Q9-

In general, using attention mechanism instead of traditional RNN and CNN's, was a breakthrough due to several reasons:
First of all, the RNN and LSTM models could not handle *parallel computing* and the training process became hard and time consuming so it could not perform well on tasks like text generation and question answering(long-dependency context), or in general, tasks in which *context* matters!
Self-attention mechanism claims that it can capture the relationship and long-context dependencies of data, *independently* and without considering their positions.
In the case of CNN's, the attention mechanism could help the network to capture the important features and attend to the main information in the picture, it may lead to better results and a more general model in case of object detection, semantic segmentation and other vision tasks. It's almost like the attention mechanism in the human vision system(focus and attending to objects).
So, three main pros of attention over RNN and CNN, is:
1- Parallel processing(over RNN), 2- capture relationships independently, 3- position independence .
The question is, how do we handle the positional information in the attention mechanism?
So we will discuss this in **Q11.**
But where do we use RNN instead of attention?
In the tasks in which *local information* matters more than general context, the RNN could perform better since it has a strong *inductive bias* on the sequential input data.

Q10-

1- **Dot product:** The default similarity score in attention mechanism.
Here is the formulation:

$$Sim(Q, K) = Q.K^T$$

Which calculates the similarity between Query and the Key. It is mostly used in *machine translation* tasks to help the model to focus on relevant parts of sentences to generate new tokens.

2 - Scaled Dot product: which is exactly same as simple dot product with only one difference:

$$Sim(Q, K) = (Q.K^T)/sqrt(keydimension)$$

Which scales the input of the network to have a stable network which could handle variants of input lengths.

3 - Cosine similarity: it focuses on the direction instead of magnitude of vectors by measuring the angle between vectors:

$$Sim(Q, K) = (Q.K^T)/(||Q||||K||)$$

Which is used in tasks like *Text similarity* and *recommendation systems*.
4- Additive (Bahdanau) Attention: it uses a Feed Forward Neural Network(FFNN) To calculate the similarity between two vectors, which can capture more complex relationships in data than simple dot products by adding non-linearity.
$[\text{Similarity}(Q, K) = v^T \tanh(W_Q Q + W_K K + b)]$
where $(W_Q), (W_K), and (v)$ are learned weight matrices and vectors, and (b) is a bias term.
Which is quite useful in complex tasks like *speech recognition*.

Q11-

Relative positional encoding is a method used in Transformer models to encode the position of tokens in a sequence, but unlike *absolute positional* encoding, it focuses on the relative distances between tokens rather than their *absolute positions*. This approach is designed to improve the model's ability to generalize across different input lengths and to better capture the relative relationships between tokens.

The question is “Why Relative Positional Encoding is Used?”

1. Contextual Relevance: In many natural language processing tasks, the meaning of a token can be influenced more by its relative position to other tokens rather than its absolute position. For instance, in syntactic parsing or question-answering, the relationship between words (e.g., subject-verb-object) is often more important than their exact positions in the sentence.
2. Generalization Across Lengths: Absolute positional encodings tie the model to specific input lengths, making it less flexible when handling sequences longer than those seen during training. Relative positional encoding, however, allows the model to generalize better to varying sequence lengths.
3. Improved Performance: Empirical studies have shown that models using relative positional encoding often outperform those using absolute positional encoding on various NLP tasks. This improvement is due to the more dynamic and context-aware nature of relative positional encodings.

Formulation of Relative Positional Encoding

In relative positional encoding, instead of encoding the absolute positions of tokens, we encode the relative distances between pairs of tokens. Let's denote:
- (i) as the position of the query token.
- (j) as the position of the key token.

The goal is to modify the attention mechanism to take into account the relative position $(j - i)$.

Implementation

1. Positional Embeddings:

- Define a set of learnable vectors $(\mathbf{r}_k)$ for each possible relative position (k) in the range $([-L, L])$, where (L) is the maximum relative distance considered. These vectors are shared across all layers and **heads** in the model(only the first block of the network).

2. Attention Calculation:

- The attention score between a query at position (i) and a key at position (j) is modified to include the relative positional embedding:

$$e_{ij} = \frac{(\mathbf{q}_i \cdot \mathbf{k}_j) + (\mathbf{q}_i \cdot \mathbf{r}_{j-i})}{\sqrt{d_k}}$$

Where $(\mathbf{q}_i)$ is the query vector at position (i), $(\mathbf{k}_j)$ is the key vector at position (j), and $(\mathbf{r}_{j-i})$ is the relative positional embedding for the distance $(j - i)$.

