Character level language model - Dinosaurus Island¶

Welcome to Dinosaurus Island! 65 million years ago, dinosaurs existed, and in this assignment, they have returned.

The special task: Leading biology researchers are creating new breeds of dinosaurs and bringing them to life on earth, and the distributed job is to give names to these dinosaurs. If a dinosaur does not like its name, it might go berserk, so choose wisely!

- Store text data for processing using an RNN
- Build a character-level text generation model using an RNN
- Sample novel sequences in an RNN
- Explain the vanishing/exploding gradient problem in RNNs
- Apply gradient clipping as a solution for exploding gradients

1 - Problem Statement

```
import numpy as np
from utils import *
import random
import pprint
import copy
```

```
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)
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)
```

2 - Building Blocks of the Model

```
def clip(gradients, maxValue):
    '''
    Clips the gradients' values between minimum and maximum.

    Arguments:
    gradients -- a dictionary containing the gradients "dWaa", "dWax",
"dWya", "db", "dby"
    maxValue -- everything above this number is set to this number, and
everything less than -maxValue is set to -maxValue

    Returns:
    gradients -- a dictionary with the clipped gradients.
    '''
    gradients = copy.deepcopy(gradients)

    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

# Test with a max value of 10
def clip_test(target, mValue):
    print(f"\nGradients for mValue={mValue}")
    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}

    gradients2 = target(gradients, mValue)
    print("gradients[\"dWaa\"][1][2] =", gradients2["dWaa"][1][2])
    print("gradients[\"dWax\"][3][1] =", gradients2["dWax"][3][1])
    print("gradients[\"dWya\"][1][2] =", gradients2["dWya"][1][2])
    print("gradients[\"db\"][4] =", gradients2["db"][4])
    print("gradients[\"dby\"][1] =", gradients2["dby"][1])

    for grad in gradients2.keys():
        valuei = gradients[grad]
        valuef = gradients2[grad]
        mink = np.min(valuef)
```

```
        maxk = np.max(valuef)
        assert mink >= -abs(mValue), f"Problem with {grad}. Set a_min to
-mValue in the np.clip call"
        assert maxk <= abs(mValue), f"Problem with {grad}.Set a_max to
mValue in the np.clip call"
        index_not_clipped = np.logical_and(valuei <= mValue, valuei >=
-mValue)
        assert np.all(valuei[index_not_clipped] ==
valuef[index_not_clipped]), f" Problem with {grad}. Some values that
should not have changed, changed during the clipping process."

    print("\033[92mAll tests passed!\x1b[0m")

 clip_test(clip, 10)
clip_test(clip, 5)


# UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: sample

def sample(parameters, char_to_ix, seed):
    """
    Sample a sequence of characters according to a sequence of probability
distributions output of the RNN

    Arguments:
    parameters -- Python dictionary containing the parameters Waa, Wax,
Wya, by, and b.
    char_to_ix -- Python dictionary mapping each character to an index.
    seed -- Used for grading purposes. Do not worry about it.

    Returns:
    indices -- A list of length n containing the indices of the sampled
characters.
    """

    # Retrieve parameters and relevant shapes from "parameters" dictionary
    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]

    # Step 1: Create the a zero vector x that can be used as the one-hot
vector
    # Representing the first character (initializing the sequence
generation).
    x = np.zeros((vocab_size, 1))
```

```
    # Step 1': Initialize a_prev as zeros
    a_prev = np.zeros((n_a, 1))

    # Create an empty list of indices. This is the list which will contain
the list of indices of the characters to generate
    indices = []

    # idx is the index of the one-hot vector x that is set to 1
    # All other positions in x are zero.
    # Initialize idx to -1
    idx = -1

    # Loop over time-steps t. At each time-step:
    # Sample a character from a probability distribution
    # And append its index (`idx`) to the list "indices".
    # You'll stop if you reach 50 characters
    # (which should be very unlikely with a well-trained model).
    # Setting the maximum number of characters helps with debugging and
prevents infinite loops.
    counter = 0
    newline_character = char_to_ix['\n']

    while (idx != newline_character and counter != 50):

        # Step 2: Forward propagate x using the equations (1), (2) and (3)
        a = np.tanh(np.dot(Wax, x) + np.dot(Waa, a_prev) + b)
        z = np.dot(Wya, a) + by
        y = softmax(z)

        # For grading purposes
        np.random.seed(counter + seed)

        # Step 3: Sample the index of a character within the vocabulary
from the probability distribution y
        # (see additional hints above)
        idx = np.random.choice(list(range(vocab_size)), p = y.ravel())

        # Append the index to "indices"
        indices.append(idx)

        # Step 4: Overwrite the input x with one that corresponds to the
sampled index `idx`.
        # (see additional hints above)
        x = np.zeros((vocab_size, 1))
        x[idx] = 1
```

```
            # Update "a_prev" to be "a"
            a_prev = a

            # for grading purposes
            seed += 1
            counter +=1

        if (counter == 50):
            indices.append(char_to_ix['\n'])

        return indices

def sample_test(target):
    np.random.seed(24)
    _, n_a = 20, 100
    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 = target(parameters, char_to_ix, 0)
    print("Sampling:")
    print("list of sampled indices:\n", indices)
    print("list of sampled characters:\n", [ix_to_char[i] for i in
indices])

    assert len(indices) < 52, "Indices length must be smaller than 52"
    assert indices[-1] == char_to_ix['\n'], "All samples must end with
\\n"
    assert min(indices) >= 0 and max(indices) < len(char_to_ix), f"Sampled
indexes must be between 0 and len(char_to_ix)={len(char_to_ix)}"
    assert np.allclose(indices[0:6], [23, 16, 26, 26, 24, 3]), "Wrong
values"

    print("\033[92mAll tests passed!")
```

