CS5242_Assignment_2

March 21, 2018

CHANGE LOG

20 March 2018: - change one testing code in RNN forward section from `h0=np.random.uniform(size=(N, H))` to `h0=np.random.uniform(size=(H,))`, because the shape of RNN initial state `h0` shall not be dependent on the batch size. - correct and clean up some comments in `rnn_layers.py`.

21 March 2018: - strengthen the function `rel_error(x, y)` to check whether both `x` and `y` have `NaN` or not in corresponding position, which will help you debug the codes. **Since the later parts of this assignment are based on the former parts, make sure your implementation fully pass the current test case and then proceed to the next test case.** - improve the keras part in the forward section of Bidirectional RNN, so that its output is padded by `NaN` as expected. - change `datasets.py` to always rebuild dictionary.

1 Introduction

ASSIGNMENT DEADLINE: 2 APRIL 2018 (MON) 17:00PM

In this assignment we will be coding the building blocks for the Recurrent Neural Network (RNN) in `rnn_layers.py` and putting them together to train a RNN on sentiment analysis.

Attention: Only python3 will be allowed to use in this assignment. And we use numpy to store and caculate data and parameters. You do not need a GPU for this assignment. CPU is enough. To run this Jupyter notebook, you need to install the depedent libraries in requiremets.txt via pip (or pip3). Note: If you don't implement all the codes, running the codes might occur some errors.

For each layer we will implement a forward and a backward function. The forward function will receive inputs and will return the outputs of this layer(loss layer will be a little different), and the backward pass will receive upstream derivatives and inputs and will return gradients with respect to the inputs. Gradients for weights or bias will be stored in parameters in this layer:

```python
class SomeLayer(Layer):
    # some layer type inherited from Layer class
    def __init__(self, params):
        # set up specific layer parameters
        # initialize variables for the layer weights
        # initialize variables for storing the gradients
        # initialize other necessary variables
    def forward(self, inputs):
        # Receive inputs and return output
        # Do some computations ...
        z = # ... some intermediate value
        # Do some more computations ...
        outputs = # the outputs
        return outputs
    def backward(self, in_grads, inputs):
        # Receive derivative of loss with respect to outputs,
        # and compute derivative with respect to inputs.
        # Use values in cache to compute derivatives
        out_grads = # Derivative of loss with respect to inputs
        self.w_grad = # Derivative of loss with respect to self.weights
        return out_grads
```

After implementing a bunch of layers (i.e. `RNN Cell`, `RNN`, `Bidirectional RNN`) in this way, we will be able to easily combine them to build classifiers for various applications whose input are sequential data (e.g. Sentiment Analysis).

This iPython notebook serves to: - explain the questions - explain the function APIs and implementation examples - provide helper functions to piece functions together and check your code

2 RNN Cell Layer

RNN cell is the basic building block of RNN, which implements the specific operation at each time step of RNN. It has an hidden states of dimension `H` and accepts inputs of dimension `D`. In this assignment, you are required to implement a simple type of RNN cell, formulated as follows:

$$y = tanh(Wx + Uh + b),$$

where `x` and `h` are the inputs and hidden states respectively, and `W`, `U` and `b` are trainable kernel, recurrent_kernel and bias respectively.

2.1 Forward

Please implement the function `RNNCell.forward(self, inputs)` and test your implementation using the following code. (`inputs` is a list of two numpy arrays, `[x, h]`).

```
In [2]: import numpy as np
        import keras
        from keras import layers
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell
        from utils.tools import rel_error

        N, D, H = 3, 10, 4
        x = np.random.uniform(size=(N, D))
        prev_h = np.random.uniform(size=(N, H))

        rnn_cell = RNNCell(in_features=D, units=H)
        out = rnn_cell.forward([x, prev_h])
        # compare with the keras implementation
        keras_x = layers.Input(shape=(1, D), name='x')
        keras_prev_h = layers.Input(shape=(H,), name='prev_h')
        keras_rnn = layers.RNN(layers.SimpleRNNCell(H),
                               name='rnn')(keras_x, initial_state=keras_prev_h)
        keras_model = keras.Model(inputs=[keras_x, keras_prev_h],
                                  outputs=keras_rnn)
        keras_model.get_layer('rnn').set_weights([rnn_cell.kernel,
                                                  rnn_cell.recurrent_kernel,
                                                  rnn_cell.bias])
        keras_out = keras_model.predict_on_batch([x[:, None, :], prev_h])

        print('Relative error (<1e-5 will be fine): {}'.format(rel_error(keras_out, out)))
```

```
Relative error (<1e-5 will be fine): 7.925405100795174e-08
```

