

# Guide to Chip Huyen's Machine Learning Interview Book

Crafted with ❤️ by Kyaw Htet Paing Win

## ▼ Introduction

Hello, friends 😊!

I recently came across Chip Huyen's book which covers over 300 technical questions for machine learning interviews. I realized there was no solutions, so I tried working through a few questions on my own. Since ChatGPT gave some detailed explanations on these

questions that I liked when I checked my answers, I figured, why not get the solutions for all of these interview questions?

So, I wrote a Python script to scrape all the questions and used ChatGPT to get potential solutions to each one. I compiled these machine learning interview preparations into this handy guide, with the goal of helping you prepare for and excel in your upcoming interviews!

I hope this accompanying guide saves you time in prepping for interviews and helps you land your dream ML role. I will also aim to continue improving the completeness of the explanations over time.



**Caution:** Make sure to sense check the answers while doing the exercises and the math questions are not quite as complete as I would like.

Before you start preparing, be sure to check out Chip's helpful guide on the machine learning interview prep, **Wishing you the best of luck with your upcoming interviews!**



Thanks to **Chip Huyen** for compiling these questions for the machine learning community. Check out the original sources below.



Machine Learning Interview Book: <https://huyenchip.com/ml-interviews-book/>

## ▼ Methodology

### Data Collection

- Because the questions in the book are spread over multiple web pages, I wrote a Python script using the `requests` and `BeautifulSoup` libraries to make HTTP GET requests and parse the web content for the questions. I also decided to parse the difficulty level of each question, as this could be useful when prompting the language model later. While I was able to parse most of the questions, there were some with mathematical notation that I decided to handle manually, as parsing them would not be a significant time savings.

### Prompt

- You're a skilled machine learning engineer answering technical questions in an interview. Difficulty levels, if available, are indicated as: [E] for easy, [M] for medium, and [H] for hard.

### Guidelines:

- For [E] questions, demonstrate a fundamental understanding of machine learning concepts.
- For [M] and [H] questions, offer details to prove your expertise.
- Use concrete examples and emphasize real-world application when applicable.
- Include code snippets if relevant.
- Quantify and qualify achievements: discuss specific improvements and how you overcame challenges.

- If a question seems incorrect or ambiguous, request clarification.

Adopt a conversational and professional tone, as if you are a candidate in an interview. Provide clear recommendations and conclusions in your answers, using first-person narrative and colloquial expressions where appropriate. Ensure your responses demonstrate both technical expertise and effective communication skills.

## ▼ Machine Learning Interview Book Questions & Answers

Questions are classified into three difficulty levels. Easy [E], Medium [M], and Hard [H]

### ▼ Chapter 5: Math Questions Answer

#### ▼ Vectors

##### Dot product

- **[E] What's the geometric interpretation of the dot product of two vectors?**

The geometric interpretation of the dot product of two vectors is it measures the product of their magnitudes with the cosine of the angle between them. For two vectors  $u$  and  $v$ , the dot product  $u \cdot v$  is the magnitude of  $u$  times the magnitude of  $v$  times the cosine of the angle between them. When the vectors point in the same direction, the cosine is 1 and the dot product is at its maximum.

- **[E] Given a vector  $u$ , find vector  $v$  of unit length such that the dot product of  $u$  and  $v$  is maximum.**

Given a vector  $u$ , the vector  $v$  of unit length that maximizes the dot product  $u \cdot v$  is the vector that points in the same direction as  $u$ . The vector  $v$  can be found by dividing  $u$  by its own magnitude, making it a unit vector.

## Outer product

- [E] Given two vectors  $a = [3, 2, 1]$  and  $b = [-1, 0, 1]$ . Calculate the outer product ?

◦ The outer product of two vectors  $a = [3, 2, 1]$  and  $b = [-1, 0, 1]$  is calculated as follows:

$$\mathbf{a} \otimes \mathbf{b} = \begin{bmatrix} 3 \times -1 & 3 \times 0 & 3 \times 1 \\ 2 \times -1 & 2 \times 0 & 2 \times 1 \\ 1 \times -1 & 1 \times 0 & 1 \times 1 \end{bmatrix} = \begin{bmatrix} -3 & 0 & 3 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- [M] Give an example of how the outer product can be useful in ML.

An example of how the outer product can be useful in ML is in creating a Gram matrix, which represents all possible inner products between a set of vectors and is useful in kernel methods like support vector machines.

- [E] What does it mean for two vectors to be linearly independent?

Two vectors are linearly independent if no scalar multiple of one vector equals the other. This means they do not lie on the same line through the origin and neither can be formed by scaling the other.

- [M] Given two sets of vectors  $A = a_1, a_2, a_3, \dots, a_n$  and  $B = b_1, b_2, b_3, \dots, b_m$ . How do you check that they share the same basis?

To check if two sets of vectors A and B share the same basis, you must prove that each set spans the same subspace and has the same dimensionality. This typically involves showing that each vector in one set can be written as a linear combination of vectors in the other set and vice versa, and that both sets have the same number of linearly independent vectors.

- [M] Given n vectors, each of d dimensions. What is the dimension of their span?

Given  $n$  vectors, each of  $d$  dimensions, the dimension of their span depends on their linear independence. If all vectors are linearly independent, the dimension of their span is  $\min(n, d)$ . If they are not all linearly independent, the dimension will be less than  $\min(n, d)$ .

## Norms and metrics

- **[E] What's a norm? What is  $L_0, L_1, L_2, L_{norm}$ ?**

A norm is a function that assigns a strictly positive length or size to each vector in a vector space, except for the zero vector, which is given a length of zero. The  $L_0$  norm refers to the number of non-zero elements in a vector,  $L_1$  norm is the sum of the absolute values of the vector elements,  $L_2$  norm (also known as the Euclidean norm) is the square root of the sum of the squares of the vector elements, and the  $L_\infty$  norm is the maximum absolute value of the elements in the vector.

- **[M] How do norm and metric differ? Given a norm, make a metric. Given a metric, can we make a norm?**

A norm is a function that measures the size of vectors in a space, while a metric measures the distance between two points in a space. To make a metric from a norm, define the metric  $d(u, v)$  as the norm of the difference between  $u$  and  $v$ ,  $d(u, v) = \|u - v\|$ . Not all metrics come from norms, but for a function to be a norm, it must satisfy certain properties such as absolute scalability and the triangle inequality, and it must be zero only for the zero vector.

## ▼ Matrices

- **[E] Why do we say that matrices are linear transformations?**

Matrices represent linear transformations because they map linear combinations of vectors to other linear combinations.

Specifically, for a matrix  $\mathbf{M}$  and vectors  $\mathbf{u}$  and  $\mathbf{v}$ , and a scalar  $c$ , the following properties hold true, which are the properties of linear transformations:

- $\mathbf{M}(\mathbf{u} + \mathbf{v}) = \mathbf{Mu} + \mathbf{Mv}$ 
  - Applying  $\mathbf{M}$  to the sum of two vectors is the same as summing the results of applying  $\mathbf{M}$  to each vector individually.
- $\mathbf{M}(c\mathbf{u}) = c(\mathbf{Mu})$ 
  - Applying  $\mathbf{M}$  to a scalar multiple of a vector is the same as multiplying the result of applying  $\mathbf{M}$  by that same scalar.

These properties demonstrate that matrix multiplication exhibits the key aspects of a linear transformation:

1. **Additivity** - Applying the transformation to vector sums gives the sum of individually transformed vectors
2. **Homogeneity** - Scalars distribute across the transformation

Thus, matrices and matrix multiplication inherently represent and operate as linear transformations between vector spaces.

- **[E] What's the inverse of a matrix? Do all matrices have an inverse? Is the inverse of a matrix always unique?**

The inverse of a matrix  $\mathbf{A}$  is another matrix  $\mathbf{A}^{-1}$  such that  $\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ , where  $\mathbf{I}$  is the identity matrix. Not all matrices have an inverse; only square matrices that are non-singular (with a non-zero determinant) have an inverse. When a matrix has an inverse, it is always unique.

- **[E] What does the determinant of a matrix represent?**

The determinant of a matrix can be interpreted as a scaling factor for the transformation that the matrix represents. Geometrically, it represents the volume scaling factor of the linear transformation described by the matrix, including whether the transformation preserves or reverses the orientation of the space.

- **[E] What happens to the determinant of a matrix if we multiply one of its rows by a scalar  $t \times R$ ?**

Multiplying a row of a matrix by a scalar  $t$  scales the determinant of the matrix by that scalar. So if the original determinant was  $d$ , the new determinant will be  $t \times d$ .

- **[M] A  $4 \times 4$  matrix has four eigenvalues  $3, 3, 2, -1$ . What can we say about the trace and the determinant of this matrix?**

The trace of the matrix, which is the sum of its eigenvalues, would be  $3+3+2-1=7$ . The determinant of the matrix, which is the product of its eigenvalues, would be  $3 \times 3 \times 2 \times (-1) = -18$

- **[M] Given the following matrix:**

1	4	-2
-1	3	2
3	5	6

Without explicitly using the equation for calculating determinants, what can we say about this matrix's determinant?

Hint: rely on a property of this matrix to determine its determinant.

Since the sum of the first and second row gives the third row (i.e.,  $(1)+(-1)=3$ ,  $(4)+(3)=5$ ,  $(-2)+(2)=0$ ), we can conclude that the rows are linearly dependent. Therefore, the determinant of this matrix is zero.

- **[M] What's the difference between the covariance matrix  $A^T A$  and the Gram matrix  $AA^T$ ?**

- The covariance matrix  $A^T A$  measures the variance and covariance among the columns of  $A$ , assuming the rows represent different observations. The Gram matrix  $AA^T$ , on the other hand, measures the inner products between the rows of  $A$ , representing the similarity between different observations.
- Given  $A \in \mathbb{R}^{n \times m}$  and  $b \in \mathbb{R}^n$ 
  - [M] Find  $x$  so that:  $Ax = b$ 

To find  $x$ , we can use matrix algebra techniques such as Gaussian elimination, or for overdetermined systems, methods like least squares. If  $A$  is square and invertible, we can directly calculate  $x$  as  $A^{-1}b$ .
  - [E] When does this have a unique solution?

This equation has a unique solution when  $A$  is a square matrix ( $n = m$ ) and has full rank, meaning it is invertible.
  - [M] Why is it when  $A$  has more columns than rows,  $Ax = b$  has multiple solutions?

When  $A$  has more columns than rows, it means there are more variables than equations. This usually leads to an infinite number of solutions because there are free variables that are not constrained by the equations.
  - [M] Given a matrix  $A$  with no inverse. How would you solve the equation  $Ax = b$ ? What is the pseudoinverse and how to calculate it?

If  $A$  is not invertible or not square, we can still find a solution using the pseudoinverse, denoted as  $A^+$ . The pseudoinverse provides the least squares solution and can be calculated using singular value decomposition (SVD).
- Derivative is the backbone of gradient descent.

- **[E] What does derivative represent?**

The derivative represents the rate of change of a function with respect to a variable. It tells us how a function changes as its input changes.

- **[M] What's the difference between derivative, gradient, and Jacobian?**

The derivative refers to the rate of change for single-variable functions. The gradient is a generalization of the derivative for multivariable functions, indicating the direction of the steepest ascent. The Jacobian is a matrix of all first-order partial derivatives of a vector-valued function and generalizes the gradient for functions that output vectors.

- **[H] Say we have the weights  $w \in \mathbb{R}^{d \times m}$  and a mini-batch  $x$  of  $n$  elements, each element is of the shape  $1 \times d$  so that  $x \in \mathbb{R}^{n \times d}$ . We have the output  $y = f(x; w) = xw$ . What's the dimension of the Jacobian  $\delta y / \delta x$ ?**

The dimension of the Jacobian  $\delta y / \delta x$  for the function  $y = f(x; w) = xw$ , with  $x \in \mathbb{R}^{(n \times d)}$  and  $w \in \mathbb{R}^{(d \times m)}$ , would be  $n \times m \times n \times d$ . This is because the Jacobian needs to describe how each element of the output  $y$  changes with respect to each element of the input  $x$ .

- **[H] Given a very large symmetric matrix  $A$  that doesn't fit in memory, say  $A \in \mathbb{R}^{1M \times 1M}$  and a function  $f$  that can quickly compute  $f(x) = Ax$  for  $x \in \mathbb{R}^{1M}$ . Find the unit vector  $x$  so that  $x^T Ax$  is minimal.**

**Hint:** Can you frame it as an optimization problem and use gradient descent to find an approximate solution?

To find the unit vector  $x$  that minimizes  $x^T Ax$ , we can frame this as an optimization problem and approach it using an iterative algorithm like gradient descent or the conjugate gradient method. These methods update  $x$  in a direction that minimizes the function  $x^T Ax$ , and in the case of gradient descent, this involves

computing the gradient  $-2Ax$  at each step. We would iterate this process until convergence, ensuring at each step that  $x$  remains a unit vector by normalizing it.

#### ▼ Dimensionality reduction

- **[E] Why do we need dimensionality reduction?**

Dimensionality reduction is used to reduce the number of random variables under consideration and can be divided into feature selection and feature extraction. It helps in reducing the time and storage space required, removes multicollinearity, enhances the interpretation of the parameters, helps in visualizing data, and most importantly, it can help in avoiding the curse of dimensionality.

- **[E] Eigen decomposition is a common factorization technique used for dimensionality reduction. Is the eigen decomposition of a matrix always unique?**

- The eigen decomposition of a matrix is not always unique. For a given matrix, there can be multiple sets of eigenvectors that correspond to the same eigenvalue, especially if the eigenvalues are degenerate (i.e., multiple eigenvalues are the same). The set of all eigenvectors corresponding to an eigenvalue can form a subspace, and any orthogonal basis of this subspace can be a valid set of eigenvectors. However, the eigenvalues themselves are unique.

- **[M] Name some applications of eigenvalues and eigenvectors.**

- Eigenvalues and eigenvectors are widely used in many fields. They appear in stability analysis, structural analysis, vibration analysis, and electrical circuits. In machine learning, they are used in algorithms like PCA for dimensionality reduction, in Google's PageRank algorithm, and in spectral clustering.

- **[M] We want to do PCA on a dataset of multiple features in different ranges. For example, one is in the range 0-1 and one is in the range 10 - 1000. Will PCA work on this dataset?**

- PCA can be applied to this dataset, but the features should be scaled or normalized first. If features are on different scales, PCA will be biased towards the features with larger variances, so standardizing the features is a crucial preprocessing step.

- **[H] Under what conditions can one apply eigen decomposition? What about SVD?**

1. Eigen decomposition can be applied to square matrices and is particularly used for symmetric matrices, where the eigenvalues are real numbers. SVD, on the other hand, can be applied to any matrix, regardless of its shape and whether it's square or not.
2. The relationship between SVD and eigen decomposition is that for a symmetric matrix, the singular values in the SVD are the absolute values of the eigenvalues, and the left and right singular vectors correspond to the eigenvectors. SVD is a generalization of eigen decomposition to non-square matrices.
3. PCA is closely related to SVD. When performing PCA, the eigendecomposition is essentially applied to the covariance matrix of the data, which can also be obtained from the SVD of the data matrix. The principal components correspond to the singular vectors associated with the largest singular values.

- **[H] How does t-SNE (T-distributed Stochastic Neighbor Embedding) work? Why do we need it?**

- t-SNE works by converting the high-dimensional Euclidean distances between points into conditional probabilities that represent similarities. The similarity of datapoint  $x_j$  to datapoint  $x_i$  is the conditional probability  $p_{j|i}$ , that  $x_i$  would pick  $x_j$  as its neighbor. In the low-dimensional space, t-SNE then tries to reproduce these probabilities using a Student-t distribution. This method is particularly useful for visualizing high-dimensional data in two or three dimensions and can reveal clusters or groupings within the data. We need it because it helps to understand the structure of complex data by revealing patterns that are difficult to detect in high-dimensional spaces.

▼ Calculus and Convex Optimization

## Differentiable functions

- [E] What does it mean when a function is differentiable?
  - A function is differentiable at a point if it has a derivative at that point. This means the function's graph has a tangent at that point and the function changes at a rate that can be described by that tangent.
- [E] Give an example of when a function doesn't have a derivative at a point.
  - An example is the function  $f(x) = |x|$ , which doesn't have a derivative at  $x = 0$ . The graph of this function has a sharp corner at  $x = 0$ , which means there is no single tangent line at that point.
- [M] Give an example of non-differentiable functions that are frequently used in machine learning.  
How do we do backpropagation if those functions aren't differentiable?

An example is the ReLU (Rectified Linear Unit) function, which is non-differentiable at  $x = 0$ . In machine learning, backpropagation with such functions often uses a concept called subgradient, which allows the algorithm to bypass non-differentiability at certain points. For ReLU, the derivative is defined as 0 for  $x < 0$  and 1 for  $x > 0$ , and at  $x = 0$ , any value between 0 and 1 can be used.

## Convexity

- [E] What does it mean for a function to be convex or concave? Draw it.

A function is convex if the line segment between any two points on the graph of the function lies above or on the graph. A function is concave if this line segment lies below or on the graph.

- [E] Why is convexity desirable in an optimization problem?

Convexity is desirable because it ensures that any local minimum is also a global minimum, making the optimization problem easier to solve. Convex problems have well-understood properties and can often be solved efficiently.

- **[M] Show that the cross-entropy loss function is convex.**

The cross-entropy loss function for binary classification, defined as  $L(y, p) = -[y \log(p) + (1 - y) \log(1 - p)]$ , is convex with respect to  $p$ . This can be shown by taking the second derivative of the loss function with respect to  $p$  and proving that it is non-negative for  $p$  in the range  $(0, 1)$ .

- **Most ML algorithms we use nowadays use first-order derivatives (gradients) to construct the next training iteration.**

- **[E] How can we use second-order derivatives for training models?**

Second-order derivatives can be used in optimization algorithms to better understand the curvature of the loss function. This information can be used to adjust the learning rate and the direction of the update steps, potentially leading to faster convergence.

- **[M] Pros and cons of second-order optimization.**

- Pros: Can lead to faster convergence and more informed update steps.
- Cons: Computationally expensive, as it requires calculating and inverting the Hessian matrix.

- **[M] Why don't we see more second-order optimization in practice?**

Second-order optimization methods are not used as frequently because they are computationally intensive, especially for large models, and the benefit in convergence speed does not always justify the additional computational cost.

- **[M] How can we use the Hessian (second derivative matrix) to test for critical points?**

The Hessian matrix can be used to test the nature of critical points. If the Hessian at a point is positive definite, the point is a local minimum; if it is negative definite, the point is a local maximum; and if it is indefinite, the point is a saddle point.

- [M] Let  $x \in \mathbb{R}^n$ ,  $L = \text{crossentropy}(\text{softmax}(x), y)$  in which  $y$  is a one-hot vector. Take the derivative of  $L$  with respect to  $x$ .

The derivative of the cross-entropy loss with respect to  $x$ , when using a softmax output layer, simplifies to the difference between the predicted probabilities and the actual labels (in the one-hot vector  $y$ ).

- [M] Given the function  $f(x, y) = 4x^2 - y$  with the constraint  $x^2 + y^2 = 1$ . Find the function's maximum and minimum values.

use the method of Lagrange multipliers. This involves setting up the Lagrangian  $L(x, y, \lambda) = 4x^2 - y + \lambda(1 - x^2 - y^2)$ , taking its derivatives with respect to  $x$ ,  $y$ , and  $\lambda$ , setting them to zero, and solving the resulting system of equations. The solutions will give the maximum and minimum values of  $f$  under the given constraint.

## ▼ Probability Questions

1. [E] Given a uniform random variable  $X$  in the range of  $[0,1]$  inclusively. What's the probability that  $X=0.5$ ?

For a continuous uniform distribution, the probability of  $X$  being exactly any specific value, including 0.5, is 0. This is because the probability for a continuous distribution is defined over intervals, not specific points.

2. [E] Can the values of PDF be greater than 1? If so, how do we interpret PDF?

Yes, the values of a Probability Density Function (PDF) can be greater than 1. The key point is that the area under the PDF curve over the entire range must integrate to 1. A high PDF value does not represent probability but rather indicates a higher density of the variable at that point.

3. [E] What's the difference between multivariate distribution and multimodal distribution?

A multivariate distribution is a probability distribution with more than one random variable, each with its range of values. A multimodal distribution is a probability distribution with more than one peak or mode,

regardless of how many variables it has.

4. **[E] What does it mean for two variables to be independent?**

Two variables are independent if the occurrence or the value of one does not affect the occurrence or value of the other. In probabilistic terms, two variables X and Y are independent if the probability of X and Y occurring together is the product of their individual probabilities.

5. **[E] It's a common practice to assume an unknown variable to be of the normal distribution. Why is that?**

This assumption is often made due to the Central Limit Theorem, which states that the distribution of the sum (or average) of a large number of independent, identically distributed variables tends toward a normal distribution, regardless of the shape of the original distribution.

6. **[E] How would you turn a probabilistic model into a deterministic model?**

To convert a probabilistic model into a deterministic one, you typically use expected values, mode, or median of the probability distributions as fixed values instead of random variables. This approach ignores the variability and uncertainty represented by the probability distributions.

7. **[H] Is it possible to transform non-normal variables into normal variables? How?**

Yes, it's possible through transformations such as the Box-Cox transformation or the log transformation. These methods adjust the distribution of the data to approximate a normal distribution more closely.

8. **[M] When is the t-distribution useful?**

The t-distribution is particularly useful when dealing with small sample sizes or when the population standard deviation is unknown. It's commonly used in hypothesis testing and confidence interval estimation for means, especially in scenarios with limited data.

