Q1 A single perceptron cannot solve XOR because as we can see and conclude from the truth table that XOR is not linearly separable function.

XOR

0	0	0
0	1	1
1	1	0
1	0	1

Whereas a single perceptron can only solve linearly separable functions, because it uses linear activation function.

For eg: $z = \sum w_i x_i + b$, $y = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$

With introducing multi-layer perceptrons we bring in hidden layers which allow us to even represent functions which are not linear, by using activation functions like ReLU, sigmoid, tanh, etc. that are non-linear.

$$z = \sum w_i x_i + b \quad , \quad h = \sigma(z)$$

$\sigma \to$ non-linear activation function

Q2 Stacking linear layers : $y_1 = w_1 x + b_1$

$$y_2 = w_2 y_1 + b_2 = w_2(w_1 x + b_1) + b_2$$

$$= w_1 w_2 x + \underbrace{w_2 b_1 + b_2}$$

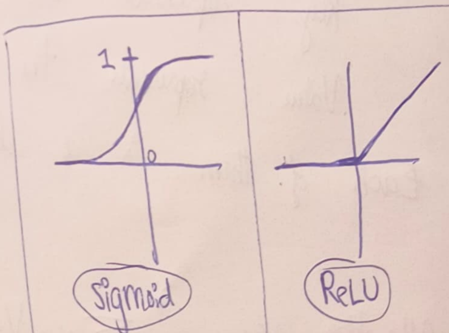
$$y_2 = w_3 x + b_3 \longleftarrow \text{Linear}$$

∴ Each layer just does some linear function on the input resulting in the stack still being linear as shown for arbitrary linear function above.

Since in backpropagation we use chain rule, which involves multiplication and if the derivative is <1, repeated multiplication results in the shrinking of gradients (deep networks)

sigmoid $\sigma_1(x) = \frac{1}{1+e^{-x}}$ $\quad \sigma_1'(x) = \sigma(x)[1-\sigma(x)]$

ReLU $\sigma_2(x) = \max(0, x)$ $\quad \sigma_2'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$



as we can see max derivative in case of sigmoid $\to$ 0.25

ReLU $\to$ 1

So ReLU solves vanishing gradient better than sigmoid since in case of active neurons derivative $= 1$ hence allowing gradients to go through many layers.

(Q3) We need positional encoding since transformers do not know the order of words by default. If two sentences have same words but different order the transformer will not be able to differentiate without positional encodings.

Sinusoidal PE $\Rightarrow$ cannot be learned. It is a fixed mathematical function. It has no max sentence length and is better in understanding relative awareness. Extrapolation = good.

Absolute PE $\Rightarrow$ It is learned from training examples. It has a max training length and learns vectors for encoding. Poor at extrapolation.

RoPE $\Rightarrow$ Instead of directly adding positional encoding with token embeddings, it rotates query and key vectors based on position.

$$\begin{pmatrix} q'_1 \\ q'_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \leftarrow \text{rotates like this}$$

$\theta \to$ relation with position i

$\theta \to f(i)$

In this way, relative position is naturally encoded and learned. Since rotation is periodic and continuous ($\therefore$ not fixed encodings) so better for long contexts and extrapolation since attention score now depends on relative positions.

(Q4) Query represents what information the token needs
Key represents what info the token has/offers to other tokens
Value represents the actual info that is being passed forward.

Each of them is a learned linear projection $Q = XW_Q$, $K = XW_K$, $V = XW_V$ $X \to$ Input token embeddings

Attention Score $= \sum_j \alpha_{ij} V_j$ $\alpha_{ij} = \text{softmax}\left(\frac{Q_i \cdot K_j}{\sqrt{d_k}}\right)$

Assuming they are normally distributed so mean $= 0$
variance $= 1$

but dot product $Q \cdot K = \sum_{i=1}^{d_k} q_i k_i$ $\leftarrow$ variance $= d_k$

if not scaling then as d_k increases, dot product increases, gradients vanish.

by scaling $\text{var}\left(\frac{Q \cdot K}{\sqrt{d}}\right) \simeq 1$ resulting in better gradient flow by avoiding softmax saturation

Diagonal entry $\Rightarrow Q_i \cdot K_i \Rightarrow$ a token attention to itself $\therefore$ highest attention score (self)

(Q5) Splitting attention into multiple heads allows us to learn many patterns at the same time by running in parallel. So it is fast because parallel and learns complex relations better.

$\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$ $\quad$ $\text{Multihead} = \text{concat}(\text{head}_1, \text{head}_2, \ldots \text{head}_h) W_0$

$d_{model} \Rightarrow$ dimension of token embedding

$h \Rightarrow$ no. of heads

$d_{head} \Rightarrow$ dimensions per head $= \frac{d_{model}}{h}$

(Q6) Greedy decoding is suboptimal ~~because~~ because it selects the token with max probability at each time step t. But what we actually want is to maximise the joint probability of all tokens. Basically the problem with greedy, although it feels the best intuitively ~~local maximum~~

$\hookrightarrow$ Local maximum $\neq$ Global maximum

Beam search is better because instead of selecting the best one, it keeps the '~~top~~ k' sequences at each step then expand each, score all and keep the best k. so it goes through multiple options in parallel and chooses the globally optimum one.

Q1 (a) $d\_head = \frac{d\_model}{h} = \frac{768}{12} = \boxed{64}$

(b) Parameters (One layer) $\Rightarrow W_Q, W_K, W_V \in \mathbb{R}^{d_{model} \times d_{model}}$

$= 768 \times 768$

$= \boxed{589824}$ each

Q2 $softmax = \frac{e^{z_i}}{\sum_i e^{z_i}}$

$$\left.\begin{aligned} e^2 &= 7.389 \\ e^1 &= 2.718 \\ e^0 &= 1 \end{aligned}\right\} add = 11.107$$

$softmax(2.0) = \frac{7.389}{11.107} = \boxed{0.665}$

$softmax(1.0) = \frac{2.718}{11.107} = \boxed{0.245}$

$softmax(0.0) = \frac{1}{11.107} = \boxed{0.090}$