2.2 Backward

Please implement the function `RNNCell.backward(self, in_grads, inputs)` and test your implementation using the following code. You need to compute the gradients to both the inputs and hidden states, as well as those trainable weights.

```
In [3]: import numpy as np
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell
        from utils.check_grads import check_grads_layer

        N, D, H = 3, 10, 4
        x = np.random.uniform(size=(N, D))
        prev_h = np.random.uniform(size=(N, H))
        in_grads = np.random.uniform(size=(N, H))

        rnn_cell = RNNCell(in_features=D, units=H)
        check_grads_layer(rnn_cell, [x, prev_h], in_grads)
```

```
<1e-8 will be fine
Gradients to inputs 0: 2.5639354037879172e-11
Gradients to inputs 1: 9.38322890811756e-12
Gradients to -:rnn_cell/kernel: 1.9758904801264367e-11
Gradients to -:rnn_cell/recurrent_kernel: 1.3650539132781635e-11
Gradients to -:rnn_cell/bias: 3.020072109479231e-11
```

Then please improve your implementation of RNN cell so that it can properly handle `NaN` input, and test it with the following code. **The gradients to those `NaN` input units are supposed to be zeros.**

```
In [4]: import numpy as np
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell
        from utils.check_grads import check_grads_layer

        N, D, H = 3, 10, 4
        x = np.random.uniform(size=(N, D))
        # set part of input to NaN
        # this situation will be encountered in the following work
        x[1:, :] = np.nan
        prev_h = np.random.uniform(size=(N, H))
        in_grads = np.random.uniform(size=(N, H))

        rnn_cell = RNNCell(in_features=D, units=H)
        check_grads_layer(rnn_cell, [x, prev_h], in_grads)
```

```
<1e-8 will be fine
Gradients to inputs 0: 1.841448845509418e-11
Gradients to inputs 1: 9.257412008245938e-12
Gradients to -:rnn_cell/kernel: 1.608549362557875e-11
Gradients to -:rnn_cell/recurrent_kernel: 2.2382520516256812e-11
Gradients to -:rnn_cell/bias: 2.8963646612775508e-11
```

3 RNN Layer

RNN layer wraps any type of RNN cell so that it can operate over a sequence of input data of different length. In particular, it runs a instance of RNN cell over the inputs, holds and updates the hidden states for the RNN cell. In this assignment, you are required to implement such a general RNN layer that is able to wrap your implemented RNN cell above.

3.1 Forward

Please implement the function `RNN.forward(self, inputs)` and test your implementation using the following code. Since NN layers generally proceed on a batch of data simultaneously, and for RNN, each input data may have different length, we define the input data format as an array of `(N, T, D)`, where `N` is the number of samples in a batch, `T` is the maximum length of input sequences, and `D` is the dimension of features at each time step. `NaN` is used to pad input sequences of different lenghts, so that the resulting length equals to `T`, e.g. `(x1, x2, ..., xk, NaN, NaN)`. **Hint: you can utilze `np.nan_to_num(x)` to easily convert NaNs to zeros in a numpy array, and `np.isnan(x)` to get binary mask indicating which elements are NaNs.**

```
In [5]: import numpy as np
        import keras
        from keras import layers
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell, RNN
        from utils.tools import rel_error

        N, T, D, H = 2, 3, 4, 5
        x = np.random.uniform(size=(N, T, D))
        x[0, -1:, :] = np.nan
        x[1, -2:, :] = np.nan
        h0 = np.random.uniform(size=(H,))

        rnn_cell = RNNCell(in_features=D, units=H)
        rnn = RNN(rnn_cell, h0=h0)
        out = rnn.forward(x)

        keras_x = layers.Input(shape=(T, D), name='x')
        keras_h0 = layers.Input(shape=(H,), name='h0')
        keras_rnn = layers.RNN(layers.SimpleRNNCell(H), return_sequences=True,
                               name='rnn')(keras_x, initial_state=keras_h0)
        keras_model = keras.Model(inputs=[keras_x, keras_h0],
                                  outputs=keras_rnn)
        keras_model.get_layer('rnn').set_weights([rnn.kernel,
                                                  rnn.recurrent_kernel,
                                                  rnn.bias])
        keras_out = keras_model.predict_on_batch([x, np.tile(h0, (N, 1))])

        print('Relative error (<1e-5 will be fine): {}'.format(rel_error(keras_out, out)))
```

```
Relative error (<1e-5 will be fine): 5.590215723424332e-08
```