9. Assume you manage an unreliable file storage system that crashed 5 times in the last year, each crash happens independently.
  - a. [M] What's the probability that it will crash in the next month?

Assuming a Poisson distribution, with an average rate of 5 crashes per year, the probability of a crash in any given month (1/12 of a year) can be calculated using the Poisson probability formula.
  - b. [M] What's the probability that it will crash at any given moment?

This is difficult to quantify without a specific time frame. The Poisson distribution can provide probabilities over a specified interval but not for a 'moment.'
10. [M] Say you built a classifier to predict the outcome of football matches. In the past, it's made 10 wrong predictions out of 100. Assume all predictions are made independently, what's the probability that the next 20 predictions are all correct?

Assuming the past error rate of 10% continues, the probability of making 20 correct predictions in a row can be calculated as  $(0.9)^{20}$ , as each prediction is independent.
11. [M] Given two random variables X and Y. We have the values P(X|Y) and P(Y) for all values of X and Y. How would you calculate P(X)?

To calculate  $P(X)$ , you can use the law of total probability.  $P(X)$  is the sum of  $P(X|Y)P(Y)$  over all possible values of  $Y$ . Mathematically, it's expressed as  $P(X) = \sum P(X|Y=y)P(Y=y)$ , summed over all  $y$ .
12. [M] You know that your colleague Jason has two children and one of them is a boy. What's the probability that Jason has two sons? Hint: it's not 1/2.

The sample space of possibilities, given that one child is a boy, includes boy-boy, boy-girl, and girl-boy (the order matters). Out of these three possibilities, only one is the event where both children are boys. Therefore, the probability that Jason has two sons is 1/3.

13. **There are only two electronic chip manufacturers: A and B, both manufacture the same amount of chips. A makes defective chips with a probability of 30%, while B makes defective chips with a probability of 70%.**
- [E] If you randomly pick a chip from the store, what is the probability that it is defective?**

The probability that a randomly picked chip is defective is the average of the defect rates, as both companies produce the same number of chips. So, the probability is  $(30\% + 70\%) / 2 = 50\%$ .
  - [M] Suppose you now get two chips coming from the same company, but you don't know which one. When you test the first chip, it appears to be functioning. What is the probability that the second electronic chip is also good?**
  - The probability that both chips are good depends on which company they came from. For company A, the probability of two good chips is  $0.7 \times 0.7 = 0.49$ , and for B, it's  $0.3 \times 0.3 = 0.09$ . The total probability is the average,  $(0.49 + 0.09) / 2 = 0.29$ .
14. **There's a rare disease that only 1 in 10,000 people get. Scientists have developed a test to diagnose the disease with the false positive rate and false negative rate of 1%.**
- [E] Given a person is diagnosed positive, what's the probability that this person actually has the disease?**

This can be solved using Bayes' theorem. Let D be the event of having the disease and + be a positive test.  $P(D|+) = [P(+|D)P(D)] / [P(+|D)P(D) + P(+|\text{not } D)P(\text{not } D)]$ . Substituting the values, we get  $P(D|+) = [0.99 \times 0.0001] / [0.99 \times 0.0001 + 0.01 \times 0.9999] \approx 0.0098$  or 0.98%.
  - [M] What's the probability that a person has the disease if two independent tests both come back positive?**

If two independent tests come back positive, the probability of having the disease is higher. Using Bayes' theorem, let's denote  $P(D|++, \text{not } D)$  as the probability of not having the disease after two positive tests.  $P(D|++) = 1 - P(\text{not } D|++)$ .  $P(\text{not } D|++) = [P(++|\text{not } D)P(\text{not } D)] / [P(++|D)P(D) + P(++|\text{not } D)P(\text{not } D)]$ . Substituting the values, we get a much higher probability than a single test.

15. **[M] A dating site allows users to select 10 out of 50 adjectives to describe themselves. Two users are said to match if they share at least 5 adjectives. If Jack and Jin randomly pick adjectives, what is the probability that they match?**

The probability that Jack and Jin match (share at least 5 adjectives) can be calculated using combinatorics. We calculate the total number of ways they can choose adjectives and the number of ways they can choose so that they share at least 5. The probability is then the number of matching scenarios divided by the total number of scenarios. This requires summing over the scenarios where they share exactly 5, 6, 7, 8, 9, and all 10 adjectives, each of which can be calculated using combinations.

16. **[M] Consider a person A whose sex we don't know. We know that for the general human height, there are two distributions: the height of males follows  $h_m = N(\mu_m, \sigma_m^2)$  and the height of females follows  $h_j = N(\mu_j, \sigma_j^2)$ . Derive a probability density function to describe A's height.**
17. **[H] There are three weather apps, each the probability of being wrong 1/3 of the time. What's the probability that it will be foggy in San Francisco tomorrow if all the apps predict that it's going to be foggy in San Francisco tomorrow and during this time of the year, San Francisco is foggy 50% of the time? (Hint: consider both independent and dependent cases.)**

- For independent cases: Assume each app independently has a 1/3 chance of being wrong. The probability that all three are correct simultaneously is  $(2/3) \times (2/3) \times (2/3)$ . Given that they all agree it will be foggy, the probability of it actually being foggy, using Bayes' theorem, is  $P(\text{Foggy}|\text{All say Foggy}) = [P(\text{All say Foggy}|\text{Foggy}) \times P(\text{Foggy})] / P(\text{All say Foggy})$ . Substitute the probabilities and calculate.

- For dependent cases: If the apps' predictions are not independent (e.g., they all use the same data source), the scenario changes. We treat the situation as if there's only one source. So, the probability they're all correct is 2/3, and we use Bayes' theorem as above but with this adjusted probability.

18. **[M] Given n samples from a uniform distribution  $[0, d]$ . How do you estimate d? (Also known as the German tank problem)**

An estimator for  $d$  can be the maximum value observed in the  $n$  samples (let's call it  $m$ ) adjusted with the sample size. A commonly used formula is an unbiased estimator:  $d \approx m + (m/n) - 1$ .

19. **[M] You're drawing from a random variable that is normally distributed,  $X \sim N(0, 1)$ , once per day. What is the expected number of days that it takes to draw a value that's higher than 0.5?**

First, find the probability of drawing a value higher than 0.5 from a standard normal distribution. This is 1 minus the cumulative distribution function (CDF) value at 0.5. Let this probability be  $p$ . The number of days is a geometric distribution with parameter  $p$ , and the expected number of days is  $1/p$ .

20. **[M] You're part of a class. How big the class has to be for the probability of at least a person sharing the same birthday with you is greater than 50%?**

To find the class size where the probability of at least one person sharing the same birthday with you is greater than 50%, use the complement of the probability that no one shares your birthday. The probability that a single individual doesn't share your birthday is  $364/365$ . For a class of  $n$  people, the probability that none of them share your birthday is  $(364/365)^n$ . Set this less than 0.5 and solve for  $n$ .

21. **[H] You decide to fly to Vegas for a weekend. You pick a table that doesn't have a bet limit, and for each game, you have the probability  $p$  of winning, which doubles your bet, and  $1-p$  of losing your bet. Assume that you have unlimited money, is there a betting strategy that has a guaranteed positive payout, regardless of the value of  $p$ ?**

This is a scenario of the Martingale betting strategy, where you double your bet after each loss. With unlimited money and no bet limit, theoretically, this strategy can guarantee a positive payout. However, it requires an infinite bankroll, an infinite amount of time, and the absence of a house betting limit, which are not practical.

22. **[H] Given a fair coin, what's the number of flips you have to do to get two consecutive heads?**

This problem can be solved using Markov chains or recursive probability. Denote the expected number of additional flips needed after getting a head as  $E(H)$  and after getting a tail or starting fresh as  $E(T)$ . Set up the equations  $E(T) = 1 + 0.5E(H) + 0.5E(T)$  and  $E(H) = 1 + 0.5 + 0.5E(T)$ , and solve for  $E(T)$ .

23. **[H] In national health research in the US, the results show that the top 3 cities with the lowest rate of kidney failure are cities with populations under 5,000. Doctors originally thought that there must be something special about small town diets, but when they looked at the top 3 cities with the highest rate of kidney failure, they are also very small cities. What might be a probabilistic explanation for this phenomenon? (Hint: The law of small numbers.)**

This phenomenon can be explained by the law of small numbers, which is a type of statistical fluctuation. Small sample sizes (like small cities) are more likely to have extreme values purely by chance. Therefore, both the lowest and highest rates of kidney failure being in small towns could be due to random variation rather than any specific cause related to the towns themselves.

24. **[M] Derive the maximum likelihood estimator of an exponential distribution.**

▼ Stats

1. **[E] Explain frequentist vs. Bayesian statistics.**

- Frequentist statistics interprets probability as the long-run frequency of events occurring in repeated experiments or trials. It does not incorporate prior beliefs or information.
- Bayesian statistics, on the other hand, interprets probability as a measure of belief or certainty about a hypothesis, given the current evidence. It incorporates prior beliefs or information and updates these beliefs with new data.

2. **[E] Given the array [1,5,3,2,4,4], find its mean, median, variance, and standard deviation.**

- Mean:  $(1 + 5 + 3 + 2 + 4 + 4) / 6$
- Median: The middle value when the data is sorted (in this case, 3 and 4), so the median is  $(3 + 4) / 2$
- Variance: The average of the squared differences from the mean
- Standard deviation: The square root of the variance

3. **[M] When should we use median instead of mean? When should we use mean instead of median?**

- Use median instead of mean when the data is skewed or has outliers, as the median is more robust against them. The median represents the middle point and is not influenced by extreme values.
- Use mean instead of median when the data is symmetrically distributed and lacks outliers, as the mean considers all values and is a better measure of the central tendency for such data.

4. **[M] What is a moment of function? Explain the meanings of the zeroth to fourth moments.**

Moments of a function are quantitative measures related to the shape of the function's graph.

- The zeroth moment is the total area under the curve (for probability distributions, this is 1).
- The first moment is the mean, representing the center of mass.
- The second moment is the variance, representing the spread or dispersion.

- The third moment is the skewness, indicating the asymmetry of the distribution.
  - The fourth moment is the kurtosis, indicating the 'tailedness' of the distribution.
5. **[M] Are independence and zero covariance the same? Give a counterexample if not.**
- Independence and zero covariance are not the same. Independence implies zero covariance, but zero covariance does not necessarily imply independence.
  - Counterexample: Consider X as a random variable uniformly distributed between -1 and 1, and let Y =  $X^2$ . Here, the covariance between X and Y is zero, but they are not independent.
6. **[E] Suppose that you take 100 random newborn puppies and determine that the average weight is 1 pound with the population standard deviation of 0.12 pounds. Assuming the weight of newborn puppies follows a normal distribution, calculate the 95% confidence interval for the average weight of all newborn puppies.**
- The 95% confidence interval can be calculated using the formula: mean  $\pm$  (Z-score \* standard deviation / sqrt(sample size)). For a 95% confidence interval, the Z-score is typically 1.96 for a normal distribution. So, the interval is  $1 \pm (1.96 * 0.12 / \sqrt{100})$ . Calculate the exact values to get the interval.
7. **[M] Suppose that we examine 100 newborn puppies and the 95% confidence interval for their average weight is [0.9,1.1] pounds. Which of the following statements is true?**
- a. **Given a random newborn puppy, its weight has a 95% chance of being between 0.9 and 1.1 pounds.**
  - b. **If we examine another 100 newborn puppies, their mean has a 95% chance of being in that interval.**
  - c. **We're 95% confident that this interval captured the true mean weight.**

**Hint: This is a subtle point that many people misunderstand. If you struggle with the answer, Khan Academy has a great article on it.**

The correct interpretation of the 95% confidence interval [0.9,1.1] pounds is option 3: "We're 95% confident that this interval captured the true mean weight." This means that if we were to take many such samples and calculate a 95% confidence interval for each, about 95% of these intervals would contain the true mean weight. It does not make specific predictions about individual measurements or another sample's mean.

8. **[H] Suppose we have a random variable  $X$  supported on  $[0, 1]$  from which we can draw samples. How can we come up with an unbiased estimate of the median of  $X$ ?**

To come up with an unbiased estimate of the median, you can draw a large number of samples from  $X$  and then use the empirical median (the middle value of the ordered samples) as an estimate. As the number of samples increases, this empirical median will converge to the true median of  $X$  due to the law of large numbers.

9. **[H] Can correlation be greater than 1? Why or why not? How to interpret a correlation value of 0.3?**

No, correlation values range between -1 and 1. A correlation greater than 1 or less than -1 is not possible. A correlation of 0.3 indicates a positive but weak linear relationship between two variables.

10. **The weight of newborn puppies is roughly symmetric with a mean of 1 pound and a standard deviation of 0.12. Your favorite newborn puppy weighs 1.1 pounds.**

- a. **[E] Calculate your puppy's z-score (standard score).**

The z-score is calculated as (puppy's weight - mean) / standard deviation. So,  $z = (1.1 - 1) / 0.12$ .

- b. **[E] How much does your newborn puppy have to weigh to be in the top 10% in terms of weight?**

To be in the top 10%, find the z-score corresponding to the 90th percentile, which is approximately 1.28. Then use the formula: weight = mean + (z-score x standard deviation).

- c. [M] Suppose the weight of newborn puppies followed a skew distribution. Would it still make sense to calculate z-scores?

In a skewed distribution, z-scores can still be calculated, but they might not be as informative as in symmetric distributions. The mean and standard deviation are influenced by skewness, and z-scores might not accurately reflect the distribution's characteristics.

11. [H] Tossing a coin ten times resulted in 10 heads and 5 tails. How would you analyze whether a coin is fair?

To analyze whether the coin is fair, you can perform a hypothesis test, such as a chi-square test or a binomial test. The null hypothesis would be that the coin is fair (equal probability for heads and tails). Calculate the test statistic and the p-value to determine if there's significant evidence to reject the null hypothesis in favor of the alternative hypothesis (the coin is not fair).

12. Statistical significance.

- a. [E] How do you assess the statistical significance of a pattern whether it is a meaningful pattern or just by chance?

To assess the statistical significance, you typically use hypothesis testing. This involves formulating a null hypothesis (usually representing no effect or no difference) and an alternative hypothesis. You then calculate a p-value, which is the probability of observing the data, or something more extreme, under the null hypothesis. If this p-value is below a certain threshold (commonly 0.05), the result is considered statistically significant, indicating that the observed pattern is unlikely to have occurred by chance.

- b. [E] What's the distribution of p-values?

Under the null hypothesis, p-values follow a uniform distribution between 0 and 1. This is because every probability value is equally likely if the null hypothesis is true.

- c. **[H] Recently, a lot of scientists started a war against statistical significance. What do we need to keep in mind when using p-value and statistical significance?**

The "war" against statistical significance includes concerns about over-reliance on p-values. Key points to remember are that a statistically significant result does not imply practical significance, p-values do not measure the size of an effect or the importance of a result, and they do not provide a good measure of evidence regarding a model or hypothesis.

### 13. **Variable correlation.**

- a. **[M] What happens to a regression model if two of their supposedly independent variables are strongly correlated?**

Strong correlation between independent variables (multicollinearity) can make it difficult to determine the individual effect of each variable on the dependent variable. It can inflate the variance of the coefficient estimates and make them unstable and sensitive to changes in the model.

- b. **[M] How do we test for independence between two categorical variables?**

Independence between two categorical variables can be tested using the chi-square test of independence. This test compares the observed frequencies in each category against the frequencies expected if the two variables were independent.

- c. **[H] How do we test for independence between two continuous variables?**

To test independence between two continuous variables, correlation coefficients (like Pearson's or Spearman's correlation) can be used. Another approach is to use hypothesis testing with a model (like linear regression) and check if the coefficient of one variable is significantly different from zero.

14. **[E] A/B testing is a method of comparing two versions of a solution against each other to determine which one performs better. What are some of the pros and cons of A/B testing?**

- Pros: A/B testing provides a straightforward way to compare two versions directly against each other in a real-world setting. It helps in making data-driven decisions and can reveal user preferences and behavior.
- Cons: A/B testing can be time-consuming and requires a large sample size to detect small differences. It only tests what is measurable in the short term and might not capture long-term effects or user satisfaction comprehensively.

15. **[M] You want to test which of the two ad placements on your website is better. How many visitors and/or how many times each ad is clicked do we need so that we can be 95% sure that one placement is better?**

To determine the required number of visitors or clicks for a 95% confidence level, you need to conduct a power analysis, which takes into account the expected effect size, the power of the test (commonly set at 0.8), and the significance level. Tools and statistical software packages can perform these calculations.

16. **[M] Your company runs a social network whose revenue comes from showing ads in newsfeed. To double revenue, your coworker suggests that you should just double the number of ads shown. Is that a good idea? How do you find out?**

To evaluate this idea, conduct experiments or A/B tests to measure user engagement, satisfaction, and ad revenue with different numbers of ads. It's essential to consider user experience and long-term engagement, as an increase in ads might lead to a negative user experience and potential long-term revenue loss.

17. **Imagine that you have the prices of 10,000 stocks over the last 24 month period and you only have the price at the end of each month, which means you have 24 price points for each stock. After**

calculating the correlations of  $10,000 * 9,999$  pairs of stock, you found a pair that has the correlation to be above 0.8.

a. [E] What's the probability that this happens by chance?

The probability of finding a high correlation by chance increases with the number of comparisons. In big data, even rare events become probable. For 10,000 stocks compared pairwise, the chance of finding a correlation above 0.8 by random chance is not negligible.

b. [M] How to avoid this kind of accidental patterns?

Hint: Check out the curse of big data.

To avoid such accidental patterns, use techniques like Bonferroni correction to adjust the significance level for multiple comparisons. Another approach is to use cross-validation or out-of-sample testing to confirm whether the observed correlations hold in different data sets.

18. [H] How are sufficient statistics and Information Bottleneck Principle used in machine learning?

Sufficient statistics and the Information Bottleneck Principle are used to extract relevant information from data efficiently. Sufficient statistics summarize data in a way that no other data about the original distribution is needed. The Information Bottleneck Principle is used to find a trade-off between compression and retention of relevant information, often applied in deep learning to understand and optimize neural networks. These concepts help in reducing the complexity of data and models, leading to more efficient and interpretable machine learning algorithms.

## ▼ Chapter 6: Computer Science Answer

### ▼ Algorithms

- Write a Python function to recursively read a JSON file.

```
import json

def read_json_recursively(file_path):
    with open(file_path, 'r') as file:
        data = json.load(file)

    def parse_json(data):
        if isinstance(data, dict):
            for key, value in data.items():
                print(f"Key: {key}")
                parse_json(value)
        elif isinstance(data, list):
            for item in data:
                parse_json(item)
        else:
            print(f"Value: {data}")

    parse_json(data)
```

- Implement an O(NlogN) sorting algorithm, preferably quick sort or merge sort.

```
def merge_sort(arr):
    if len(arr) > 1:
```

```
mid = len(arr) // 2
L = arr[:mid]
R = arr[mid:]

merge_sort(L)
merge_sort(R)

i = j = k = 0

while i < len(L) and j < len(R):
    if L[i] < R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

while i < len(L):
    arr[k] = L[i]
    i += 1
    k += 1

while j < len(R):
    arr[k] = R[j]
    j += 1
    k += 1
```

```
    k += 1
return arr
```

- Find the longest increasing subsequence in a string.

```
def longest_increasing_subsequence(s):
    dp = [1] * len(s)
    for i in range(1, len(s)):
        for j in range(0, i):
            if s[i] > s[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

- Find the longest common subsequence between two strings.

```
def longest_common_subsequence(s1, s2):
    dp = [[0] * (len(s2)+1) for _ in range(len(s1)+1)]
    for i in range(1, len(s1)+1):
        for j in range(1, len(s2)+1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[-1][-1]
```

- Traverse a tree in pre-order, in-order, and post-order.

```

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def preorder_traversal(root):
    return [root.val] + preorder_traversal(root.left) + preorder_traversal(root.right)

def inorder_traversal(root):
    return inorder_traversal(root.left) + [root.val] + inorder_traversal(root.right)

def postorder_traversal(root):
    return postorder_traversal(root.left) + postorder_traversal(root.right) + [root.val]

```

- Given an array of integers and an integer k, find the total number of continuous subarrays whose sum equals k. The solution should have O(N) runtime.

