CS587 - Neural Networks & Learning of Hierarchical Representation

Spring Semester 2024
Assignment 5

Papageridis Vasileios - 4710
csd4710@csd.uoc.gr

June 15, 2024

Introduction

This assignment aims to teach us about RNN models. More specifically we will implement a RNN cell and the key functionalities of an RNN model. In this way we will gain intuition about how a RNN actually works and the advantages that RNN models offer in problems such as text generation, query answering, video analysis etc.
In the first part of this assignment, *RNN cells*, we create a RNN cell from scratch. This task will be achieved only by using basic numpy and python functions in order to build the cell's core functionalities. At the end of this part, we answer some theoretical questions in order to confirm our knowledge and that we fully understand how an RNN cell works.
In the second part, we will use the RNN model created in *Part A* to create a character-level language model that is going to generate new text. The algorithm will learn the text patterns in our training dataset and generate new text randomly. We will use datasets that include names of existing chemical elements and a list of movie titles. The model's task is to create plausible new names for chemical elements and generate new movie titles. The pipeline of the algorithm includes loading the names, splitting them into single characters, and identifying unique characters to define the alphabet. We will define the model, including forward and backward propagation operations, to establish the loss function and compute gradients for updating the model's parameters. Additionally, we will implement gradient clipping to ensure the model learns effectively and generalizes well.

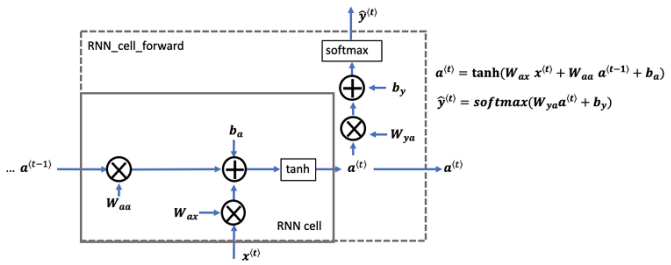

Figure 1: A vanilla RNN cell.

Part A: RNN Cells

The goal of this part of the assignment is to implement the Recurrent Neural Network (RNN) cell, as described in Figure 2 of the assignment. We aim to:

- Implement the forward step of the RNN cell.
- Understand the functionality, limitations, and advantages of RNN cells.

In this section we are just going to explain the implementation steps, which all can be found in the corresponding *.ipynb* file of this assignment.

Implementation Steps

Forward Step of the RNN Cell

The forward step of the RNN cell involves the following key computations:

- **Hidden State**: The hidden state $a^{\langle t \rangle}$ is calculated using the current input $x^{\langle t \rangle}$, the previous hidden state $a^{\langle t-1 \rangle}$, and the parameters W_{ax}, W_{aa}, and b_a. All of this is done using the $\tanh$ activation function.
- **Prediction**: The prediction $\hat{y}^{\langle t \rangle}$ is computed using the hidden state $a^{\langle t \rangle}$ and the parameters W_{ya} and b_y, followed by a softmax function.
- **Storing Values**: The values of $a^{\langle t \rangle}$, $a^{\langle t-1 \rangle}$, $x^{\langle t \rangle}$, and the parameters are stored in a tuple for use in backward propagation.

0.0.1 RNN Forward Pass

The forward pass of the RNN uses the cell forward function and the steps that we had to implement are:

- **Initialize Storage Arrays**: Arrays are created to store the hidden states and predictions across all time steps.
- **Initialize Hidden State**: The initial hidden state a_{next} is set to a_0.
- **Iterate Through Time Steps**: For each time step t:
  - The we obtain the input $x^{\langle t \rangle}$.
  - The RNN cell forward function is called to compute the next hidden state $a^{\langle t \rangle}$ and prediction $\hat{y}^{\langle t \rangle}$.
  - The hidden state and prediction are stored in the arrays.
  - Values needed for backward propagation are saved in a list of caches.