3 - Building the Language Model

```
# UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: optimize

def optimize(X, Y, a_prev, parameters, learning_rate = 0.01):
    """
    Execute one step of the optimization to train the model.
```

```
    Arguments:
    X -- list of integers, where each integer is a number that maps to a
character in the vocabulary.
    Y -- list of integers, exactly the same as X but shifted one index to
the left.
    a_prev -- previous hidden state.
    parameters -- python dictionary containing:
                        Wax -- Weight matrix multiplying the input, numpy
array of shape (n_a, n_x)
                        Waa -- Weight matrix multiplying the hidden state,
numpy array of shape (n_a, n_a)
                        Wya -- Weight matrix relating the hidden-state to
the output, numpy array of shape (n_y, n_a)
                        b --  Bias, numpy array of shape (n_a, 1)
                        by -- Bias relating the hidden-state to the
output, numpy array of shape (n_y, 1)
    learning_rate -- learning rate for the model.

    Returns:
    loss -- value of the loss function (cross-entropy)
    gradients -- python dictionary containing:
                        dWax -- Gradients of input-to-hidden weights, of
shape (n_a, n_x)
                        dWaa -- Gradients of hidden-to-hidden weights, of
shape (n_a, n_a)
                        dWya -- Gradients of hidden-to-output weights, of
shape (n_y, n_a)
                        db -- Gradients of bias vector, of shape (n_a, 1)
                        dby -- Gradients of output bias vector, of shape
(n_y, 1)
    a[len(X)-1] -- the last hidden state, of shape (n_a, 1)
    """

    ### START CODE HERE ###

    # Forward propagate through time (≈1 line)
    loss, cache = rnn_forward(X, Y, a_prev, parameters)

    # Backpropagate through time (≈1 line)
    gradients, a = rnn_backward(X, Y,  parameters, cache)

    # Clip your gradients between -5 (min) and 5 (max) (≈1 line)
    gradients = clip(gradients, 5)

    # Update parameters (≈1 line)
    parameters = update_parameters(parameters, gradients, learning_rate)
```

```
    ### END CODE HERE ###


    return loss, gradients, a[len(X)-1]
def optimize_test(target):
    np.random.seed(1)
    vocab_size, n_a = 27, 100
    a_prev = np.random.randn(n_a, 1)
    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}
    X = [12, 3, 5, 11, 22, 3]
    Y = [4, 14, 11, 22, 25, 26]
    old_parameters = copy.deepcopy(parameters)
    loss, gradients, a_last = target(X, Y, a_prev, parameters,
learning_rate = 0.01)
    print("Loss =", loss)
    print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
    print("np.argmax(gradients[\"dWax\"]) =",
np.argmax(gradients["dWax"]))
    print("gradients[\"dWya\"][1][2] =", gradients["dWya"][1][2])
    print("gradients[\"db\"][4] =", gradients["db"][4])
    print("gradients[\"dby\"][1] =", gradients["dby"][1])
    print("a_last[4] =", a_last[4])

    assert np.isclose(loss, 126.5039757), "Problems with the call of the
rnn_forward function"
    for grad in gradients.values():
        assert np.min(grad) >= -5, "Problems in the clip function call"
        assert np.max(grad) <= 5, "Problems in the clip function call"
    assert np.allclose(gradients['dWaa'][1, 2], 0.1947093), "Unexpected
gradients. Check the rnn_backward call"
    assert np.allclose(gradients['dWya'][1, 2], -0.007773876), "Unexpected
gradients. Check the rnn_backward call"
    assert not np.allclose(parameters['Wya'], old_parameters['Wya']),
"parameters were not updated"

    print("\033[92mAll tests passed!")
# UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: model

def model(data_x, ix_to_char, char_to_ix, num_iterations = 35000, n_a =
50, dino_names = 7, vocab_size = 27, verbose = False):
    """
    Trains the model and generates dinosaur names.
```

```
    Arguments:
    data_x -- text corpus, divided in words
    ix_to_char -- dictionary that maps the index to a character
    char_to_ix -- dictionary that maps a character to an index
    num_iterations -- number of iterations to train the model for
    n_a -- number of units of the RNN cell
    dino_names -- number of dinosaur names you want to sample at each
iteration.
    vocab_size -- number of unique characters found in the text (size of
the vocabulary)

    Returns:
    parameters -- learned parameters
    """

    # 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).
    examples = [x.strip() for x in data_x]

    # 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))

    # for grading purposes
    last_dino_name = "abc"

    # 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 = [char_to_ix[ch] for ch in single_example]
        single_example_ix = idx
        X = [None] + single_example_chars

        # Set the labels Y (see instructions above)
        ix_newline = [char_to_ix['\n']]
        Y = X[1:] + [char_to_ix['\n']]

        # 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")

        # to keep the loss smooth.
        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)
                last_dino_name = get_sample(sampled_indices, ix_to_char)
                print(last_dino_name.replace('\n', ''))

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

