Homework 2: Recurrent Neural Networks
CSC 266 / 466
Frontiers in Deep Learning
Spring 2023

Junfei Liu - jliu137@u.rochester.edu

Deadline: See Blackboard

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using http://overleaf.com, which is free and can be accessed online.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. The relevant code should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

You may wish to use the program *MathType*, which can easily export equations to AMS LaTeX so that you don't have to write the equations in LaTeX directly: http://www.dessci.com/en/products/mathtype/

Problem 1 - Recurrent Neural Networks (10 points)

In this one you will find the parameters for an RNN that implements binary addition, but rather than using a toolbox, you will find them by hand!

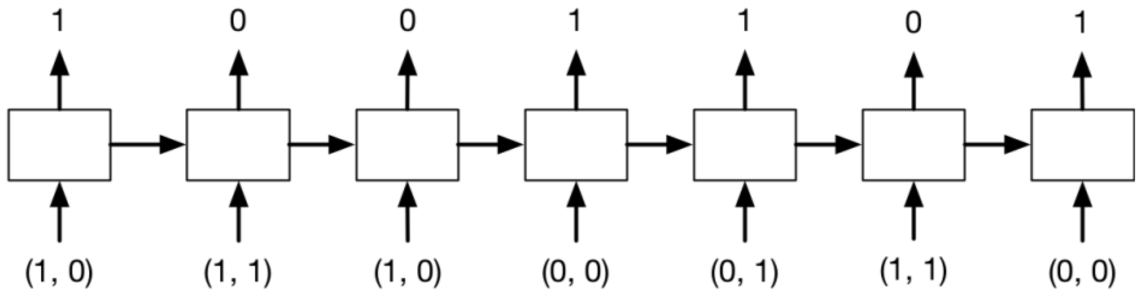
The input to your RNN will be two binary numbers, starting with the *least* significant bit. You will need to pad the largest number with an additional zero on the left side and you should make the other number the same length by padding it with zeros on the left side. For instance, the problem

$$100111 + 110010 = 1011001$$

would be input to your RNN as:

- Input 1: 1, 1, 1, 0, 0, 1, 0
- Input 2: 0, 1, 0, 0, 1, 1, 0
- Correct output: 1, 0, 0, 1, 1, 0, 1

The RNN has two input units and one output unit. In this example, the sequence of inputs and outputs would be:



The RNN that implements binary addition has three hidden units, and all of the units use the following non-differentiable hard-threshold activation function

$$\sigma(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

The equations for the network are given by

$$\begin{aligned} y_t &= \sigma\left(\mathbf{v}^T \mathbf{h}_t + b_y\right) \\ \mathbf{h}_t &= \sigma\left(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{b}_\mathbf{h}\right) \end{aligned}$$

where $\mathbf{x}_t \in \mathbb{R}^2$, $\mathbf{U} \in \mathbb{R}^{3\times 2}$, $\mathbf{W} \in \mathbb{R}^{3\times 3}$, $\mathbf{b}_\mathbf{h} \in \mathbb{R}^3$, $\mathbf{v} \in \mathbb{R}^3$, and $b_y \in \mathbb{R}$.

Before backpropagation was invented, neural network researchers using hidden layers would set the weights by hand (at least that is what Prof. Kanan's PhD advisor told him). Your job is to find the settings for all of the parameters *by hand*, including the value of $\mathbf{h}_0$. Give the settings for all of the matrices, vectors, and scalars to correctly implement binary addition. Note that since the activation function is non-differentiable, backpropagation would not work.

Assume that all of the coefficients in the matrices are integers and that they are either -2, -1, 0, 1, or 2.

Hint: All elements of **U** are the same value. Have one hidden unit activate if the sum is at least 1, one hidden unit activate if the sum is at least 2, and one hidden unit if it is 3, which you can do using $\mathbf{b_h}$ as it acts as the firing threshold for each unit. You may wish to review how a binary adder works. If you are struggling with the problem, I recommend programming up the forward pass of the network, e.g., using NumPy or in Julia, to assess your solution, along with writing some tests to verify that your matrices are producing the correct outputs. If you do this, you do not need to give us the code.