Questions Answers

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:
Recurrent Neural Network (RNN) cells are designed to process sequences of data by maintaining a hidden state that captures information from previous time steps. This capability allows RNNs to model temporal dependencies and patterns in sequential data. In order to answer the questions we have to explain some parts of a RNN cell a little bit more:

1. **Hidden State ($a^{(t)}$)**:
    - It's a vector that stores information from the previous time steps.
    - It is used as the memory of the cell, because it captures information of the sequence up to the current time step.
2. **Input ($x^{(t)}$)**:
    - The input at the current time step t.
3. **Weights and Biases**:
    - W_{ax}: Weight matrix for the input $x^{(t)}$.
    - W_{aa}: Weight matrix for the hidden state $a^{(t-1)}$.
    - b_a: Bias vector for the hidden state computation.
4. **Output ($\hat{y}^{(t)}$)**:
    - The predicted output at time step t.

How RNN Cells Achieve Memorization
RNN cells achieve memorization by updating their hidden state $a^{(t)}$ at each time step using the equation $a^{(t)} = \tanh(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b_a)$. This is done by using the previous hidden state $a^{(t-1)}$, which acts as a memory as we said earlier, so we can use the information from the previous time steps and then combine them with the current input $x^{(t)}$. The $\tanh$ function keeps the hidden state values within a specific range, stabilizing the learning process. The updated hidden state $a^{(t)}$ is then used to compute the output $\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$. This process enables the RNN to capture and utilize temporal dependencies in sequential data.

Purpose of Each Function in the Memorization Task

- **Hidden State Update Function ($\tanh$)**: It takes the input and previous hidden state to update the current hidden state, allowing the RNN to remember information from previous time steps.
- **Softmax Function**: Converts the hidden state into a probability distribution, in order to make the prediction.

In summary, RNN cells achieve memorization by maintaining and updating a hidden state that captures information from previous inputs. The hidden state, combined with the current input, allows the RNN to retain context and make predictions.

Question B: List the limitations of RNNs, and briefly mention why these issues arise.
Answer:

1. **Vanishing and Exploding Gradients**:
    - **Description**: Gradients can become very small (vanishing gradients) or very large (exploding gradients) during training.
    - **Cause**: Repeated multiplication of gradient terms over many time steps leads to exponential shrinkage or growth.
    - **Impact**: Vanishing gradients block learning long-range dependencies, while exploding gradients cause instability.
2. **Difficulty in Capturing Long-Term Dependencies**:
    - **Description**: RNNs struggle to maintain information over long sequences.
    - **Cause**: Vanishing gradients prevent propagation of information from earlier to later time steps.
    - **Impact**: Limits the model's ability to learn long-term patterns.
3. **Training Instability and Slow Convergence**:
    - **Description**: Training RNNs can be unstable and slow.
    - **Cause**: Issues with gradients and the sequential nature of RNNs.
    - **Impact**: Requires significant computational resources and careful tuning of hyperparameters.
4. **Limited Parallelization**:
    - **Description**: RNNs process data sequentially.
    - **Cause**: Hidden states depend on previous time steps.
    - **Impact**: Limits the use of parallel computing resources, slowing training and inference.

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:

RNNs are appropriate for problem (b) (Recognizing whether a movie review says that the movie is worth watching or not) because they are designed to process and understand sequential data like text, capturing temporal dependencies and context. For problem (a) (Recognizing images between 10 different species of birds), CNNs are the better choice due to their effectiveness in handling spatial data and image features.

- **Nature of the Problem**: Sentiment analysis of movie reviews to classify them as positive (worth watching) or negative (not worth watching).
- **Appropriate Model**: Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, or Gated Recurrent Units (GRUs).
- **Justification**: RNNs are suitable for text processing tasks due to their ability to capture temporal dependencies and context in sequential data like sentences and paragraphs. They can remember the context and previous words, making informed predictions about the sentiment of the review.