3.2 Backward

Please implement the function `RNN.backward(self, in_grads, inputs)` and test your implementation using the following code (**note the internal gradients passed from next time steps**). Once again: the gradients to those `NaN` input units are supposed to be zeros

```
In [6]: import numpy as np
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell, RNN
        from utils.check_grads import check_grads_layer

        N, T, D, H = 2, 3, 4, 5
        x = np.random.uniform(size=(N, T, D))
        x[0, -1:, :] = np.nan
        x[1, -2:, :] = np.nan
        in_grads = np.random.uniform(size=(N, T, H))

        rnn_cell = RNNCell(in_features=D, units=H)
        rnn = RNN(rnn_cell)
        check_grads_layer(rnn, x, in_grads)
```

```
<1e-8 will be fine
Gradients to inputs: 9.530037985918532e-12
Gradients to -:rnn/kernel: 1.4097573404577487e-11
Gradients to -:rnn/recurrent_kernel: 7.11401260189477e-12
Gradients to -:rnn/bias: 3.198714096705757e-11
```

4 Bi-directional RNN Layer

Vallina RNN operates over input sequence in one direction, so it has limitations as the future input information cannot be reached from the current state. On the contrary, Bi-directional RNN addresses this shortcoming by operating the input sequence in both forward and backward directions.

Usually, Bi-directional RNN is implemented by running two independent RNNs in opposite direction of input data, and concatenating the outputs of the two RNNs. Since you have implemented RNN layer above, implementing Bi-directional RNN layer is not hard, which just requires certain manipulation of input data. A useful function that can reverse a batch of sequence data is provided for your easy implementation.

```
def _reverse_temporal_data(self, x, mask):
    num_nan = np.sum(~mask, axis=1)
    reversed_x = np.array(x[:, ::-1, :])
    for i in range(num_nan.size):
        reversed_x[i] = np.roll(reversed_x[i], x.shape[1]-num_nan[i], axis=0)
    return reversed_x
```

4.1 Forward

We provided the function `BidirectionalRNN.forward(self, inputs)` and the following code for testing. Note that `H` is the dimension of the hidden states of one internal RNN, so the actual dimension of the hidden states (or outputs) of Bidirectional RNN is `2*H`.

```
In [7]: import numpy as np
        import keras
        from keras import layers
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell, BidirectionalRNN
        from utils.tools import rel_error

        N, T, D, H = 2, 3, 4, 5
        x = np.random.uniform(size=(N, T, D))
        x[0, -1:, :] = np.nan
        x[1, -2:, :] = np.nan
        h0 = np.random.uniform(size=(N, H))
        hr = np.random.uniform(size=(N, H))

        rnn_cell = RNNCell(in_features=D, units=H)
        brnn = BidirectionalRNN(rnn_cell, h0=h0, hr=hr)
        out = brnn.forward(x)

        keras_x = layers.Input(shape=(T, D), name='x')
        keras_h0 = layers.Input(shape=(H,), name='h0')
        keras_hr = layers.Input(shape=(H,), name='hr')
        keras_x_masked = layers.Masking(mask_value=0.)(keras_x)
        keras_rnn = layers.RNN(layers.SimpleRNNCell(H), return_sequences=True)
        keras_brnn = layers.Bidirectional(keras_rnn, merge_mode='concat', name='brnn')(
                keras_x_masked, initial_state=[keras_h0, keras_hr])
        keras_model = keras.Model(inputs=[keras_x, keras_h0, keras_hr],
                                  outputs=keras_brnn)
        keras_model.get_layer('brnn').set_weights([brnn.forward_rnn.kernel,
                                                   brnn.forward_rnn.recurrent_kernel,
                                                   brnn.forward_rnn.bias,
                                                   brnn.backward_rnn.kernel,
                                                   brnn.backward_rnn.recurrent_kernel,
                                                   brnn.backward_rnn.bias])
        keras_out = keras_model.predict_on_batch([np.nan_to_num(x), h0, hr])
        nan_indices = np.where(np.any(np.isnan(x), axis=2))
        keras_out[nan_indices[0], nan_indices[1], :] = np.nan

        print('Relative error (<1e-5 will be fine): {}'.format(rel_error(keras_out, out)))
```

```
Relative error (<1e-5 will be fine): 8.834344845009141e-08
```

