1. A
2. C
3. A
4. D
5. C
6. B
7. B
8. A
9. A,C
10. A,B
11. If we don’t use activation function in neural network, every neuron will only be performing a linear transformation on the inputs using the weights and biases. Although linear transformations make the neural network simpler, but this network would be less powerful and will not be able to learn the complex patterns from the data.
12. **Forward Propagation:** In the forward propagation, we check what the neural network predicts for the first training example with initial weights and bias. First, we initialize the weights and bias randomly, then we calculate z, the weighted sum of activation and bias, then we apply the activation function (Relu, sigmoid, etc).

**Backward Propagation:** We can define a cost function that measures how good our neural network performs. For an input, x, and desired output, y, we can calculate the cost of a specific training example as the square of the difference between the network’s output and the desired output,

Ck = (output – y)2

Where k stands for training example, and the output is assumed to be the activation of the output neuron, and y is the actual desired output. The total cost of a training set is the average of the individual cost functions of the data in the training set, We want to improve the performance of the neural network on the training examples, so that we can change the weights and bias, and hopefully, lower the total cost. We want to know how much the specific weights and bias affect the total cost, so we need to calculate the partial derivatives of the total cost with respect to the weights and bias. To do this, we can apply the chain rule, Now, we calculate the partial derivatives with respect to the total cost, Then we update the weights and bias: we multiply the partial derivatives with some learning rate and subtract the results from the weights and bias.

1. Gradient Descent: The goal of the gradient descent is to minimise a given function which, in our case, is the loss function of the neural network. To achieve this goal, it performs two steps iteratively.
   1. Compute the slope (gradient) that is the first-order derivative of the function at the current point
   2. Move-in the opposite direction of the slope increase from the current point by the computed amount

Batch Gradient Descent: In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that’s just one step of gradient descent in one epoch.

Stochastic Gradient Descent: In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good. Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way. To tackle this problem, we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in one epoch for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it’s gradient
4. Use the gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for all the examples in training dataset

Mini Batch Gradient Descent: We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used. Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. So, after creating the mini-batches of fixed size, we do the following steps in one epoch:

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created
6. 1. Easily fits in the memory.

2. It is computationally efficient.

1. Benefit from vectorization.
2. If stuck in local minimums, some noisy steps can lead the way out of them.
3. Average of the training samples produces stable error gradients and convergence.
4. Transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task. It is an optimization that allows rapid progress or improved performance when modelling the second task.