LHW 1 : Convolutional Neural Networks for text classification

In this homework, you will be implementing the *forward pass*, *backpropagation*, and *gradient checking* for a convolutional neural network with sparse inputs for text classification.

The setup

Let's define parameters for the Convolutional Neural Network. You do not need to modify them.

[1]
```
import sys
import math
import numpy as np
import pickle

# window size for the CNN
width = 2

# number of filters
F = 100

# learning rate
alpha = 1e-1

# vocabsize: size of the total vocabulary
vocabsize = 10000

# vocab: the vocabulary dictionary with the word as key and its index as
# the input will be transformed into respective positional indices using
# as the input for the forward and backward algorithm
# e.g. if vocab = {'a': 0, 'simple': 1, 'sentence': 2} and the training d
# "a simple simple sentence a",
# the input to the forward and backward algorithm will be [0,1,1,2,0]
vocab = {}

np.random.seed(1)

# U and V are weight vectors of the hidden layer
# U: a matrix of weights of all inputs for the first
# hidden layer for all F filters in the
# where each filter has the size of vocabsize by width (window size)
```

```
# U[i, j, k] represents the weight of filter u_j
# for word with vocab[word] = i when the word is
# at the position k of the sliding window
# e.g. for the example, "a simple simple sentence a",
# if the window size is 4 and we are looking at the first sliding window
# of the 9th filter, the weight for the last "sentence" will be U[2, 8, 3
# i.e U[index of the word in vocab, index of the filter, position of the
U = np.random.normal(loc=0, scale=0.01, size=(vocabsize, F, width))

# V: the the weight vector of the F filter outputs (after max pooling)
# that will produce the output, i.e. o = sigmoid(V*h)
V = np.random.normal(loc=0, scale=0.01, size=(F))
```

Let's define some utility functions that may be useful. You don't need to modify them.

[2]

```
def sigmoid(x):
    """
    helper function that computes the sigmoid function
    """
    return 1. / (1 + math.exp(-x))


def read_vocab(filename):
    """
    helper function that builds up the vocab dictionary for input transfo
    """
    file = open(filename)
    for line in file:
        cols = line.rstrip().split("\t")
        word = cols[0]
        idd = int(cols[1])
        vocab[word] = idd
    file.close()


def read_data(filename):
    """
    :param filename: the name of the file
    :return: list of tuple ([word index list], label)
    as input for the forward and backward function
    """
    data = []
    file = open(filename)
    for line in file:
        cols = line.rstrip().split("\t")
        label = int(cols[0])
        words = cols[1].split(" ")
        w_int = []
```

```
            for w in words:
                # skip the unknown words
                if w in vocab:
                    w_int.append(vocab[w])
            data.append((w_int, label))
        file.close()
        return data


    def train():
        """
        main caller function that reads in the names of the files
        and train the CNN to classify movie reviews
        """
        vocabFile = "vocab.txt"
        trainingFile = "movie_reviews.train"
        testFile = "movie_reviews.dev"

        read_vocab(vocabFile)
        training_data = read_data(trainingFile)
        test_data = read_data(testFile)

        for i in range(50):
            # confusion matrix showing the accuracy of the algorithm
            confusion_training = np.zeros((2, 2))
            confusion_validation = np.zeros((2, 2))

            for (data, label) in training_data:
                # back propagation to update weights for both U and V
                backward(data, label)

                # calculate forward and evaluate
                prob = forward(data)["prob"]
                pred = 1 if prob > .5 else 0
                confusion_training[pred, label] += 1

            for (data, label) in test_data:
                # calculate forward and evaluate
                prob = forward(data)["prob"]
                pred = 1 if prob > .5 else 0
                confusion_validation[pred, label] += 1

            print("Epoch: {}\tTrain accuracy: {:.3f}\tDev accuracy: {:.3f}"
                .format(
                i,
                np.sum(np.diag(confusion_training)) / np.sum(confusion_traini
                np.sum(np.diag(confusion_validation)) / np.sum(confusion_vali
```

1. Forward

Given the parameters and definition of the CNN model (§2 of HW), complete the Forward Function to calculate *o* (the probability of the positive class) for an input text. You may not import any additional libraries.