4.2 Backward

Please refer to the provided forward function and implement the function `BidirectionalRNN.backward(self, inputs)`. Test your implementation using the following code.

```
In [8]: import numpy as np
        import importlib
        import rnn_layers
        importlib.reload(rnn_layers)
        from rnn_layers import RNNCell, BidirectionalRNN
        from utils.check_grads import check_grads_layer

        N, T, D, H = 2, 3, 4, 5
```

```
x = np.random.uniform(size=(N, T, D))
x[0, -1:, :] = np.nan
x[1, -2:, :] = np.nan
in_grads = np.random.uniform(size=(N, T, H*2))

rnn_cell = RNNCell(in_features=D, units=H)
brnn = BidirectionalRNN(rnn_cell)
check_grads_layer(brnn, x, in_grads)
```

```
<1e-8 will be fine
Gradients to inputs: 8.579989149209735e-12
Gradients to -:brnn/forward_kernel: 1.824662864297008e-11
Gradients to -:brnn/forward_recurrent_kernel: 5.372414025314837e-12
Gradients to -:brnn/forward_bias: 3.4250111358958327e-11
Gradients to -:brnn/backward_kernel: 1.8062326207033073e-11
Gradients to -:brnn/backward_recurrent_kernel: 5.224566005210753e-12
Gradients to -:brnn/backward_bias: 3.370979073972067e-11
```

5 Sentiment Analysis using RNNs

In this section, you are required to test your implementations above by running an ensemble NN on a sentiment analysis dataset. The dataset, `data/corpus.csv`, consists of 800 real movie comments and the corresponding labels that indicate whether the comments are positive or negative. For example:

```
POSTIVE: I absolutely LOVE Harry Potter, as you can tell already.
NEGATIVE: My dad's being stupid about brokeback mountain...
```

We provide a basic NN for your experiments, which can be found in `applications.py`. The architecture is as follow:

```
FCLayer(vocab_size, 200, name='embedding')
BidirectionalRNN(RNNCell(in_features=200, units=50))
FCLayer(100, 32, name='fclayer1')
TemporalPooling()
FCLayer(32, 2, name='fclayer2')
```

The input to the network is sequences of one-hot vectors, each of which represents a word. The 1st FC layer works as an embedding layer to learn and retrieve the word embedding vectors. After a Bi-directional RNN layer and another FC layer, a TemporalPooling layer (see `layers.py`) is used to mean-pooling a sequence of vectors into one vector, which will ignore the filling `NaN`s. The rest of the network is same as a normal NN classifier.

```
In [14]: from utils import datasets
         from applications import SentimentNet
         from loss import SoftmaxCrossEntropy, L2
         from optimizers import Adam
```