```

def subarray_sum(nums, k):
    count = 0
    current_sum = 0
    prefix_sums = {0: 1}

```

```

for num in nums:
    current_sum += num
    count += prefix_sums.get(current_sum - k, 0)
    prefix_sums[current_sum] = prefix_sums.get(current_sum, 0) + 1

return count

```

- There are two sorted arrays `nums1` and `nums2` with `m` and `n` elements respectively. Find the median of the two sorted arrays. The solution should have  $O(\log(m+n))$  runtime.

Finding the median of two sorted arrays in  $O(\log(m+n))$  time is a classic algorithm problem that is best solved by using a binary search technique. The key insight is to use the properties of a median to partition both arrays into two halves such that:

1. The length of the left half is equal to the length of the right half (or one more if the combined length is odd).
2. Every element in the left half is less than or equal to every element in the right half.

Here's a step-by-step explanation of the algorithm:

- We perform a binary search on the smaller of the two arrays to find a partition that satisfies the two conditions above.
- At each step, we calculate the median of the current partitions.
- We adjust the partition positions based on the comparison of elements adjacent to the partitions in both arrays.
- The search continues until we find a partition where the two conditions are met.

Here's the code implementing this logic:

```
def findMedianSortedArrays(nums1, nums2):
    # Make sure nums1 is the smaller array
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1
    m, n = len(nums1), len(nums2)

    imin, imax, half_len = 0, m, (m + n + 1) // 2

    while imin <= imax:
        i = (imin + imax) // 2
        j = half_len - i

        if i < m and nums2[j-1] > nums1[i]:
            # i is too small, must increase it
            imin = i + 1
        elif i > 0 and nums1[i-1] > nums2[j]:
            # i is too big, must decrease it
            imax = i - 1
        else:
            # i is perfect
            if i == 0: max_of_left = nums2[j-1]
            elif j == 0: max_of_left = nums1[i-1]
            else: max_of_left = max(nums1[i-1], nums2[j-1])
```

```

        if (m + n) % 2 == 1:
            return max_of_left

        if i == m: min_of_right = nums2[j]
        elif j == n: min_of_right = nums1[i]
        else: min_of_right = min(nums1[i], nums2[j])

    return (max_of_left + min_of_right) / 2.0

# Example usage:
nums1 = [1, 3]
nums2 = [2]
median = findMedianSortedArrays(nums1, nums2)

```

This function will find the median of the two sorted arrays `nums1` and `nums2` in  $O(\log(\min(m, n)))$  time, which is within the required time complexity. The binary search is conducted on the smaller array, which ensures the logarithmic time complexity with respect to the total length of the two arrays.

- **Write a program to solve a Sudoku puzzle by filling the empty cells. The board is of size 9×9. It contains only 1-9 numbers. Empty cells are denoted with \*. Each board has one unique solution.**

Solving a Sudoku puzzle programmatically can be efficiently achieved using a backtracking algorithm. This algorithm works by trying to fill cells one by one, moving forward when possible and backtracking when it encounters a cell that cannot be filled with any number that doesn't violate Sudoku rules.

Here's the outline of the algorithm:

1. Find an empty cell (denoted by `*` in this case).

2. Attempt to place numbers 1-9 in that cell, checking if the current board remains valid:
  - Each number must appear exactly once in each row.
  - Each number must appear exactly once in each column.
  - Each number must appear exactly once in each of the 9 3x3 subgrids.
3. If a number is valid, recursively attempt to fill in the rest of the board.
4. If no numbers are valid or if the recursion is unable to complete the board, backtrack and try a different number in the previous step.
5. If all cells are filled, the puzzle is solved.

Here is a Python program that implements this strategy:

```
def solve_sudoku(board):  
    empty = find_empty_location(board)  
    if not empty:  
        return True # Puzzle solved  
    row, col = empty  
  
    for num in range(1, 10):  
        if is_safe(board, row, col, str(num)):  
            board[row][col] = str(num)  
            if solve_sudoku(board):  
                return True  
            board[row][col] = '*' # Backtrack
```

```

        return False

def find_empty_location(board):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == '*':
                return (i, j)
    return None

def used_in_row(board, row, num):
    return num in board[row]

def used_in_col(board, col, num):
    return num in [board[i][col] for i in range(len(board))]

def used_in_box(board, box_start_row, box_start_col, num):
    for i in range(3):
        for j in range(3):
            if board[i + box_start_row][j + box_start_col] == num:
                return True
    return False

def is_safe(board, row, col, num):
    return not used_in_row(board, row, num) and \
           not used_in_col(board, col, num) and \
           not used_in_box(board, row - row % 3, col - col % 3, num)

```

```

# Example usage
sudoku_board = [
    ["5", "3", "*", "*", "7", "*", "*", "*", "*"],
    ["6", "*", "*", "1", "9", "5", "*", "*", "*"],
    ["*", "9", "8", "*", "*", "*", "*", "6", "*"],
    ["8", "*", "*", "*", "6", "*", "*", "*", "3"],
    ["4", "*", "*", "8", "*", "3", "*", "*", "1"],
    ["7", "*", "*", "*", "2", "*", "*", "*", "6"],
    ["*", "6", "*", "*", "*", "*", "2", "8", "*"],
    ["*", "*", "*", "4", "1", "9", "*", "*", "5"],
    ["*", "*", "*", "*", "8", "*", "*", "7", "9"]
]

if solve_sudoku(sudoku_board):
    for row in sudoku_board:
        print(" ".join(row))
else:
    print("No solution exists")

```

This program will fill in the `*`s in the `sudoku_board` with the correct numbers to solve the Sudoku puzzle, if a solution exists. Since Sudoku puzzles with unique solutions are a form of constraint satisfaction problems, the backtracking algorithm is a natural fit for solving them.

- Given a memory block represented by an empty array, write a program to manage the dynamic allocation of that memory block. The program should support two methods: `malloc()` to allocate

## memory and `free()` to free a memory block.

In a simplified simulation of memory allocation in Python, we can represent the memory block with a list and manage allocations using a dictionary to keep track of allocated and freed memory blocks. This program won't directly manage memory at a low level since Python abstracts away manual memory management, but it can demonstrate the logic of `malloc` and `free` functions.

Here's a basic implementation:

```
class MemoryBlock:
    def __init__(self, size):
        self.size = size
        self.memory = [None] * size # Simulate memory block with a list
        self.allocations = {} # Dictionary to store the allocation starting : 

    def malloc(self, size):
        """Allocate a block of memory."""
        # Find a block of memory large enough to allocate
        start = self._find_block(size)
        if start is not None:
            self.allocations[start] = size
            return start
        else:
            raise MemoryError("Out of memory")

    def free(self, address):
```

```

"""Free an allocated block of memory."""
if address in self.allocations:
    size = self.allocations.pop(address)
    for i in range(size):
        self.memory[address + i] = None
else:
    raise ValueError("Invalid free address")

def _find_block(self, size):
    """Find a contiguous block of memory of a certain size."""
    if not self.allocations:
        return 0 if size <= self.size else None

    # Sort allocations by address
    sorted_allocs = sorted(self.allocations.items())

    # Check for space before the first block
    if sorted_allocs[0][0] >= size:
        return 0

    # Check for space between blocks
    for i in range(len(sorted_allocs) - 1):
        start = sorted_allocs[i][0] + sorted_allocs[i][1]
        end = sorted_allocs[i + 1][0]
        if end - start >= size:
            return start

```

```

# Check for space after the last block
last_alloc_start, last_alloc_size = sorted_allocs[-1]
if self.size - (last_alloc_start + last_alloc_size) >= size:
    return last_alloc_start + last_alloc_size

return None

# Example usage:
memory = MemoryBlock(1024) # Initialize a block of memory with 1024 units
ptr1 = memory.malloc(10) # Allocate 10 units
memory.free(ptr1) # Free the allocated 10 units

```

In this code, `malloc` looks for a contiguous free block of memory by checking the gaps between already allocated blocks. If it finds a suitable block, it marks it as allocated. The `free` function marks the memory starting at the given address as free.

This is a high-level representation and does not reflect the actual complexity of memory management in a real operating system, where memory fragmentation, alignment, and other low-level considerations come into play. However, this example captures the essence of the `malloc` and `free` operations.

- **Given a string of mathematical expression, such as "10 \* 4 + (4 + 3) / (2 - 1)", calculate it. It should support four operators +, -, :, /, and the brackets ().**

Evaluating a mathematical expression with various operators and parentheses can be done by converting the expression into a form that can be computed directly, like Reverse Polish Notation (RPN), or by using a

recursive approach to handle the precedence and associativity of operators.

In Python, we can utilize the built-in `eval` function for this purpose, which can evaluate a string containing a Python expression. However, using `eval` is generally unsafe if the input is not sanitized, as it can execute arbitrary code. For the sake of this exercise, assuming the input is always a safe, well-formed mathematical expression and replacing `:` with `/` for division, you could use `eval` carefully.

Here's how you could use `eval` to calculate such an expression:

```
def calculate_expression(expr):
    # Replace ':' with '/' to handle division
    expr = expr.replace(':', '/')
    try:
        # Evaluate the expression and return the result
        return eval(expr)
    except Exception as e:
        # Handle errors like syntax errors or division by zero
        print(f"Error evaluating expression: {e}")

# Example usage:
expression = "10 * 4 + (4 + 3) / (2 - 1)"
result = calculate_expression(expression)
print(result)
```

If you want to write a function from scratch without using `eval`, you would need to implement a parser that converts the expression to RPN and then evaluate the RPN expression, or use a recursive descent parser

to evaluate the expression directly based on operator precedence and associativity rules.

Here is a simplified example of an expression evaluator without using `eval`:

```
import re
import operator

def calculate_expression(expr):
    # Define operator functions
    ops = {
        '+': operator.add,
        '-': operator.sub,
        '*': operator.mul,
        '/': operator.truediv
    }

    # Normalize the expression and split into tokens
    tokens = re.findall(r'\d+|\+|\-|\*|\//|(\(|\))', expr.replace(':', '/'))

    def precedence(op):
        precedences = {'+": 1, "-": 1, "*": 2, "/": 2}
        return precedences.get(op, 0)

    def apply_operator(operators, values):
        op = operators.pop()
```

```

        right = values.pop()
        left = values.pop()
        values.append(ops[op](left, right))

def greater_precedence(op1, op2):
    return precedence(op1) > precedence(op2)

operators = []
values = []
for token in tokens:
    if token.isdigit():
        values.append(int(token))
    elif token == '(':
        operators.append(token)
    elif token == ')':
        while operators[-1] != '(':
            apply_operator(operators, values)
        operators.pop() # Remove the '('
    else:
        while (operators and operators[-1] in ops and
               greater_precedence(operators[-1], token)):
            apply_operator(operators, values)
        operators.append(token)

while operators:
    apply_operator(operators, values)

```

```
    return values[0]

# Example usage:
expression = "10 * 4 + (4 + 3) / (2 - 1)"
result = calculate_expression(expression)
print(result)
```

This function processes the expression string and computes the result using a stack-based algorithm for operator precedence and associativity. It's a much more involved process than using `eval`, but it's a safer and more educational approach.

- [Given a directory path, descend into that directory and find all the files with duplicated content.](#)

To find all files with duplicated content within a given directory and its subdirectories, you can use a hash function to generate a unique fingerprint for each file's content. By comparing these hashes, you can identify which files are duplicates.

Here's a Python script that implements this logic:

```
import os
import hashlib
from collections import defaultdict

def hash_file(file_path):
    hasher = hashlib.md5()
```

```

        with open(file_path, 'rb') as afile:
            buf = afile.read()
            hasher.update(buf)
        return hasher.hexdigest()

def find_duplicates(root_directory):
    # Dictionary to store the hash and corresponding file paths
    hashes = defaultdict(list)

    # Traverse the directory and its subdirectories
    for subdir, dirs, files in os.walk(root_directory):
        for filename in files:
            file_path = os.path.join(subdir, filename)
            file_hash = hash_file(file_path)
            hashes[file_hash].append(file_path)

    # Filter the hash entries where there are duplicates
    return {hash_value: paths for hash_value, paths in hashes.items() if len(paths) > 1}

# Example usage
root_directory = '/path/to/directory'
duplicates = find_duplicates(root_directory)

for hash_value, file_paths in duplicates.items():
    print(f"Duplicate files for hash {hash_value}:")
    for path in file_paths:

```

```
print(f"  {path}")
```

This script uses the MD5 hash function to create a fingerprint for each file. If a file's hash is already in the `hashes` dictionary, it means that the file is a duplicate (or at least that it has the exact same content as another file). The `find_duplicates` function then returns a dictionary of hashes to lists of file paths, where each list contains the paths of files that are duplicates of each other.

Before running the script, make sure to replace `'/path/to/directory'` with the actual path to the directory you want to check for duplicates. This script is a straightforward approach that works well for small to medium-sized sets of files but may need to be optimized for performance with very large datasets.

- In Google Docs, you have the Justify alignment option that spaces your text to align with both left and right margins. Write a function to print out a given text line-by-line (except the last line) in Justify alignment format. The length of a line should be configurable.

The function will take a block of text and a line width, and it will output the text with each line except the last justified to the specified line width.

```
def justify_text(text, line_width):
    """
    Justify the given text so that each line, except the last line, aligns with the specified line width.

    :param text: The text to be justified.
    :param line_width: The length of each line.
    :return: The justified text.
    """
```

```
"""
words = text.split()
current_line = []
lines = []

# Create lines of approximately the specified width
for word in words:
    if len(' '.join(current_line + [word])) <= line_width:
        current_line.append(word)
    else:
        lines.append(current_line)
        current_line = [word]

# Add the last line
lines.append(current_line)

justified_text = ""
for i in range(len(lines) - 1): # Justify all but the last line
    line = lines[i]
    if len(line) == 1: # If line has only one word, just add it
        justified_text += line[0] + '\n'
        continue

    # Calculate spaces
    total_spaces = line_width - sum(len(word) for word in line)
    spaces_needed = len(line) - 1 # Number of spaces to insert
```

```

space_counts = [(total_spaces // spaces_needed) + (1 if i < total_spaces - 1) for i in range(len(line))]

# Build the line
for word, space_count in zip(line, space_counts):
    justified_text += word + ' ' * space_count
justified_text += line[-1] + '\n'

# Add the last line as is (not justified)
justified_text += ' '.join(lines[-1])

return justified_text

# Example usage
sample_text = "This is an example of text that needs to be justified. Justify the text to a width of 40 characters."
justified = justify_text(sample_text, 40)
print(justified)

```

This function first breaks the text into words and then forms lines that are as close to the specified line width as possible without exceeding it. For each line except the last, it distributes extra spaces between words to ensure the line stretches to the full width. The last line is left as is, without justification.

- You have 1 million text files, each is a news article scraped from various news sites. Since news sites often report the same news, even the same articles, many of the files have content very similar to each other. Write a program to filter out these files so that the end result contains only files that are sufficiently different from each other in the language of your choice. You're free to choose a metric to define the “similarity” of content between files.

To filter out similar text files from a large set, we can use a technique that involves calculating a hash or fingerprint for each file based on its contents, then comparing these to find duplicates or near-duplicates. A common approach for this is to use the SimHash algorithm or some form of shingling combined with a locality-sensitive hashing (LSH) algorithm.

For the purpose of this example, I will outline a Python program using SimHash because of its efficiency with large datasets and its ability to detect near-duplicates. The SimHash algorithm computes a fingerprint of a set of features (such as words or tokens) by hashing them and then combining these hashes into a single fingerprint.

Here's a conceptual outline of the program:

1. Compute the SimHash for each file.
2. Compare the SimHashes to determine which files are similar.
3. Keep only one file from each set of similar files.

Let's write a Python script using SimHash:

```
import os
import re
from simhash import Simhash

def get_features(s):
    width = 3
    s = s.lower()
    s = re.sub(r'^\w+', ' ', s)
```

```

        return [s[i:i + width] for i in range(max(len(s) - width + 1, 1))]

def compute_simhash(filepath):
    with open(filepath, 'r', encoding='utf-8') as f:
        content = f.read()
    return Simhash(get_features(content))

# Assuming we have a function that lists all file paths
file_paths = list_all_file_paths()

# Compute simhash for each file and store in a dictionary
simhashes = {}
for path in file_paths:
    simhashes[path] = compute_simhash(path)

# Function to find similar items based on Hamming distance
def find_similar(simhashes):
    threshold = 5 # Set a threshold for Hamming distance
    similar_pairs = []
    keys = list(simhashes.keys())
    for i in range(len(keys)):
        for j in range(i + 1, len(keys)):
            if simhashes[keys[i]].distance(simhashes[keys[j]]) < threshold:
                similar_pairs.append((keys[i], keys[j]))
    return similar_pairs

```

```
# Find similar files
similar_files = find_similar(simhashes)

# Filter out similar files, keeping only one in each pair
unique_files = set(file_paths) - {pair[1] for pair in similar_files}

