Course: DD2424 - Assignment 4

In this assignment you will train an RNN to synthesize English text character by character. You will train a *vanilla RNN* with outputs, as described in lecture 9, using the text from the book *The Goblet of Fire* by J.K. Rowling. The variation of SGD you will use for the optimization will be *AdaGrad*. The final version of your code should contain these major components:

- **Preparing Data**: Read in the training data, determine the number of unique characters in the text and set up mapping functions - one mapping each character to a unique index and another mapping each index to a character.
- **Back-propagation**: The forward and the backward pass of the back-propagation algorithm for a vanilla RNN to efficiently compute the gradients.
- **AdaGrad** updating your RNN's parameters.
- **Synthesizing text from your RNN**: Given a learnt set of parameters for the RNN, a default initial hidden state $\mathbf{h}_0$ and an initial input vector, $\mathbf{x}_0$, from which to bootstrap from then you will write a function to generate a sequence of text.

Background 1: *A Vanilla RNN*

The mathematical details of the RNN you will implement are as follows. Given a sequence of input vectors, $\mathbf{x}_1, \ldots, \mathbf{x}_\tau$, where each $\mathbf{x}_t$ has size $d \times 1$ and an initial hidden state $\mathbf{h}_0$, the RNN outputs at each time-step t a vector of probabilities, $\mathbf{p}_t$ $(K \times 1)$, for each possible character and a hidden state $\mathbf{h}_{t+1}$ for size $m \times 1$. That is

for $t = 1, 2, \ldots, \tau$

$$\mathbf{a}_t = W\,\mathbf{h}_{t-1} + U\mathbf{x}_t + \mathbf{b} \tag{1}$$
$$\mathbf{h}_t = \tanh(\mathbf{a}_t) \tag{2}$$
$$\mathbf{o}_t = V\mathbf{h}_t + \mathbf{c} \tag{3}$$
$$\mathbf{p}_t = \text{SoftMax}(\mathbf{o}_t) \tag{4}$$

The loss for a single labelled sequence is the sum of the cross-entropy loss for each

$$L(\mathbf{x}_{1:\tau}, \mathbf{y}_{1:\tau}, \Theta) = \sum_{t=1}^{\tau} l_t = -\sum_{t=1}^{\tau} \log(\mathbf{y}_t^T \mathbf{p}_t) \tag{5}$$

where $\Theta = \{\mathbf{b}, \mathbf{c}, W, U, V\}$, $\mathbf{x}_{1:\tau} = \{\mathbf{x}_1, \ldots, \mathbf{x}_\tau\}$ and $\mathbf{y}_{1:\tau}$ is defined similarly.

The equations for the gradient computations of the back-propagation algorithm for such an RNN are given in Lecture 9. Note in the lecture notes the bias vectors have been omitted. It is left as an exercise for you to compute the gradient w.r.t. the two bias vectors.

Background 2: *AdaGrad algorithm for optimization*

In this assignment you will implement the variant of SGD called *AdaGrad*. To refresh your memory the update steps for AdaGrad are defined as

$$\mathbf{m}_{\theta,t'} = \mathbf{m}_{\theta,t'-1} + \mathbf{g}_{t'}^2 \tag{6}$$

$$\boldsymbol{\theta}_{t'+1} = \boldsymbol{\theta}_{t'} - \frac{\eta}{\sqrt{\mathbf{m}_{\theta,t'} + \epsilon}} \mathbf{g}_{t'} \tag{7}$$

where

- $\boldsymbol{\theta}$ is a generic place holder for the parameter vector/matrix under consideration,
- t' refers to the iteration of the SGD update (not to be confused with the t used to denote the input and output vectors of the labelled training sequence),
- $\mathbf{g}_{t'}$ is the gradient vector $\frac{\partial L}{\partial \boldsymbol{\theta}}$ and
- in an abuse of notation the operations of division, raising to the power of two and square root are applied to each entry of the vector/matrix independently.

(You could also try `RMSProp` updates where

$$\mathbf{m}_{\theta,t'} = \gamma \mathbf{m}_{\theta,t'-1} + (1-\gamma)\, \mathbf{g}_{t'}^2 \tag{8}$$

$$\boldsymbol{\theta}_{t'+1} = \boldsymbol{\theta}_{t'} - \frac{\eta}{\sqrt{\mathbf{m}_{\theta,t'} + \epsilon}} \mathbf{g}_{t'} \tag{9}$$

Common values settings for γ and η are .9 and .001 respectively. However, I found for this update step there are more problems with exploding gradients and you have to be more careful with how you clip your gradients.)