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:

This scenario is the so-called exploding gradient problem. During the backpropagation through time, gradients are propagated backwards through many time steps. When gradients are repeatedly multiplied by values greater than one, they can grow exponentially, resulting in very large gradient values. This leads to large updates to the network's weights, causing numerical instability and overflow. When the weights are updated by these large gradients, they can become extremely large, leading to numerical instability. Eventually, the weights can overflow, resulting in NaN values in the parameters and loss function.

Possible solution

One effective method to address the exploding gradient problem is gradient clipping, which sets a threshold value for the gradients. During backpropagation, if the gradients exceed this threshold, they are rescaled to ensure they remain within the specified range. This prevents the gradients from becoming excessively large and helps maintain numerical stability. Another solution could be to use advanced RNN architectures like LSTMs or GRUs, which are designed to prevent scenarios like vanishing or exploding gradients. Lastly, we could use an optimizer that provide more robust training and stable gradients like Adam, so we can minimize the probabilities to get exploding gradients during training.

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:
Gated Recurrent Units (GRUs) give solutions to the vanishing and exploding gradients problems. Those are one of the limitations of the vanilla RNN models and prevents them to learn long-term dependecies in sequential data. GRUs add gating mechanisms that adjust the flow of information through the network, minimizing the vanishing and exploding gradient problems. In order to understand how GRUs solve those problems, we will briefly describe the gate mechanism of those models:

- **Update Gate (z_t)**:
  - **Purpose**: Controls how much of the previous hidden state is passed to the current hidden state.
  - **Equation**: $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$
- **Reset Gate (r_t)**:
  - **Purpose**: Determines how much of the previous hidden state should be forgotten.
  - **Equation**: $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
- **Candidate Hidden State ($\tilde{h}_t$)**:
  - **Purpose**: Represents the new hidden state information to be added.
  - **Equation**: $\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$
- **Final Hidden State (h_t)**:
  - **Purpose**: The actual hidden state passed to the next time step.
  - **Equation**: $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$

These gates help retain relevant information over long time steps and discard irrelevant information, leading to improved learning of long-term dependencies and more stable training.

Part B: Text Generation using a RNN model

In this part of the assignment, the goal was to use the RNN model implemented in Part A to build a character-level language model that will be generating new text. The model was trained to learn text patterns from datasets containing names of chemical elements and movie titles, and generate new names and titles.

Questions Answers

Question A: Include the computational graph of the RNN cell in your report, along with the extensive mathematical derivation process you used to get the gradients.
Answer:
The computational graph of a RNN cell as given in the exercise is the one below:

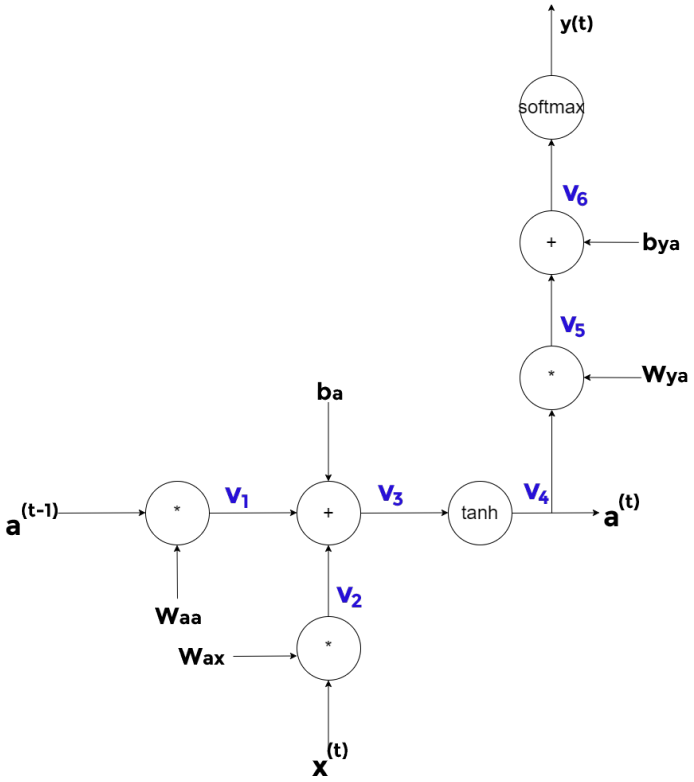

Figure 2: Computation graph of a RNN cell.

