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 *Adam*. 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.
- **Adam** updating of 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

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

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: *Adam algorithm for optimization*

In this assignment you will implement the variant of an adaptive SGD called *Adam*. To refresh your memory the update steps for *Adam* are defined as (more details are in the lecture notes)

$$\mathbf{m}_{\theta,t'} = \beta_1 \mathbf{m}_{\theta,t'-1} + (1-\beta_1)\,\mathbf{g}_{t'} \tag{6}$$

$$\mathbf{v}_{\theta,t'} = \beta_2 \mathbf{v}_{\theta,t'-1} + (1-\beta_2)\,\mathbf{g}_{t'}^2 \tag{7}$$

$$\hat{\mathbf{m}}_{\theta,t'} = \mathbf{m}_{\theta,t'}/(1-\beta_1^{t'}) \tag{8}$$

$$\hat{\mathbf{v}}_{\theta,t'} = \mathbf{v}_{\theta,t'}/(1-\beta_2^{t'}) \tag{9}$$

$$\boldsymbol{\theta}_{t'+1} = \boldsymbol{\theta}_{t'} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_{\theta,t'} + \epsilon}}\,\hat{\mathbf{m}}_{\theta,t'} \tag{10}$$

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.

The standard default setting for the hyper-parameters are $\beta_1 = .9$, $\beta_1 = .999$ and $\epsilon = 1e-8$.

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 = book_dir + 'goblet_book.txt'
fid = open(book_fname, "r")
book_data = fid.read()
fid.close()
```

All the characters of the book are now in `book_data`. To get a vector containing the unique characters in `book_data` apply

```
unique_chars = list(set(book_data))
```

Once you have this list then its length `K` corresponds to the dimensionality of the output and also the 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 two dictionaries - `char_to_ind` and `ind_to_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 `unique_chars` as its value). And similarly for `ind_to_char` fill in the integers `0` to `K-1` as its keys and assign the appropriate character value for each integer. You will use these dictionaries 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=.001` and `seq_length=25`.

In my code I found it easiest to store the parameters of the network in a dictionary 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'] = (1/np.sqrt(2*K))*rng.standard_normal(size = (m, K))
RNN['W'] = (1/np.sqrt(2*m))*rng.standard_normal(size = (m, m))
RNN['V'] = (1/np.sqrt(m))*rng.standard_normal(size = (K, m))
```

where `rng` is defined as in `Assignment 1`. Please note due to how arrays are normally stored in memory in `python` it is probably more efficient to store each input example in `X` as a row vector as opposed to a column vector. In this case then `X` would have size `seq_len` $\times$ K and the RNN's parameters should have transposed dimensionality that is `RNN['U']` has size K $\times$ m and `RNN['U']` size m $\times$ K. And then applying `RNN['U']` to `X` is done with `np.matmul(X, RNN['U'])` etc. For the rest of this document it is assumed that `X` stores each separate input as a column but converting to the row storage is straightforward and just requires converting the vectors in the equations in the first of this document from column vectors to row vectors and making the adjustments accordingly.

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 = np.cumsum(p, axis=0)
a = rng.uniform(size=1)
ii = np.argmax(cp - a > 0)
```

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 `0`$\leq$`ii`$\leq$`K-1` such that `cp[ii-1]` $\leq$ `a` $\leq$ `cp[ii]`. You should store each index you sample for `0`$\leq$`t`$\leq$`n-1` 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 `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[0:seq_length]
Y_chars = book_data[1: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. As per usual you should store the computed gradients in a dictionary. This will allow you to write more streamlined code for gradient checking and to implement the Adam updates that is something akin to (but where you will have the updates of the Adam algorithm as opposed to vanilla SGD):

```
for kk in grads.keys():
    RNN[kk] = RNN[kk] - eta * grads[kk]
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. On the Canvas website I have provided a skeleton version of the `PyTorch` code, with lines missing in the forward pass you have to fill in, needed to calculate the gradients with its automatic differentiation engine. You need to write these lines with the appropriate `torch` operations to compute them instead of `numpy` ones. To avoid numerical issues you should do your checks with a network with `m=10` and `seq_length=25`.

0.5 Train your RNN using Adam

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 0) 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]` and the labels for this sequence is `book_data[e+1:e+seq_length+1]`. 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=0` then `hprev` should be the zero vector while if `e>0` 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 Adam 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 slower and you can see the smoothed loss 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. For reference the `smooth_loss` at beginning of the 3rd epoch of training, $\sim$133,000 update steps, for me was $\sim$ `1.55`.

You should also synthesize text (of length 200 characters) from your `RNN` regularly (say after every 1,000th 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[:, 0:1]` (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>len(book_data)-seq_length-1` then you should reset `e` to be 0 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 0 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=4.604919523909445

```
HFjY .)1B}d"^Ljt'ü"iui04!.'"W(Kf
_jUPhTSnpGKnUe')SoCzklq9D-H jm6Qci}3:r)l.OYk-ornyskWnüa99n•4};JvzWzqssH 1'9"?Y/C7)cugdüh!6aapOüSwT3F7:uG2G_4"s9x!ac0bHIty2d0 0_gRzk04ZE
ITbk}qFU6}T:hQ•(DyOBYc):C9HDhYM
```

iter = 1,000, smooth_loss=3.380624181004383

```
ved wasderon.. "oo"r
sob waxe nlehe singe vfsrle dnd if id ans ofe too sHerhing dinhee hoack hinprewafed ce hotak dis woat wimedhldgeY,I. jramd doent eard.kanmth srend?no
```