3. Adding Bias:

- Alternatively, a bias term corresponding to the relative position can be added:

$$e_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j + b_{j-i}$$

where (b_{j-i}) is a learnable bias specific to the relative position $(j - i)$.

4. Attention Weights:

- The attention weights are then computed as usual using the softmax function over the modified scores:

$$\alpha_{ij} = \text{softmax}(e_{ij})$$

5. Output Computation:

- Finally, the output is computed as a weighted sum of the value vectors $(\mathbf{v}_j)$, incorporating the relative positional information:

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_j$$

Q12-

The muli-head attention is just some self-attention heads which stack on top of each other! The idea is simple. It’s like asking one question from several experts with different mindset instead of one person and integrating the whole information to provide a conclusion. It's simple and efficient in computation and implementation due to adding only a few parameters to the network. It could help the model to capture different aspects of the data, so it makes sense to use it instead of a single self-attention head!

Why do we use masked attention?
To prevent data leakage and maintain causality!
In autoregressive tasks like text generation, it's crucial to ensure that the generated tokens only depend on previous tokens and not next tokens! It makes sense, if our model could see the feature, so they would be considered as a cheater! But on other hand, tasks like sentiment analysis and information retrieval, we do not need to ensure these options! Since we need to see all the context to decide.

Why is layer normalization used?

Layer Normalization is a technique used to stabilize and accelerate the training of deep neural networks, including Transformers. In the Transformer architecture, layer normalization is typically applied to the inputs of the attention sub-layers and the feed-forward sub-layers.
Layer normalization normalizes the inputs across the features for each data point independently, which helps in reducing internal *covariate shift*. This stabilization helps in achieving faster *convergence* during training.
By normalizing the inputs, the gradients are more predictable, reducing the risk of *vanishing or exploding gradients*, which is particularly important for training very deep networks.

Q4-

The main challenge in continuous action spaces is we have **infinite actions to select**! Which causes less efficient action selection policy and also it's hard to maintain a proper *reward system* and may lead to a bad result(no convergence).
Your agent may never select the same action twice!
Another problem is the *curse of dimensionality* which is obvious due to the number of actions and it makes it hard to explore action space and find the best action!
In discrete action spaces, by generating a probability for each action using softmax function, we can guarantee the *exploration and exploitation* and interpret and control it as well, but in continuous spaces, it’s challenging since we cant use softmax to define the probability of the actions!(which action is prior). The solution of this problem is to consider a probability density function for action space(which is usually a normal distribution) and take a sample of it by generating a **mu and sigma** instead of probability of the actions. And modify a new loss function and entropy bonus to learn this network, which the loss is log probability of a normal distribution. We do this technique in *Actor-critic* approaches(so one possible solution for these action spaces is using actor-critic approaches).
Another possible solution for these challenges is using function approximation techniques instead of considering all actions. We use these approximations to approximate the **policy** and **value** functions using neural networks. And also we can

discretize our action space which adds a bias to the model and may lead to bad outcomes.

Q5-

In one sentence:

‘’In Trust Region Policy Optimization (TRPO), the concept of a "trust region" is used to ensure that updates to the policy are made in a stable and reliable manner. ‘’

What is trust region?

We have two types of optimization techniques based on some insights!
The line search method which is used in normal gradient descent and Trust regions!
The line search algorithms search for the line which have the direction of optimal point and TR methods are doing as follow:

The trust region is defined by a constraint on the *Kullback-Leibler (KL)* divergence between the old policy $(\pi_{\theta_{\text{old}}})$ and the new policy $(\pi_{\theta_{\text{new}}})$. The KL divergence measures the difference between the two probability distributions, **ensuring** that the new policy does not differ too much from the old policy.
This constraint is mathematically represented as:

$$[\mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}\left[D_{\text{KL}}\left(\pi_{\theta_{\text{old}}}(\cdot|s)\|\pi_{\theta_{\text{new}}}(\cdot|s)\right)\right] \leq \delta]$$

Here, (δ) is a small positive value that specifies the **maximum allowed divergence,** and $(\rho_{\theta_{\text{old}}})$ represents the state distribution under the old policy.