Solution:

$$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{b_h} = \begin{bmatrix} 0 \\ -1 \\ -2 \end{bmatrix}$$

$$\mathbf{v} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

$$\mathbf{b_y} = 0$$

The above answers are calculated based on the assumption that $\mathbf{h_t} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ when sum is 0, $\mathbf{h_t} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ when sum is 1, $\mathbf{h_t} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ when sum is 2, and $\mathbf{h_t} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ when sum is 3. In this case, $\mathbf{U}\mathbf{x}_t$ can produce $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$, or $\begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}$ with $\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$. Meanwhile, with

$\mathbf{W} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, we have

$$\mathbf{W}\mathbf{h}_{t-1} = \begin{cases} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \text{if } \mathbf{h}_{t-1} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} & \text{if } \mathbf{h}_{t-1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \end{cases}$$

Therefore, $\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}$ will yield a vector filled with the same value as sum. Then, by having $\mathbf{b_h} = \begin{bmatrix} 0 \\ -1 \\ -2 \end{bmatrix}$, $\mathbf{h_t}$ can successfully output the desired value.

Because desired $\mathbf{y_t}$ = sum % 2, I designed $\mathbf{v} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$ and $\mathbf{b_y} = 0$ such that $\mathbf{v}^T\mathbf{h}_t + b_y$ will yield the correct output with all four cases of $\mathbf{h_t}$.
It passed the test by a program using NumPy shown in Appendix.

Problem 2 - GRU for Sentiment Analysis

In this problem you will use a popular RNN model called the Gated Recurrent Units (GRU) to learn to predict the sentiment of a film, television, etc. review. The dataset we are using is the IMDB review dataset (link). It is a binary sentiment classification (positive or negative) dataset. We provide four text files for you to download: train_pos_reviews.txt, train_neg_reviews.txt, test_pos_reviews.txt, test_neg_reviews.txt. Each line is an independent review for a movie.

Put your code in the appendix.

Part 1 - Preprocessing (5 points)

First you need to do proper preprocessing of the sentences so that each word is represented by a single number index in a vocabulary.

Remove all punctuation from the sentences. Build a vocabulary from the unique words collected from text file so that each word is mapped to a number.

Now you need to convert the data to a matrix where each row is a review. Because reviews are of different lengths, you need to pad or truncate the reviews to make them same length. We are going to use 400 as the fixed length in this problem. That means any review that is longer than 400 words will be truncated; any review that is shorter than 400 words will be padded with 0s. Please note that your padded 0s should be placed *before* the review if they are needed.

After you prepare the data, you can define a standard PyTorch dataloader directly from numpy arrays (say you have data in train_x and labels in train_y).

```
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
```

Implement the data preprocessing procedure.

Solution:
See Appendix for preprocessing code.

Part 2 - Build A Binary Prediction RNN with GRU (20 points)

Your RNN module should contain the following modules: a word embedding layer, a GRU, and a prediction unit.

1. You should use nn.Embedding layer to convert an input word to an embedded feature vector.
2. Use nn.GRU module. Feel free to choose your own hidden dimension. It might be good to set the batch_first flag to True so that the GRU unit takes (batch, seq, embedding_dim) as the input shape.
3. The prediction unit should take the output from the GRU and produce a number for this binary prediction problem. Use nn.Linear and nn.Sigmoid for this unit.

At a high level, the input sequence is fed into the word embedding layer first. Then, the GRU is taking steps through each word embedding in the sequence and return output / feature at each step. The prediction unit should take the output from the final step of the GRU and make predictions.

Implement your GRU module, train the model and report accuracy on the test set.

Solution:
The test accuracy after 20 epochs is 0.7432065606117249.

Part 3 - Comparison with a MLP (5 points)

Since each review is a fixed length input (with potentially many 0s in some samples), we can also train a standard MLP for this task.

Train a MLP with one hidden layer on the training data and report accuracy on the test set. Make the number of parameters similar to the GRU used earlier. Use sensible choices for activation functions and your optimizer. How does it compare with the result from your GRU model?

Solution:
The test accuracy is floating around 0.5. The final test accuracy after 20 epochs is 0.49966034293174744, which is nearly a pure guessing on a binary classification problem. I trained the standard MLP with the first layer with 256 units and a hidden layer with 64 units. The MLP model without any embedding is completely outperformed by GRU model.