# Now we have a set of unique files
print(f'Total unique files: {len(unique_files)})
```

Please note that this script uses the `simhash` library, which you would need to install using `pip install simhash`. The `list_all_file_paths()` is a placeholder for a function that would return a list of file paths to process. The `find_similar` function checks for similar hashes based on the Hamming distance, which measures how many bits are different between two hashes. The threshold for similarity can be adjusted based on how similar you want the documents to be. This script removes the second file of each similar pair, but more sophisticated deduplication logic can be implemented as needed.

In practice, for a dataset of 1 million files, this approach may still be computationally intensive, and further optimizations or distributed computing might be necessary. For example, you could use a map-reduce approach to distribute the computation of SimHashes across multiple machines, then reduce by filtering out duplicates.

## ▼ Complexity and Numerical Analysis

- **Matrix multiplication**

i. [E] You have three matrices:  $A \in R^{100 \times 5}$ ,  $B \in R^{5 \times 200}$ ,  $C \in R^{200 \times 20}$  and you need to calculate the product  $ABC$ . In what order would you perform your multiplication and why?

Given the matrix dimensions for  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{C}$ , the goal is to minimize the number of scalar multiplications when computing the product  $\mathbf{ABC}$ . The dimensions are:

- $\mathbf{A}$  is  $100 \times 5$
- $\mathbf{B}$  is  $5 \times 200$
- $\mathbf{C}$  is  $200 \times 20$

If you multiply as  $(AB)\mathbf{C}$ , the first operation  $AB$  will result in a  $100 \times 200$  matrix, which will then be multiplied by a  $200 \times 20$  matrix. The total number of scalar multiplications for  $(AB)\mathbf{C}$  will be:

- First multiplication  $AB$ :  $100 \times 5 \times 200$  scalar multiplications
- Second multiplication  $(AB)\mathbf{C}$ :  $100 \times 200 \times 20$  scalar multiplications
- Total for  $(AB)\mathbf{C}$ :  $100 \times 5 \times 200 + 100 \times 200 \times 20$

Alternatively, if you multiply as  $\mathbf{A}(\mathbf{BC})$ , the first operation  $\mathbf{BC}$  will result in a  $5 \times 20$  matrix, which will then be multiplied by a  $100 \times 5$  matrix. The total number of scalar multiplications for  $\mathbf{A}(\mathbf{BC})$  will be:

- First multiplication  $\mathbf{BC}$ :  $5 \times 200 \times 20$  scalar multiplications
- Second multiplication  $\mathbf{A}(\mathbf{BC})^*$ :  $100 \times 5 \times 20$  scalar multiplications
- Total for  $\mathbf{A}(\mathbf{BC})$ :  $5 \times 200 \times 20 + 100 \times 5 \times 20$

Let's calculate the exact number of scalar multiplications required for both orders to determine which is more efficient.

The total number of scalar multiplications for computing  $(AB)\mathbf{C}$  would be 500,000, while for computing  $\mathbf{A}(\mathbf{B}\mathbf{C})$  it would be 30,000.

Therefore, it is much more efficient to compute  $\mathbf{A}(\mathbf{B}\mathbf{C})$  because it requires significantly fewer scalar multiplications compared to  $(AB)\mathbf{C}$ . The primary reason for this efficiency is that by computing  $\mathbf{B}\mathbf{C}$  first, you are reducing the size of the intermediate matrix that needs to be multiplied with  $\mathbf{A}$ , leading to fewer overall computations. This is a good example of how matrix multiplication order can have a substantial impact on computational efficiency.

ii. [M] Now you need to calculate the product of  $N$  matrices  $A_1 A_2 \dots A_n$ .

How would you determine the order in which to perform the multiplication?

- **[E] What are some of the causes for numerical instability in deep learning?**

- **Limited Precision of Floating-Point Representations:** Computers represent numbers with a finite precision which can lead to rounding errors. This is particularly problematic in deep learning where many operations can result in very small or very large numbers.
- **Gradient Exploding and Vanishing:** In deep networks, especially RNNs, gradients can become very large (explode) or very small (vanish) due to repeated multiplication through layers. This can make learning unstable or slow.
- **Poorly Scaled Inputs:** Data that is not properly normalized can lead to large variations in the activation values or gradients, causing numerical instability.

- **Inappropriate Learning Rates:** A learning rate that is too high can cause the model parameters to diverge, while a very low learning rate can lead to convergence to suboptimal weights.
- **Suboptimal Weight Initialization:** Poorly chosen initial weights can lead to saturation of neurons or poor gradient flow, contributing to instability.
- **[E] In many machine learning techniques (e.g. batch norm), we often see a small term  $\epsilon$  added to the calculation. What's the purpose of that term?**

The small term  $\epsilon$  (epsilon) is often added in machine learning calculations, such as in the denominator of batch normalization, to prevent division by zero and improve numerical stability. In many computations, especially those involving divisions or logarithms, the denominator or the argument can approach zero, which can lead to undefined behavior or numerical errors. Adding  $\epsilon$ , a very small positive number (like 1e-8), ensures that such divisions or operations remain stable and defined.

- **[E] What made GPUs popular for deep learning? How are they compared to TPUs?**
  - **GPUs:** Graphics Processing Units (GPUs) became popular in deep learning due to their ability to perform a large number of matrix and vector operations in parallel, significantly speeding up the training and inference processes of deep neural networks. They are designed for high throughput of simple, data-parallel tasks, which is common in deep learning.
  - **TPUs:** Tensor Processing Units (TPUs) are specifically designed for tensor operations in neural networks and offer higher performance and efficiency than GPUs for certain types of deep learning tasks. TPUs are tailored for a high volume of low-precision computation (like 8-bit calculations), which can be beneficial for deploying large models in production.
  - **Comparison:** While GPUs are more versatile and widely used in a range of deep learning tasks, TPUs are optimized for specific tasks, particularly in large-scale, production environments. TPUs may offer

better performance for specific applications, but GPUs are generally more accessible and support a broader range of deep learning and general compute tasks.

- **[M] What does it mean when we say a problem is intractable?**
  - The term "intractable" in the context of problem-solving, particularly in computer science and mathematics, refers to problems that are excessively difficult to solve within practical time frames. This usually means that the time required to solve the problem increases exponentially with the size of the input (exponential time complexity), making them impractical to solve as the input size grows.
  - Intractability often arises in optimization, decision, and search problems where the number of possibilities to explore grows exponentially with the input size (e.g., the Traveling Salesman Problem for large numbers of cities).
  - Intractable problems are significant because they impose fundamental limits on what can be efficiently computed. This has led to the development of approximation algorithms and heuristics that provide good, but not optimal, solutions in a reasonable amount of time for such problems.
- **[H] What are the time and space complexity for doing backpropagation on a recurrent neural network?**
  - **Time Complexity:** The time complexity of backpropagation in RNNs depends on several factors, including the sequence length ( $T$ ), the number of hidden units ( $H$ ), and the size of the input ( $D$ ). For a simple RNN, the time complexity is typically  $O(T * H^2)$  for each input sequence. This is because, for each time step, the network needs to compute the gradients with respect to the weights, which involves operations proportional to the square of the number of hidden units. When the input size  $D$  is significant, the complexity includes an additional term, making it  $O(T * (H^2 + HD))$ .
  - **Space Complexity:** The space complexity for training RNNs is also  $O(T * H^2)$ , mainly due to the need to store intermediate gradients for each time step during backpropagation through time (BPTT). This

can be quite substantial, especially for long sequences and large networks. Additionally, if the model includes layers with a large number of parameters or if there are many layers, the space required to store these parameters and their gradients must also be considered.

It's important to note that the actual complexity can vary based on the specific type of RNN architecture (like LSTM or GRU) and optimizations used in the implementation.

- **[H] Is knowing a model's architecture and its hyperparameters enough to calculate the memory requirements for that model?**
  - Knowing a model's architecture and its hyperparameters is generally a good starting point for estimating memory requirements, but it might not be sufficient for a precise calculation due to several factors:
    - **Batch Size:** The memory required for training depends significantly on the batch size. Larger batches require more memory to store the activations and gradients for each sample in the batch.
    - **Precision of Computations:** The precision (e.g., float32, float64) of the computations affects memory usage. Lower precision (like float16) can reduce memory requirements.
    - **Implementation Details:** Specific details of the implementation, such as how gradients are stored and whether certain optimizations (like gradient checkpointing) are used, can impact memory usage.
    - **Auxiliary Structures:** Additional memory might be required for things like optimization algorithms (e.g., Adam optimizer stores moving averages of both gradients and squared gradients), regularization techniques, and data augmentation.
    - **Model States:** In the case of RNNs, the memory required to maintain states over time steps also needs to be considered.

Therefore, while knowing the model's architecture and hyperparameters provides an essential framework for estimating memory requirements, a complete understanding often requires considering the training context, implementation specifics, and additional computational resources utilized during the model's training or inference phases.

- **[H] Your model works fine on a single GPU but gives poor results when you train it on 8 GPUs. What might be the cause of this? What would you do to address it?**

Several factors could lead to a model performing well on a single GPU but poorly when trained on 8 GPUs:

1. **Batch Size Effect:** When using multiple GPUs, the effective batch size increases (assuming you keep the per-GPU batch size constant). A larger batch size can change the training dynamics, potentially affecting the convergence and generalization of the model. Larger batches can lead to a decrease in the effective learning rate and an increase in generalization error.
2. **Synchronization Issues:** In multi-GPU training, especially with data-parallelism, ensuring proper synchronization and communication between GPUs is crucial. Inefficient synchronization can lead to delays or inconsistencies in gradient updates.
3. **Learning Rate Scaling:** With larger batch sizes, the learning rate often needs to be scaled appropriately. The 'linear scaling rule' suggests increasing the learning rate in proportion to the batch size, but this might require tuning.
4. **Stochasticity Reduction:** More GPUs with larger batch sizes reduce the stochasticity of gradient descent, which can sometimes lead to poorer exploration of the loss landscape.

#### **Addressing the Issue:**

- **Adjust Learning Rate:** Experiment with the learning rate. You might need to increase it as per the linear scaling rule or find an optimal value through experimentation.

- **Gradual Learning Rate Warm-up:** Gradually increase the learning rate at the beginning of training to mitigate the shocks of a suddenly large batch size.
- **Batch Normalization Adjustment:** If you're using batch normalization, consider using synchronized batch normalization across GPUs, as the statistics of batch normalization can be affected by the larger effective batch size.
- **Experiment with Batch Size:** Try reducing the batch size per GPU or using gradient accumulation to simulate a smaller effective batch size.
- **[H] What benefits do we get from reducing the precision of our model? What problems might we run into? How to solve these problems?**

#### Benefits:

1. **Reduced Memory Usage:** Lower precision formats (like float16 or bfloat16) use less memory, allowing for larger models or batches to fit in the same memory footprint.
2. **Faster Computation:** Lower precision arithmetic is generally faster on specialized hardware like GPUs and TPUs, leading to faster training and inference.
3. **Energy Efficiency:** Lower precision computations can be more energy-efficient, which is essential for mobile and edge computing.

#### Problems and Solutions:

- **Reduced Numerical Stability:** Lower precision can lead to issues like overflow or underflow in gradients and activations. This can be mitigated by:
  - **Mixed Precision Training:** Use a mix of lower and higher precision (e.g., float16 for forward pass and float32 for weight updates) to balance speed and stability.

- **Gradient Scaling:** Scale gradients in lower precision to prevent underflow, and then scale them back before the update step.
- **Loss of Accuracy:** The reduced precision might lead to a loss in the model's accuracy. Fine-tuning or retraining the model in lower precision can sometimes recover this loss.
- **Hardware Compatibility:** Not all hardware accelerates lower precision calculations, and some algorithms may not benefit as much from reduced precision.

In summary, while reducing precision can offer significant benefits in terms of memory, speed, and energy efficiency, it introduces challenges in numerical stability and model accuracy. These challenges can often be addressed through techniques like mixed precision training and gradient scaling.

- **[H] How to calculate the average of 1M floating-point numbers with minimal loss of precision?**

Calculating the average of a large number of floating-point numbers can lead to precision issues due to the inherent limitations of floating-point arithmetic in computers. Here are strategies to minimize precision loss:

1. **Kahan Summation Algorithm:** This is a method to compute the sum of a sequence of floating-point numbers with reduced numerical errors. The algorithm keeps a running compensation (a small value to correct the loss of precision in the addition). This method is more accurate than a straightforward summation, especially for large sequences.

Here's a simple implementation in Python:

```
def kahan_sum(nums):
    total = 0.0
    c = 0.0 # A running compensation for lost low-order bits.
    for num in nums:
```

```

        y = num - c      # So far, so good: c is zero.
        t = total + y   # Alas, sum is big, y small, so low-order digits of
        c = (t - total) - y # (t - total) recovers the high-order part of
        total = t       # Algebraically, c should always be zero. Beware overflow!
        return total

# Example usage
nums = [your list of 1M floating-point numbers]
sum_nums = kahan_sum(nums)
average = sum_nums / len(nums)

```

2. **Incremental Averaging:** Another approach is to use an incremental method for computing the average, which can be more numerically stable for a large number of values:

```

def incremental_average(nums):
    average = 0.0
    for i, num in enumerate(nums, 1):
        average += (num - average) / i
    return average

# Example usage
nums = [your list of 1M floating-point numbers]
average = incremental_average(nums)

```

These methods help in reducing the error accumulated due to the finite precision of floating-point representations, especially when dealing with a large number of values.

- **[H] How should we implement batch normalization if a batch is spread out over multiple GPUs?**

When a batch is distributed across multiple GPUs, implementing batch normalization correctly is crucial for maintaining consistency and performance. Here's how you can implement it:

1. **Synchronized Batch Normalization:** In this approach, batch normalization statistics (mean and variance) are computed across all the GPUs. This requires communication between GPUs to synchronize and aggregate the statistics from each batch across all GPUs.
2. **Use Deep Learning Frameworks Support:** Many modern deep learning frameworks, like TensorFlow and PyTorch, provide support for synchronized batch normalization. Utilizing this support can simplify the implementation:
  - In PyTorch, you can use `torch.nn.SyncBatchNorm` which synchronizes the mean and standard deviation of BatchNorm across multiple GPUs during training.
  - In TensorFlow, distributed strategies like `tf.distribute.MirroredStrategy` can be used to handle batch normalization in a multi-GPU setup.
3. **Careful Consideration of Batch Size:** Ensure that the batch size per GPU is not too small, as very small batches can lead to unstable estimates of the normalization statistics. The total effective batch size should be large enough for stable statistics.
4. **Alternatives to Batch Normalization:** If synchronized batch normalization is not feasible or too complex, consider alternative normalization methods that are less sensitive to batch sizes, such as Layer Normalization or Group Normalization.

Implementing batch normalization across multiple GPUs requires careful handling of the normalization statistics to ensure consistency across the distributed batches. Utilizing framework-specific features and considering alternative normalization methods can be effective strategies.

- **[M] Given the following code snippet. What might be a problem with it? How would you improve it?**  
**Hint: this is an actual question asked on StackOverflow.**

```
import numpy as np

def within_radius(a, b, radius):
    if np.linalg.norm(a - b) < radius:
        return 1
    return 0

def make_mask(volume, roi, radius):
    mask = np.zeros(volume.shape)
    for x in range(volume.shape[0]):
        for y in range(volume.shape[1]):
            for z in range(volume.shape[2]):
                mask[x, y, z] = within_radius((x, y, z), roi, radius)
    return mask
```

The provided code snippet is intended to create a binary mask for a volume (3D array) where points within a specified radius from a region of interest (ROI) are marked. While the logic of the code is sound, there are several potential issues and areas for improvement:

## **1. Syntax Errors:**

- The code contains syntax errors in the lines where `volume shape` is used. It should be `volume_shape` or another valid variable name.

## **2. Performance Inefficiency:**

- The code uses nested loops to iterate over every point in the volume, which can be highly inefficient, especially for large volumes. This approach leads to a significant computational burden as the complexity is  $O(n^3)$  for a volume of size  $n \times n \times n$ .

## **3. Vectorization for Performance Enhancement:**

- The use of NumPy suggests that vectorization could be used to improve performance. NumPy is highly optimized for vectorized operations, which are generally much faster than iterating through arrays with loops.

## **4. Memory Usage:**

- For very large volumes, creating and storing a mask of the same size might be memory-intensive. Depending on the application, it might be necessary to consider a more memory-efficient representation (like a sparse matrix) if the number of points within the radius is relatively small.

### **Improved Version:**

Here's an improved version of the `make_mask` function using vectorized operations:

```
pyt
import numpy as np

def make_mask(volume_shape, roi, radius):
```

```

# Create a grid of coordinates
x, y, z = np.indices(volume_shape)

# Calculate the squared distance from each point to the ROI
squared_dist = (x - roi[0])**2 + (y - roi[1])**2 + (z - roi[2])**2

# Create a mask where the squared distance is less than the squared radius
mask = squared_dist < radius**2
return mask.astype(int)

# Example usage
volume_shape = (100, 100, 100)
roi = (50, 50, 50)
radius = 20
mask = make_mask(volume_shape, roi, radius)

```

In this improved version, the code utilizes NumPy's array operations to compute the mask. It calculates the squared distances of all points in the volume from the ROI in a vectorized manner, which is much faster than iterating over each point. The mask is then created by checking which points have a squared distance less than the squared radius. This approach is more efficient and leverages the strengths of NumPy for handling large arrays.

## ▼ Chapter 7: Machine Learning Workflows Answer

### ▼ Basics

- [E] Explain supervised, unsupervised, weakly supervised, semi-supervised, and active learning.

Sure, these are all different types of machine learning paradigms. Let me explain each one of them:

1. Supervised Learning: This is a type of machine learning where we provide the model with labeled training data. Each example in the training data consists of an input vector and a desired output value (label). The model learns a function that maps inputs to outputs. This is used in applications like regression (predicting a continuous value) and classification (predicting a discrete value). For instance, predicting house prices based on features like location, size, etc. is a supervised learning problem.
2. Unsupervised Learning: In this type of learning, the model is given only input data and no explicit output values (labels). The goal is to discover the underlying structure in the data, like grouping or clustering of data points. It's used in exploratory analysis or when we don't have a target variable. An example could be customer segmentation in a market research study.
3. Weakly Supervised Learning: This is a type of supervised learning where the training data is noisily labeled. This means that the labels are not always accurate or reliable. This often happens when labels are obtained through heuristic methods or when the labeling process is too costly and we resort to cheaper, less accurate labeling methods.
4. Semi-Supervised Learning: This is a middle ground between supervised and unsupervised learning. Here, the model is trained on a combination of labeled and unlabeled data. This is usually done when acquiring labels for data is expensive or time-consuming. The idea is that the model can learn some underlying structure from the unlabeled data which can improve performance on the labeled data.
5. Active Learning: In active learning, the model is allowed to interact with the environment and choose the data from which it learns. The model identifies what it finds confusing or uncertain, and asks the user (or some kind of oracle) to label these instances. This is particularly useful when labeling data is costly or when we have a large amount of unlabeled data and we want the model to tell us what would be most useful to label.

Each of these learning paradigms has its own use cases and is chosen based on the problem at hand, the data available, and the resources at disposal.

- **Empirical risk minimization.**

1. **[E] What's the risk in empirical risk minimization?**

Empirical Risk Minimization (ERM) is a principle in machine learning where the goal is to find a hypothesis that minimizes the loss on the observed data, or the empirical risk.

The "risk" in Empirical Risk Minimization refers to the loss or error that the model incurs when predicting the output for a given input. This loss is calculated using a loss function, which measures the difference between the model's prediction and the actual output. The goal of ERM is to find the model that minimizes this loss on the training data.

However, there's a risk associated with this approach. The main risk is overfitting, which happens when the model learns the training data too well, to the point that it performs poorly on unseen data. This is because the model has essentially memorized the training data, rather than learning the underlying patterns. This makes it perform poorly on new, unseen data.

To mitigate this risk, we often use techniques like regularization, cross-validation, or working with a simpler model to prevent the model from overfitting to the training data.

2. **[E] Why is it empirical?**

Empirical Risk Minimization (ERM) is called "empirical" because it focuses on minimizing the "empirical" or observed risk, which is calculated based on the data sample at hand, rather than the theoretical or true risk.

In machine learning, the goal is often to find a hypothesis (or model) that minimizes the risk, or expected loss. However, the true risk, which is the expected loss over the entire population, is usually

unknown because we don't have access to the entire population of data.

Instead, we have a finite sample of data, and we calculate the empirical risk, which is the average loss over this sample. The ERM principle then suggests choosing the hypothesis that minimizes this empirical risk. This is a practical approach to learning from the data we have, hence the term "empirical".

### 3. [E] How do we minimize that risk?

Empirical Risk Minimization (ERM) is a principle in machine learning where we aim to minimize the loss function over the training set. The "risk" here refers to the expected loss, and "empirical" means that we're estimating this risk directly from the training data.

To minimize this risk, we typically use optimization algorithms such as gradient descent, stochastic gradient descent, or more advanced methods like Adam or RMSprop. These algorithms iteratively adjust the model's parameters to find the minimum of the loss function.

For example, in gradient descent, we start with initial values for the model's parameters and calculate the gradient of the loss function with respect to these parameters. The gradient points in the direction of steepest increase of the function, so we update the parameters by taking a step in the opposite direction (i.e., the direction of steepest decrease). We repeat this process until the parameters converge to a minimum.

Keep in mind that ERM can lead to overfitting if the model is too complex, because it tries to minimize the loss on the training set as much as possible. To prevent overfitting, we often use regularization techniques, which add a penalty term to the loss function that increases with the complexity of the model.

- **[E] Occam's razor states that when the simple explanation and complex explanation both work equally well, the simple explanation is usually correct. How do we apply this principle in ML?**

In machine learning, Occam's Razor principle is often applied in the form of model complexity and overfitting. The idea is that a simpler model is often better than a more complex one if they both perform equally well on unseen data.

For example, let's say we have two models: a linear regression model and a neural network. Both models have been trained on the same dataset and achieve similar accuracy on the validation set. According to Occam's Razor, we should prefer the simpler model, in this case, the linear regression model, because it's less likely to overfit the training data and is more interpretable.

This principle is also the basis for techniques like regularization, which add a penalty term to the loss function to discourage complexity and encourage the model to learn simpler patterns. Regularization methods like L1 and L2 help in preventing overfitting by adding a complexity term to the loss function, which forces the weights to be small if the model complexity increases.

So, in essence, Occam's Razor in machine learning encourages us to prefer simpler models over more complex ones, to avoid overfitting and improve generalization to unseen data.

- **[E] What are the conditions that allowed deep learning to gain popularity in the last decade?**

Deep learning has gained significant popularity over the last decade due to a confluence of several factors:

1. Availability of Data: The digital age has led to an explosion in the amount of data we generate. Social media, smartphones, and the Internet of Things (IoT) devices are continuously producing a vast amount of data. Deep learning algorithms thrive on large datasets, and the availability of such data has been a key driver in the rise of deep learning.
2. Computational Power: Deep learning models are computationally intensive. The advent of GPUs (Graphics Processing Units) has made it feasible to train complex deep learning models. GPUs are particularly well-suited to the parallel computations required for deep learning.

3. Software Libraries and Frameworks: The development of open-source deep learning libraries and frameworks such as TensorFlow, PyTorch, and Keras has made it easier for researchers and practitioners to design, train, and deploy deep learning models. These libraries provide high-level APIs and take care of many of the underlying details, making deep learning more accessible.

4. Breakthroughs in Research: There have been several key breakthroughs in deep learning research over the past decade. For example, the development of techniques such as convolutional neural networks (CNNs) for image processing and recurrent neural networks (RNNs) for sequence data have opened up new possibilities for the application of deep learning.

5. Success in Benchmarks and Competitions: Deep learning models have achieved state-of-the-art results in a number of benchmarks and competitions, such as the ImageNet Large Scale Visual Recognition Challenge. These successes have helped to demonstrate the power of deep learning and have contributed to its popularity.

- [M] If we have a wide NN and a deep NN with the same number of parameters, which one is more expressive and why?

The expressiveness of a neural network is largely determined by its architecture, including its depth and width. However, it's not as simple as saying one is always more expressive than the other. The expressiveness of a neural network depends on the specific problem it's trying to solve.

In general, deeper networks are considered more expressive than wider ones given the same number of parameters. This is because depth allows for hierarchical feature learning, where each layer of the network can learn increasingly complex features from the output of the previous layer. For example, in a deep convolutional neural network used for image recognition, the first layer might learn to recognize edges, the second layer might learn to recognize textures or shapes formed by these edges, the third layer might learn to recognize more complex structures like parts of objects, and so on. This hierarchical feature learning can be very powerful for many tasks.

On the other hand, wider networks can also be expressive, but they tend to be more prone to overfitting, especially when the number of parameters is large. Wider networks have more units per layer, which allows them to learn more diverse features, but this can also lead to learning irrelevant or noisy features if not properly regularized.

So, while deeper networks are generally more expressive given the same number of parameters, the optimal architecture (i.e., the balance between depth and width) depends on the specific problem and the amount and quality of available training data.

- [H] **The Universal Approximation Theorem states that a neural network with 1 hidden layer can approximate any continuous function for inputs within a specific range. Then why can't a simple neural network reach an arbitrarily small positive error?**

The Universal Approximation Theorem indeed states that a neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function to any desired precision, given the function is defined on a compact subset of real numbers. However, there are a few key points to consider when interpreting this theorem:

1. **\*\*Approximation vs. Learning\*\***: The theorem is about the approximation capabilities of a neural network, not its learning capabilities. It doesn't say anything about how easy it is to learn the parameters of such an approximation through training. The learning process, which often involves methods like gradient descent, might not converge to the optimal solution due to reasons like local minima, saddle points, or improper learning rates.
2. **\*\*Finite vs. Infinite Neurons\*\***: The theorem states that a neural network can approximate any continuous function with a finite number of neurons, but it doesn't specify how many neurons are required. In practice, the number of neurons required to achieve a certain level of accuracy might be astronomically large, making it infeasible to implement.

3. **Overfitting**: Even if we could theoretically reach an arbitrarily small error on the training set, this does not mean that the model will generalize well to unseen data. Overfitting is a common problem in machine learning where the model learns the noise in the training data, leading to poor performance on the test data.

4. **Noise in Data**: Real-world data often contains noise. Even if we could fit a function perfectly to the noisy data, the error with respect to the underlying true function might still be large.

In conclusion, while the Universal Approximation Theorem provides a theoretical guarantee about the capabilities of neural networks, there are practical issues that prevent us from reaching an arbitrarily small positive error in real-world applications.

- [E] What are saddle points and local minima? Which are thought to cause more problems for training large NNs?

Saddle points and local minima are two types of points that can be encountered during the optimization process of training a neural network.

A local minimum is a point where the value of the function is lower than at nearby points, but possibly higher than at some other point further away. In the context of neural networks, this means that the weights and biases of the network at this point produce a lower error than other nearby configurations, but there may be a better configuration elsewhere.

A saddle point, on the other hand, is a point where the gradient of the function is zero but it's not a local maximum or a local minimum. In higher dimensions, a saddle point is a point where the function has a local minimum in one direction and a local maximum in another direction.

In the context of training large neural networks, both local minima and saddle points can pose problems, but saddle points are generally thought to cause more issues. This is because in high-dimensional spaces, which are typical in large neural networks, saddle points are much more common than local minima.

At a saddle point, the gradient is zero, so gradient-based optimization methods like stochastic gradient descent (SGD) can get stuck. However, unlike at a local minimum, it's not clear from the gradient which direction to move to improve the function value. This can lead to slow progress and poor final performance of the network.

On the other hand, while local minima can also cause training to get stuck, recent research suggests that in high-dimensional spaces, most local minima are actually pretty good, i.e., they correspond to a low value of the loss function. Therefore, getting stuck in a local minimum is less of a problem than getting stuck at a saddle point.

- **Hyperparameters.**

1. **[E] What are the differences between parameters and hyperparameters?**

In machine learning, both parameters and hyperparameters play crucial roles, but they serve different purposes.

Parameters are the internal variables that the model learns during the training process. For example, in a linear regression model, the coefficients of the predictors are parameters. These are learned from the data and are not set by the practitioner. They are internal to the model and are estimated or learned from the data.

On the other hand, hyperparameters are external to the model and cannot be learned from the data. They are configuration variables, which we use to control the learning process. For example, the learning rate in gradient descent, the complexity parameter in a support vector machine (C in SVM), or the depth of a decision tree, are all hyperparameters. These are set by the practitioner before the learning process begins.

In summary, parameters are learned from the data, and they change during the learning process. Hyperparameters, however, are set beforehand and remain constant during the learning process. They

are used to control the learning process and can be tuned for optimal model performance.

## 2. [E] Why is hyperparameter tuning important?

Hyperparameter tuning is a crucial step in the process of building a machine learning model.

Hyperparameters are the configuration variables that govern the training process itself, and they can significantly influence the performance of a model. They are not learned from the data but are set prior to the start of training.

The importance of hyperparameter tuning lies in the fact that the right choice of hyperparameters can make the difference between an average model and a highly accurate one. For instance, in a neural network, hyperparameters like the learning rate, the number of hidden layers, or the number of nodes in each layer can greatly affect the model's ability to learn effectively from the data.

In a decision tree algorithm, the depth of the tree or the minimum number of samples required to split an internal node are examples of hyperparameters. If these are not set correctly, the model might underfit or overfit the data, leading to poor predictions.

So, hyperparameter tuning is essentially about finding the optimal combination of hyperparameters that yields a model with the best performance. This is usually done through techniques like grid search, random search, or more advanced methods like Bayesian optimization.

In summary, hyperparameter tuning is important because it directly impacts the model's performance, and hence, the quality of predictions it makes.

## 3. [M] Explain algorithm for tuning hyperparameters.

Hyperparameters are parameters that are not learned from the data. They are set prior to the commencement of the learning process. Tuning hyperparameters is an important step in optimizing a machine learning algorithm. The goal is to find the optimal hyperparameters that provide the best performance for your model on unseen data.

There are several strategies for tuning hyperparameters:

1. **Grid Search**: This is the most traditional method. You specify a set of values for each hyperparameter that you want to tune. The algorithm will then try all possible combinations of these values and return the combination that gives the best performance. The downside of grid search is that it can be very time-consuming, especially if you have many hyperparameters to tune or if each model takes a long time to train.
2. **Random Search**: Instead of trying all combinations, random search selects random combinations of the hyperparameters to train the model and keeps track of the performance. The advantage is that it's less time-consuming than grid search, but the downside is that it might not find the optimal combination.
3. **Bayesian Optimization**: This is a more advanced method. It builds a probability model of the objective function and uses it to select the most promising hyperparameters to evaluate in the true objective function. It balances exploration against exploitation. It's more efficient than random or grid search but requires more complex software.
4. **Gradient-based optimization**: Some hyperparameters are amenable to gradient-based optimization, just like the parameters of a neural network during the training phase. This is a more recent development and can be more efficient than other methods, but it's not applicable to all hyperparameters and can be more complex to implement.
5. **Evolutionary algorithms**: These are optimization algorithms based on the principles of evolution, such as genetic algorithms. They can be very effective for hyperparameter tuning, but they can also be more complex to implement and understand.

Here's an example of how you might use grid search with scikit-learn:

```

from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Create a base model
rf = RandomForestClassifier()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           cv=3, n_jobs=-1, verbose=2)
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
# Get the best parameters
best_params = grid_search.best_params_

```

In this example, we're tuning a random forest classifier. We define a grid of hyperparameters to search over, including the number of trees in the forest (`'n\_estimators'`), the maximum depth of the trees (`'max\_depth'`), and the minimum number of samples required to split an internal node (`'min\_samples\_split'`). We then fit the grid search object to our training data and retrieve the best hyperparameters.

- **Classification vs. regression.**

1. **[E] What makes a classification problem different from a regression problem?**

Classification and regression are two types of supervised learning problems in machine learning, but they are used for different types of prediction tasks.

1. Classification: In a classification problem, the output variable is a category, such as "spam" or "not spam" for email filtering. The goal is to predict the class label of an instance. Classification is a discrete problem, meaning the output is from a limited set of categories. For example, if we are building a machine learning model to predict whether an email is spam or not, we are dealing with a binary classification problem.

2. Regression: In a regression problem, the output variable is a real or continuous value, such as "salary" or "weight". The goal is to predict a quantity. For instance, if we are building a machine learning model to predict the house price based on various features like the number of bedrooms, square footage, location, etc., we are dealing with a regression problem.

So, the key difference between classification and regression lies in the type of output they predict - classification predicts a category, while regression predicts a quantity.

2. **[E] Can a classification problem be turned into a regression problem and vice versa?**

Yes, in some cases, a classification problem can be turned into a regression problem and vice versa, but it's important to note that it depends on the specific problem and the data at hand.

For instance, let's consider a binary classification problem where we want to predict whether an email is spam or not. If we assign the classes, spam and not-spam, values of 1 and 0 respectively, we can turn this into a regression problem. The output of the regression model can be a continuous value between 0 and 1, which can be interpreted as the probability of the email being spam. We can then set a threshold (like 0.5) to decide the class. This is essentially what logistic regression does.

On the other hand, a regression problem can be turned into a classification one by binning the output values. For example, if we're predicting house prices (a regression problem), we could create different bins (or classes) like "low-priced", "medium-priced", and "high-priced", and turn it into a classification problem. However, this might lead to loss of information because we're converting a continuous variable into a categorical one.

In general, whether or not to convert a problem from classification to regression (or vice versa) depends on the specific problem, the data, and the business requirements. It's crucial to remember that each type of problem has its own strengths and weaknesses, and the choice should be based on what makes the most sense for the task at hand.

- **Parametric vs. non-parametric methods.**

1. **[E] What's the difference between parametric methods and non-parametric methods? Give an example of each method.**

Parametric methods and non-parametric methods are two broad classifications of statistical learning techniques, each with its own strengths and weaknesses.

Parametric methods make certain assumptions about the underlying data's distribution. They involve a two-step model-based approach: first, we make an assumption about the functional form, or shape, of the population from which our sample is drawn. This could be a linear relationship, a polynomial relationship, etc. Then, we use the training data to fit or train the model, often by estimating the parameters of the model. An example of a parametric method is Linear Regression, where we assume a linear relationship between the independent and dependent variables.

Non-parametric methods, on the other hand, do not make strong assumptions about the data's functional form. They seek to avoid the potential pitfalls of parametric methods that can arise from incorrect assumptions about the data's distribution. Instead, they try to learn the data's structure

directly from the data itself. An example of a non-parametric method is the Decision Tree, which makes no assumptions about the functional form of the data and instead splits the data based on certain criteria.

In summary, the choice between parametric and non-parametric methods depends on the specific problem, the nature of the data, and the trade-off between interpretability and model accuracy.

## 2. [H] When should we use one and when should we use the other?

Parametric and non-parametric methods are two broad classifications of statistical techniques.

Parametric methods make certain assumptions about the underlying data. They assume that the data follows a certain distribution, typically a Gaussian or normal distribution. Examples of parametric methods include linear regression, logistic regression, and analysis of variance (ANOVA).

Non-parametric methods, on the other hand, do not make strong assumptions about the underlying distribution of the data. They are more flexible and can be used when the assumptions of parametric methods are not met. Examples of non-parametric methods include decision trees, k-nearest neighbors, and support vector machines.

The choice between parametric and non-parametric methods depends on the characteristics of your data and the specific problem you're trying to solve.

1. **\*\*Data Characteristics:\*\*** If your data is normally distributed or if you have a large sample size, parametric methods are often a good choice. They are simpler and more interpretable, and they can provide more statistical power if their assumptions are met. On the other hand, if your data is not normally distributed or if you have a small sample size, non-parametric methods may be more appropriate. They are more robust to outliers and can model complex, non-linear relationships.
2. **\*\*Problem Specifics:\*\*** If you're interested in understanding the influence of specific predictors, parametric methods like regression can provide clear and interpretable results. If you're more

interested in prediction accuracy, non-parametric methods can often provide better performance, as they can model complex, non-linear relationships.

In practice, it's often a good idea to try both parametric and non-parametric methods and see which one works best for your specific problem. You can use cross-validation or other model selection techniques to compare the performance of different methods and choose the one that gives the best results.

- [M] **Why does ensembling independently trained models generally improve performance?**

Ensembling is a technique in machine learning where multiple models are trained independently and their predictions are combined to make a final prediction. The main reason why ensembling improves performance is due to the concept of "wisdom of the crowd".

In simple terms, the idea is that a group of models, each with their own strengths and weaknesses, can collectively make better decisions than any single model alone. This is because each model may capture different aspects of the data, and by combining their predictions, we can leverage the strengths of each model and mitigate their individual weaknesses.

There are a few key reasons why ensembling works:

1. **\*\*Diversity\*\***: Each model in the ensemble might learn and capture different patterns in the data. When the models' predictions are combined, the ensemble can capture a wider range of patterns and complexities in the data.
2. **\*\*Reduced Overfitting\*\***: Overfitting occurs when a model learns the training data too well, including its noise and outliers, and performs poorly on unseen data. By combining multiple models, the ensemble can generalize better to unseen data, reducing the risk of overfitting.
3. **\*\*Reduced Variance\*\***: The ensemble's final prediction is usually the average of the predictions of individual models. This averaging process can reduce the variance of the predictions, leading to more

stable and reliable predictions.

4. **Handling High Dimensionality**: In high-dimensional spaces, individual models may struggle to find patterns due to the curse of dimensionality. Ensembles can help overcome this by allowing each model to focus on different subsets of the feature space.

For example, in a random forest, which is an ensemble of decision trees, each tree is trained on a different subset of the data. Some trees might make errors on certain instances, but other trees might correct these errors. The final prediction of the random forest is typically the majority vote (for classification) or the average (for regression) of the predictions of the individual trees. This approach usually results in a more accurate and robust model than any single decision tree.

- **[M] Why does L1 regularization tend to lead to sparsity while L2 regularization pushes weights closer to 0?**

L1 and L2 regularization are both techniques used to prevent overfitting in machine learning models by adding a penalty term to the loss function. However, they have different effects on the model parameters due to the nature of their penalty terms.

L1 regularization, also known as Lasso regularization, adds an absolute value of the magnitude of coefficient as penalty term to the loss function. The L1 penalty has the property of pushing weights to become exactly zero. This is because the absolute value function is not differentiable at zero, and so the optimization algorithm can "jump" to zero weights. This leads to sparser solutions, meaning that it can reduce some feature coefficients to zero and effectively exclude them from the model. This can be particularly useful when you have a dataset with many features and you want to automatically perform feature selection.

On the other hand, L2 regularization, also known as Ridge regularization, adds the squared magnitude of coefficient as penalty term to the loss function. The L2 penalty tends to spread the weight values out more

evenly and push them towards smaller values, but not exactly zero. This is because the square function is differentiable everywhere, and so the optimization algorithm will gradually decrease the weight values. This can be beneficial when all features are important and you want to include all of them in the model, but prevent any one feature from having too much influence.

In summary, the difference between L1 and L2 regularization in terms of their effect on the weights is due to the properties of their penalty terms. L1 regularization can lead to sparser solutions with some weights being exactly zero, while L2 regularization tends to push weights towards smaller values, but not exactly zero.

- **[E] Why does an ML model's performance degrade in production?**

There are several reasons why a machine learning model's performance might degrade in production:

1. **\*\*Concept Drift\*\***: This refers to the change in the underlying statistical properties of the data over time. This can happen due to various reasons such as changes in user behavior, seasonality, or other external factors. If the model is not updated to reflect these changes, its performance can degrade.
2. **\*\*Data Skew\*\***: This occurs when the distribution of values in the training data is different from the distribution in the production data. This can lead to a model that performs well on the training data but poorly on the production data.
3. **\*\*Overfitting\*\***: This happens when a model learns the noise in the training data instead of the underlying pattern. Such a model will perform well on the training data but poorly on unseen data.
4. **\*\*Quality of Input Data\*\***: If the data quality degrades (missing values, incorrect entries, etc.), the model's performance will also degrade.
5. **\*\*Software or Infrastructure Changes\*\***: Changes in the software or infrastructure that the model relies on can also lead to performance degradation. For example, if a feature used by the model is no longer available due to a software update, the model's performance will suffer.

To mitigate these issues, it's important to monitor the model's performance continuously and retrain it regularly with fresh data. It's also crucial to ensure the quality of the input data and to keep the software and infrastructure up to date.

- **[M] What problems might we run into when deploying large machine learning models?**

Deploying large machine learning models can present several challenges:

1. **\*\*Computational Resources\*\***: Large models often require significant computational resources, both in terms of processing power and memory. This can be expensive and may not be feasible for all organizations. Additionally, the time required to train and deploy these models can be substantial.
2. **\*\*Overfitting\*\***: Large models with many parameters are more prone to overfitting. They may perform extremely well on the training data but fail to generalize to unseen data. This can lead to poor performance in production.
3. **\*\*Data Requirements\*\***: Large models typically require large amounts of data to train effectively. Gathering and cleaning this data can be a significant task. Additionally, privacy concerns may limit the amount of data that can be used.
4. **\*\*Maintenance and Updates\*\***: Large models can be more difficult to maintain and update. Changes in the underlying data distribution may require retraining the model, which can be time-consuming and resource-intensive.
5. **\*\*Interpretability\*\***: Large, complex models can be "black boxes," making it difficult to understand why they're making certain predictions. This can be a problem in industries where interpretability is important, such as healthcare or finance.
6. **\*\*Latency\*\***: In real-time applications, large models can cause latency issues. The time taken to load the model into memory and make predictions can be significant, potentially leading to user experience issues.

To mitigate these issues, it's important to thoroughly test the model in an environment that closely mirrors production before deployment. Techniques like regularization, early stopping, or using simpler models can be used to prevent overfitting. For computational issues, techniques like model pruning, quantization, or using more efficient architectures can help. For interpretability, techniques like SHAP, LIME, or model-agnostic methods can be used.

- **Your model performs really well on the test set but poorly in production.**

1. **[M] What are your hypotheses about the causes?**

There could be several reasons why a model performs well on the test set but poorly in production.

Here are a few hypotheses:

1. **\*\*Data Drift\*\***: The data in production could be different from the data used for training and testing the model. This is known as data drift. It can occur due to changes in user behavior, seasonal effects, or other external factors. For example, a model trained on summer clothing sales data might not perform well in winter.
2. **\*\*Concept Drift\*\***: This is when the statistical properties of the target variable, which the model is trying to predict, change over time. For example, in a loan default prediction model, if the economy suddenly takes a downturn, the model might start performing poorly because the underlying assumptions have changed.
3. **\*\*Overfitting\*\***: The model might be overfitting the training data, meaning it has learned the noise in the training data rather than the underlying pattern. This would result in high performance on the test set but poor performance on new, unseen data in production.
4. **\*\*Quality of Test Data\*\***: The test set might not be a good representative of the real-world data. If the test set is too small or not diverse enough, the model might perform well on it but fail to generalize to new data in production.

5. **Feedback Loops**: Sometimes, the predictions made by the model can influence future input data. This is known as a feedback loop. For example, in a recommendation system, the items recommended by the model can influence the user's future behavior, which in turn influences the input data for the model.
6. **Model Decay**: Models tend to 'decay' or become less effective over time if they're not updated regularly. This is especially true for models dealing with data that changes frequently, like stock prices or social media trends.

To diagnose and fix these issues, it's important to monitor the model's performance in production and regularly retrain the model with fresh data. Also, using techniques like cross-validation during model development can help prevent overfitting.

## 2. [H] How do you validate whether your hypotheses are correct?

When a model performs well on the test set but poorly in production, it typically indicates a problem with overfitting, data leakage, or a shift in the underlying data distribution. Here are some steps I would take to validate these hypotheses:

1. **Overfitting**: If the model is overfitting, it has learned the noise in the training data, leading to high performance on the test set but poor generalization to unseen data in production. To check for overfitting, I would examine the learning curves for the training and validation sets. If the model performs well on the training set but poorly on the validation set, it's a clear sign of overfitting. To mitigate this, I could introduce regularization, use a simpler model, or gather more data.
2. **Data Leakage**: Data leakage occurs when information from the future or the target variable accidentally ends up in the training data. This can lead to overly optimistic test set performance. To check for data leakage, I would conduct a thorough audit of the data collection and preprocessing

pipeline to ensure that no information from the future or the target variable is being inadvertently included in the training data.

3. **Distribution Shift**: If the distribution of the data in production is different from the distribution of the data used for training and testing, the model may perform poorly in production. This is often due to temporal changes or non-stationarity in the data. To check for distribution shift, I would compare the distributions of the training and production data. If they are significantly different, I might need to retrain the model more frequently, use a model that can adapt to changing distributions, or collect more representative data.

In all these cases, I would use a combination of visualizations, statistical tests, and additional model training to validate my hypotheses. It's also important to maintain good logging and monitoring practices in production to quickly identify and diagnose these issues.

3. **[M] Imagine your hypotheses about the causes are correct. What would you do to address them?**

There could be several reasons why a model performs well on the test set but poorly in production. Here are a few hypotheses and the corresponding actions I would take to address them:

1. **Data Drift**: The production data may have drifted from the data used to train the model. This means the distribution of the data in production is different from the distribution of the training data. To address this, I would first set up a monitoring system to track the statistical properties of the input data in production. If data drift is detected, I would retrain the model with new data that reflects the current situation.

2. **Concept Drift**: The underlying concept that the model has learned may have changed in the new data. For example, a model predicting house prices may no longer be accurate if there's a sudden

economic downturn. To address this, I would monitor the model's performance over time and retrain it as necessary when performance degrades.

3. **Overfitting**: The model may have overfit to the training data, meaning it learned the noise in the training data and fails to generalize to unseen data. To address this, I would use techniques like regularization, dropout (for neural networks), or gathering more data. I would also make sure to use a validation set during training to tune hyperparameters and prevent overfitting.
4. **Feedback Loops**: Sometimes, the model's predictions can influence the future input data. For example, a recommendation system may suggest certain products to users, which in turn influences their future behavior. To address this, I would try to incorporate the feedback loop into the model, or use techniques like multi-armed bandit algorithms that can handle changing environments.
5. **Mismatched Evaluation Metrics**: The evaluation metric used during model development may not align with the business objective. For instance, accuracy might not be the best metric if the classes are imbalanced. In this case, I would work with the business stakeholders to define a more appropriate evaluation metric.
6. **Poor Quality of New Data**: The production data might be noisier or have more missing values than the training data. To address this, I would improve the data cleaning and preprocessing steps, and possibly retrain the model with more representative data.

In all these cases, it's crucial to have a robust monitoring system in place to detect when the model's performance starts to degrade, and a process to retrain and redeploy the model as needed.

#### ▼ Sampling and creating training data

- **[E] If you have 6 shirts and 4 pairs of pants, how many ways are there to choose 2 shirts and 1 pair of pants?**

This is a basic combinatorics problem. To solve it, we need to use the combination formula, which is:

$$C(n, k) = n! / [k!(n-k)!]$$

where  $n$  is the total number of items,  $k$  is the number of items to choose, and " $!$ " denotes factorial, which is the product of all positive integers up to that number.

So, for choosing 2 shirts out of 6, the combination would be:

$$C(6, 2) = 6! / [2!(6-2)!] = 15$$

And for choosing 1 pair of pants out of 4, the combination would be:

$$C(4, 1) = 4! / [1!(4-1)!] = 4$$

Since these are independent events, we multiply the combinations together to get the total number of ways to choose 2 shirts and 1 pair of pants:

$$15 * 4 = 60$$

So, there are 60 different ways to choose 2 shirts and 1 pair of pants from your given wardrobe.

- **[M] What is the difference between sampling with vs. without replacement? Name an example of when you would use one rather than the other?**

Sampling with replacement means that after we select an instance from our dataset for our sample, we put it back in the dataset. This allows it to be picked again for our sample. On the other hand, sampling without replacement means that once we select an instance from our dataset, it's removed and can't be selected again for our sample.

The choice between these two methods depends on the specific requirements of your analysis or model.

For example, if you're creating a bootstrap sample (a common technique in statistics and machine learning to estimate the sampling distribution), you would sample with replacement. This is because bootstrapping is based on the idea of creating 'pseudo-populations' from your original sample by drawing observations

with replacement, which allows for the possibility of the same observation appearing more than once in your resample.

On the other hand, if you're splitting your dataset into a training set and a test set, you would sample without replacement. This is to ensure that your test set is entirely distinct from your training set, which helps to provide a more robust evaluation of your model's performance. If you were to sample with replacement in this scenario, you could end up with some instances appearing in both your training and test sets, which could lead to overly optimistic performance estimates.

- **[M] Explain Markov chain Monte Carlo sampling.**

Markov Chain Monte Carlo (MCMC) is a method used in computational statistics to sample from complex probability distributions. It's particularly useful in Bayesian statistics where we often have to deal with high-dimensional posterior distributions.

The basic idea behind MCMC is to generate a Markov chain of samples in such a way that the stationary distribution of the chain is the target distribution we want to sample from. A Markov chain is a sequence of random variables where the distribution of each variable only depends on the value of the previous variable.

The Monte Carlo part of MCMC refers to the use of randomness in the algorithm. Each new sample in the chain is generated from the previous sample by a random process, which involves some form of random number generation.

One of the most common MCMC methods is the Metropolis-Hastings algorithm. It starts with an arbitrary initial point, then it proposes a new point based on a proposal distribution. The new point is accepted as the next point in the chain with a probability that depends on the ratio of the target distribution at the new point and the current point. If the new point is not accepted, the current point is repeated in the chain.

The beauty of MCMC methods is that they allow us to sample from complex distributions without knowing the normalization constant, which is often the case in Bayesian statistics where the normalization constant of the posterior distribution is the evidence or marginal likelihood, a high-dimensional integral that is intractable for most realistic models.

However, MCMC methods also have their challenges. They can be slow to converge, especially in high-dimensional spaces, and it can be difficult to determine when they have converged. They can also get stuck in local modes of multi-modal distributions. There are various diagnostic methods and advanced MCMC techniques to address these issues.

- **[M] If you need to sample from high-dimensional data, which sampling method would you choose?**

When dealing with high-dimensional data, traditional sampling methods often struggle due to the "curse of dimensionality," where the volume of the data grows exponentially with the increase in dimensions. This can make the sampled data sparse and not representative of the overall data distribution.

In such cases, I would consider using a method called Markov Chain Monte Carlo (MCMC) sampling. MCMC is a class of algorithms for sampling from a probability distribution based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. The state of the chain after a large number of steps is then used as a sample of the desired distribution. The quality of the sample improves as a function of the number of steps.

MCMC methods are particularly useful when the dimension of the data is high because they construct samples sequentially and use information from previous samples to inform the location of the next. This allows MCMC methods to effectively explore the high-dimensional space and generate samples that are representative of the data distribution.

However, it's worth noting that MCMC methods can be computationally expensive and might require careful tuning to work effectively. Therefore, they might not always be the best choice, depending on the specific

context and constraints of the problem at hand.

For instance, if computational resources or time are a constraint, a simpler method like Latin Hypercube Sampling (LHS) might be a better choice. LHS is a stratified sampling method that ensures a good coverage of the parameter space and can be more efficient than random sampling in high dimensions.

In conclusion, the choice of sampling method for high-dimensional data would depend on the specific requirements and constraints of the problem, including the available computational resources, the time constraints, and the complexity of the data distribution.

- [H] Suppose we have a classification task with many classes. An example is when you have to predict the next word in a sentence -- the next word can be one of many, many possible words. If we have to calculate the probabilities for all classes, it'll be prohibitively expensive. Instead, we can calculate the probabilities for a small set of candidate classes. This method is called candidate sampling. Name and explain some of the candidate sampling algorithms.

Hint: check out this great article on candidate sampling by the TensorFlow team.

Candidate sampling is a useful technique for dealing with multi-class classification problems where the number of classes is extremely large. It helps to reduce the computational cost by only considering a small subset of negative classes (the classes to which the target class does not belong) along with the positive class (the target class) during training. Here are some of the candidate sampling algorithms:

1. **\*\*Negative Sampling\*\***: This is a simplified version of Noise Contrastive Estimation (NCE). In negative sampling, we randomly select a small number of negative classes to update the weights in our model. For example, if we are trying to predict the next word in a sentence and the target word is 'dog', we will update the weights for 'dog' and a small number of randomly chosen words that are not 'dog'.
2. **\*\*Noise Contrastive Estimation (NCE)\*\***: NCE is a more sophisticated method that introduces noise distributions to estimate the parameters of the model. It converts the multi-class classification problem into

a binary classification problem by distinguishing the target class from artificially generated noise.

3. **Hierarchical Softmax**: Hierarchical Softmax uses a binary tree representation of the output layer where each node represents a class and each leaf node represents a path from the root to the node. It significantly reduces the computational cost from linear to logarithmic time in terms of the number of classes.

4. **Importance Sampling**: This method assigns an importance weight to each class and uses these weights to sample a subset of classes. The classes with higher weights are more likely to be sampled.

5. **Self-Normalizing Importance Sampling (SNIS)**: SNIS is a variant of importance sampling that normalizes the importance weights to make them sum to one. This ensures that the sampled classes are a good representation of the entire class distribution.

All these methods have their pros and cons and the choice of method depends on the specific requirements of the task. For example, Negative Sampling and NCE are easy to implement and computationally efficient, but they may not be as accurate as Hierarchical Softmax or Importance Sampling. On the other hand, Hierarchical Softmax and Importance Sampling may provide more accurate results, but they are more complex and computationally expensive.

- Suppose you want to build a model to classify whether a Reddit comment violates the website's rule. You have 10 million unlabeled comments from 10K users over the last 24 months and you want to label 100K of them.

1. [M] How would you sample 100K comments to label?

Sampling the right data is crucial to building a good machine learning model. Given that we have 10 million unlabeled comments from 10K users over the last 24 months, there are several strategies we could employ to sample 100K comments to label.

1. **Random Sampling**: This is the simplest method where we randomly select 100K comments from the entire dataset. This method assumes that all comments are equally likely to violate Reddit's rules, which may not be the case.
  2. **Stratified Sampling**: In this method, we divide the entire dataset into different groups or strata, and then sample from each stratum. For instance, we could stratify by users, assuming that some users are more likely to violate rules than others. We could also stratify by time, assuming that rule violations might be more common at certain times. This method ensures that our sample is representative of the entire dataset.
  3. **Time-based Sampling**: Given that the data spans over 24 months, it might be useful to sample comments from different time periods to capture any temporal trends or changes in behavior. For instance, we could sample equally from each month to ensure that our sample is representative of the entire time period.
  4. **User-based Sampling**: Since we have data from 10K users, another strategy could be to sample comments from a diverse range of users. This could help ensure that our model generalizes well across different users.
  5. **Active Learning**: This is a more sophisticated approach where we initially train a model on a small set of labeled data, and then use the model to identify the most informative examples to label next. This method can be more efficient than random sampling, especially when labeling data is costly.
- In practice, I would recommend a combination of these methods. For instance, we could use stratified sampling to ensure that our sample is representative of different users and time periods, and then use active learning to iteratively improve our model. This would help ensure that our model is both accurate and generalizable.

2. **[M] Suppose you get back 100K labeled comments from 20 annotators and you want to look at some labels to estimate the quality of the labels. How many labels would you look at? How would you sample them?**

**Hint: This article on different sampling methods and their use cases might help.**

To estimate the quality of the labels, we would need to review a subset of the labeled comments. The number of labels to look at would depend on the confidence level and margin of error we are comfortable with. For instance, if we want a 95% confidence level with a 5% margin of error, we would need to review approximately 400 labels, according to sample size calculation.

However, considering the large number of labels we have (100K), it might be more practical to review a larger sample, say 1,000 or even 2,000 labels, to get a more accurate estimate of the quality. This would reduce our margin of error and increase our confidence level.

As for how to sample them, we would want to use stratified random sampling. This is because we have labels from 20 different annotators and we want to ensure that our sample accurately represents the population.

Stratified random sampling involves dividing the population into homogeneous subgroups (strata) and then taking a simple random sample from each subgroup. In this case, we would divide the 100K labeled comments into 20 strata, each representing one annotator. Then, we would randomly sample from each stratum. This ensures that our sample includes labels from all annotators and that no annotator is over- or under-represented in the sample.

Here's a simple way to do this in Python using pandas:

