Neural Networks & Representation Learning (CS-587)
Assignment #4

Due on June 30, 2023

Ass. Prof. N. Komontakis

University of Crete
Department of Computer Science

Nikolaos Kougioulis (ID 1285)

June 30, 2023

Part A: RNN cells

For the first part of the programming assignment, the task is to implement an RNN cell and an RNN model.

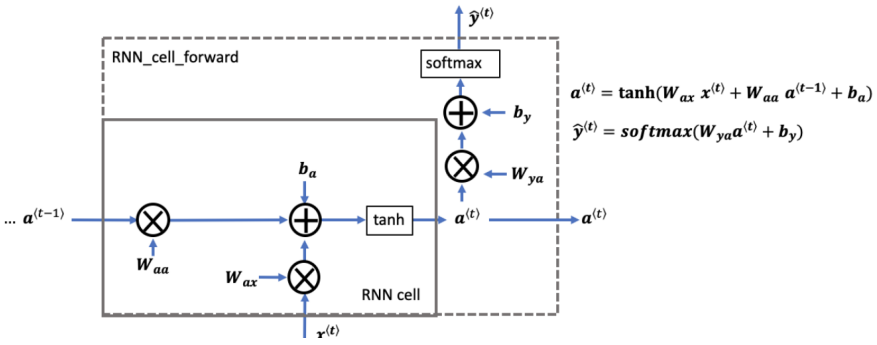

Figure 1: Illustration of the given RNN cell.

1. At this step we have to implement the forward step of the RNN cell illustrated in Figure 1. We compute the hidden state $a^{\langle t \rangle}$ (substep 1), which is the output of the RNN cell in the figure, compute the prediction `yt_pred` $\hat{y}^{\langle t \rangle}$ (substep 2) and then return the tuple $(a^{\langle t \rangle}, a^{\langle t \rangle}, x^{\langle t \rangle}, \text{parameters})$ as a python tuple (substep 3). This is illustrated in the code snippet below:

```
# 1. compute next activation state using the formula in the RNN cell figure

a_next = np.tanh(np.dot(Waa, a_prev) + np.dot(Wax, xt) + ba)

# 2. compute output of the current cell using the formula given above

yt_pred = softmax(np.dot(Wya, a_next) + by)

# 3. store values you need for backward propagation in cache

cache = (a_next, a_prev, xt, parameters)
```

2. We now need to fill-in the `rnn_forward` function to implement the forward pass of the model. We create two arrays a and y of zeros with shape (n_a, m, T_x) for storing the hidden states and (n_y, m, T_x) respectively that will store the predictions (substep 1).

```
a, y_pred = np.zeros((n_a, m, T_x)), np.zeros((n_y, m, T_x))
```

3. We initialize the 2-dimensional hidden state a_{next} by setting it equal to the initial hidden state a_0 simply by `a_next = a_0` (substep 2).

4. For the next step, again in the forward pass function, we begin by iterating through every timestep T_x (first line of code snippet). We obtain the data to be fed to the RNN cell as a slice of x, with x having a shape of (n_x, m, Tx) and update the hidden state by calling the `rnn_cell_forward` function and storing the next hidden state and currect $\hat{y}^{\langle t \rangle}$ prediction. Values are then appended to the list of cache for the backward pass.

```
    for t in range(T_x):
        a_next, yt_pred, cache = rnn_cell_forward(x[:,:,t], a_next, parameters)
        a[:,:,t], y_pred[:,:,t] = a_next, yt_pred
        caches.append(cache)
```

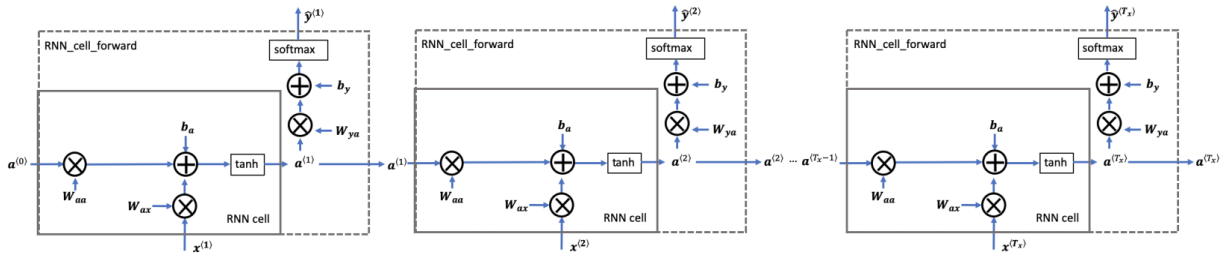

Figure 2: Illustration of the complete RNN model.