```
            print('\n')

    return parameters, last_dino_name

parameters, last_name = model(data.split("\n"), ix_to_char, char_to_ix,
22001, verbose = True)

assert last_name == 'Trodonosaurus\n', "Wrong expected output"
print("\033[92mAll tests passed!")
```

```
j =  0 idx =  0
single_example = turiasaurus
single_example_chars [20, 21, 18, 9, 1, 19, 1, 21, 18, 21, 19]
single_example_ix 0
 X =  [None, 20, 21, 18, 9, 1, 19, 1, 21, 18, 21, 19]
 Y =        [20, 21, 18, 9, 1, 19, 1, 21, 18, 21, 19, 0]

Iteration: 0, Loss: 23.087336

Nkzxwtdmfqoeyhsqwasjkjvu
Kneb
Kzxwtdmfqoeyhsqwasjkjvu
Neb
Zxwtdmfqoeyhsqwasjkjvu
Eb
Xwtdmfqoeyhsqwasjkjvu


j =  1535 idx =  1535
j =  1536 idx =  0
Iteration: 2000, Loss: 27.884160

Liusskeomnolxeros
Hmdaairus
Hytroligoraurus
Lecalosapaus
Xusicikoraurus
Abalpsamantisaurus
Tpraneronxeros


Iteration: 4000, Loss: 25.901815

Mivrosaurus
```

Inee
Ivtroplisaurus
Mbaaisaurus
Wusichisaurus
Cabaselachus
Toraperlethosdarenitochusthiamamumamaon

Iteration: 6000, Loss: 24.608779

Onwusceomosaurus
Lieeaerosaurus
Lxussaurus
Oma
Xusteonosaurus
Eeahosaurus
Toreonosaurus

Iteration: 8000, Loss: 24.070350

Onxusichepriuon
Kilabersaurus
Lutrodon
Omaaerosaurus
Xutrcheps
Edaksoje
Trodiktonus

Iteration: 10000, Loss: 23.844446

Onyusaurus
Klecalosaurus
Lustodon
Ola
Xusodonia
Eeaeosaurus
Troceosaurus

Iteration: 12000, Loss: 23.291971

Onyxosaurus
Kica
Lustrepiosaurus
Olaagrraiansaurus

Yuspangosaurus
Eealosaurus
Trognesaurus

Iteration: 14000, Loss: 23.382338

Meutromodromurus
Inda
Iutroinatorsaurus
Maca
Yusteratoptititan
Ca
Troclosaurus

Iteration: 16000, Loss: 23.224544

Meusspanchodtashuarhiaspantaxia
Indaa
Iuspsauhosaurus
Macacosaurus
Yusoconikaulrit
Cacasoceimurus
Trrasaurus

Iteration: 18000, Loss: 22.904954

Pivrrong
Llecanosaurus
Myssocilindus
Peeaishidanagtallsaurus
Ytrong
Eg
Trojichus

Iteration: 20000, Loss: 23.005394

Nkytrohelosaurus
Lolaagosaurus
Lyusochosaurus
Necakson
Yussangosaurus
Eiagosaurus
Trodon

```
Iteration: 22000, Loss: 22.728886

Onustreofkelus
Llecagosaurus
Mystolojmiaterltasaurus
Ola
Yuskeolongus
Eiacosaurus
Trodonosaurus


All tests passed!
```

Expected output

```
...
Iteration: 22000, Loss: 22.728886

Onustreofkelus
Llecagosaurus
Mystolojmiaterltasaurus
Ola
Yuskeolongus
Eiacosaurus

   Trodonosaurus
```