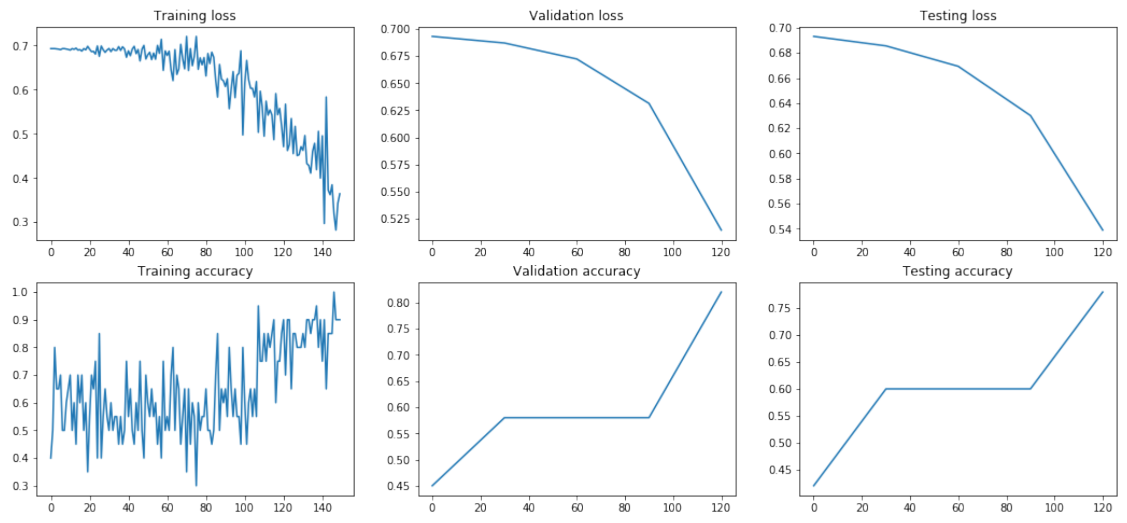
```
import numpy as np
np.random.seed(5242)

dataset = datasets.Sentiment()
model = SentimentNet(dataset.dictionary)
loss = SoftmaxCrossEntropy(num_class=2)

adam = Adam(lr=0.001, decay=0,
            scheduler_func=lambda lr, it: lr*0.5 if it%1000==0 else lr)
model.compile(optimizer=adam, loss=loss, regularization=L2(w=0.001))
train_results, val_results, test_results = model.train(
        dataset,
        train_batch=20, val_batch=100, test_batch=100,
        epochs=5,
        val_intervals=100, test_intervals=300, print_intervals=5)
```

```
[nltk_data] Downloading package punkt to /home/xindi/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
Number of training samples: 600
Number of validation samples: 100
Number of testing samples: 100
Epoch 0:
Test accuracy=0.42000, loss=0.69315
Validation accuracy: 0.45000, loss: 0.69315
Iteration 0:        accuracy=0.40000, loss=0.69315, regularization loss= 0.009036046209358467
Iteration 5:        accuracy=0.70000, loss=0.69070, regularization loss= 0.0022410672725193975
Iteration 10:        accuracy=0.70000, loss=0.68973, regularization loss= 0.00041878472374745683
Iteration 15:        accuracy=0.60000, loss=0.69091, regularization loss= 0.0001413690914479824
Iteration 20:        accuracy=0.55000, loss=0.69177, regularization loss= 0.00020796768528298125
Iteration 25:        accuracy=0.85000, loss=0.67568, regularization loss= 0.0006681557886230196
Epoch 1:
Test accuracy=0.60000, loss=0.68559
Validation accuracy: 0.58000, loss: 0.68705
Iteration 0:        accuracy=0.50000, loss=0.69345, regularization loss= 0.001613390747216352
Iteration 5:        accuracy=0.45000, loss=0.69772, regularization loss= 0.002402786281599502
Iteration 10:        accuracy=0.55000, loss=0.68806, regularization loss= 0.0030177369004393824
Iteration 15:        accuracy=0.50000, loss=0.69112, regularization loss= 0.005082075466266946
Iteration 20:        accuracy=0.60000, loss=0.67920, regularization loss= 0.008247960253956459
Iteration 25:        accuracy=0.45000, loss=0.70043, regularization loss= 0.014103194247666671
Epoch 2:
Test accuracy=0.60000, loss=0.66931
Validation accuracy: 0.58000, loss: 0.67233
Iteration 0:        accuracy=0.55000, loss=0.67761, regularization loss= 0.015688502132330688
Iteration 5:        accuracy=0.70000, loss=0.63469, regularization loss= 0.022206020852305094
Iteration 10:        accuracy=0.35000, loss=0.72084, regularization loss= 0.027938583705597174
Iteration 15:        accuracy=0.30000, loss=0.72080, regularization loss= 0.03369917402022893
Iteration 20:        accuracy=0.65000, loss=0.63110, regularization loss= 0.04102514636303885
Iteration 25:        accuracy=0.70000, loss=0.62178, regularization loss= 0.052299307999537234
```

```
Epoch 3:
Test accuracy=0.60000, loss=0.63007
Validation accuracy: 0.58000, loss: 0.63132
Iteration 0:        accuracy=0.65000, loss=0.60729, regularization loss= 0.0660883834625426
Iteration 5:        accuracy=0.65000, loss=0.58193, regularization loss= 0.08139344458128718
Iteration 10:        accuracy=0.60000, loss=0.61475, regularization loss= 0.09359006369435258
Iteration 15:        accuracy=0.65000, loss=0.58354, regularization loss= 0.11112877887105403