First of all we are going to define the forward pass based on the graph, as we have done in previous assignments:

Forward Pass

$$
\begin{aligned}
v_1 &= a^{(t-1)} \cdot W_{aa} \\
v_2 &= x^{(t)} \cdot W_{ax} \\
v_3 &= v_1 + v_2 + b_a \\
v_4 &= \tanh(v_3) \\
v_5 &= v_4 \cdot W_{ya} \\
v_6 &= v_5 + b_{ya} \\
v_7 &= \text{softmax}(v_6)
\end{aligned}
$$

Then the backward pass can be calculated as:

Backward Pass

We define: $\bar{v} = \frac{\partial f}{\partial v} \cdot f$

$$
\begin{aligned}
\bar{v}_6 &= \frac{\partial L}{\partial v_6} = dy \\
\bar{v}_5 &= \frac{\partial v_6}{\partial v_5} = \frac{\partial v_6}{\partial v_5} \cdot \bar{v}_6 = 1 \cdot \bar{v}_6 = dy \\
\bar{v}_4 &= \frac{\partial v_5}{\partial v_4} \cdot \bar{v}_5 = \frac{\partial v_5}{\partial v_4} \cdot \bar{v}_5 = W_{ya} \cdot dy \\
\bar{v}_3 &= \frac{\partial L}{\partial v_3} \cdot \bar{v}_4 = \frac{\partial v_4}{\partial v_3} \cdot \bar{v}_4 = (1 - v_3^2) \cdot W_{ya} \cdot dy \\
\bar{v}_2 &= \frac{\partial L}{\partial v_2} \cdot \bar{v}_3 = \frac{\partial v_3}{\partial v_2} \cdot \bar{v}_3 = (1 - v_3^2) \cdot W_{ya} \cdot dy \\
\bar{v}_1 &= \frac{\partial v_3}{\partial v_1} \cdot \bar{v}_3 = \frac{\partial v_3}{\partial v_1} \cdot \bar{v}_3 = (1 - v_4^2) \cdot W_{ya} \cdot dy
\end{aligned}
$$

We have that:

$$da\_next = W_{ya} \cdot dy$$

$$
\begin{aligned}
\bar{W}_{ax} &= \frac{\partial v_2}{\partial W_{ax}} \cdot \bar{v}_2 = x^{(t)} \cdot (1 - v_3^2) \cdot W_{ya} \cdot dy \\
\bar{x}^{(t)} &= \frac{\partial v_2}{\partial x^{(t)}} \cdot \bar{v}_2 = W_{ax} \cdot (1 - v_3^2) \cdot W_{ya} \cdot dy \\
\bar{W}_{aa} &= \frac{\partial v_1}{\partial W_{aa}} \cdot \bar{v}_1 = a^{(t-1)} \cdot (1 - v_3^2) \cdot W_{ya} \cdot dy \\
\bar{W}_{ya} &= \frac{\partial v_5}{\partial W_{ya}} \cdot \bar{v}_5 = v_4 \cdot dy = tanh(v_3) \cdot dy \\
\bar{a}^{(t-1)} &= \frac{\partial v_1}{\partial a^{(t-1)}} \cdot \bar{v}_1 = W_{aa} \cdot \bar{v}_1 \\
\bar{b}_{ya} &= \frac{\partial v_6}{\partial b_{ya}} \cdot \bar{v}_6 = 1 \cdot dy = dy
\end{aligned}
$$

$$\bar{b}_a = \frac{\partial v_3}{\partial b_a} \cdot \bar{v}_3 == 1 \cdot (1 - v_3^2) \cdot W_{ya} \cdot dy$$

Additional notes:
We use the notation dy for the gradient of the $softmax$ function, in order to simplify our calculations.
We also compute the backpropagation for a single RNN cell. In the next paragraphs we explain how backpropagtion actually works for all the cells of a RNN model.