Testing the given input (hidden states and predictions) and output, it appears everything works as expected: `a[4][1] = [-0.99999375 0.77911235 -0.99861469 -0.99833267]`, `y_pred[1][3] = [0.79560373 0.86224861 0.11118257 0.81515947]`.

- **Question A:** Explain briefly the general functionality of RNN cells (how do we achieve memorization, the purpose of each function on the memorization task).

  Answer: The general functionality of RNN cells is to process sequential data by maintaining and utilizing information from previous steps or time points. In more detail, an RNN cell is define by a hidden state function

  $$h_t = \phi(x_t, h_{t-1}; w)$$

  and a prediction function $y_t = \psi(h_t; w)$ where $h_t \in \mathbb{R}^Q$, $x_t \in \mathbb{R}^D$ and w the parameters. For instance, *Elman Networks* are of the form

  $$h_t = \phi(W_h x_t + U_h y_{t-1} + b_h)$$
  $$y_t = \psi(W_y h_t + b_y)$$

  where x_t represents the input vector, h_t the hidden layer vector, y_t the output vector, W, U and b are the parameter matrices and parameter vectors respectively and ϕ, ψ the (non-linear) activation functions.

  To better understand the model, the feedback loop of the RNN cell is *unrolled*, with the length of the unrolled cell being equal to the number of time steps of the input sequence (Figure 3).

  Memorization is evident by observing the underlying temporal dependencies in the formulas of the RNN cell, as well as the illustration of the unrolled model. The hidden state of the unfolded network is formed by incorporating previous observations of the input sequence. Within each cell, the current input x, the previous hidden state h and a bias b are combined and transformed by a (non-linear) activation function to determine the current hidden state.

- **Question B:** List the limitations of RNNs, as well as try to briefly mention why these issues arise.

  Answer: Limitations of RNNs include[1]:

[1] Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." Andrej Karpathy blog, May 21, 2015. https://karpathy.github.io/2015/05/21/rnn-effectiveness/ (accessed May 15, 2023).

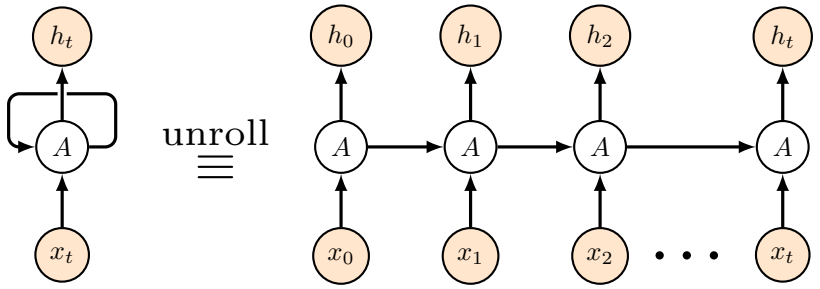

Figure 3: Unfolding a recurrent neural network. The length of the unrolled cell is equal to the time steps of the input sequence.

– Vanishing/Exploding Gradient Problem: RNNs suffer from difficulties in learning long-term dependencies due to the vanishing or exploding gradient problem. When gradients become too small or too large during backpropagation, it becomes challenging for the network to propagate useful information across long sequences.

This can be illustrated using a "vanilla" RNN model. The basic formulation of a vanilla RNN block is the following:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b) \ \forall t = 1, 2, \dots$$

Notice that the matrices W_h, W_x and the bias b are independent of t which means they are parameters shared across all timestamps of the RNN model. Without loss of generality, considering h_t and x_t to be scalar values, the gradient, which is necessary for back-propagation, of the output h_t with respect to W_h is

$$\nabla_{W_h}(h_t) = \tanh'(W_h h_{t-1} + W_x x_t + b) \cdot \frac{\partial(W_h h_{t-1} + W_x x_t + b)}{\partial W_h} \tag{1}$$

with

$$\frac{\partial(W_h h_{t-1} + W_x x_t + b)}{\partial W_h} = h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h}\,. \tag{2}$$