```
# Assume df is a pandas DataFrame containing the labeled comments  
# and 'annotator' is the column containing the annotator IDs
```

```
# Number of samples from each annotator  
n_samples = 50 # Adjust this based on the total sample size you want  
  
# Perform stratified random sampling  
sample_df = df.groupby('annotator').apply(lambda x: x.sample(n_samples))  
  
# Reset the index  
sample_df = sample_df.reset_index(drop=True)
```

This will give you a DataFrame `sample_df` containing the sampled labels, which you can then review to estimate the quality of the labels.

- **[M] Suppose you work for a news site that historically has translated only 1% of all its articles. Your coworker argues that we should translate more articles into Chinese because translations help with the readership. On average, your translated articles have twice as many views as your non-translated articles. What might be wrong with this argument?**

**Hint: think about selection bias.**

The argument seems to suggest that translating more articles into Chinese will increase readership because the current translated articles have twice as many views. However, this argument might be flawed due to a concept called selection bias.

Selection bias occurs when the samples we're using to make conclusions aren't representative of the population we're interested in. In this case, we're only translating 1% of all articles. It's likely that the articles chosen for translation are the ones that are expected to be most popular or relevant to the Chinese readership. This could be because of the topic, the author, or any number of other factors.

If we're selectively translating only the best or most relevant articles, it's not surprising that these articles get more views. This doesn't necessarily mean that translating more articles, including potentially less popular or relevant ones, will result in the same increase in views.

To truly test the hypothesis that translating articles increases readership, we would need to conduct a more controlled experiment. For example, we could randomly select a portion of articles to be translated and compare their readership to those that weren't translated. This would help to ensure that the increase in readership is due to the translation itself, and not due to the inherent qualities of the articles chosen for translation.

- **[M] How to determine whether two sets of samples (e.g. train and test splits) come from the same distribution?**

There are several statistical tests to determine whether two sets of samples come from the same distribution. Here are a few:

1. **\*\*Kolmogorov-Smirnov Test (K-S Test)\*\***: This is a non-parametric test that compares the cumulative distributions of two data sets. The K-S test returns a D statistic and a p-value corresponding to the D statistic. A low D value suggests that the two samples were drawn from the same distribution.
2. **\*\*Chi-Square Test\*\***: This test is used to determine if there is a significant difference between the expected frequencies and the observed frequencies in one or more categories of a contingency table. In the context of comparing two distributions, the categories would be intervals of values, and the frequencies would be the number of data points in each interval.
3. **\*\*Anderson-Darling Test\*\***: This is another non-parametric test that can be used to compare the goodness of fit of a sample data set to a distribution. It gives more weight to the tails than the K-S test.
4. **\*\*Mann-Whitney U Test\*\***: This is a non-parametric test that can be used to determine if there are differences between two independent, non-normally distributed variables.

In Python, you can use the `scipy.stats` library to perform these tests. Here's an example of how to use the K-S test:

```
from scipy import stats

# Assume train and test are your two samples
D, p_value = stats.ks_2samp(train, test)

print('D statistic:', D)
print('p-value:', p_value)
```

The p-value is the probability of observing a statistic (or one more extreme) assuming the null hypothesis is true. A small p-value (typically  $\leq 0.05$ ) indicates strong evidence against the null hypothesis, so you reject the null hypothesis. In this case, the null hypothesis is that the two samples come from the same distribution. So, a small p-value would indicate that the two samples likely come from different distributions.

Remember, these tests assume that your data meets certain conditions (e.g., independence of observations), so it's important to ensure these conditions are met before you apply these tests.

- **[H] How do you know you've collected enough samples to train your ML model?**

Determining whether you've collected enough samples to train your machine learning model is a complex task. It depends on several factors including the complexity of the problem, the quality of the data, the diversity of the data, and the type of machine learning model you're using. Here are some general guidelines I follow:

1. **Complexity of the problem**: If the problem is simple and the data features are not too high-dimensional, fewer samples might be needed. For complex problems with high-dimensional data, more

samples are typically required.

2. **Quality of the data**: High-quality data can lead to better model performance even with fewer samples. Quality refers to how clean, relevant, and well-labeled the data is. If the data is noisy, has many missing values, or is poorly labeled, you may need more samples to achieve good performance.
3. **Diversity of the data**: If your data is not diverse and does not represent the full range of possible input values, your model may not generalize well to unseen data. In this case, you would need more diverse samples.
4. **Type of model**: Some models, like deep learning models, typically require a large amount of data. Others, like decision trees or SVMs, might perform well with fewer samples.
5. **Performance on a validation set**: This is a practical way to determine if you have enough data. Split your data into a training set and a validation set. Train your model on the training set and evaluate it on the validation set. If the model's performance on the validation set is satisfactory, you might have enough data. If not, collecting more data could help.
6. **Learning curves**: Plotting learning curves is another practical method. As you incrementally add more data, plot the model's performance on both the training set and validation set. If the performance continues to improve as you add more data, it's a sign that you could benefit from more data. If the curves plateau, additional data might not help much.

In practice, collecting more data is often expensive and time-consuming, so it's important to also consider these practical constraints. Furthermore, it's also crucial to remember that more data is not always the solution. Sometimes, improving the quality of the data, engineering better features, or choosing a more appropriate model can lead to better performance.

- **[M] How to determine outliers in your data samples? What to do with them?**

Outliers are data points that are significantly different from other observations. They can occur due to variability in the data or due to measurement errors. Identifying and handling outliers is a crucial step in data preprocessing as they can significantly impact the results of a machine learning model.

There are several ways to detect outliers:

1. **Statistical Methods**: One common method is to use the IQR (Inter Quartile Range). The IQR is the range between the first quartile (25th percentile) and the third quartile (75th percentile). Any data point that falls below the first quartile minus 1.5 times the IQR or above the third quartile plus 1.5 times the IQR is considered an outlier.
2. **Z-Score**: The Z-score is a measure of how many standard deviations an element is from the mean. A Z-score greater than 3 or less than -3 is typically considered to be an outlier.
3. **Visual Methods**: Box plots and scatter plots can also be used to identify outliers visually.

Once you've identified outliers, there are several strategies for handling them:

1. **Removal**: If you're confident the outlier is due to incorrectly entered or measured data, you might choose to exclude the outliers from your analysis.
2. **Imputation**: You can replace the outlier with statistical measures such as the mean, median, or mode.
3. **Binning**: You can group the data into bins and then replace the outliers with the border values.
4. **Transformation**: Apply a mathematical function to reduce the impact of the outlier.
5. **Model Robustness**: Use models that are less sensitive to outliers, like tree-based models.

The choice of method depends on the nature of the data and the specific requirements of the analysis. For instance, in fraud detection, outliers could be the most important observations, so you wouldn't want to remove or adjust them.

- Sample duplication

1. [M] When should you remove duplicate training samples? When shouldn't you?

Removing duplicate training samples in machine learning is a decision that depends on the specific context of the problem you are trying to solve. Here are some considerations:

1. **When to remove duplicates:**

- \* **When duplicates could skew the model:** If the dataset is small and there are many duplicates, the model might overfit to these instances and perform poorly on unseen data. For example, in a fraud detection scenario, if fraudulent transactions are over-represented in the training set, the model might overestimate the likelihood of fraud in the real world.

- \* **When duplicates are a result of data collection error:** If the duplicates are not representative of the underlying distribution and are simply errors, they should be removed to prevent the model from learning incorrect patterns.

2. **When not to remove duplicates:**

- \* **When duplicates represent common instances:** In some cases, duplicate instances might be common in the real-world scenario the model will be deployed in. For example, in a product recommendation system, there might be many instances of popular items being bought together. Removing these duplicates might cause the model to underestimate the popularity of these items.

- \* **When duplicates carry meaningful information:** In time series data, duplicates might indicate recurring patterns that are important for the model to learn. For example, in stock price prediction, similar patterns might occur at different times, and removing these duplicates might cause the model to miss these patterns.

In conclusion, the decision to remove duplicates should be made carefully, considering the nature of the data and the problem at hand. It's also a good practice to experiment with both approaches and validate the model's performance using a hold-out validation set or cross-validation to make an informed decision.

## 2. [M] What happens if we accidentally duplicate every data point in your train set or in your test set?

If every data point in the training set is duplicated, it would not fundamentally change the model's understanding of the underlying patterns in the data. However, it could potentially lead to overfitting, especially if the model is complex and has a high capacity to learn from data. This is because the model could start to learn the noise in the data or the specificities of the duplicated examples rather than the general patterns.

For example, in a decision tree, a duplicated data point could lead to a more complex tree, as it might split based on the specificities of the duplicated data point. In a linear regression model, the duplicated data points would have a higher influence on the model parameters, potentially leading to a biased model.

If every data point in the test set is duplicated, it would not affect the training of the model, but it would artificially reduce the variance of the test error estimate, making the model appear more accurate than it actually is. This is because the test set is supposed to provide an unbiased estimate of the model's performance on unseen data, and duplicating data points does not provide any new information to the model.

In both cases, it's important to ensure that the data is correctly preprocessed and that no duplicates are present, unless they represent genuinely repeated observations in the underlying population. Data preprocessing is a crucial step in the machine learning pipeline, and it's always important to check for and handle duplicates appropriately.

- Missing data

1. [H] In your dataset, two out of 20 variables have more than 30% missing values. What would you do?

Handling missing data is a critical step in the data preprocessing phase of any machine learning project. When faced with a situation where two out of twenty variables have more than 30% missing values, there are several strategies I could employ:

1. **\*\*Imputation\*\***: This is the process of replacing missing data with substituted values. The simplest form of imputation is to fill missing values with the mean, median, or mode of the column. However, this might not be the best approach when dealing with a high percentage of missing values, as it can lead to biased or incorrect results. More sophisticated imputation methods include using algorithms like k-Nearest Neighbors (where missing values are replaced with those of the nearest neighbors), or using regression models.
2. **\*\*Deletion\*\***: If the variables with missing values are not critical to the analysis or prediction task, it might be best to simply remove these variables from the dataset. However, this decision should be made carefully, as it could potentially lead to loss of information.
3. **\*\*Predictive Modeling\*\***: We could treat the variable with missing values as a target variable and build a predictive model to estimate those missing values. This could be a regression model if the variable is continuous, or a classification model if the variable is categorical.
4. **\*\*Understanding the Nature of the Missing Data\*\***: It's important to understand the nature of the missing data: is it Missing Completely At Random (MCAR), Missing At Random (MAR), or Missing Not At Random (MNAR)? This understanding can guide the strategy for handling missing data.

In this particular case, with more than 30% missing values, I would first try to understand why these values are missing. If they are MCAR or MAR, I might consider imputation or predictive modeling,

depending on the nature of the variables and the rest of the dataset. If they are MNAR, it might be more appropriate to delete these variables, as imputation or predictive modeling could lead to biased results.

In all cases, it's important to validate the chosen approach by comparing models trained on the processed data with models trained on the original data (with missing values), to ensure that the handling of missing data does not lead to worse performance.

## 2. [M] How might techniques that handle missing data make selection bias worse? How do you handle this bias?

When we handle missing data, we need to be careful about introducing selection bias. Selection bias occurs when the data you're working with is not representative of the population you're trying to understand or make predictions about.

In the context of missing data, selection bias can be exacerbated when the missingness is not random, which is often the case. For example, if you're working with a dataset of patient health records and the patients who missed their appointments (and thus have missing data) are more likely to be unhealthy, simply dropping those records or imputing their missing values could introduce bias. If you drop the records, your dataset will be biased towards healthier patients. If you impute the missing values, the imputed values may not accurately reflect the health status of the patients who missed their appointments.

To handle this bias, we need to understand the mechanism of missingness. There are three types: Missing Completely at Random (MCAR), Missing at Random (MAR), and Missing Not at Random (MNAR).

1. MCAR: The missingness is not related to any values, observed or missing. In this case, we can safely delete the missing data or use simple imputation methods without introducing bias.

2. MAR: The missingness is related to the observed data but not the missing data. In this case, we can use methods like multiple imputation or model-based methods which take into account the observed data.
3. MNAR: The missingness is related to the missing data. This is the most challenging scenario and requires more sophisticated techniques. We might use methods like Full Information Maximum Likelihood (FIML) or Multiple Imputation using Chained Equations (MICE), which can handle MNAR under certain conditions.

In all cases, it's crucial to understand the reason for the missingness and choose an appropriate method to handle it. It's also important to perform sensitivity analysis to assess the robustness of your results to different assumptions about the missing data.

- **[M] Why is randomization important when designing experiments (experimental design)?**

Randomization plays a crucial role in experimental design for several reasons:

1. **Reduction of Bias**: Randomization helps to eliminate the effect of confounding variables. These are variables that you're not interested in, but that might affect the outcome of the experiment. By randomizing, you ensure that these confounding variables are equally distributed among your experimental groups, reducing their impact.
2. **Validity of Statistical Tests**: Many statistical tests assume that your data is randomly sampled. If this assumption is violated, the results of your tests may not be valid.
3. **Generalizability of Results**: Randomization helps to ensure that your results are representative of the larger population. If your sample is not randomized, it may be biased in some way, and the results of your experiment may not generalize to the larger population.
4. **Avoidance of Accidental Bias**: Sometimes, even with the best intentions, experimenters can introduce bias into their experiments without realizing it. Randomization helps to avoid this by ensuring that

assignment to experimental groups is completely random and not influenced by any potential unconscious biases of the experimenter.

For instance, let's say we're conducting an A/B test on a website to determine whether a new design (version B) leads to more user sign-ups than the current design (version A). If we don't randomize the assignment of users to version A or B, we might unintentionally introduce bias. For example, if more tech-savvy users tend to visit the website during certain hours, and we deploy version B during those hours, we might mistakenly attribute an increase in sign-ups to the new design, when it's actually due to the higher tech-savviness of the users during those hours. Random assignment would help to ensure that both versions A and B are tested on a representative sample of users.

- **Class imbalance.**

1. **[E] How would class imbalance affect your model?**

Class imbalance can significantly affect the performance of a machine learning model. In a binary classification problem, if one class has significantly more samples than the other, the model may become biased towards the majority class. This is because the model is trained to minimize error and, in the presence of class imbalance, it can achieve a lower error rate by simply predicting the majority class most of the time.

For example, let's say we have a dataset where 95% of the samples are of class A and only 5% are of class B. A model could achieve 95% accuracy by simply predicting class A all the time. However, this model would be completely useless for predicting class B, which could be a problem if class B is the class of interest (like detecting a rare disease in medical diagnosis).

This is why accuracy is not always a good metric to evaluate model performance, especially in the presence of class imbalance. Other metrics like precision, recall, F1 score, or the area under the ROC curve (AUC-ROC) can provide a more balanced view of the model's performance across all classes.

To handle class imbalance, we can use techniques like resampling (either oversampling the minority class or undersampling the majority class), generating synthetic samples with SMOTE, or using class weights in the model to give higher importance to the minority class.

## 2. [E] Why is it hard for ML models to perform well on data with class imbalance?

Class imbalance is a common problem in machine learning where the number of observations in one class significantly outweighs the observations in the other class. For example, in a binary classification problem, you might have 95% of observations in class 0 and only 5% in class 1.

The main issue with class imbalance is that it makes the machine learning model biased towards the majority class. This happens because most machine learning algorithms are designed to maximize overall accuracy, which can be achieved by simply predicting the majority class in the case of imbalanced datasets.

For instance, in the aforementioned example, a model that always predicts class 0 would achieve an accuracy of 95%, which seems impressive on the surface. However, this model would be completely useless for predicting class 1, which could be the class of interest in many cases (like detecting a rare disease or fraudulent transactions).

In such scenarios, the minority class, even though it's the class of interest, is often treated as noise and ignored. This leads to poor predictive performance for the minority class, which is often the more important or more interesting class in real-world applications.

Therefore, it's crucial to use appropriate techniques to handle class imbalance, such as resampling methods (oversampling the minority class or undersampling the majority class), using different performance metrics (like precision, recall, F1 score, or AUC-ROC instead of accuracy), or implementing cost-sensitive learning where misclassifying the minority class is penalized more heavily.

- 3. [M] Imagine you want to build a model to detect skin legions from images. In your training dataset, only 1% of your images shows signs of legions. After training, your model seems to make a lot more false negatives than false positives. What are some of the techniques you'd use to improve your model?**

Class imbalance is a common problem in machine learning, especially in medical imaging where the presence of a disease is often the minority class. In this case, our model seems to be biased towards predicting the majority class, leading to a high number of false negatives. Here are some techniques we can use to address this issue:

1. **Resampling Techniques**: We can either oversample the minority class, undersample the majority class, or do a combination of both. However, oversampling can lead to overfitting due to duplicate instances, while undersampling can lead to loss of information.
2. **Synthetic Minority Over-sampling Technique (SMOTE)**: This is a type of oversampling where, instead of creating exact replicas of minority class instances, we create synthetic instances that are combinations of the closest minority class instances.
3. **Cost-Sensitive Training**: We can assign a higher cost to misclassifying the minority class. Many machine learning algorithms have a parameter to set the class weights, which can be used to make the algorithm more sensitive to the minority class.
4. **Ensemble Methods**: Techniques like bagging and boosting can be used to create a strong classifier from multiple weak classifiers. In particular, boosting algorithms like AdaBoost or Gradient Boosting can be effective as they focus on instances that are hard to classify.
5. **Evaluation Metrics**: Instead of accuracy, we should use metrics that are more informative for imbalanced datasets. Precision, recall, F1-score, and AUC-ROC are all good options. In this case, since we're particularly concerned about false negatives, recall (also known as sensitivity) might be a good metric to focus on.

6. **Data Augmentation**: In the context of image data, we can create new training samples by applying transformations such as rotations, translations, zooming, etc. This can help increase the size of our minority class.

7. **Transfer Learning**: If we have a small dataset, we can use a pre-trained model on a similar task as a starting point. This model will have learned useful features which can help even when data is imbalanced.

In practice, we might need to try a combination of these techniques to achieve the best results. It's also important to use cross-validation to get a reliable estimate of the model's performance, as the class imbalance can make the model's performance highly variable across different splits of the data.

- **Training data leakage.**

1. [M] Imagine you're working with a binary task where the positive class accounts for only 1% of your data. You decide to oversample the rare class then split your data into train and test splits. Your model performs well on the test split but poorly in production. What might have happened?

Data leakage might have occurred in this scenario. This happens when information from outside the training dataset is used to create the model. This can lead to overly optimistic performance estimates, as the model has effectively been given the answer in addition to the problem.

In this case, the oversampling of the rare class was done before splitting the data into training and test sets. This means that some of the same data points that were replicated in the oversampling process could have ended up in both the training and test sets. This would give an artificially high estimate of model performance, as the model would already have seen the 'new' data in the test set during training.

In production, however, the model would be dealing with completely new data, and the performance would likely drop as a result. This is because the model has been trained and evaluated on a dataset that is not representative of the actual data it will encounter.

To avoid this, you should first split your data into training and test sets, and then apply oversampling only to the training set. This way, the test set remains a realistic and independent evaluation of your model's performance.

Here's an example of how you might do this using Python's scikit-learn and imbalanced-learn libraries:

```
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE

# split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# apply SMOTE to the training data only
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train)
```

In this code, `x` and `y` are your features and labels, and `SMOTE` is a method for oversampling the minority class. The key point is that the oversampling is done after the train-test split, so the test data remains independent.

2. **[M] You want to build a model to classify whether a comment is spam or not spam. You have a dataset of a million comments over the period of 7 days. You decide to randomly split all your**

data into the train and test splits. Your co-worker points out that this can lead to data leakage.  
How?

Hint: You might want to clarify what oversampling here means. Oversampling can be as simple as duplicating samples from the rare class.

Data leakage refers to a mistake in the process of preparing data and creating the model where information from outside the training dataset is used to create the model. This can lead to overly optimistic performance estimates.

In our case, if we randomly split all our data into train and test splits, we might end up with a situation where comments from the same user, or comments that are very similar, are present in both the training and test datasets. This is because comments from the same user or similar comments are likely to be highly correlated.

For example, if a spammer posted similar spam comments throughout the 7 days, and we randomly split the data, it's likely that some of the spammer's comments will end up in the training set and some in the test set. Our model might then do well on the test set not because it's good at detecting spam in general, but because it's seen very similar examples in the training set. This is a form of data leakage because we're essentially using information from the test set (the spammer's commenting pattern) to build our model.

To avoid this, we could ensure that all comments from the same user end up in the same split. Or, if we have timestamps for the comments, we could split the data chronologically, using comments from the first few days for training and comments from the last few days for testing. This would better simulate the real-world scenario where we need to predict whether new, unseen comments are spam or not.

- [M] How does data sparsity affect your models?

Hint: Sparse data is different from missing data.

Data sparsity can significantly impact the performance of machine learning models. Here are a few ways how:

1. **Model Complexity**: Sparse data can lead to more complex models. For instance, in a high-dimensional sparse dataset, many features have zero or near-zero variance. These features can increase the complexity of the model without adding much predictive power, leading to overfitting.
2. **Computational Efficiency**: Sparse data can be computationally expensive to process. Many machine learning algorithms are not optimized for sparse data and can be inefficient or slow when dealing with it. This is particularly true for algorithms that rely on matrix operations, as sparse matrices can lead to a lot of wasted computation on zero values.
3. **Model Performance**: Sparse data can negatively impact the performance of some models. For example, distance-based algorithms like K-Nearest Neighbors (KNN) or Support Vector Machines (SVM) can perform poorly with sparse data because the high dimensionality can make it difficult to calculate meaningful distances.
4. **Feature Selection**: Sparse data can make feature selection challenging. With many features having zero or near-zero variance, it can be difficult to determine which features are truly important for prediction.

To handle sparse data, we can use techniques such as dimensionality reduction (e.g., PCA), feature selection, or use models that are designed to handle sparsity (e.g., Lasso regression, which can perform feature selection by driving less important feature coefficients to zero).

Also, we can use sparse data representations to improve computational efficiency. For example, in Python's Scipy library, there are several data structures like CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column) that are designed specifically for sparse data. These structures only store non-zero elements, which can significantly reduce memory usage and improve computation speed.

In my previous project, I worked with a high-dimensional text dataset that was very sparse. I used TF-IDF for feature extraction, which resulted in a sparse matrix. To handle this, I used Lasso regression for its ability to handle sparsity and perform feature selection. This approach improved the model's performance and computational efficiency.

- **Feature leakage**

1. **[E] What are some causes of feature leakage?**

Feature leakage, also known as data leakage, is a problem in machine learning where information from the future, or from the target variable, is accidentally used to train the model. This can lead to overly optimistic performance estimates that don't hold up in real-world applications. Here are some common causes of feature leakage:

1. **Time series data:** If your model is trained on future data, it will have an unfair advantage as it's using information that wouldn't be available at the time of prediction. For example, using tomorrow's stock price to predict today's stock price.
2. **Preprocessing:** If you preprocess your entire dataset before splitting it into training and testing sets, you can introduce leakage. For instance, if you normalize your entire dataset using the mean and standard deviation, the test set's information is used to transform the training set.
3. **Incorrect feature engineering:** If you create features based on information that includes the target variable, you can introduce leakage. For example, if you're predicting whether a patient will develop a disease and one of your features is 'number of doctor visits in the next year', that's leakage because you wouldn't know that information at the time of prediction.
4. **Data errors:** Sometimes, data errors can cause leakage. For example, if there's a data entry error and the target variable is accidentally included in the feature set.

5. \*\*Overfitting:\*\* Overfitting can also be a form of leakage, where the model learns to perform exceptionally well on the training data, but poorly on unseen data. This is because it has effectively 'leaked' information from the training set into the model's predictions.

To prevent feature leakage, it's important to carefully consider the data you're using to train your model, how you're preprocessing it, and how you're creating your features. It's also crucial to validate your model on unseen data to ensure it generalizes well.

## 2. [E] Why does normalization help prevent feature leakage?

Normalization is a preprocessing step in machine learning that helps to standardize the range of input features or variables. This is particularly important when the features have different scales, as it can help the algorithm to converge faster and perform better.

However, normalization itself doesn't directly prevent feature leakage. Feature leakage is a different problem that occurs when information from outside the training dataset is used to create the model. This can lead to overly optimistic performance estimates that don't hold up when the model is applied to new data.

Normalization and feature leakage are related in the sense that they both involve careful management of your data during the preprocessing and model training stages. When normalizing data, it's important to only use information from the training set to avoid leakage. For example, if you're scaling features to have zero mean and unit variance, you should calculate the mean and variance using only the training data, not the entire dataset.

In summary, normalization helps to improve model performance by ensuring all features are on the same scale, but it doesn't directly prevent feature leakage. To avoid leakage, you should be careful to only use information from your training set when preprocessing your data.

## 3. [M] How do you detect feature leakage?

Feature leakage, also known as data leakage, is a problem in machine learning where information from outside the training dataset is used to create the model. This can lead to overly optimistic performance estimates, as the model may simply be using leaked information to make accurate predictions.

Detecting feature leakage can be challenging, but there are several strategies that can be used:

1. **Understanding the data and the features**: This is the most important step. You need to understand each feature, its source, and how it was generated. If a feature was generated using information that won't be available at prediction time, then it's a leaky feature. For example, if you're predicting whether a patient will get a disease in the future and you have a feature that was generated after the disease was diagnosed, then that's a leaky feature.
2. **Temporal validation**: If your problem is time-sensitive (like forecasting or predicting future events), you should make sure that your validation strategy reflects this. A common mistake is to use future data to predict the past. In this case, you can use techniques like time-based cross-validation to detect leakage.
3. **Model performance too good to be true**: If your model is performing exceptionally well, it might be a sign of data leakage. If your model's accuracy is near 100% on your validation set, it's probably because it's using leaked information.
4. **Feature importance**: If a feature is extremely important in your model, it might be because it's leaking target information. You can use techniques like permutation importance, SHAP, or LIME to understand what features your model is using to make predictions.
5. **Statistical tests**: You can use statistical tests to check if a feature and the target variable are independent. If they are not, it might be a sign of leakage.
6. **Manual checks**: After creating features, you can manually check them for leakage. This can be time-consuming, but it's a surefire way to detect leakage.

In conclusion, detecting feature leakage requires a good understanding of your data, careful feature engineering, and rigorous validation. It's a crucial step in the machine learning process, as leakage can lead to overly optimistic results that don't hold up in the real world.

- [M] Suppose you want to build a model to classify whether a tweet spreads misinformation. You have 100K labeled tweets over the last 24 months. You decide to randomly shuffle on your data and pick 80% to be the train split, 10% to be the valid split, and 10% to be the test split. What might be the problem with this way of partitioning?

The problem with this approach is that it doesn't take into account the time series nature of the data. Tweets are time-stamped and the data is inherently sequential. By randomly shuffling the data, you're disrupting the temporal order of the data. This could lead to a situation known as "data leakage" where information from the future (test set) could be used to make predictions in the past (training set). This would not be possible in a real-world scenario where you're predicting on future unseen data, and it could lead to overly optimistic performance metrics.

In addition, trends and patterns in social media data can change over time. For example, the way people use language, the topics they discuss, and even the nature of misinformation can evolve. By training on a random subset of the data, the model might not learn these temporal patterns effectively.

A better approach would be to perform a time-based split for this kind of data. For instance, you could use the first 18 months of data for training, the next 3 months for validation, and the final 3 months for testing. This way, the model is trained on past data and validated and tested on future unseen data, which more accurately reflects how the model would be used in a real-world scenario.

- [M] You're building a neural network and you want to use both numerical and textual features. How would you process those different features?

Processing numerical and textual features for a neural network involves different preprocessing steps.

For numerical features, you would typically start with some form of normalization or standardization. This is because neural networks are sensitive to the scale of input features. For example, if one feature ranges from 0 to 1 and another from 1 to 1000, the network might focus more on the larger-scale feature, which can lead to suboptimal results. Standardization transforms the data to have zero mean and unit variance, while normalization scales the data to a fixed range, usually 0 to 1.

Here's a simple example using Scikit-learn's StandardScaler:

```
from sklearn.preprocessing import StandardScaler  
  
scaler = StandardScaler()  
numerical_features = scaler.fit_transform(numerical_features)
```

For textual features, you would typically convert the text into a numerical representation that the neural network can understand. This process is called vectorization. One common method is Bag of Words, which represents each document as a vector in a high-dimensional space, with each dimension corresponding to a unique word in the corpus. Another common method is TF-IDF, which weights each word by how informative it is in the given document. More advanced methods include word embeddings like Word2Vec or GloVe, which represent words in a continuous vector space where semantically similar words are mapped to nearby points.

Here's an example using Scikit-learn's TfidfVectorizer:

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
vectorizer = TfidfVectorizer()  
textual_features = vectorizer.fit_transform(textual_documents)
```

After preprocessing, you can concatenate the numerical and textual features and feed them into the neural network. Depending on the specific task, you might also want to use different architectures or layers for the different types of features. For example, Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs) can be particularly effective for textual data.

- [H] Your model has been performing fairly well using just a subset of features available in your data. Your boss decided that you should use all the features available instead. What might happen to the training error? What might happen to the test error?

Hint: Think about the curse of dimensionality: as we use more dimensions to describe our data, the more sparse space becomes, and the further are data points from each other.

Adding more features to a model can have a variety of effects on both the training and test error, and it's important to consider the concept of the curse of dimensionality when making this decision.

Firstly, regarding the training error, it's likely that it will decrease. This is because, with more features, the model can fit the training data more closely. It has more information to learn from, and therefore, it can capture more complex patterns present in the training data. This is especially true for models with high capacity, such as deep neural networks.

However, while the training error might decrease, the test error might actually increase. This is due to overfitting, where the model learns the noise in the training data rather than the underlying pattern.

Overfitting is more likely to occur when we have a high number of features relative to the number of training examples. This is a manifestation of the curse of dimensionality, where the data becomes increasingly sparse as the dimensionality increases, making it harder for the model to generalize from the training data to unseen data.

In addition, using all available features can also lead to increased computational cost and complexity. It can make the model slower to train and more difficult to interpret.

In practice, it's often beneficial to perform some form of feature selection or dimensionality reduction to keep only the most informative features. This can help to mitigate overfitting, reduce computational cost, and improve the interpretability of the model.

So, while it might seem like a good idea to use all available features to get the most out of our data, it's important to consider the potential downsides and make an informed decision based on the specific context and requirements of the project.

▼ Objective functions, metrics, and evaluation

- Convergence.

1. [E] When we say an algorithm converges, what does convergence mean?

In the context of machine learning, when we say an algorithm converges, we're referring to the process where the algorithm iteratively improves its performance (or reduces its error) over time and eventually reaches a state where further iterations do not significantly improve the performance or reduce the error. This state is often referred to as a "minimum" in the error or cost function that the algorithm is trying to optimize.

For example, in gradient descent, which is a common optimization algorithm used in machine learning, the algorithm starts with a random point in the error function and iteratively moves towards the direction of steepest descent (i.e., the gradient) until it reaches a point where the error is minimized. This point is considered as the convergence point of the algorithm.

It's important to note that the convergence point may not always be the global minimum of the error function. It could be a local minimum, depending on the starting point and the nature of the error

function. This is a common challenge in machine learning and there are various techniques to mitigate this, such as using different initialization strategies or optimization algorithms.

## 2. [E] How do we know when a model has converged?

In machine learning, a model is said to have converged when the error, or loss, stops decreasing significantly or remains constant over iterations. This means that the model's learning has stabilized and further training is unlikely to significantly improve its performance.

There are a few ways to determine if a model has converged:

1. **Monitoring the loss function**: The most common way is to monitor the loss function. If the loss function value stops decreasing and remains approximately constant, then the model has likely converged.
2. **Validation set performance**: Another way is to monitor the performance on a validation set. If the performance on the validation set stops improving or starts to degrade (indicating overfitting), then the model has likely converged.
3. **Early stopping**: This is a form of regularization where training is stopped as soon as the validation error reaches its minimum. If the validation error starts to increase, it means the model is starting to overfit the training data and it's best to stop training.
4. **Change in parameters**: You can also monitor the change in model parameters. If the parameters stop changing significantly, then the model has likely converged.

It's important to note that convergence doesn't necessarily mean that the model is good or that it will perform well on unseen data. It just means that further training on the same data is unlikely to improve the model's performance.

- [E] Draw the loss curves for overfitting and underfitting.

1. Underfitting: In an underfitted model, both the training and validation losses are high because the model fails to capture the underlying pattern of the data. The loss curves for training and validation would be high and very close to each other. The model is too simple to understand the complexity of the data. This is often a result of a model with too few parameters or features.

2. Overfitting: In an overfitted model, the training loss is very low because the model learns the training data too well, including its noise and outliers. However, the validation loss is high because the model fails to generalize to unseen data. The loss curve for training will be low and decreasing, while the validation loss curve will decrease initially but then start to increase. This is often a result of a model with too many parameters or features, or a model that has been trained for too long.

In an ideal scenario, both the training and validation losses decrease to a point of stability with a minimal gap between them. The exact values can vary based on the specific problem and model, but the key is that the validation loss is low and stable, indicating that the model is generalizing well to unseen data.

- **Bias-variance trade-off**

1. **[E] What's the bias-variance trade-off?**

The bias-variance trade-off is a fundamental concept in machine learning that describes the balance that must be achieved between bias and variance.

Bias refers to the error introduced by approximating a real-world problem, which may be extremely complicated, by a much simpler model. For example, assuming data is linear when it actually has a more complex structure. High bias can cause an algorithm to miss relevant relations between features and target outputs, leading to underfitting.

Variance, on the other hand, refers to the error introduced by the model's sensitivity to fluctuations in the training set. High variance can cause overfitting, where the model performs well on the training data but poorly on unseen data because it's too tuned to the training set and fails to generalize.

The trade-off comes in because minimizing bias leads to more complex models, which can increase variance, and vice versa. So, we need to find a balance where we minimize both as much as possible. This is typically done through techniques like regularization and cross-validation.

## 2. [M] How's this tradeoff related to overfitting and underfitting?

The bias-variance tradeoff is a fundamental concept in machine learning that deals with the balance between a model's complexity and its ability to generalize from training data to unseen data. It's directly related to the issues of overfitting and underfitting.

Bias refers to the error introduced by approximating a real-world problem, which may be extremely complicated, by a much simpler model. High bias can cause an algorithm to miss relevant relations between features and target outputs, leading to underfitting. In other words, if a model has high bias, it's too simple, doesn't learn enough from the training data, and therefore performs poorly on both the training and testing data.

Variance, on the other hand, refers to the amount by which our model would change if we estimated it using a different training dataset. High variance can cause an algorithm to model the random noise in the training data, leading to overfitting. If a model has high variance, it's too complex, learns too much from the training data including noise and outliers, and therefore performs well on the training data but poorly on unseen data.

The tradeoff comes into play because increasing the complexity of the model will decrease bias but increase variance, and vice versa. The goal is to find the right balance where total error, a combination of bias and variance, is minimized. This is where techniques like regularization, cross-validation, and ensemble methods can be very helpful in managing the bias-variance tradeoff and mitigating overfitting and underfitting.