Backpropagation through time in a complete RNN
At the final time step T, the objective function L is dependent on the hidden state h_T solely through o_T. Thus, we can determine the gradient $\frac{\partial L}{\partial h_T} \in \mathbb{R}^h$ by applying the chain rule:

$$\frac{\partial L}{\partial h_T} = \prod \left(\frac{\partial L}{\partial o_T}, \frac{\partial o_T}{\partial h_T} \right) = W_{qh}^T \frac{\partial L}{\partial o_T}$$

For any time step $t < T$, where L depends on h_t via both h_{t+1} and o_t, the gradient $\frac{\partial L}{\partial h_t} \in \mathbb{R}^h$ is computed recursively using the chain rule:

$$\frac{\partial L}{\partial h_t} = \prod \left(\frac{\partial L}{\partial h_{t+1}}, \frac{\partial h_{t+1}}{\partial h_t} \right) + \prod \left(\frac{\partial L}{\partial o_t}, \frac{\partial o_t}{\partial h_t} \right) = W_{hh}^T \frac{\partial L}{\partial h_{t+1}} + W_{qh}^T \frac{\partial L}{\partial o_t}$$

Expanding this computation for any time step $1 \leq t \leq T$ gives:

$$\frac{\partial L}{\partial h_t} = \sum_{i=t}^{T} \left(W_{hh}^T \right)^{T-i} W_{qh}^T \frac{\partial L}{\partial o_{T+t-i}}$$

These equations illustrate how gradients are propagated back through the network over multiple time steps, enabling the training of the entire RNN model using BPTT. This applies not just to a single RNN cell, but to the entire sequence of cells in the RNN.

Softmax gradient: In order to gain more intuition about the computations we will briefly explain the computation of the gradient of the softmax function. In our case, we use just the notation dy in order to simplify our calculations. When discussing the derivative of the softmax function, we refer to its Jacobian matrix. Let $\mathbf{z} = [z_1, z_2, \ldots, z_n]$ be the input vector and $\mathbf{s} = [s_1, s_2, \ldots, s_n]$ be the output vector where each s_i is given by:

$$s_i = \frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}, \quad \forall i = 1, \ldots, n$$

The Jacobian matrix J_{softmax} is defined as:

$$J_{\text{softmax}} = \begin{pmatrix} \frac{\partial s_1}{\partial z_1} & \frac{\partial s_1}{\partial z_2} & \cdots & \frac{\partial s_1}{\partial z_n} \\ \frac{\partial s_2}{\partial z_1} & \frac{\partial s_2}{\partial z_2} & \cdots & \frac{\partial s_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s_n}{\partial z_1} & \frac{\partial s_n}{\partial z_2} & \cdots & \frac{\partial s_n}{\partial z_n} \end{pmatrix}$$

Each element of the Jacobian matrix can be computed as follows. First, consider the partial derivative of s_i with respect to z_j:

$$\frac{\partial s_i}{\partial z_j} = s_i \left(\delta_{ij} - s_j \right)$$

where δ_{ij} is the Kronecker delta, which is 1 if $i = j$ and 0 otherwise.
To derive this, note that:

$$\log s_i = z_i - \log \left(\sum_{l=1}^{n} e^{z_l} \right)$$

Taking the partial derivative with respect to z_j:

$$\frac{\partial}{\partial z_j} \log s_i = \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} \log \left(\sum_{l=1}^{n} e^{z_l} \right)$$

$$\frac{\partial}{\partial z_j} \log s_i = \delta_{ij} - \frac{e^{z_j}}{\sum_{l=1}^{n} e^{z_l}} = \delta_{ij} - s_j$$

Since:

$$\frac{\partial \log s_i}{\partial z_j} = \frac{1}{s_i} \frac{\partial s_i}{\partial z_j}$$