The form of a previous timestamp $\dfrac{\partial h_{t-1}}{\partial W_h}$ will be similar to the one of $\nabla_{W_h}(h_t)$ and after letting $W_h h_{t-1} + W_x x_t + b \equiv f_t$ we obtain:

$$\begin{aligned}
\nabla_{W_h}(h_t) &= \tanh'(f_t) \cdot \left(h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h}\right) && (3)\\
&= \tanh'(f_t) \cdot \big(h_{t-1} + W_h \cdot \tanh'(f_{t-1}) \cdot (h_{t-2} + W_h \cdot (\dots))\big) && (4)\\
&= h_{t-1} \tanh'(f_t) + h_{t-2} W_h \tanh'(f_t) \tanh'(f_{t-1}) + \dots && (5)\\
&= \sum_{t'=1}^{t-1} h_{t'} \left(W_h^{t-t'-1} \tanh'(f_{t'+1}) \cdot \dots \cdot \tanh'(f_t)\right) && (6)
\end{aligned}$$

In other words, the influence of $h_{t'}$ will be mitigated by a factor $W_h^{t-t'-1} \tanh'(f_{t'+1}) \cdot \dots \cdot \tanh'(f_t)$. Since $(\tanh x)' = 1 - \tanh^2 x$, the whole expression will result highly close to zero (thus vanish, See [2]). The above expression makes clear of the limited ability of RNNs to remember long term

[2] Romain Tavenard, Deep Learning Lecture Notes, Université de Rennes 2. Available at https://rtavenar.github.io/deep_book/book_en.pdf

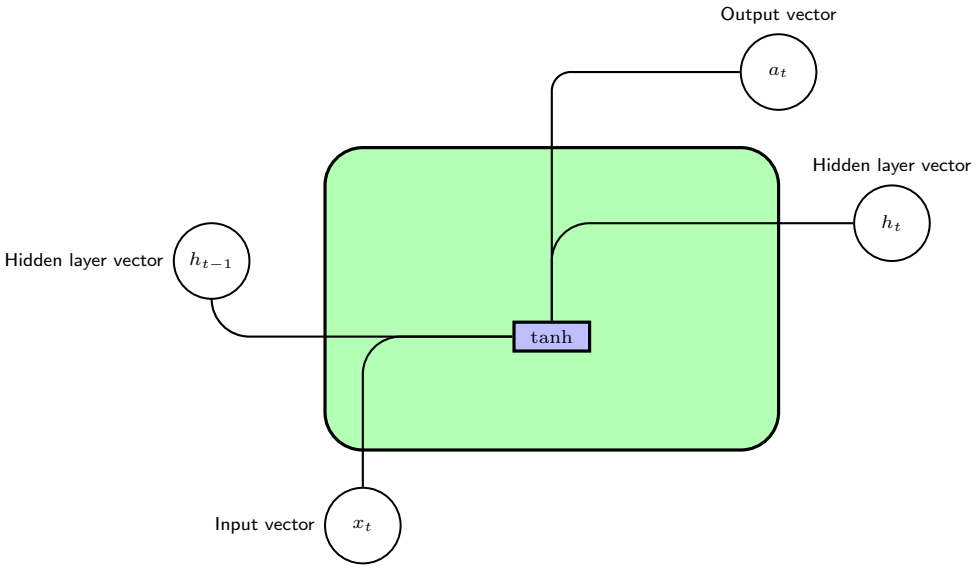

Figure 4: Illustration of an rNN cell.

dependencies from past steps in the sequence. As the length of the input increases, the ability to retain information from earlier steps vanishes.

– Another limitation of recurrent neural network architectures is the lack of parallelism. Information is processed in a sequential and serial manner, through a series of repeated layers, thus making training and inference slower. The sequential order of the execution inspired the introduction of the Attention mechanism and the transformers architecture that take the whole input sequence at once. Quoting from the 2017 paper[3] of Vaswani, Shazeer et. al:

> *Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at long sequence lengths, as memory constraints limit batching across examples.*

These issues that arise in practice can be handled by GRUs (Gated Recurrent Units) and Long Short Term Memory (LSTM) cells [4].