Code Appendix

Discussed with Yuze Wang
Reference:
`https://www.geeksforgeeks.org/python-remove-punctuation-from-string/`

Problem2 code (in ipynb):

```
# In[1]:


import re
import numpy as np
import torch
import torch.nn as nn
from torchmetrics.functional import accuracy
import torchmetrics
from torch.utils.data import TensorDataset, DataLoader
from tqdm import tqdm
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


# ## Data preprocessing

# In[2]:


def load_reviews(path, review_length, vocab):
    reviews = []
    f = open(path, "r", encoding='UTF-8')
    for line in f:
        review = []
        line = re.sub(r'[^\w\s]', '', line).lower().strip()
        for word in line.split():
            review.append(vocab[word])
        if len(review) < review_length: #pad in front
            for i in range(review_length - len(review)):
                review.insert(0, "0")
        else: # truncate
            review = review[:review_length]
        reviews.append(review)
    return np.array(reviews)


def find_vocab(path):
    f = open(path, "r", encoding='UTF-8')
    vocab = set()
    for line in f:
        line = re.sub(r'[^\w\s]', '', line).lower().strip() # reference to geeksforgeeks
        for word in line.split():
            vocab.add(word)
    return {word: i for i, word in enumerate(vocab, 1)} # enumerate start at 1 to leave 0 f


def prepare_train(data_path, review_length=400):
    all_vocab = find_vocab(data_path + "all_merged.txt")
    train_positive_reviews = load_reviews(data_path + "train_pos_merged.txt", review_length
    train_negative_reviews = load_reviews(data_path + "train_neg_merged.txt", review_length
    test_positive_reviews = load_reviews(data_path + "test_pos_merged.txt", review_length,
    test_negative_reviews = load_reviews(data_path + "test_neg_merged.txt", review_length,

    train_x = np.concatenate((train_positive_reviews, train_negative_reviews), axis=0)
    train_y = np.concatenate(((np.full((len(train_positive_reviews), 1), 1)), (np.full((len
    test_x = np.concatenate((test_positive_reviews, test_negative_reviews), axis=0)
    test_y = np.concatenate(((np.full((len(test_positive_reviews), 1), 1)), (np.full((len(t
    print("Vocabulary size: ", len(all_vocab))
    # print(train_positive_reviews[:100])
    #print(train_x.shape)
    #print(train_y.shape)
    #print(test_x.shape)
    #print(test_y.shape)
    return train_x, train_y, test_x, test_y, len(all_vocab)

# train_x, train_y, test_x, test_y = prepare_train(data_path)


# ## Build RNN with GRU

# In[3]:


class MyGRU(nn.Module):
    def __init__(self, vocab_size, hidden_dim, output_dim, num_layers, batch_size=512): # T
        super().__init__()
        self.last_hidden = torch.zeros(num_layers, batch_size, hidden_dim).to(device)
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, hidden_dim)
        self.gru = nn.GRU(hidden_dim, hidden_dim, num_layers=num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        # self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.embedding(x)
        output, hidden = self.gru(x, self.last_hidden)
        output = output[:, -1]
        self.last_hidden = hidden.data # TODO: .data?
        #print(self.last_hidden.shape)

        return self.fc(output)


# In[4]:


def train_test_GRU(train_loader, test_loader, learning_rate, epochs, vocab_size, hidden_dim
    model = MyGRU(vocab_size=vocab_size, hidden_dim=hidden_dim, output_dim=1, num_layers=2,
    loss_fn = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

    model.to(device)
    train_loss_all = []
    test_acc_all = []
    for epoch in range(1, epochs+1):
        #train
        train_loss = 0
        #train_acc = 0
        model.train()
        for batch in tqdm(train_loader):
            x, y = batch
            x = x.to(device)
            y = y.to(device)
            optimizer.zero_grad()
            output = model(x)
            loss = loss_fn(output, y.float())# TODO: squeeze?
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
            #train_acc += accuracy(output, y, task='binary')
        print(f"Epoch: {epoch}, Train Loss: {train_loss / len(train_loader)}")#, Train Acc:

        #test
        #test_loss = 0
        test_acc = 0
        for batch in tqdm(test_loader):
            x, y = batch
            x = x.to(device)
            y = y.to(device)
            output = torch.sigmoid(model(x)) # TODO: squeeze?
            loss = loss_fn(output, y.float())

            #test_loss += loss.item()
            test_acc += accuracy(output, y, task='binary')
        print(f"Epoch: {epoch}, Test Acc: {test_acc / len(test_loader)}") #Test Loss: {test

        train_loss_all.append(train_loss / len(train_loader))
        test_acc_all.append(test_acc / len(test_loader))

    return train_loss_all, test_acc_all


# In[7]:
```

```
def main():
    data_path = "./data/"
    review_length = 400
    batch_size = 128
    train_x, train_y, test_x, test_y, vocab_size = prepare_train(data_path, review_length)

    train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
    train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size, drop_last=Tr
    test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))
    test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size, drop_last=True

    lr = 0.0001
    epochs = 20
    hidden_dim = 256


    train_loss_all, test_acc_all = train_test_GRU(train_loader, test_loader, learning_rate=

main()


# In[8]:


## Comparison with a MLP


# In[9]:


class mlp(nn.Module):
    def __init__(self):
        super().__init__()
        self.first = nn.Linear(400, 64)
        self.second = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.to(torch.float32)
        x = self.first(x)
        x = self.sigmoid(x)
        x = self.second(x)
        x = self.sigmoid(x)
        return x

def train_test_mlp(train_loader, test_loader, epochs):
    model = mlp()
    loss_fn = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.AdamW(model.parameters())

    model.to(device)
    train_loss_all = []
    test_acc_all = []

    print("Now training MLP model...")

    for epoch in range(1, epochs+1):
        #train
        train_loss = 0
        #train_acc = 0
        model.train(True)
        for batch in tqdm(train_loader):
            x, y = batch
            x = x.to(device)
            y = y.to(device)
            optimizer.zero_grad()
            output = model(x)
            loss = loss_fn(output, y.float())
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
            #train_acc += accuracy(output, y, task='binary')
        print(f"Epoch: {epoch}, Train Loss: {loss}")#, Train Acc: {train_acc / len(train_lo

        #test
        #test_loss = 0
        test_acc = 0
        model.train(False)
        for batch in tqdm(test_loader):
            x, y = batch
            x = x.to(device)
            y = y.to(device)
            output = model(x)
            # loss = loss_fn(output, y.float())

            #test_loss += loss.item()
            test_acc += accuracy(output, y, task='binary')
        print(f"Epoch: {epoch}, Test Acc: {test_acc / len(test_loader)}") #Test Loss: {test

        train_loss_all.append(train_loss / len(train_loader))
        test_acc_all.append(test_acc / len(test_loader))

    return train_loss_all, test_acc_all

def mlp_main():
    data_path = "./data/"
    review_length = 400
    batch_size = 128
    train_x, train_y, test_x, test_y, vocab_size = prepare_train(data_path, review_length)

    train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
    train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size, drop_last=Tr
    test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))
    test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size, drop_last=True

    epochs = 20

    train_loss_all, test_acc_all = train_test_mlp(train_loader, test_loader, epochs=epochs)

mlp_main()
```