Exercise 1: *Implement and train a vanilla RNN*

In the following I will sketch the different parts you will need to write to complete the assignment. Note it is a guideline. You can, of course, have a different design, but you should read the outline to help inform how different parameters and design choices are made.

0.1 Read in the data

First you need to read in the training data from the text file of *The Goblet of Fire* available for download at the Canvas webpage. To save you some time here is code that will read in the contents of this text file.

```
book_fname = 'data/Goblet.txt';
fid = fopen(book_fname,'r');
book_data = fscanf(fid,'%c');
fclose(fid);
```

All the characters of the book are now in the vector `book_data`. To get a vector containing the unique characters in `book_data` apply the *Matlab* function `unique`. Once you have this list, which we will denote by `book_chars`, then its length `K` corresponds to the dimensionality of the output (input) vector of your RNN.

To allow you to easily go between a character and its one-hot encoding and in the other direction you should initialize map containers of the form:

```
char_to_ind = containers.Map('KeyType','char','ValueType','int32');
ind_to_char = containers.Map('KeyType','int32','ValueType','char');
```

Then for `char_to_ind` you should fill in the characters in your alphabet as its keys and create an integer for its value (keep things simple and use where the character appears in the vector `book_chars` as its value). And similarly for `ind_to_char` fill in the integers `1` to `K` as its keys and assign the appropriate character value for each integer. You will use these map containers when you convert a sequence of characters into a sequence of vectors of one-hot encodings and then when you convert a synthesized sequence of one-hot encodings back into a sequence of characters.

0.2 Set hyper-parameters & initialize the RNN's parameters

The one hyper-parameter you need to define the RNN's architecture is the dimensionality of its hidden state `m`. For this assignment you should set `m=100`. The other hyper-parameters you need to set are those associated with training and these are the learning rate `eta` and the length of the input sequences (`seq_length`) you use during training. Here are the default settings for this assignment `eta=.1` and `seq_length=25`.

In my code I found it easiest to store the parameters of the model in an object called `RNN`. I initialized the bias vectors `RNN.b` and `RNN.c` to be zero vectors of length `m`$\times$`1` and `K`$\times$`1`. Note for this task the dimensionality of the input and output vectors are the same. While the weight matrices are randomly initialized as

```
RNN.U = randn(m, K)*sig;
RNN.W = randn(m, m)*sig;
RNN.V = randn(K, m)*sig;
```

where I set `sig = .01`.

0.3 Synthesize text from your randomly initialized RNN

Before you begin training your RNN, you should write a function that will synthesize a sequence of characters using the current parameter values in your RNN. Besides `RNN`, it will take as input a vector `h0` (the hidden state at time 0), another vector `x0` which will represent the first (dummy) input vector to your RNN (it can be some character like afull-stop), and an integer `n` denoting the length of the sequence you want to generate. In the body of the function you will write code to implement the equations (1-4). There is just one major difference - you have to generate the next input vector `xnext` from the current input vector `x`. At each time step t when you generate a vector of probabilities for the labels, you then have to sample a label (i.e. an integer) from this discrete probability distribution. This sample will then be the $(t+1)$th character in your sequence and will be the input vector for the next time-step of your RNN.

Here is one way to randomly select a character based on the output probability scores `p`:

```
cp = cumsum(p);
a = rand;
ixs = find(cp-a >0);
ii = ixs(1);
```

First you compute the vector containing the cumulative sum of the probabilities. Then you generate a random draw, `a`, from a uniform distribution in the range 0 to 1. Next you find the index $1 \leq$ `ii` $\leq$ `K` such that `cp(ii-1)` $\leq$ `a` $\leq$ `cp(ii)` where we assume for notational convenience `cp(0)=0`. You should store each index you sample for $1 \leq$ `t` $\leq$ `n` and let your function output the matrix `Y` (size `K`$\times$`n`) where `Y` is the one-hot encoding of each sampled character. Given `Y` you can then use the map container `ind_to_char` to convert it to a sequence of characters and view what text your RNN has generated.