We have:

$$\frac{\partial s_i}{\partial z_j} = s_i \left(\delta_{ij} - s_j \right)$$

Therefore, the Jacobian matrix for the softmax function can be written as:

$$J_{\text{softmax}} = \begin{pmatrix} s_1(1 - s_1) & -s_1 s_2 & \cdots & -s_1 s_n \\ -s_2 s_1 & s_2(1 - s_2) & \cdots & -s_2 s_n \\ \vdots & \vdots & \ddots & \vdots \\ -s_n s_1 & -s_n s_2 & \cdots & s_n(1 - s_n) \end{pmatrix}$$

In summary, the Jacobian matrix J_{softmax} consists of diagonal elements $s_i(1 - s_i)$ and off-diagonal elements $-s_i s_j$, reflecting the dependency of each softmax output on all input values.

Question B: What is the purpose of gradient clipping? What learning limitation we are aiming to solve?
Answer:
Gradient clipping is a technique used during the training of neural networks to tackle the issue of exploding gradients. The primary purpose of gradient clipping is to prevent gradients from becoming excessively large.
Exploding gradients occur when the gradients of the loss function with respect to the model parameters become excessively large during backpropagation. This can lead to very large updates to the model's weights, causing instability and potential overflow, resulting in NaN values. So in order to avoid this problem and achieve stable training, gradient clipping sets a threshold for the gradients. During backpropagation, if gradients exceed this threshold, they are rescaled to remain within the specified range, preventing excessively large gradients.
How gradient clipping works:

1. **Calculate Gradients**: Compute the gradients of the loss function with respect to the model parameters.
2. **Check Gradient Norm**: Calculate the norm of the gradients.
3. **Rescale Gradients**: If the gradient norm exceeds a threshold, rescale all gradients by a factor that ensures the norm equals the threshold.
4. **Update Parameters**: Use the clipped gradients to update the model parameters.

Question C: What do you observe about the new generated chemical names? How do you interpret the format of the new generated chemical element names?
Answer:

Observations

- **Iteration 2000, Loss: 19.119649**
  - Names such as "Metsroen", "Ium", "Iumium", "Mabadosin" start to show some basic structure.
  - Some names are repetitive or simple, indicating initial learning.
- **Iteration 4000, Loss: 16.106072**
  - Names like "Onustokmerium", "Lild", "Lutrofkium" display increased complexity and variety.
  - The names seem more cohesive and show emerging patterns typical of chemical elements.
- **Iteration 6000, Loss: 13.883858**
  - Names like "Piun", "Nelc", "Otroum" continue to evolve in complexity.
  - The model is generating shorter and more varied names, indicating improved learning.

- **Iteration 8000, Loss: 12.746395**
  - Names such as ”Phturoneghner”, ”Ondc”, ”Ostrulanomerium” demonstrate high complexity and length.
  - Some names are excessively long, suggesting the model might be overfitting.
- **Iteration 10000, Loss: 12.763164**
  - Names like ”Piur”, ”Ondcantum”, ”Otium” show a mix of reasonable and complex structures.
  - The model appears to balance between generating realistic names and overfitting.
- **Later Iterations (14000 to 70000)**
  - Names display varying complexity and length.
  - The loss fluctuates, indicating the model might be struggling with overfitting and stability.
  - Examples: ”Plutonerbordun”, ”Liumium”, ”Nivossanesmetesium”, ”Inmrclenaniovenium”.
  - The generated names often have familiar prefixes and suffixes but sometimes include nonsensical patterns.

Interpretation

- **Initial Learning Phase (Up to Iteration 2000)**
  - The model begins by learning basic patterns and structures from the training data.
  - Names are simple, repetitive, and show initial signs of capturing the format of chemical names.
- **Mid Learning Phase (Iterations 4000 to 10000)**
  - The model improves in generating more complex and varied names.
  - The generated names increasingly resemble real chemical element names, both in structure and length.
  - There is a balance between overfitting and generating realistic names, as seen by the variety and complexity.
- **Advanced Learning Phase (Beyond Iteration 10000)**
  - The model starts to overfit, generating excessively complex or nonsensical names.
  - The fluctuation in loss indicates instability in learning.
  - Despite overfitting, some names still follow realistic patterns, showing that the model has captured significant aspects of the training data.