Problem1 verification code:

```
import numpy as np


class RNN():
    def __init__(self, W, U, bh, V, by, h0):
        self.W = W
        self.U = U
        self.bh = bh
        self.V = V
        self.by = by
        self.ht_1 = h0

    def sigmoid(self, input_vector):
        out = np.zeros(np.size(input_vector))
        if (np.size(input_vector) > 1):
            for i, value in enumerate(input_vector):
                if value > 0:
                    out[i] = 1
                else:
                    out[i] = 0
        else:
            if input_vector > 0:
                out = 1
            else:
                out = 0
        return out

    def forward(self, x):
        h = self.sigmoid(np.dot(self.U, x) +
                         np.dot(self.W, self.ht_1) + self.bh)
        y = self.sigmoid(self.V.dot(h) + self.by)
        print(
            f"Ux: {np.dot(self.U, x)}\n Wht-1: {np.dot(self.W, self.ht_1)}\n h: {h}\n Vh: {
        self.ht_1 = h
        return y


def main():
    W = np.array([[0, 1, 0], [0, 1, 0], [0, 1, 0]])
    U = np.array([[1, 1], [1, 1], [1, 1]])
    bh = np.array([0, -1, -2])
    by = 0
    V = np.array([1, -1, 1])
    h0 = np.array([0, 0, 0])

    rnn = RNN(W, U, bh, V, by, h0)
    input1 = np.array([1, 0, 0, 1, 1, 1])
    input2 = np.array([1, 1, 0, 0, 1, 0])
    output = np.zeros(len(input1)+1)
    for i in reversed(range(np.size(input1))):
        output[i+1] = rnn.forward(np.array([input1[i], input2[i]]))
    if np.array_equal(np.dot(W, rnn.ht_1), np.array([1, 1, 1])):
        output[0] = 1

    print(output)


if __name__ == '__main__':
    main()
```