1 Question 1: Implementing RNN from scratch [70 marks]

Recurrent Neural Networks (RNNs) represent a class of neural networks crafted specifically for processing sequential data, such as time series or natural language text. They accomplish this feat by incorporating a hidden state that undergoes updates for each time step of the input sequence, thereby enabling the network to retain a memory of prior inputs.

For this problem, you are supposed to implement an RNN from scratch that is tailored for language modelling. It takes in a series of characters and produces a probability distribution over the subsequent characters in the sequence. The network undergoes training on a corpus of text and subsequently serves to generate new text that exhibits a comparable distribution of characters to the original training corpus.

1.1 RNN Implementation

You are expected to build a basic one-layer **RNN** for this problem. It consists of an input layer, a hidden layer, and an output layer. The input layer takes in a one-hot encoded vector representing a character in the input sequence. This vector is multiplied by a weight matrix $W_{\mathbf{ax}}$ to produce a hidden state vector $\mathbf{a}$. The hidden state vector is then passed through a non-linear activation function (in this case, the hyperbolic tangent function) and updated for each time step of the input sequence. The updated hidden state is then multiplied by a weight matrix $W_{\mathbf{ya}}$ to produce the output probability distribution over the next character in the sequence.

The **RNN** is trained using stochastic gradient descent with the cross-entropy loss function. During training, the self takes in a sequence of characters and outputs the probability distribution over the next character. The true next character is then compared to the predicted probability distribution, and the parameters of the network are updated to minimize the cross-entropy loss.

1.2 Activation Functions

Softmax Activation Function

The softmax function is commonly employed as an activation function in neural networks, especially in the output layer for classification tasks. Given an input array x, the softmax function calculates the probability distribution of each element in the array according to the formula:

$$\text{softmax}_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Tanh Activation

The hyperbolic tangent activation function is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

where x represents the input to the function. The output of the function falls within the range of -1 to 1. Tanh activation is frequently utilized in neural networks as an alternative to the sigmoid activation function due to its steeper gradient, enabling it to better capture non-linear relationships within the data.

1.3 Forward propagation

During forward propagation, the input sequence is processed through the RNN to generate an output sequence. At each time step, the hidden state and the output are computed using the input, the previous hidden state, and the RNN's parameters.

The equations for forward propagation in a basic RNN are as follows:

At time step t, the input to the RNN is x_t, and the hidden state at time step $t-1$ is a_{t-1}. The hidden state at time step t is computed as:

$$a_t = \tanh(W_{aa}a_{t-1} + W_{ax}x_t + b_a)$$

where W_{aa} is the weight matrix for the hidden state, W_{ax} is the weight matrix for the input, and b_a is the bias vector for the hidden state.

The output at time step t is computed as:

$$y_t = \text{softmax}(W_{ya} a_t + b_y)$$

where W_{ya} is the weight matrix for the output, and b_y is the bias vector for the output.

1.4 Backward propagation

The objective of training an RNN is to minimize the loss between the predicted sequence and the ground truth sequence. Backward propagation calculates the gradients of the loss with respect to the RNN's parameters, which are then used to update the parameters using an optimization algorithm such as Adagrad or Adam.

The equations for backward propagation in a basic RNN are as follows:

At time step t, the loss with respect to the output y_t is given by:

$$\frac{\partial L}{\partial y_t} = \begin{cases} -\frac{1}{y_{t,i}} & \text{if } i = t_i \\ 0 & \text{else} \end{cases}$$

where L is the loss function, $y_{t,i}$ is the i-th element of the output at time step t, and t_i is the index of the true label at time step t.

The loss with respect to the hidden state at time step t is given by:

$$\frac{\partial L}{\partial a_t} = \frac{\partial L}{\partial y_t} W_{ya} + \frac{\partial L}{\partial h_{t+1}} W_{aa}$$

where $\frac{\partial L}{\partial a_{t+1}}$ is the gradient of the loss with respect to the hidden state at the next time step, which is back-propagated through time.

The gradient with respect to tanh is given by: $\frac{\partial \tanh(a)}{\partial a}$.