Conclusion

As we can observe initially, the names are simple and repetitive, but as training progresses, the names become more complex and varied. The mid-phase names are the most realistic, indicating a good balance in learning. However, as training continues, the model starts to overfit, generating overly complex or nonsensical names. This suggests that while the RNN has learned the general structure of chemical names, it needs adjustments to prevent overfitting and maintain stability.

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

Answer:

Observations

- **Iteration 0, Loss: 26.171162**
  - Generated titles are mostly random sequences of characters.
  - Examples: ”Kféyxt0i3ol2é?rox rdecwt”, ”Féyxt0i3ol2é?rox rdecwt”, ”Éyxt0i3ol2é?rox rdecwt”.
  - The titles show no meaningful structure or resemblance to movie titles.
- **Iteration 2000, Loss: 43.591103**
  - Titles start to show some recognizable patterns but still appear largely nonsensical.
  - Examples: ”Liwtor meon werot”, ”Iwtor meon werot”, ”Wstoe doofthhes”.
  - Some titles show partial words but lack coherence.
- **Iteration 4000, Loss: 42.182893**
  - Some titles begin to resemble phrases or word structures.
  - Examples: ”Olxtor kerkaverno’the to”, ”Kif 2boul”, ”Lvous”.
  - Titles have more recognizable words but still lack overall meaning.
- **Iteration 6000, Loss: 40.640555**
  - Titles start to look more like potential movie titles.
  - Examples: ”Ontrsife rigvenet”, ”Lie”, ”Mutrighape werst”.
  - The structure improves, and some titles almost make sense.
- **Iteration 8000, Loss: 40.122038**
  - Titles show further improvement in structure and coherence.
  - Examples: ”Onotrn”, ”Lia”, ”Moton”.
  - Titles are shorter and more varied, indicating learning progress.
- **Iterations 10000 to 68000**

  - Titles vary significantly in coherence and structure.
  - Examples from later iterations: ”Metowtanare u irt sin or”, ”Havuthesesisthest thesthesthise”.
  - Titles sometimes regress to randomness, showing fluctuating learning stability.

Interpretation

- **Initial Phase (Up to Iteration 2000)**
  - The RNN model begins with a high degree of randomness. Titles are essentially random character sequences.
  - The loss is initially high, reflecting the model's unfamiliarity with the data patterns.
- **Early Learning (Iterations 2000 to 6000)**
  - The model starts to learn basic patterns and structures. Titles begin to show word-like formations.
  - There is a significant drop in loss, indicating that the model is learning and improving.
- **Mid Learning (Iterations 6000 to 10000)**
  - The titles become more structured and start to resemble real movie titles.
  - The learning progress is evident as the loss continues to decrease, and the generated titles make more sense.
- **Later Learning (Iterations 10000 to 68000)**
  - Titles fluctuate between coherent and nonsensical, suggesting overfitting and instability.
  - The loss increases and decreases irregularly, indicating that the model struggles to generalize well and maintain stable learning.
  - The generated titles sometimes regress to randomness or repetitive patterns, such as ”thesthesthise”, showing potential overfitting and inability to generalize properly.

Question E: Can you think of ways to improve the model for the case of the movie title dataset?
Answer:

1. **Use of LSTM or GRU Cells**: Replace the simple RNN cells with Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells so we can handle long-term dependencies better and prevent the vanishing and exploding gradient problems.
2. **Increase the Number of Units**: Increase the number of units n_a in the RNN cells from 100 to a higher value (e.g., 256 or 512). This can provide the model with more capacity to capture complex patterns in the data.
3. **Regularization**: Regularization methods such as dropout or L2 regularization to prevent overfitting and improve generalization.

4. **Batch Training**: Instead of updating the model parameters after every single training example, we could use mini-batch training to provide more stable updates and faster convergence.

