***Solution* 1 :**

Recurrences using RNNs

Given - All inputs are integers, hidden states are scalars, all biases are zero, and all weights are indicated by the numbers on the edges.

$$h_1 = x_1 - h_0 \Rightarrow y_1 = sigmoid(1000h_1) \tag{1}$$

$$From\ (1),\ h_2 = x_2 - h_1 = x_2 - x_1 + h_0 \Rightarrow y_2 = sigmoid(1000h_2) \tag{2}$$

$$From\ (2),\ h_3 = x_3 - h_2 = x_3 - x_2 + x_1 - h_0 \Rightarrow y_3 = sigmoid(1000h_3) \tag{3}$$

$$From\ (3),\ h_4 = x_4 - h_3 = x_4 - x_3 + x_2 - x_1 + h_0 \Rightarrow y_4 = sigmoid(1000h_4) \tag{4}$$

Continuing the above sequence, for even terms, we get,

$$h_t = x_t - x_{t-1} + x_{t-2} - x_{t-3} + ... + x_2 - x_1 + h_0 \Rightarrow y_t = sigmoid(1000h_t) \tag{5}$$

We can re-group the terms into even and odd terms -

$$h_t = (x_t + x_{t-2} + ... + x_2) - (x_{t-1} + x_{t-3} + ... + x_1) + h_0 \Rightarrow y_t = 1/(1 + e^{-1000h_t}) \tag{6}$$

If $h_0 = 0$, the value of h_t would depend on the sum of even terms and odd terms. If sum of odd coefficients, is greater than even coefficients, then the probability value would be closer to 0 and hence binary classified as 0. Vice versa, if sum of even coefficients, is greater than odd coefficients, then probability would be closer to 1 and hence binary classified as 1.

Also, if the sequence of inputs is 1,2,3,4... then for t = 2n terms, $h_t = n$
$\Rightarrow y_t = sigmoid(1000h_t) = 1/(1 + e^{-1000n})$

***Solution* 2 :**

1. Encoder-Decoder Architecture for Translation

- Encoder-decoder architecture have the ability to process variable length input and output sequences of text.
- Using encoder-decoder architecture, long range dependencies can be supported.

- Also, plain sequence-to-sequence RNNs have a vanishing gradient that issue that can be resolved using encoder-decoder architecture.
- Encoder-decoder architecture based models can be trained faster.
- Parallel computation can be implemented.

2. An Attention mechanism is used to construct a sequence of vectors from the encoder RNN from every time-step of the input sequence. The Decoder then learns to pay selective "attention" to the vectors and produces the output at each time step. The key idea is that on each step of the decoder, we intend to focus on a particular part of the source sequence rather than the complete source. Hence, we only utilize the most relevant parts of the input sequence. This enables faster and easier learning as well as improves its quality.

The attention mechanism addresses the bottleneck problem that occurs with the usage of a fixed-length encoding vector, where the decoder has limited access to the input. In the standard seq2seq model, we only use the last hidden state of the encoder RNN (as context vector ind decoder), and as result it is generally unable to accurately process long input sequences. Whereas, the Attention Mechanism directly retains and uses all the hidden states of the input sequence during the decoding process and selectively pick out specific elements from that sequence.

Reference - https://machinelearningmastery.com/the-attention-mechanism-from-scratch/

***Solution* 3 :**

1. Embeddings for NLP applications

Differences between embeddings generated by BERT and Word2Vec:

- Word2Vec embeddings are independent of position of word in sentence. Whereas, BERT have a bidirectional encoding strategy and thus depends on the position of the word in the sentence.
- Word2Vec doesn't support out-of-vocabulary (OOV) words since it cannot generate vectors for words that are outside the provided dictionary. Whereas, BERT is not limited to the provided dictionary.
- Word2Vec offers pre-trained word embeddings where it takes a single word as input and outputs a single vector representation of that word. But in case of BERT, we require the complete sentence.

- Word2Vec models generate embeddings that are context-independent. Whereas, BERT produces embeddings that allows multiple vector representations for the same word, and are context-dependent.

Reference - https://www.saltdatalabs.com/blog/word2vec-vs-bert

2. One-Hot Encoding is a relatively simple technique where in order to create and update the vectorization, only a new entry is to be added in the vector with a one for each new category. Here, every word has its own value in a vector. But, this also creates a problem where we end up creating a new dimension for each category and hence ends up being inefficient.

On the other hand, Embedding is a complex technique which involves large amounts of data and also has repeated occurrences of individual elements, and hence as a result has long training time. The output is a quite dense vector but with a fixed number of dimensions.

Also, One-Hot Encoding provides no information about the items where each vectorization is an orthogonal representation in another dimension. On the other hand, Embeddings place commonly co-occurring items together in the representation space.

Hence, Embeddings are preferred in situations where are given enough training time, training data, and complex training algorithm.

Reference - https://datascience.stackexchange.com/questions/29851/one-hot-encoding-vs-word-embedding-when-to-choose-one-or-another

---

***Solution* 4 :**

1. Every token only attends to its own position and all previous positions.

Hence, at position i, we compare i terms - Number of combinations = $\Sigma_{i=1}^{n} i = 1 + 2 + 3 + ...n$

Hence, number of dot-products = $\frac{n(n+1)}{2}$

2. Every token attends to at most t positions prior to it, plus itself -

For the cases where $i \leq t$, the number of dot products = i, whereas from t + 1, the number of dot products = t + 1

Number of terms till t = t, number of terms from t+1 to n = n - (t + 1) + 1 = n - t

Number of dot-products till t $= \Sigma_{i=1}^{t} i = 1 + 2 + 3 + ...t = \frac{t(t+1)}{2}$

Number of dot-products from t+1 to n $= (n - t) \times (t + 1)$

Hence, number of dot-products $= \frac{t(t+1)}{2} + (n - t) \times (t + 1)$

3. N tokens are partitioned into windows of size w and every token attends to all positions within its window and prior to it, plus itself-

If the question is interpreted as all positions in window w and elements in prior windows, for a window i, it has 1 to w elements in it.

Total number of combinations = Number of windows $\times \Sigma_{i=1}^{w} i$

$\frac{n}{w} \times \Sigma_{i=1}^{w} i = \frac{n}{w} \times \frac{w(w+1)}{2}$

Hence, total number of combinations $= \frac{n(w+1)}{2}$
