique characters present in the data (including newline characters, spaces, etc.).
* **data\_size, vocab\_size**: The total number of characters in the data and the number of unique characters.

1. Character-to-Index and Index-to-Character Mapping

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

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

pp = pprint.PrettyPrinter(indent=4)

pp.pprint(ix\_to\_char)

* **char\_to\_ix**: Maps each character to a unique integer (its index).
* **ix\_to\_char**: Maps indices back to their corresponding characters.

These mappings are necessary for converting characters to numerical representations (indices) and vice versa when training and generating text.

1. Clipping Function

def clip(gradients, maxValue):

dWaa, dWax, dWya, db, dby = gradients['dWaa'], gradients['dWax'], gradients['dWya'], gradients['db'], gradients['dby']

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

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

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

return gradients

* This function clips the values of the gradients between -maxValue and maxValue to mitigate the issue of **exploding gradients** in RNN training. Exploding gradients can cause instability during training by producing excessively large updates to parameters.
* **np.clip()**: Limits the values of each gradient to lie between -maxValue and maxValue.

1. Testing the Clipping Function

mValue = 10

np.random.seed(3)

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

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

gradients = clip(gradients, mValue)

print(gradients)

Random gradient matrices are generated using np.random.randn(), scaled by a factor of 10, and then clipped with the clip() function. The clipped values are printed to verify that clipping is correctly applied.

1. Matrix and Vector Operations

matrix1 = np.array([[1,1],[2,2],[3,3]])

matrix2 = np.array([[0],[0],[0]])

vector1D = np.array([1,1])

vector2D = np.array([[1],[1]])

print(np.dot(matrix1,vector1D))

print(np.dot(matrix1,vector2D))

This part shows matrix-vector multiplication and explores the differences between 1D and 2D arrays when performing dot products. The code highlights **broadcasting** behavior in NumPy, which happens when adding arrays of different shapes.

1. Sampling Function

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

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

vocab\_size = by.shape[0]

n\_a = Waa.shape[1]

x = np.zeros((vocab\_size, 1)) # Initial input is a one-hot encoded vector

a\_prev = np.zeros((n\_a, 1)) # Initial hidden state is zero

indices = [] # List to store sampled 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) # Hidden state update

z = np.dot(Wya, a) + by # Output layer pre-activation

y = softmax(z) # Softmax gives probabilities for each character

np.random.seed(counter+seed) # Set seed for reproducibility

idx = np.random.choice(list(range(vocab\_size)), p=y.ravel()) # Sample index from probability distribution

indices.append(idx) # Add sampled index to list

x = np.zeros((vocab\_size, 1)) # Create one-hot vector for the sampled character

x[idx] = 1 # Set sampled index to 1

a\_prev = a # Update hidden state for the next timestep

seed += 1

counter += 1

if (counter == 50): # If max length is reached, append newline

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

return indices

1. Testing the Sampling Function

np.random.seed(2)

Wax, Waa, Wya = np.random.randn(n\_a, vocab\_size), np.random.randn(n\_a, n\_a), np.random.randn(vocab\_size, n\_a)

b, by = np.random.randn(n\_a, 1), np.random.randn(vocab\_size, 1)

parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "b": b, "by": by}

indices = sample(parameters, char\_to\_ix, 0)

print("list of sampled characters:\n", [ix\_to\_char[i] for i in indices])

**Testing**: The sampling function is tested using randomly initialized parameters. It samples a list of character indices and converts them back to characters using ix\_to\_char.

1. Forward and backward propagation

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

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

The function rnn\_forward executes the forward propagation for the entire sequence X. It computes the hidden states and the loss for the current example X and Y. The forward pass also stores intermediate results in cache, which will be needed for backpropagation.

Next, the rnn\_backward function computes the gradients of the loss with respect to the parameters by backpropagating through time. These gradients will be used to update the model’s parameters.

1. 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):

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.seed(0)

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\_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

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

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

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

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")

loss = smooth(loss, curr\_loss)

if j % 2000 == 0:

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

seed = 0

for name in range(dino\_names):

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

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

seed += 1

print('\n')

return parameters

 **Initialization:**

* Sets input/output sizes (n\_x, n\_y) based on the vocabulary size.
* Initializes model parameters and initial loss.

 **Data Preparation:**

* Loads dinosaur names from dinos.txt, converts them to lowercase, and shuffles them.

 **Training Loop:**

* Iterates for a specified number of training steps (num\_iterations):
  + **Index Selection:** Cycles through the training examples.
  + **Input/Output Preparation:** Encodes the current name into indices for the RNN input (X) and output (Y).
  + **Optimization:** Performs forward and backward propagation using the optimize function to update model parameters and calculate loss.
  + **Loss Smoothing:** Smooths the loss for stability.
  + **Sampling:** Every 2000 iterations, samples and prints generated dinosaur names.

 **Return:**

* Returns the trained model parameters after completing all iterations.

**Conclusion:**

The dinosaur name generation project effectively utilizes an LSTM model to learn and generate unique dinosaur names from a dataset. Through iterative training, the model demonstrates improvement by decreasing loss and producing contextually relevant names. Techniques like gradient clipping and loss smoothing enhance training stability, showcasing the potential of RNNs in natural language processing. This project not only deepens understanding of text generation but also paves the way for exploring more complex applications in AI-driven creative fields.

**Github Link:**

**https://github.com/Bhargava-Srinivasan-26/Deep\_learning\_elective/tree/dae0c20e72279c2e20ab55ec626f24939f5780e8/Unit%203/Lab%208**