[3]
```
def forward(word_indices):
    """
    :param word_indices: a list of word indices, i.e. idx = vocab[word]
    :return: a result dictionary containing 3 items -
    result['prob']: output of the CNN algorithm.
    result['h']: the hidden layer output after max pooling, h = [h1, ...,
    result['hid']: argmax of F filters, e.g. j of x_j
    e.g. for the ith filter u_i, tanh(word[hid[i], hid[i] + width]*u_i) =
    """

    h = np.zeros(F, dtype=float)
    hid = np.zeros(F, dtype=int)
    prob = 0.0
    # treating word_indices as telling us which entry of each w_i has "1"

    # step 1. compute h and hid
    # loop through the input data of word indices and
    # keep track of the max filtered value h_i and its position index x_j
    # h_i = max(tanh(weighted sum of all words in a given window)) over a

    """
    Type your code below
    """

    for i in range(F):
        """
        for a filter i, we compute corresponding h_i and hid_i
        first, we calculate the sum of weights corresponding to words in
        """
        weighted_sum = 0
        for k in range(width):
            """
            we add the weight corresponding to the (k+1)th word in the fi
            """
            weighted_sum += U[word_indices[k], i, k]
        """
        then we find tanh(sum of weights of words in the first window)
        and we assume hid[i] = 0
        """
        h[i] = math.tanh(weighted_sum)
        hid[i] = 0
        for j in range(len(word_indices) - width):
            """
            Now we go through each window x_j
            and calculate the sum of the weights of words in this window
            """
            weighted_sum = 0
```

```
                for k in range(width):
                    weighted_sum += U[word_indices[j + k], i, k]
                """
                We compare the tanh of this sum to the "current" h[i]
                """
                if math.tanh(weighted_sum) > h[i]:
                    h[i] = math.tanh(weighted_sum)
                    hid[i] = j
        # step 2. compute probability
    # once h and hid are computed, compute the probabiliy by sigmoid(h^TV
    """
    Type your code below
    """
    prob = sigmoid(np.dot(V, h))

    # step 3. return result
    return {"prob": prob, "h": h, "hid": hid}
```

2. Backward

Using the gradient update equations for V (§3 in HW) and U (§3.1), implement the updates for U and V in the backward function.

[12]
```
def backward(word_indices, true_label):
    """
    :param word_indices: a list of word indices, i.e. idx = vocab[word]
    :param true_label: true label (0, 1) of the movie reviews
    :return: None
    update weight matrix/vector U and V based on the loss function
    """
    global U, V
    pred = forward(word_indices)
    prob = pred["prob"]
    h = pred["h"]
    hid = pred["hid"]
    y = true_label

    # update U and V here
    # loss_function = y * log(o) + (1 - y) * log(1 - o)
    #               = true_label * log(prob) + (1 - true_label) * log(1 -
    # to update V: V_new = V_current + d(loss_function)/d(V)*alpha
    # to update U: U_new = U_current + d(loss_function)/d(U)*alpha
    # Make sure you only update the appropriate argmax term for U
    for i in range(F):
        for j in range(width):
            U[word_indices[hid[i]+j], i, j] = U[word_indices[hid[i]+j], i
```

```
    V = V + y*h-prob*h

    """
    Type your code below
    """
```

3. Gradient Checking

Now that you have implemented the forward and backward function, you are going to check the correctness of the implementation by calculating numerical gradients and comparing them with the analytical values. Refer to §4 in HW.

Implement the functions that calculate numerical gradients for V and U.