- **Question C:** Consider the two following problems, (a) Recognizing images between 10 different species of birds, and (b) Recognizing whether a movie review says that the movie is worth watching or not. For which of the two problems can we use a RNN-based model? Justify your answer.

Answer:

As we have seen in the previous assignment, image recognition is more appropriate to be handled by convolutional neural networks (cNNs). Due to their architecture to capture local characteristics (such

[3] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). *Attention Is All You Need*. Advances in Neural Information Processing Systems, 30. Preprint available at https://arxiv.org/pdf/1706.03762.pdf

[4] Xiao, Fanyi,. "Deep Neural Networks Basics" ECS 289G - Visual Recognition, Computer Science Department at UC Davis https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-fall2016/DNN.pdf (accessed June 2, 2023)

as edges) and proceed to capture more global and invariant characteristics at the deepest layers, their handling of hierarchical makes them more suited towards distincting between different species of birds.

For recognizing whether a movie review says that the movie is worth watching or not, RNN models (and their variations) are an appropriate option. Since movie reviews are written in text, this task requires handling sequences of words in order to model their temporal dependencies sequentially over the whole review text.

- **Question D:** While training an RNN, you observe an increasing loss trend. Upon looking you find out that at some point the weight values increase suddenly very much and finally, take the value `NaN`. What kind of learning problem does this indicate? What can be a solution? (**HINT:** answer this after you get a glimpse of Part B).

  Answer: An increasing loss trend followed by weight values suddenly becoming very large and resulting in `NaN` indicates weight values that are outside the floating-point range, thus depicts an exploding gradient problem during back-propagation. To combat this one can use gradient clipping (see Part B).

- **Question E:** Gated Recurrent Units (GRUs) have been proposed as a solution on a specific problem of RNNs. What is the problem they solve? And how?

  Answer: As mentioned previously, GRUs[5] were introduced as a solution to address the vanishing gradient problem and the challenge of capturing and retaining information of long-term dependencies in traditional RNNs.

  GRUs introduce gating mechanisms that regulate the flow of information within the recurrent units. They incorporate two main gates: the update gate and the reset gate. These gates control the flow of information from the previous hidden state and the current input to the current hidden state.

  For a fully gated unit, initially, for $t = 0$ the output vector is $h_0 = 0$. Then

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] \\
\tilde{h}_t &= \tanh(W \cdot [r_t \odot h_{t-1}, x_t] \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

  where we have two types of gates, an update gate and a reset gate (A GRU is like a long short-term memory (LSTM) with a forget gate but with fewer parameters than an LSTM, as it lacks the output gate.). The update gate determines the amount of information to be retained from the previous state, and the reset gate the amount of information to be ignored from the previous hidden state when computing the current hidden state. Variations exist, such as the minimal gated unit and the light gated recurrent unit, in the way they implement the gating (e.g the minimal gated unit merges the update and reset gate to a forget gate).

  The above allow GRUs to retain and update information from previous timesteps in a selective manner, thus effectively capturing both short-term and long-term dependencies, controlling the flow of information much more robustly than traditional RNNs.

[5] Cho, Kyunghyun; van Merrienboer, Bart; Bahdanau, DZmitry; Bougares, Fethi; Schwenk, Holger; Bengio, Yoshua (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* arXiv:1406.1078.

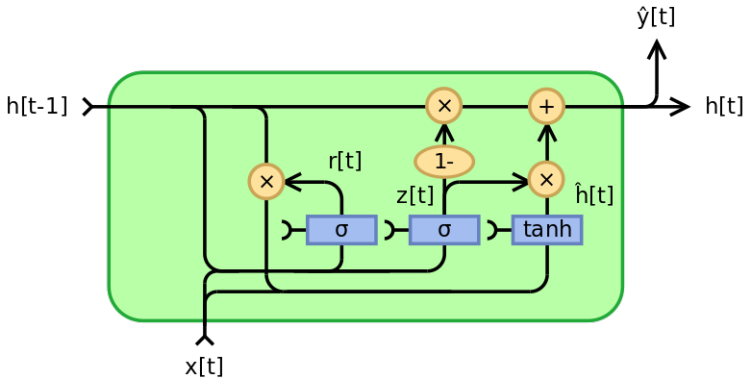

Figure 5: Illustration of a Fully Gated GRU cell.

