Student Name: Arash Sajjadi
Student ID: 400422096

Artificial Neural Network (ANN)
Assignment V

I have handed in assignment number 4, but for personal practice, I tried to answer the assignments that are not time-consuming from the fifth assignment series.

Question 1

A stateful RNN maintains the internal state across time steps, while a stateless RNN does not. In a stateful RNN, the hidden state from the previous time step is passed as an input to the current time step, allowing the network to maintain information across long sequences. In a stateless RNN, the hidden state is reset at each time step, so the network cannot maintain information across long sequences.

One of the main advantages of using a stateful RNN is that it can handle input sequences of varying lengths, while a stateless RNN requires input sequences to be of fixed length. Stateful RNNs are also useful for tasks such as language modeling and speech recognition, where the meaning of a word or phrase depends on the context in which it is used.

On the other hand, stateless RNNs are more computationally efficient and easier to train, since the hidden state is reset at each time step and does not need to be propagated through the network. They are also less prone to overfitting and are more robust to noise in the input data.

In summary, Stateful RNNs are better suited to tasks that require maintaining context across time steps, while stateless RNNs are better suited to tasks that do not require this type of context.

Question 2

Encoder-Decoder RNNs and plain sequence-to-sequence RNNs are both architectures that are commonly used for sequence-to-sequence tasks such as machine translation, text summarization, and image captioning. The main difference between the two is that Encoder-Decoder RNNs use two separate RNNs, one for encoding the input sequence and one for decoding the output sequence. In contrast, plain sequence-to-sequence RNNs use a single RNN for both encoding and decoding.

An Encoder-Decoder RNN first encodes the input sequence into a fixed-length context vector, which is then passed to the decoder RNN to generate the output sequence. The encoder RNN compresses the input sequence into a fixed-length vector representation, which captures the most important information of the input sequence. The decoder RNN then generates the output sequence based on the context vector.

In contrast, plain sequence-to-sequence RNNs use a single RNN that processes the input sequence from left to right and generates the output sequence from right to left. This architecture cannot compress the input sequence into a fixed-length vector representation. It may be harder to train and less effective for tasks that require capturing a lot of contexts.

For automatic translation, I would prefer Encoder-Decoder RNNs, as they have been shown to be more effective than plain sequence-to-sequence RNNs in machine translation tasks. The encoder-decoder architecture allows the model to capture the meaning of the source sentence in a fixed-length vector, which can be used to generate the target sentence. Additionally, the attention

mechanism is often used in Encoder-Decoder RNNs to focus on specific parts of the input sequence while generating the output, which can improve the model's performance.

Question 3

A possible implementation for a gated RNN cell that sums its inputs over time is to use a combination of two gates: an update gate and a reset gate. The update gate controls how much of the previous state should be retained, while the reset gate controls how much of the previous state should be forgotten. The gating values can be represented as scalars between 0 and 1, where 0 means "no effect" and 1 means "full effect". The equations for the gating values can be represented as:

$$\texttt{update\_gate} = \sigma(W_u \cdot [h_{t-1}, x_t] + b_u)$$

$$\texttt{reset\_gate} = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

where h_{t-1} is the previous hidden state, x_t is the current input, W_u and b_u are the weight matrix and bias for the update gate, and W_r and b_r are the weight matrix and bias for the reset gate. σ is the `sigmoid` function, which maps the input to a value between 0 and 1.

Once we have the gating values, we can use them to calculate the new hidden state. The new hidden state is calculated as a linear combination of the previous hidden state, the current input, and the reset gate.

$$h_t = \texttt{update\_gate} * h_{t-1} + (1 - \texttt{update\_gate}) * (\texttt{reset\_gate} * h_{t-1} + (1 - \texttt{reset\_gate}) * (W_h \cdot [h_{t-1}, x_t] + b_h))$$

where W_h and b_h are the weight matrix and bias for the new hidden state calculation.

This gated RNN cell will sum its inputs over time as the update gate allows the previous state to be retained, and the reset gate allows the previous state to be forgotten.

In the LSTM architecture, the input gate, forget gate, and output gate are all scalar values between 0 and 1 that control the flow of information in and out of the cell state.

The input gate controls how much of the new input should be added to the cell state. The input gate value is calculated using a `sigmoid` function:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

Where W_i and b_i are the weight matrix and bias for the input gate, h_{t-1} is the previous hidden state, and x_t is the current input.

The forget gate controls how much of the previous cell state should be forgotten. The forget gate value is also calculated using a `sigmoid` function:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Where W_f and b_f are the weight matrix and bias for the forget gate.

The forget and input gates work together to update the cell state. The new cell state is calculated as:

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

Where c_t is the new cell state, c_{t-1} is the previous cell state, $\tilde{c}_t$ is the candidate cell state, and i_t and f_t are the input and forget gate values respectively.

It's worth noting that the `sigmoid` activation function is used for the gates but the LSTM architecture can be implemented with other activation functions such as `ReLU`, `tanh`, etc.

In the Gated Recurrent Unit (GRU) architecture, the reset gate and update gate are both scalar values between 0 and 1 that control the flow of information in and out of the hidden state.

The update gate controls how much of the previous hidden state should be retained, and how much of the new input should be used to update the hidden state. The update gate value is calculated using a `sigmoid` function:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

Where W_z and b_z are the weight matrix and bias for the update gate, h_{t-1} is the previous hidden state, and x_t is the current input.

The reset gate controls how much of the previous hidden state should be forgotten. The reset gate value is also calculated using a `sigmoid` function:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

Where W_r and b_r are the weight matrix and bias for the reset gate.

The update and reset gates work together to update the hidden state. The new hidden state is calculated as:

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

Where h_t is the new hidden state, h_{t-1} is the previous hidden state, $\tilde{h}_t$ is the candidate hidden state, and z_t and r_t are the update and reset gate values respectively.

It's worth mentioning that the `sigmoid` activation function is used for the gates but the GRU architecture can be implemented with other activation functions such as `ReLU`, `tanh`, etc.

Qestion 5

Since this exercise is time consuming and I have handed the fourth series exercise, I am writing the general steps.

To create a Persian language model using RNNs or LSTM cells for sequence prediction, one could follow these steps:

1. Collect and preprocess a large corpus of Persian text, such as the Persian Wikipedia dataset. This may involve cleaning the text, tokenizing it, and creating a vocabulary.
2. Split the corpus into training, validation, and test sets.
3. Choose an architecture for the model, such as a multi-layer RNN or LSTM, and initialize the model with randomly-initialized weights.
4. Train the model on the training data using a suitable optimization algorithm, such as Adam, and monitor the model's performance on the validation set during training.
5. Select the best-performing model on the validation set and use it to make predictions on the test set.

6. Use the trained model to predict the entire sentence by giving the first 3 to 5 words of a sentence in the corpus.

7. Finally, evaluate the model's performance using metrics such as perplexity and BLEU score to measure the quality of the generated text.