iter = 4,000, smooth_loss=2.2598227679427607

```
It tily."
"ht blantehe hhinn whe five hess add bony.

He," s." Heofing, cher comt twer suvery thouket llichuther. Wetsext Mrpange coemetwoo dha fioul ofon the ickearimtly of I to Harov statteapeters. e
```

iter = 30,000, smooth_loss=1.7861039794613953:

```
less at they lords, Harry punte bubben angry baster mince wroue a sawn to be shony there zork. .  stow hit lead ressered to loos.
"I ow whase, seid like, tid squacly mpeoved lark that the glorn, sum th
```

iter = 150,000, smooth_loss=1.5343960296531385:

```
ngraction thank amplewing.
"This moing the mart noomed almant wesesiven name.  You took air had bark"
Harry. . . But the toreed an hour a saver to weet Hawr down said, spook fintort.  "Sire Cumbbattani
```

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*

Improve the performance of the network

There are lots of ways the RNN you have just trained could be improved upon. Here are some avenues you can explore:

1. In the basic assignment each training sequence was visited in the same order for each epoch of training. It might be better to train with sequences from random locations in the text at each update iteration. Training in this fashion means `hprev` has to be re-set before calculating the gradients. Or one could use a compromise solution and split the book into L chunks (perhaps randomise this for each epoch) and then randomly choose the order of these chunks to use for training and then run through each chunk sequentially as in the basic assignment. For this scenario then `hprev` only has to be re-set for each chunk. I have not tried either approach so it would be interesting to see if either has any effect on speed of convergence or the results. To get a measure of performance you could set aside one chunk of the book as a validation set and use this to see if there is any quantitative difference between the predictions made on this set with the two different ways of training. (1 point)

2. In the basic assignment we effectively had batches of size 1. In conjunction with the previous approach it would be interesting to see if batches bigger than one could be used to speed up convergence (with respect to the number of update steps) and also result in a better trained model see the previous comments. (1 point)

3. When generating text people play around with how they sample from the discrete probability vector output by the RNN. One simple idea is to introduce a temperature parameter T when applying the SoftMax operator that is:

$$\tilde{\mathbf{p}}_t = \frac{\exp(\mathbf{o}_t/T)}{\mathbf{1}^T \exp(\mathbf{o}_t/T)} \tag{11}$$

Setting $T \in [0, 1)$ makes the distribution more peaky then the default SoftMax and the "peakiness" increases as T decreases. Low temperatures skew the distribution towards the high probability characters

and if you sample from $\tilde{\mathbf{p}}_t$ instead of $\mathbf{p}_t$ then you will probably increase the quality of the text generated, however, it will decrease the diversity of the text generated.

The paper The Curious Case of Neural Text Degeneration by Ari Holtzman et al., ICLR 2020 proposes a variation on this alternative called *Nucleus Sampling*. Here you define a threshold $\theta \in (0, 1]$. Let the indices $i_1, i_2, \ldots, i_K$ correspond to the indices of $\mathbf{p}_t$ when it is sorted in descending order (ie p_{t,i_1} is the highest entry in $\mathbf{p}_t$ and p_{t,i_K} is the lowest). Then find the smallest integer k_t s.t.

$$\sum_{j=1}^{k_t} p_{t,i_j} \geq \theta \tag{12}$$

then let

$$p' = \sum_{j=1}^{k_t} p_{t,i_j} \tag{13}$$

Create a new probability vector $\tilde{\mathbf{p}}_t$ to sample from during the text generation phrase by setting for $i = 1, \ldots, K$

$$\tilde{p}_{t,i} = \begin{cases} p_{t,i}/p' & \text{if } p_{t,i} \geq p_{t,i_{k_t}} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

In effect you will only sample from the top k_t probabilities. The size of the sampling set will differ from one time step to the next depending on the shape of $\mathbf{p}_t$ and the threshold θ.

You should use the two different strategies for sampling, temperature and Nucleus Sampling, with a low, medium and high values of their respective parameters T and θ and show examples of the qualitative effect on the text generated. (If you were to continue in this direction in a more quantitative manner then you would to define a quality metric such as BLEU/PERPLEXITY or human evaluation.) (2 points)

4. You can speed up your gradient computations with the following

   - Some of the computations in the forward and backward pass can be pre-computed with matrix operations and stored before entering the for loop through time and then accessing the pre-computed quantities. Similarly some operations can be computed after the `for` loop and applied as a batch matrix operation if the appropriate data is stored such as a the final output and `softmax` layers in the forward pass.

- The matrix `X` is a one-hot encoding of the character data and is very sparse. Thus in the forward and backward passes instead of computing matrix multiplications with sparse matrices one can instead use the indices of the non-zero entries of `X` to speed up computations. This is especially relevant for the gradient computations of `U`.
- In the backward pass the gradient computations for `W` require computing the outer product of two vectors. If you use `np.matmul(.,.)` to compute this outer product, replacing it with `np.outer(.,.)` should lead to speed ups.

Make these changes and report by how much you were able to speed up your code. It would be also fun to compare the speed of your fast gradient computations to those of `pytorch`. (1 point)

Bonus Points Available: You can complete any of the above suggestions and earn to a maximum of 4 bonus points. To get the bonus point you must submit:

1. Your code.
2. A pdf document reporting briefly on the upgrades you performed to the basic assignment and the results you achieved.