0.4 Implement the forward & backward pass of back-prop

Next up is writing the code to compute the gradients of the loss w.r.t. the parameters of the model. While you write this code, you should use the first

seq_length characters of book_data as your labelled sequence for debugging that is

```
X_chars = book_data(1:seq_length);
Y_chars = book_data(2:seq_length+1);
```

Note the label for an input character is the next character in the book. Once you have X_chars and Y_chars, you then have to convert them to the matrices X and Y containing the one-hot encoding of the characters of the sequence. Both X and Y have size K×seq_length and each column of the respective matrices corresponds to an input vector and its target output vector. You should also set h0 to the zero vector. Given this labelled sequence and initial hidden state you are in a position to write and call a function that performs the forward-pass of the back-prop algorithm. This function should apply the equations (1-4) to the input data just described and return the loss and also the final and intermediary output vectors at each time step needed by the backward-pass of the algorithm.

Once you have computed the forward-pass then the next step is to write the code for the backward pass of the back-prop algorithm. Here you should implement the equations given in Lecture 9.

One piece of advice is that you should store each of the computed gradients in an object (with the same names as for your RNN object), such as grads.W etc. This will allow you to write more streamlined code to check your analytical gradients against their numerical counterparts and to implement the AdaGrad updates. This is because you can access the names within an object easily, so for instance, to perform a vanilla SGD update step for all the parameters the code would look something like:

```
for f = fieldnames(RNN)'
  RNN.(f{1}) = RNN.(f{1}) - eta * grads.(f{1});
end
```

After you have written the code to compute the forward and backward pass, you then have to, as per usual check your gradient computations numerically. On the Canvas website I have provided a *Matlab* function that computes the gradients numerically. The function assumes you have stored the parameters of your RNN in an object called RNN and also it calls a function ComputeLoss that computes the loss for an input sequence of length 25 stored in a matrix X of size K×25 and corresponding ground truth output matrix, Y of size K×25, that has the label for each input in the sequence. The numerical gradient computations also sets h0 to be zero for each gradient computation. In my code I set the step size for the numerical computations to h=1e-4 and I get a max relative error of around 8.5344e-06 (for RNN.V) when I set m=5. If you increase the value of m then you will need to increase the value of sig used in the parameter initialization to combat numerical precision issues.

Once you are sure your gradient computations are correct then you should clip your gradients to avoid the exploding gradient problem. Something like this should work:

```
for f = fieldnames(grads)'
   grads.(f1) = max(min(grads.(f1), 5), -5);
end
```

0.5 Train your RNN using AdaGrad

You are now ready to write the high-level loop to train your RNN with the text in `book_data`. The general high-level approach will be as follows. Let `e` (initialized to 1) be the integer that keeps track of where in the book you are. At each iteration of the SGD training you should grab a labelled training sequence of length `seq_length` characters. Thus your sequence of input characters corresponds to `book_data(e:e+seq_length-1)` and the labels for this sequence is `book_data(e+1:e+seq_length)`. You should convert these sequence of characters into the matrices `X` and `Y` (the one-hot encoding vectors of each character in the input and output sequences).

However, before you pass this labelled sequence into your forward and backward functions you also need to define `hprev`. If `e=1` then `hprev` should be the zero vector while if `e>1` then `hprev` should be set to the last computed hidden state by the forward pass in the previous iteration. Thus (hopefully) you have a `hprev` that has stored the context of all the prior characters it has seen so far in the book! Now you have all the inputs needed for the forward and backward pass functions to compute the gradient. Once you have computed the gradients then you can apply the AdaGrad update step to all the parameters of your RNN.

Your forward pass function should also return the loss for the labelled training sequence. As we are implementing SGD the loss from one training sequence to the next will vary alot and also it is too expensive to compute the loss of the entire training data, it useful to keep track of a smoothed version of the loss over the iterations with a weighted sum of the smoothed loss and the current loss such as:

```
smooth_loss = .999* smooth_loss + .001 * loss;
```

You should print out `smooth_loss` regularly (say after every 100th update step) to see if the smoothed loss is, in general, reducing. What I found is that learning is initially very fast and then it slows. After the 1st epoch learning is much slower and you can see the smoothed low going up and down according to which part of the novel is harder or easier to predict,

but at corresponding points in the novel there is a general trend for the smoothed loss to get gradually smaller. (Note the lower values I saw for the `smooth_loss` (∼ 7th epoch of training, 300,000 update steps) were ∼39. You will probably reach this loss value at a much earlier stage of training though maybe not as consistently throughout the text.)

You should also synthesize text (of length around 200 characters) from your `RNN` regularly (say after every 500th update step) during training (you can let out a shout of hurrah when you see your first synthesized `Harry`, `Hermione`, `Dumbledore`, or ...). This allows you to see if your training is doing something sensible. You can do this by calling your function where `h0` is the same `hprev` as used in the forward pass and `x0` is `X(:, 1)` (the first character of the labelled input sequence for the current iteration).