The trust region helps stabilize policy updates in several ways:

By constraining the KL divergence, TRPO prevents the policy from making large, abrupt changes. This is crucial because drastic policy changes can lead to instability, especially in environments where small differences in policy can lead to significantly different outcomes.
The trust region constraint helps ensure that each policy update improves the expected reward or at least does not degrade it significantly. This is achieved by maintaining the policy updates within a region where the approximation used to estimate the improvement is reliable.
By limiting the change in policy, TRPO improves the robustness of the learning process. This makes the optimization process less sensitive to noise in the gradient estimates and other sources of variability.
The trust region allows the agent to explore new policies safely without straying too far from the known, effective policies. This balanced exploration and exploitation strategy is essential for effective reinforcement learning.

Q6-

Entropy regularization is a technique to encourage exploration and prevent premature convergence to *suboptimal policies*. Entropy regularization plays a crucial role in maintaining a balance between exploration and exploitation, thereby affecting the overall performance and robustness of the learned policy.
Entropy in the context of RL measures the *randomness* or *uncertainty* in the **action distribution of the policy**.

''High entropy means the policy is exploring different actions more uniformly, while low entropy indicates that the policy is more deterministic, favoring certain actions over others".

This technique involves adding some sort of randomness in policies objective function.
The modified objective function with entropy regularization is typically expressed as:

$$[L(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t\right] + \beta \mathbb{E}\left[H(\pi_\theta(\cdot|s_t))\right]]$$

Here, $(H(\pi_\theta(\cdot|s_t)))$ represents the entropy of the policy (π_θ) at state (s_t), and (β) is a hyperparameter that controls the weight of the entropy term.

The hyperparameter β determines the trade-off between the reward maximization and the entropy maximization. A higher β places more emphasis on exploration, while a lower β focuses more on exploitation.

How Entropy Regularization Affects Exploration and Policy Performance?
By adding entropy penalty to the objective function, it enforces the model to search a wider range of policies and actions(more exploration).
Without entropy regularization, the policy might quickly converge to a **suboptimal** policy because it prematurely focuses on actions that appear to be the best based on limited experience. Entropy regularization keeps the policy's action distribution more *stochastic* for a longer period during training.

Q7-

PPO was introduced by openAI to address the problem of complexity of computation and implementation of TRPO; it has some joint attributes with the TRPO.

PPO mainly use two methods to achieve stabilize PG updates which is as follows:

1.Clipped Surrogate Objective:

PPO uses a clipped objective function to limit the extent of policy updates. This is done by clipping the probability ratio between the new policy and the old policy to be within a specified range. The clipped objective function is given by:

$$[L^{\mathrm{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)A_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]]$$

Here, $(r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)})$ is the probability ratio, (A_t) is the advantage function, and (ϵ) is a hyperparameter that specifies the clipping range.

2.Adaptive Kullback-Leibler Penalty:

Alternatively, PPO can use an adaptive KL penalty to constrain updates. Instead of hard clipping, this method penalizes updates that result in a large KL divergence, adjusting the penalty coefficient dynamically during training. The objective function with KL penalty is:

$$[L^{\mathrm{KL}}(\theta) = \mathbb{E}\left[r_t(\theta)A_t - \beta D_{\mathrm{KL}}\left(\pi_{\theta_{\mathrm{old}}}(\cdot|s_t)||\pi_\theta(\cdot|s_t)\right)\right]]$$

Where (β) is a coefficient that adjusts the weight of the KL penalty.

PPO addresses the following shortcomings of TRPO:

Simplicity and Ease of Implementation: TRPO Involves solving a constrained optimization problem using a complex conjugate gradient algorithm and line search to ensure the KL divergence constraint is met. This makes TRPO more challenging to implement and tune. While PPO simplifies this process by using either the clipped surrogate objective or an adaptive KL penalty, which are easier to implement using standard optimization techniques such as stochastic gradient descent. This reduces the complexity of the algorithm.
Computational Efficiency: TRPO The constrained optimization process and the need for multiple backtracking line searches make TRPO computationally expensive.
By using simpler objective functions (clipped or with an adaptive penalty), PPO reduces the computational overhead, allowing for more frequent and efficient policy updates.