Part B

For the second part of the assignment, as an application of the RNN model that was implemented in the previous part of the assignment, we will use it to build a character level language model to generate new text, more precisely generating chemical element names and then movie names.

We specify the corresponding path of the text file containing known chemical names using the `os` library, `path_data = os.path.join('data','element_list.txt')`.

From the provided list of element names, the splitting results in 25 characters, including spaces, special characters and the newline character, to form the chemical names. The complete RNN model handling the characters is illustrated in Figure 2. At each step, the RNN tries to predict the next candidate character. The given dataset $X = (x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, ..., x^{\langle T_x \rangle})$ corresponds to the list of characters in the training set, while $Y = (y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, ..., y^{\langle T_x \rangle})$ is such that at every time-step t, we have $y^{\langle t \rangle} = x^{\langle t+1 \rangle}$ (see the provided comments at the accompanied `.ipynb` notebook).

Regarding the coding pipeline, after loading the text file and obtaining the unique characters, we must define the forward and backward operations for constructing the model. As mentioned before, gradient clipping is utilized, leading us to answer the following questions:

- **Question A:** What is the purpose of gradient clipping? What learning limitation we are aiming to solve?

  Answer: The purpose of gradient clipping is to address the exploding gradient problem during the training of deep neural networks, particularly recurrent neural networks (RNNs). Gradient clipping is a technique that constraints the numerical magnitude of the gradients during back-propagation within an imposed threshold, preventing them from becoming too large [6] and thus facilitating the learning process. The exploding gradient problem can result in divergence from a local maxima and unstable training and as seen before, in extreme cases can lead to `NaN` weight values.

Following the rest of the guidelines for the coding pipeline, we have:

For the `rrn_step_forward` function, we define the operations for the hidden state and the prediction for every time step, as shown in Figure 1. For the backward step, in `rrn_step_backward`, the partial derivatives

[6] https://angelina-yang.medium.com/what-is-gradient-clipping-d676eeaeb294

are computed for each of the parameters $dWya, dby, da_next, db, dWax, dWaa, da_next$ and stored in a python dictionary. The gradients of this step are then used on `update_parameters` to define the updates for the parameters of the RNN cell from gradient descent, stored in a dictionary too and returned by the function. Gradient clipping is then implemented in the `clip` function by iterating with a `for` loop over the gradients of each parameter using `np.clip`[7].

Finally, in the `optimize` functions, by calling the appropriate functions according to the provided comments, we perform a forward pass through time, then a backward pass, perform gradient clipping for absolute values over 5 and update the parameters, using the functions we have filled before.

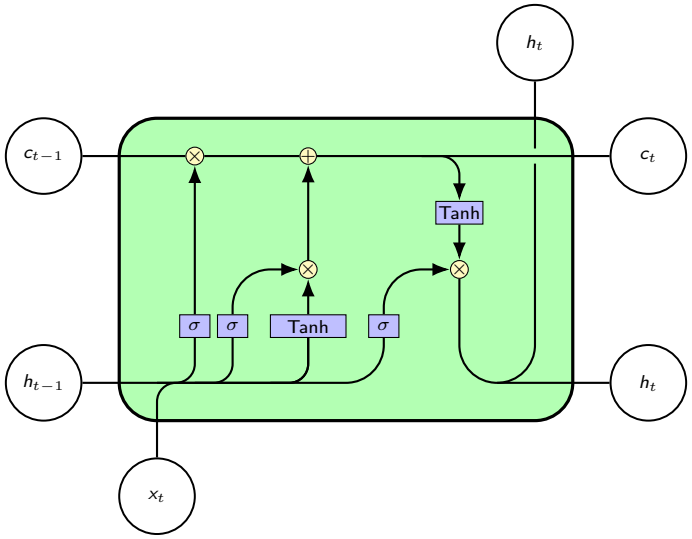

Figure 6: Illustration of an LSTM Cell.

- **Question B:** What do you observe about the new generated chemical names? How do you interpret the format of the new generated chemical element names?