At the end of an update step you should then increase your the counter `e` by `seq_length`. If this results in `e>length(book_data)-seq_length-1` then you should reset `e` to be 1 and loop through the characters in the book again. When you reset `e` you have completed one epoch of training. Also when you reset `e` to 1 you should also reset `hprev` to the zero-vector.

To help you debug here are snapshots of text sequences I generated at the different stages of my training:

iter = 1, smooth_loss=109.236

```
 C':}3 By/KOFOeUOyb eD6GJ_yq ^oQIf.pFHWZaz(rPD2c,:CVjdAg);Q!?x3eqS9qyDme;g)L-XUo}t Cg }F.Bz3wEXO!Yqs aiZ -wZwfZ
)eLy}t/e)w-}b7MCt7Mu^GQ(/SFB"9nx
exEoA/ 3Hn7lxcsuQD/^SDKn;K/EsRKjEH b'Q:cR-wWzrp
dUs-zd!
```

iter = 1000, smooth_loss=86.412

```
 inntho hordcad ghhoke cohe Ho Haou oban? nuithicwtcmar thim Brrtorter ighatulf aid thane ci,. aog goed ait Pmhath ae he cetheag Wel land toinn PhTcouc
```

iter = 4000, smooth_loss=60.063

```
 olryy.  He qalice men?"
bigpoud.
Souf'gh ait."d ath asing. Har or ande, the fiop ""I. "Seang,.
.
bage; fomtef pood Dotly, Mv. Houndny.
"."
horandy, bay I or."Woom Mroighin Hasrotvart fowhertent. Hadry
```

iter = 30000, smooth_loss=47.612:

```
 iless youd ane he parefins that ware Dues are Ske he flrars out..  Then , Harry in to, tham ham was lly doue - feepemer lacply into meach that preen a
```

iter = 363000, smooth_loss=39.450996:

```
 ore was and Encorks.
"Eut shave thinks his fist theaghed of Harry.
"Mr. If ack," said Ron beroncy, no tischen want they strating Serming.  "De Dragry," said Ron.  I would to wizards spmetwer scrough t
```

To complete the assignment:

To pass the assignment you need to upload:

1. The code for this assignment.

2. A brief pdf report with the following content:

   i) State how you checked your analytic gradient computations and whether you think that your gradient computations are bug free for your RNN.

   ii) Include a graph of the smooth loss function for a longish training run (at least 2 epochs).

   iii) Show the evolution of the text synthesized by your RNN during training by including a sample of synthesized text (200 characters long) before the first and before every 10,000th update steps when you train for 100,000 update steps.

   iv) A passage of length 1000 characters synthesized from your best model (the one that achieved the lowest loss).

Exercise 2: *Optional for bonus points*

1. **Synthesize *Game of Throne* tweets instead of Harry Potter**

   It would be fun to take the same approach in the assignment and apply it to generating tweets. Here we have the constraint that each sequence can be at most 140 characters long. I haven't done this myself but the adjustments I would initially make to the assignment implementation would be

   - Add an *end-of-tweet* character to your set of possible characters.
   - When you synthesize text have a hard limit of 140 characters.
   - During training I would re-initialize `hprev` to its default value after each training tweet.
   - Maybe play around with the length of the sequence during training.

   Here is a link to a repository of tweets from people who were watching season 8 of *Game of Thrones* Game of Thrones S8 (Twitter). When you download the data you will have the repository file `'gotTwitter.csv'` which contains the text of the tweets and lots of other information. In total there are 760,660. I don't think you need to use all the data to train your network. Use the number your computer can cope with! To process the file you should probably use `python3` and the `pandas` library. To get a list with the text of all the tweets you can use some code like

```
df1 = pd.read_csv('gotTwitter.csv', delimiter=',')
list_of_tweets = df1['text'].to_list()
print(len(list_of_tweets))
```

Bonus Points Available: (**4 points**) If you are able to generate tweets that bare some resemblance to a tweet. This is, of course, subjective but I will be very lenient in my judgments. It will be fun to see if you are able to generate tweets with names of the characters from the series and then see if other words pop up. BTW I have not *moderated* or scanned in total these tweets so apologies for the bad language and angry sentiments that might be embedded in them. I am also more that fine if you use another repository of tweets from a particular person or on a topic to perform the same task.

To get the bonus point you must submit

(a) Your code.

(b) A pdf document reporting briefly what changes (beyond cosmetic trivial ones) to the assignment implementation you made and containing synthesized tweets at different stages during training.