[13]
```
def calc_numerical_gradients_V(V, word_indices, true_label):
    """
    :param true_label: true label of the data
    :param V: weight vector of V
    :param word_indices: a list of word indices, i.e. idx = vocab[word]
    :return V_grad:
    V_grad =    a vector of size length(V) where V_grad[i] is the numeric
                gradient approximation of V[i]
    """
    # you might find the following variables useful
    x = word_indices
    y = true_label
    eps = 1e-4

    V_grad = np.zeros(F, dtype=float)

    """
    Type your code below
    """

    pred = forward(word_indices)
    prob = pred["prob"]
    h = pred["h"]
    hid = pred["hid"]

    for i in range(len(V_grad)):
        V_plus = V
        V_minus = V
        V_plus[i] = V[i] + eps
        V_minus[i] = V[i] - eps
        prob_plus = sigmoid(np.dot(V_plus, h))
        prob_minus = sigmoid(np.dot(V_minus, h))
        V_grad[i] = (y * math.log(prob_plus) + (1 - y) * math.log(1 - pro
                     - y * math.log(prob_minus) - (1 - y) * math.log(1 -
```

```
    return V_grad


def calc_numerical_gradients_U(U, word_indices, true_label):
    """
    :param U: weight matrix of U
    :param word_indices: a list of word indices, i.e. idx = vocab[word]
    :param true_label: true label of the data
    :return U_grad:
    U_grad =    a matrix of dimension F*width where U_grad[i, j] is the n
                approximation of the gradient for the argmax of
                each filter i at offset position j
    """
    # you might find the following variables useful
    x = word_indices
    y = true_label
    eps = 1e-4

    pred = forward(x)
    prob = pred["prob"]
    h = pred["h"]
    hid = pred["hid"]
    U_grad = np.zeros((F, width))

    """
    Type your code below
    """
    """" Need to change each h_i to reflect adding/subtracting epsilon to
    """"notice that for a fixed i, with each j, we get the same change in
    because we are simply adding epsilon each time to the same sum"""
    for i in range(F):
        for j in range(width):
            sum = 0
            for k in range(width):
                sum += U[x[hid[i] + k], i, k]
            h_i_plus = math.tanh(sum + eps)
            h_i_minus = math.tanh(sum - eps)
            h_plus = h
            h_minus = h
            h_plus[i] = h_i_plus
            h_minus[i] = h_i_minus
            prob_plus = sigmoid(np.dot(V, h_plus))
            prob_minus = sigmoid(np.dot(V, h_minus))
            U_grad[i, j] = (y * math.log(prob_plus) + (1 - y) * math.log(
                            - y * math.log(prob_minus) - (1 - y) * math.l

    return U_grad
```

Now that we have functions to calculate gradients, implement the check function to compare the numerical gradients with their respective analytical values. Be sure to update the analytical and numerical gradients below using the functions we wrote above.

[16]
```
def check_gradient():
    """
    :return (diff in V, diff in U)
    Calculate numerical gradient approximations for U, V and
    compare them with the analytical values
    check gradient accuracy; for more details, cf.
    http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanc
    """
    x = []
    for i in range(100):
        x.append(np.random.randint(vocabsize))
    y = 1

    pred = forward(x)
    prob = pred["prob"]
    h = pred["h"]
    hid = pred["hid"]

    """
    Update 0s below with your calculations
    """
    # check V
    # compute analytical and numerical gradients and compare their differ
    ana_grad_V = y * h - prob * h  # <-- Update
    numerical_grad_V = calc_numerical_gradients_V(V, x, y)  # <-- Update
    sum_V_diff = sum((numerical_grad_V - ana_grad_V) ** 2)

    # check U
    # compute analytical and numerical gradients and compare their differ

    ana_grad_U = np.zeros((F, width))  # <-- Update
    for i in range(F):
        for j in range(width):
            ana_grad_U[i, j] = (y - prob) * V[i] * (1 - h[i] * h[i])
    numerical_grad_U = calc_numerical_gradients_U(U, x, y)  # <-- Update
    sum_U_diff = sum(sum((numerical_grad_U - ana_grad_U) ** 2))
    print("V difference: {:.8f}, U difference: {:.8f} (these should be cl
          .format(sum_V_diff, sum_U_diff))
```

Result

Let's check the difference between the numerical gradients and the analytical gradients using the function completed above. Report the numbers in the writeup.

[17]
```
check_gradient()
```

```
V difference: 0.06165559, U difference: 0.30760193 (these should be close
to 0)
```

[9]
```

```

[ ]
```

```