The gradients with respect to the parameters are then computed using the chain rule:

$$\frac{\partial L}{\partial W_{ya}} = \sum_t \frac{\partial L}{\partial y_t} a_t$$

$$\frac{\partial L}{\partial b_y} = \sum_t \frac{\partial L}{\partial y_t}$$

$$\frac{\partial L}{\partial W_{ax}} = \sum_t \frac{\partial L}{\partial a_t} \frac{\partial a_t}{\partial W_{ax}}$$

$$\frac{\partial L}{\partial W_{aa}} = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_{aa}}$$

$$\frac{\partial L}{\partial b_a} = \sum_t \frac{\partial L}{\partial a_t} \frac{\partial h_t}{\partial b_a}$$

where $\frac{\partial h_t}{\partial W_{ax}}$, $\frac{\partial a_t}{\partial W_{aa}}$, and $\frac{\partial h_t}{\partial b_a}$ can be computed as:

$$\frac{\partial a_t}{\partial W_{ax}} = x_t$$

$$\frac{\partial a_t}{\partial W_{aa}} = a_{t-1}$$

$$\frac{\partial a_t}{\partial b_a} = 1$$

These gradients are then used to update the parameters of the RNN using an optimization algorithm such as gradient descent, Adagrad, or Adam. We will be using Adam for this lab.

1.5 Loss

The cross-entropy loss between the predicted probabilities y_{pred} and the true targets y_{true} at a single time step t is given by:

$$H(y_{\text{true},t}, y_{\text{pred},t}) = -\sum_i y_{\text{true},t,i} \log(y_{\text{pred},t,i})$$

where $y_{\text{pred},t}$ is the predicted probability distribution at time step t, $y_{\text{true},t}$ is the true probability distribution at time step t (i.e., a one-hot encoded vector representing the true target), and i ranges over the vocabulary size.

The total loss is then computed as the sum of the cross-entropy losses over all time steps:

$$L = \sum_{t=1}^{T} H(y_{\text{true},t}, y_{\text{pred},t})$$

where T is the sequence length.

1.6 Train

The 'train' method trains the RNN on a dataset using backpropagation through time. The method takes an instance of 'DataReader' containing the training data as input. It initializes a hidden state vector a_{prev} at the beginning of each sequence to zero. Then, it iterates until the smooth loss is less than a threshold value.

During each iteration, it retrieves a batch of inputs and targets from the data reader. The RNN performs a forward pass on the input sequence and computes the output probabilities. The backward pass is performed using the targets and output probabilities to calculate the gradients of the parameters of the network. The Adagrad algorithm is used to update the weights of the network.

The method then calculates and updates the loss using the updated weights. The previous hidden state is updated for the next batch. The method prints the progress every 500 iterations by generating a sample of text using the 'sample' method and printing the loss.

The 'train' method can be summarized by the following steps:

1. Initialize a_{prev} to zero at the beginning of each sequence.
2. Retrieve a batch of inputs and targets from the data reader.
3. Perform a forward pass on the input sequence and compute the output probabilities.
4. Perform a backward pass using the targets and output probabilities to calculate the gradients of the parameters of the network.
5. Use the Adagrad algorithm to update the weights of the network.
6. Calculate and update the loss using the updated weights.
7. Update the previous hidden state for the next batch.
8. Print progress every 10000 iterations by generating a sample of text using the 'sample' method and printing the loss.
9. Repeat steps 2-8 until the smooth loss is less than the threshold value.

1.7 Tasks to be completed

- Task 1: Complete the `softmax` function using the formula discussed. Do not use for loops. Also, subtract each value in x with the maximum value of x in the first line, then implement your softmax function.

  [10 marks]

- Task 2: Complete the `loss` function using the formula described in the theory section above. Do not use for loops. [10d marks]

- Task 3: Complete the `forward` and the `backward` function. You can refer to the theory section above. You will only get marks for this part if both functions are correct. The train function has been completed for you. [25 marks]
- Task 4: Complete the `predict` function. You can refer to the section above. Once you are done with the predict function, your entire RNN class will be verified. Thus you will get marks for this part if your entire class is correctly implemented. [30 marks]
- Comments have been provided at each step to guide you further.