  Answer: We run the model for 100k iterations, 7 samples at each iteration, 65 units at each RNN cell, with a learning rate of 0.01. Every 2000 iterations generated characters are printed. Notice that during the first iterations, the generated names do not make much sense, for example

```
Iteration: 0, Loss: 22.538572

Nkzxwucmeroeygtrw
Knea
Kzxwucmeroeygtrw
Nea
Zxwucmeroeygtrw
Ea
Xwucmeroeygtrw
```

  But as the loss gets lower, many generated names at the final iterations are plausible candidates for naming an unknown element (probably an element not found to occur naturally on Earth), for instance

```
Iteration: 98000, Loss: 10.196881

```

[7] https://numpy.org/doc/stable/reference/generated/numpy.clip.html

```
Potponium
Nohodium
Nopsenchordonium
Pelaerium
Totabtinum
Leaburm
Totabtunum
```

Notice how the names of the elements end in "um" "-ium" etc, since these are the final characters found in most elements provided in our test data.

- **Question C:** What happens when you train your model for the movie titles dataset? what do you observe and how do you interpret this observation?

Answer:

In order to train our model for the movie titles, we notice that instead of 25 we have 42 unique characters in our data, and as such must modify our code accordingly (see comments on the notebook): The parameter `vocab_size` when performing the forward pass in the function `rnn_forward` and when training the model in the function `model`, to accommodate for the new dimensions of the tensors.

As before, for 60k iterations, 10 samples at each iteration, 30 units at each RNN cell, with a learning rate of 0.01, at the first iterations it generates random names, and at the final iterations more plausible movie names, like the following:

```
Iteration: 68000, Loss: 34.098450

Onyiss of the spuathers alory lamans
Inca
Istrgand of the blandes
On -axse berro javen
The eplk grapis
Ca
The park
A
The greaundes
Aity a prathersa fuperteior die the larna the poce
```

Observe that the generated titles and "creativity" is limited to the provided list of movies, more specifically the words and phrases that appear more frequently and hence the generated titles do not always produce meaningful or plausible movie titles. Some generated names are definetely candidates for a possible movie, such as "The park", "Inca" and "Istrgrand of the blandes" (a name for a Norwegian movie about Black Metal perhaps?). This is obviously a more complex task than the previous one, as it involves more complex names and more words.

- **Question D:** Can you think of ways to improve the model for the case of the movie title dataset?

Answer: Some ways to improve the model is firstly by improving the quality of the provided dataset, by adding more disparity to the movie list. An improvement for the careful reader is to choose independent words instead of characters only, using splitting, to generate more plausible movie names using the words present in the movie list, instead of trying to construct a semantically correct word character by character. One other way, as we have seen in the previous assignments, would be to perform transfer learning for a model that has been already trained on a similar dataset (such as song names or music album titles) and then perform fine-tuning on that model. Finally, one can alter the model and choose an LSTM, as it captures long range dependencies more accurately than an RNN

model (thus generating titles of multiple words in a adequate manner) as we have already seen so far. This is also more useful and powerful when looking to generate much longer text, such as whole paragraphs of movie descriptions.

Finally, regarding the bonus part:

- b) Briefly explain what is different between a GRU cell and a Long-Short Term Memory (LSTM) cell.

  Answer: The main differences between GRUs and LSTMs are:

  GRUs use two gates, compared to an LSTM cell which has three gates (an input, a forget and an output gate). GRUs do not store memory, unlike LSTMs, as they lack and output gate. Moreover, in GRUs the reset gate is directly applied to the previous hidden state, while in an LSTM the input and forget gates are the ones responsible of the resetting (see the illustrations of the GRU and LSTM cells in Figures 5 and 6).

- a) The implementation of the GRU cell, on Figure 5, and the formulas defined in page 6, we have:

```
    # 1.  Concatenate a_prev and xt (3 lines)
    concat = np.concatenate((a_prev, xt), axis=0)

    # 2. Compute values for zt, rt, ht_sl, a_next using the formulas given figure  (6
    lines)
    zt = sigmoid(np.dot(Wz, concat))
    rt = sigmoid(np.dot(Wr, concat))
    ht_sl = np.tanh(np.dot(Wh, np.concatenate((rt * a_prev, xt), axis=0)))
    a_next = (1 - zt) * a_prev + zt * ht_sl

    # 3. Compute prediction of the GRU cell (1 line)
    yt_pred = np.dot(Wy, a_next) + by
```

References

[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.