5. **Data Augmentation**: Increase the variability in the training data by slightly modifying the movie titles, such as by adding synonyms or altering punctuation.

6. **Early Stopping**: We could use early stopping to stop the training process once the validation loss stops improving.

7. **Fine-tuning Hyperparameters**: In order to find the optimal settings for the model.

Bonus:

Part (a) of the bonus section is included in the corresponding *.ipynb* file and is fully implemented.

(b): Differences Between GRU and LSTM Cells

Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) cells are advanced Recurrent Neural Networks (RNNs) designed to address the vanishing and exploding gradient problems. The main differences between those 2 types of RNNs are:

Structure

- **Number of Gates**:
  - **LSTM**: Three gates—input gate, forget gate, and output gate.
  - **GRU**: Two gates—reset gate and update gate.
- **Cell State**:
  - **LSTM**: Maintains a separate cell state (C_t) in addition to the hidden state (h_t).
  - **GRU**: Only maintains a hidden state (h_t).

Gate Functions and Operations

- **LSTM Gates**:
  - **Input Gate**:
    $$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
  - **Forget Gate**:
    $$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
  - **Output Gate**:
    $$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
- **GRU Gates**:
  - **Update Gate**:
    $$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$
  - **Reset Gate**:
    $$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

Hidden State Update

- **LSTM**:
  - **Cell State Update**:

$$C_t = f_t * C_{t-1} + i_t * \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

  - **Hidden State Update**:

$$h_t = o_t * \tanh(C_t)$$

- **GRU**:
  - **Candidate Hidden State**:

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

  - **Hidden State Update**:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Computational Efficiency

- **LSTM**: More parameters and computation due to its complex structure.
- **GRU**: Simpler and faster to train due to fewer parameters.

Summary

- **LSTM Cells**:
  - Three gates (input, forget, output) and maintain both a cell state and a hidden state.
  - More powerful but computationally heavier.
  - Better at capturing long-term dependencies.
- **GRU Cells**:
  - Two gates (reset, update) and maintain only a hidden state.
  - Simpler and computationally more efficient.
  - Often perform comparably to LSTMs with fewer parameters.

Both GRUs and LSTMs address the limitations of standard RNNs by providing mechanisms to control the flow of information, thereby improving the ability to learn from long sequences. Below we present 2 figures that illustrate the structure of each cell, in order to gain more intuition about them.

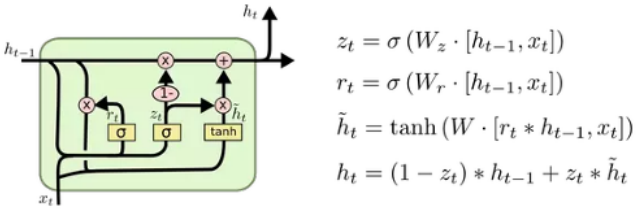

Figure 3: GRU Cell Structure

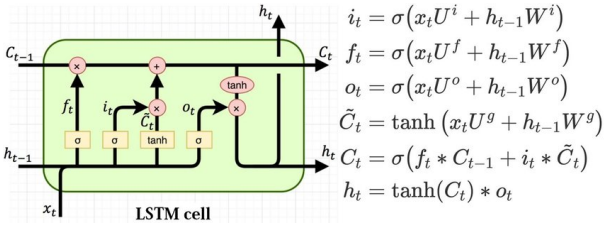

Figure 4: LSTM Cell Structure

References

[1] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*, ICLR Workshop 2014, https://arxiv.org/pdf/1312.6034.

[2] Russakovsky, O., Deng, J., Su, H., *ImageNet Large Scale Visual Recognition Challenge*, International Journal of Computer Vision (IJCV). Vol 115, Issue 3, 2015, pp. 211-252, https://arxiv.org/pdf/1409.0575.

[3] Simon J.D. Prince, *Understanding Deep Learning*, MIT Press, 2023, https://udlbook.github.io/udlbook/.

[4] Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015, http://neuralnetworksanddeeplearning.com/.