2 Question 2: RNN using torch implementation [30 marks]

For this question, we will use the torch implementation of RNN to explore the application of the PyTorch library as it is used in basic deep learning projects.

2.1 Time series prediction - setup

The problem that we will be looking at is fairly straightforward and a popular problem in literature. Given a time series, we want to predict the next "value". A more generalised problem will be **looking at a token sequence and predicting the next token**. This is the fundamental principle of Natural Language Processing, where every text sequence is tokenised into a sequence of numbers called tokens. Then, based on some context, the model learns to predict the next token.

However, formulating the problem requires some **pre-processing and preparation** of the dataset. Generally, the data given is in the form of a long sequence of values. An example dataset for this problem is provided with this question. We have given you the historical data of "BTCUSDT" i.e. the ticker with which stocks of Bitcoin are traded in US markets. This data can easily be downloaded from Binance. We have filtered data and provided the opening price of BTCUSDT at every 20-hour interval.

- Due to the volatile nature of bitcoin and stocks in general, the values vary across a huge range ranging from \$6,000 to \$40,000. Often, it is a good practice to **normalise the data** such that this range is reduced to a considerably smaller interval (for example [-1,1]), which makes it easier for the model to converge quickly. An optimal solution will still be reached if we remove the normalisation step, but the time and number of computations required will also increase considerably.
- Secondly, however trivial, the way data is given to the model needs some more processing. We need to make sequence target pairs that are essentially sequences of length `seq_length` and corresponding targets. Typically, X[i:i+seq_length] and $X[i+1]$.

Once this setup is done, we move to creating the model (using torch modules). Instead of guiding you through the process as an ungraded learning exercise, we expect you to look up PyTorch Documentation and example implementations to understand how to use these modules. Additionally, a very informative resource to start is this tutorial. Refer to the figure 1 for model architecture:

2.2 Code

In the skeleton code provided, we have provided the torch Dataset object, which creates a processed dataset as specified previously. This is then batched appropriately to help reduce the time the model requires to converge.

- `StockDataset`: This object extends the torch.Dataset class and is used to process the data.
- `RNN`: This extends the torch.nn.module. Essentially, any custom model that we wish to tailor for our task needs to be created similarly, with a `forward` pass. All the layers are initialised in the constructor, and connections among them are tailored to the forward pass.
- `train`: The training loop comes here. This loop is instrumental to train the model once it is created and instantiated, along with the specification of the loss function to be optimised and the optimiser to be used. Note that we also use a validation set to select the best model based on its performance.
- `test`: Testing the model will be fairly straightforward. It is essentially similar to how we calculated the validation loss.

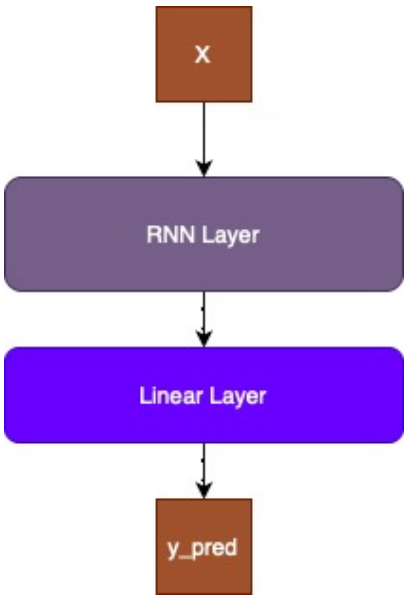

Figure 1: Model Architecture

2.3 Tasks to be completed

- Complete the model instantiation and the forward pass (The correctness of this is tested solely based on the correct implementation of subsequent functions).
- Complete the training loop using the model instantiation. [25 marks]
- Complete the testing loop carefully. [5 marks]

Following a similar pattern as previous labs, the evaluation will be done by comparing the validation loss histories thus obtained during the training loop.