Iteration 20:        accuracy=0.85000, loss=0.49460, regularization loss= 0.13094445978583277
Iteration 25:        accuracy=0.90000, loss=0.48669, regularization loss= 0.1521807189338592
Epoch 4:
Test accuracy=0.78000, loss=0.53882
Validation accuracy: 0.82000, loss: 0.51451
Iteration 0:        accuracy=0.90000, loss=0.47099, regularization loss= 0.17514628267298024
Iteration 5:        accuracy=0.85000, loss=0.45507, regularization loss= 0.19911478609797545
Iteration 10:        accuracy=0.85000, loss=0.46247, regularization loss= 0.2250560789075794
Iteration 15:        accuracy=0.90000, loss=0.46028, regularization loss= 0.2528770682403159
Iteration 20:        accuracy=0.75000, loss=0.49559, regularization loss= 0.28189964550639507
Iteration 25:        accuracy=0.85000, loss=0.38463, regularization loss= 0.304186624250363
```

```
In [15]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.figure(2, figsize=(18, 8))
         plt.subplot(2, 3, 1)
         plt.title('Training loss')
         plt.plot(train_results[:,0], train_results[:,1])
         plt.subplot(2, 3, 4)
         plt.title('Training accuracy')
         plt.plot(train_results[:,0], train_results[:,2])
         plt.subplot(2, 3, 2)
         plt.title('Validation loss')
         plt.plot(val_results[:,0], val_results[:,1])
         plt.subplot(2, 3, 5)
         plt.title('Validation accuracy')
         plt.plot(val_results[:,0], val_results[:,2])
         plt.subplot(2, 3, 3)
         plt.title('Testing loss')
         plt.plot(test_results[:,0], test_results[:, 1])
         plt.subplot(2, 3, 6)
         plt.title('Testing accuracy')
         plt.plot(test_results[:, 0], test_results[:,2])
```

```
Out[15]: [<matplotlib.lines.Line2D at 0x7f85f8a73f60>]
```



5.1 Train your best Sentiment Analysis net

Based on the provided NN, tweak the hyperparameters and use what you've learnt to train the best net on sentiment analysis (you should not need to wait half a day for the training to complete). You are free to use more features, hidden units, layers etc. In your report, please write the following: - Training and test accuracy over iterations - Architecture and training method (eg. optimization scheme, data augmentation): explain your design choices, what has failed and what has worked and why you think they worked/failed

Credits will be given based on your test accuracy and explanation on your network architecture and training method. Large component of the grading will subject to the latter. Use only the code you have written and any helper functions provided in this assignment and **assignment 1**. Do not use external libraries like Tensorflow and Pytorch.

6 Final submission instructions

Upon completion, please submit the following: - Your code files in a folder `codes`; - A short report (1-2 pages) in pdf titled `report.pdf`, explaining the logic (expressed using mathematical formulation) of your implementation (including the forward and backward function like ReLU) and the findings from training your best net.

Please zip up the abovementioned files under a folder named with your NUSNET ID: eg. 'e0123456.zip' and submit the zipped folder to IVLE/workbin/assignment 2 submission. The submission deadline is 2 APRIL 2018 (MON) 17:00PM.