**3. [M] How do you know that your model is high variance, low bias? What would you do in this case?**

A model with high variance and low bias is typically overfitting the training data. This means it performs very well on the training data but poorly on unseen data or test data. It has low bias because it doesn't make many assumptions about the data and tries to capture all the patterns in the training data, including the noise. This leads to high variance because it will produce very different outputs for small changes in the input.

There are a few ways to identify if your model is experiencing high variance:

1. **\*\*Cross-validation:\*\*** You can use cross-validation techniques like k-fold cross-validation. If your model's performance is excellent on the training set but significantly worse on the validation set, it is likely overfitting and has high variance.
2. **\*\*Learning curves:\*\*** Plotting learning curves, which are graphs that show the performance of a model on the training and validation sets over many iterations, can also help identify high variance. If the model performs well on the training set but poorly on the validation set, and the gap between the two persists even as more data is added, the model likely has high variance.
3. **\*\*Complexity curves:\*\*** These are plots of the model's performance on the training and validation sets as a function of model complexity. If the model's performance on the training set is much better than on the validation set and the gap increases with increasing model complexity, the model likely has high variance.

If your model has high variance, there are several strategies you can use to address this:

1. **\*\*Collect more data:\*\*** More data can help the model generalize better and reduce overfitting.
2. **\*\*Feature selection:\*\*** Reducing the number of features can simplify the model and reduce overfitting. Techniques like backward elimination, forward selection, or using L1 regularization (Lasso)

can help with this.

3. **Regularization:** Techniques like L2 regularization (Ridge) or L1 regularization (Lasso) can be used to penalize complex models and reduce overfitting.
4. **Simplifying the model:** If your model is too complex (e.g., a deep neural network with many layers), you might want to try a simpler model (e.g., a linear regression or a shallow neural network).
5. **Ensemble methods:** Techniques like bagging and boosting can help reduce variance by combining the predictions of several models.
6. **Early stopping:** In gradient descent, instead of iterating until the very end, you can stop the training early, for example, when the performance on the validation set starts to degrade.

Remember, the goal is to find a good balance between bias and variance, not necessarily to have the lowest possible variance. A model with very low variance might have high bias, and vice versa. The best model usually lies somewhere in the middle.

4. **[M] How do you know that your model is low variance, high bias? What would you do in this case?**

A model with high bias and low variance typically underfits the data. This means that the model is too simple to capture the underlying patterns in the data, leading to poor performance.

You can identify a high bias, low variance model through the following indicators:

1. **Poor performance on the training set:** Since the model is too simple, it may not perform well even on the training data.
2. **Little difference between training and test performance:** A high bias model will perform poorly on both the training and test sets. If there is little to no difference between the performance on these two sets, that's a good indication of high bias.

3. **Visual inspection**: If you can plot the model's predictions, a high bias model will often appear overly simplified compared to the actual data.

To address a high bias, low variance model, you can:

1. **Increase model complexity**: This could be achieved by adding more features, creating interaction features, or increasing the order of existing features. For example, if you're using a linear regression model, you could add polynomial features to increase complexity.
2. **Change the model**: If the current model is too simple, you could consider using a more complex model. For example, you could switch from linear regression to a decision tree or a neural network.
3. **Reduce regularization**: If you're using a model with regularization (like Ridge or Lasso regression), reducing the regularization strength could help the model fit the data better.

Remember, the goal is to find the right balance between bias and variance. Making the model too complex can lead to overfitting, where the model starts to learn the noise in the data, leading to high variance. So, it's important to monitor the model's performance on both the training and validation sets as you make adjustments.

- **Cross-validation**.

1. **[E] Explain different methods for cross-validation.**

Cross-validation is a technique used in machine learning to assess the predictive performance of a model and check for overfitting. It involves partitioning the dataset into two sets: a training set used to train the model, and a validation set used to evaluate the model. Here are some common methods for cross-validation:

1. **K-Fold Cross-Validation**: In this method, the dataset is randomly partitioned into 'k' equal size subsamples. Of the 'k' subsamples, a single subsample is retained as the validation data for testing the

model, and the remaining ' $k-1$ ' subsamples are used as training data. The cross-validation process is then repeated ' $k$ ' times, with each of the ' $k$ ' subsamples used exactly once as the validation data. The ' $k$ ' results can then be averaged to produce a single estimation.

2. **\*\*Stratified K-Fold Cross-Validation\*\*:** This is a variation of K-Fold that can produce more reliable results when dealing with binary classification problems or when the data is imbalanced. In this method, each fold contains roughly the same proportions of the different types of class labels.
3. **\*\*Leave One Out Cross-Validation (LOOCV)\*\*:** This is a special case of K-Fold cross-validation where ' $k$ ' is set to the total number of observations in the dataset. In other words, for a dataset with ' $n$ ' observations, ' $n$ ' folds are created. In each iteration, one observation is used for validation while the remaining ' $n-1$ ' observations are used for training. This method can be computationally expensive for large datasets.
4. **\*\*Leave P Out Cross-Validation (LPOCV)\*\*:** This is a more general case of LOOCV where ' $p$ ' observations are left out in each fold. These ' $p$ ' observations serve as the validation set, and the remaining observations serve as the training set.
5. **\*\*Time Series Cross-Validation\*\*:** This method is used for time series data where the order of observations matters. The training set consists of observations from the start up to a given point, and the validation set consists of observations after that point.

Each of these methods has its own advantages and disadvantages, and the choice of method depends on the specific problem and dataset at hand.

## 2. **[M] Why don't we see more cross-validation in deep learning?**

Cross-validation is a powerful technique in machine learning that helps to avoid overfitting and provides a robust estimate of model performance. It involves partitioning the dataset into subsets, training the model on some subsets (training set) and validating the model on the remaining subsets (validation

set). This process is repeated multiple times, each time with a different subset used as the validation set.

However, cross-validation is not commonly used in deep learning for several reasons:

1. **Computational Cost**: Deep learning models are computationally expensive to train. They often involve large amounts of data and complex architectures. Cross-validation, especially k-fold cross-validation, would require the model to be trained k times, which could be computationally prohibitive.
2. **Data Abundance**: Deep learning models often require large amounts of data, and in many cases, the datasets used are so large that a simple train/test split is sufficient to get a reliable estimate of the model performance. In such cases, the added benefit of cross-validation may not justify the additional computational cost.
3. **Overfitting is less of a concern**: With large datasets and proper regularization, overfitting is less of a concern in deep learning. Therefore, the main benefit of cross-validation, which is to provide a more robust estimate of the model performance and prevent overfitting, may not be as critical.
4. **Time complexity**: Training deep learning models can take a long time, even days or weeks. Cross-validation would multiply this time, which is often not practical.

That said, it's not that cross-validation is never used in deep learning. In certain cases, such as when the available data is limited, cross-validation can still be beneficial. It's just that the circumstances where cross-validation is most beneficial are less common in deep learning compared to other areas of machine learning.

- **Train, valid, test splits.**

1. **[E] What's wrong with training and testing a model on the same data?**

Training and testing a machine learning model on the same data is a bad practice because it can lead to overfitting. Overfitting is a scenario where the model learns the training data so well, including its noise and outliers, that it performs poorly on unseen data.

When you train and test on the same data, the model will have an artificially high performance metric because it's essentially being tested on data it has already seen. This doesn't give a true measure of how well the model has learned to generalize from the training data to unseen data, which is the ultimate goal of machine learning.

The proper practice is to split your data into at least two sets: a training set and a test set. The model is trained on the training set and then tested on the test set. This gives a more realistic measure of the model's performance on unseen data.

In many cases, it's also beneficial to have a third, validation set. The model is trained on the training set, tuned with the validation set, and finally, the performance is tested on the test set. This helps to tune the model's hyperparameters without leaking information from the test set into the model during training.

## 2. [E] Why do we need a validation set on top of a train set and a test set?

The main reason we need a validation set, in addition to a training set and a test set, is to tune the hyperparameters of our model and prevent overfitting.

Here's a more detailed explanation:

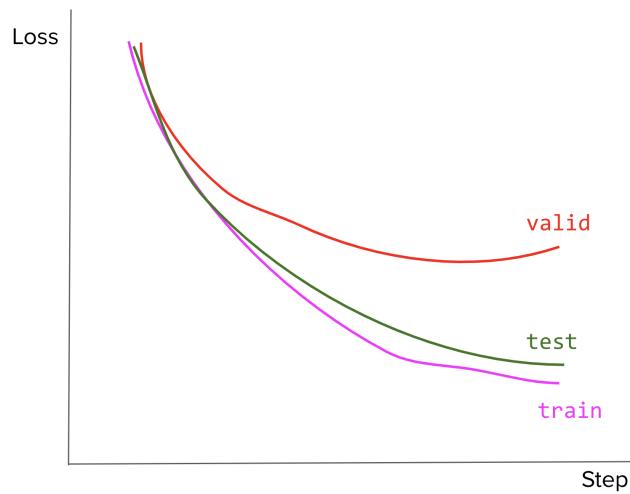
1. \*\*Training Set\*\*: This is the data on which we train our model. The model learns from this data.
2. \*\*Validation Set\*\*: After we've trained our model, we need to tune the hyperparameters (like learning rate, number of layers in a neural network, number of trees in a random forest, etc.). We use the validation set for this purpose. It acts as a check during training to see how well the model is

generalizing to data it hasn't seen before. It's important to note that the model doesn't learn from this data, but we adjust the model based on the performance on the validation set.

3. **Test Set**: Finally, after all the training and validation, we need a final check to see how well our model performs on completely unseen data. This is where the test set comes in. It's used only once, after we've finalized our model, to evaluate its performance.

In summary, the validation set helps us tune our model and prevent overfitting on the training data, and the test set gives us a final, unbiased estimate of our model's performance on unseen data.

3. **[M] Your model's loss curves on the train, valid, and test sets look like this. What might have been the cause of this? What would you do?**



The graph shows the loss curves for training, validation, and test sets decreasing over time, which is generally a good sign in machine learning models, as it suggests that the model is learning from the

data. However, the fact that the validation and test loss are higher than the training loss indicates that the model may be overfitting to the training data.

Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise specific to it. This typically results in high accuracy on the training data but poorer performance on unseen data. In the given graph, the validation and test losses are converging, which suggests that the model is generalizing somewhat properly on unseen data, but not as well as on the training data.

To address this issue, I would consider the following steps:

1. **Regularization:** Implement techniques such as L1 or L2 regularization to penalize complex models and thus prevent overfitting.
2. **Dropout:** Use dropout in neural networks, which randomly disables neurons during training, forcing the network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
3. **Data Augmentation:** Increase the size and variability of the training set by augmenting the data, thereby providing the model with more examples to learn from and improving its generalization capabilities.
4. **Early Stopping:** Stop the training process before the model has a chance to overfit. This involves monitoring the validation loss and stopping training when the validation loss begins to increase, despite improvements in training loss.
5. **Hyperparameter Tuning:** Adjust learning rates, batch sizes, or model architectures to see if a different set of parameters would help the model generalize better.
6. **Cross-validation:** Instead of a single validation set, use cross-validation to ensure that the model's performance is consistent across different subsets of the data.

7. **Model Complexity:** If the model is too complex for the given data, simplifying the model (using fewer layers or neurons) might help to prevent overfitting.

8. **Ensemble Methods:** Combine the predictions of multiple models to improve generalization.

For instance, if I were using a neural network, I might introduce dropout layers into the architecture like so:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential([
    Dense(128, activation='relu', input_shape=(input_shape,)),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
```

In this code snippet, `Dropout(0.5)` randomly sets a fraction `0.5` of input units to 0 at each update during training time, which helps prevent overfitting.

By implementing these techniques, I would aim to reduce the gap between training and validation/test loss, thereby improving the model's ability to generalize to new data

- **[E] Your team is building a system to aid doctors in predicting whether a patient has cancer or not from their X-ray scan. Your colleague announces that the problem is solved now that they've built a**

## system that can predict with 99.99% accuracy. How would you respond to that claim?

While a 99.99% accuracy rate sounds impressive, it's important to remember that accuracy isn't the only metric we should be considering, especially in a problem like this where the cost of false negatives (predicting a patient doesn't have cancer when they actually do) is extremely high.

In medical diagnostics, we often deal with imbalanced classes - where the number of negative cases (no cancer) far outweighs the positive cases (cancer). In such scenarios, a model could achieve high accuracy by simply predicting the majority class (no cancer) for all patients. However, this would be disastrous as it would miss all actual cases of cancer.

Instead, we should look at other metrics such as precision, recall, and the F1 score. Precision tells us how many of the predicted positive cases were actually positive, while recall tells us how many of the actual positive cases we were able to catch. The F1 score is the harmonic mean of precision and recall, and gives us a single metric that balances the two.

So, while the 99.99% accuracy is a good start, I would suggest that we also look at these other metrics to get a more complete picture of our model's performance. We should also consider using techniques like oversampling the minority class or using cost-sensitive learning to handle the class imbalance.

- F1 score.

1. [E] What's the benefit of F1 over the accuracy?

The F1 score is a more robust metric for evaluating the performance of a model, particularly in cases where the data is imbalanced. It's a measure of a model's precision and recall, and it's the harmonic mean of these two metrics.

Let's consider an example where we have a dataset with 95 negative instances and 5 positive instances. If we have a model that simply classifies all instances as negative, it would have an

accuracy of 95%. However, this model is not useful at all because it fails to identify any of the positive instances, which could be crucial in certain scenarios like fraud detection or disease diagnosis.

In such cases, the F1 score is a better metric because it takes into account both false positives and false negatives. A model with a high F1 score is both good at precision (it doesn't label many negative instances as positive) and recall (it finds a large number of the positive instances). So, even if the model's accuracy is high, a low F1 score would indicate that the model is not performing well in terms of precision and/or recall.

In summary, while accuracy can be a useful metric, it can be misleading when the classes are imbalanced. The F1 score, on the other hand, provides a more balanced measure of a model's performance.

## 2. [M] Can we still use F1 for a problem with more than two classes. How?

Yes, the F1 score can be extended to problems with more than two classes. The F1 score is the harmonic mean of precision and recall, and both of these metrics can be calculated for multi-class problems. There are a few different ways to do this:

1. **\*\*Micro-average\*\***: This method aggregates the contributions of all classes to compute the average metric. In a multi-class classification setup, micro-average is preferable if you suspect there might be class imbalance (i.e., you may have many more examples of one class than of other classes).

```
from sklearn.metrics import f1_score  
f1_score(y_true, y_pred, average='micro')
```

2. **\*\*Macro-average\*\***: This method computes the metric independently for each class and then takes the average (hence treating all classes equally), regardless of class imbalance.

```
from sklearn.metrics import f1_score  
f1_score(y_true, y_pred, average='macro')
```

3. **\*\*Weighted average\*\***: This method computes the F1 score for each class independently but when it adds them together uses a weight that depends on the number of true labels of each class. This takes into account the class imbalance.

```
from sklearn.metrics import f1_score  
f1_score(y_true, y_pred, average='weighted')
```

4. **\*\*None\*\***: This returns the score of each class.

```
from sklearn.metrics import f1_score  
f1_score(y_true, y_pred, average=None)
```

In all these cases, `y_true` is the list of true labels and `y_pred` is the list of predicted labels. The choice of method should depend on your specific needs and the nature of your problem.

- Given a binary classifier that outputs the following confusion matrix.

	Predicted True	Predicted False
Actual True	30	20
Actual False	5	40

1. [E] Calculate the model's precision, recall, and F1.

First, let's define precision, recall, and F1 score:

- **Precision** is the number of true positives divided by the number of true positives plus the number of false positives. It measures the accuracy of positive predictions.

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

- **Recall** (also known as sensitivity) is the number of true positives divided by the number of true positives plus the number of false negatives. It measures the ability of a classifier to find all the positive samples.

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

- **F1 Score** is the harmonic mean of precision and recall. It is a measure of a test's accuracy and considers both the precision and the recall.

$$\text{F1} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$$

Given the confusion matrix:

- True Positives (TP) = 30
- False Positives (FP) = 5
- False Negatives (FN) = 20
- True Negatives (TN) = 40

Let's calculate:

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP}) = 30 / (30 + 5)$$

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN}) = 30 / (30 + 20)$$

$$\text{F1} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$$

Let's perform the calculations for precision, recall, and F1 score.

1. The calculated values are as follows:

- **Precision:** Approximately 0.857
- **Recall:** Approximately 0.600
- **F1 Score:** Approximately 0.706

These metrics indicate that when the model predicts an instance as positive, it is correct about 85.7% of the time. However, it only identifies 60% of all actual positive instances. The F1 score, which balances precision and recall, is approximately 0.706, indicating there is room for improvement, especially in terms of recall.

## 2. [M] What can we do to improve the model's performance?

To improve the model's performance, we need to consider strategies that could either increase precision, recall, or both. Here are some approaches:

- **Feature Engineering:** Review and improve the feature set. Adding relevant features or removing noisy features could help the model make better predictions.
- **Resampling:** If the dataset is imbalanced, applying oversampling for the minority class or undersampling for the majority class could help the model learn more about the underrepresented class.
- **Threshold Adjustment:** Adjusting the decision threshold for classification. If the model is too conservative in predicting positives, lowering the threshold might improve recall.
- **Model Complexity:** Evaluate if the model is too simple or too complex. A more complex model might have a better capacity to learn from the data, while a simpler model might generalize better.

- **Cross-validation:** Use cross-validation to tune hyperparameters and avoid overfitting, ensuring that the model performs well across different subsets of the data.
- **Error Analysis:** Perform a thorough error analysis to understand where the model is making mistakes and why. This might provide insights into specific adjustments needed.
- **Regularization Techniques:** Use regularization methods like L1 or L2 to prevent overfitting and help the model generalize better.
- **Ensemble Methods:** Use ensemble learning methods to combine the strengths of multiple models, which can often yield better performance than any single model.

Each of these strategies might help improve the model's precision and recall in different ways, and they often need to be balanced against each other since improving one might lead to a decrease in the other. A thorough examination of the model's performance on a validation set would help in choosing the right approach.

- **Consider a classification where 99% of data belongs to class A and 1% of data belongs to class B.**
  1. **[M] If your model predicts A 100% of the time, what would the F1 score be? Hint: The F1 score when A is mapped to 0 and B to 1 is different from the F1 score when A is mapped to 1 and B to 0.**

The F1 score is the harmonic mean of precision and recall. In this case, if we map A to 0 and B to 1, and our model predicts A 100% of the time, we have a situation of extreme class imbalance and our model is essentially ignoring the minority class B.

In this scenario, the precision for class B is 0 because we have no true positives - we never predict class B. The recall for class B is also 0 because, again, we have no true positives.

The F1 score is calculated as:  $2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$ . Given both precision and recall are 0, the F1 score for class B is undefined or can be considered as 0 because we're not correctly predicting class B at all.

On the other hand, for class A, the precision is 0.99 because we have a lot of true positives (99% of data belongs to class A and we predict A all the time). The recall is 1 because we have no false negatives (we never predict B when it's actually A). So, the F1 score for class A would be:  $2 * (0.99 * 1) / (0.99 + 1) = 0.995$ .

So, if we're considering class B (the minority class) as the positive class, our F1 score is 0. If we're considering class A (the majority class) as the positive class, our F1 score is 0.995. This highlights the importance of carefully choosing which class is considered positive in the context of the problem you're solving, especially in cases of severe class imbalance.

2. **[M] If we have a model that predicts A and B at a random (uniformly), what would the expected F1 be?**

The F1 score is the harmonic mean of precision and recall. In this case, let's assume that the model predicts class A and class B uniformly at random. This means that the model will predict class A 50% of the time and class B 50% of the time.

Let's calculate precision and recall for class B (the minority class), as it's often the class of interest in imbalanced datasets.

Precision is the number of true positives (correctly predicted as B) divided by the total number of elements labeled as B (true positives + false positives). Since we're predicting B 50% of the time, and the actual proportion of B is 1%, the precision would be  $0.01 / 0.5 = 0.02$ .

Recall is the number of true positives divided by the total number of elements that actually belong to the positive class (true positives + false negatives). Since we're predicting B 50% of the time, and the

actual proportion of B is 1%, the recall would be  $0.01 / 0.01 = 1$ .

The F1 score is the harmonic mean of precision and recall, calculated as  $2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$ . Substituting the values we calculated, the F1 score would be  $2 * (0.02 * 1) / (0.02 + 1) = 0.0392$ .

So, if we have a model that predicts A and B uniformly at random in this scenario, we would expect the F1 score to be approximately 0.0392. This low score reflects the poor performance of the model, particularly in identifying the minority class (B), which is often the more important class in imbalanced datasets.

- **[M] For logistic regression, why is log loss recommended over MSE (mean squared error)?**

Logistic regression is a classification algorithm, where the goal is to predict a binary outcome - 1 (success) or 0 (failure). The output of logistic regression is a probability that the given input point belongs to a certain class.

Mean Squared Error (MSE) is more suitable for regression problems where we have continuous output. In classification problems, like the ones solved by logistic regression, we need a loss function that is more suitable for binary outcomes. That's where log loss (also known as logistic loss or cross-entropy loss) comes in.

Log loss provides a heavy penalty for predictions that are both wrong and confident. For example, if the actual class is 1, and the prediction is close to 0, the log loss becomes a large positive number, which is what we want in a cost function - higher cost for worse predictions.

On the other hand, MSE would not punish misclassifications as harshly. In the case of confident and wrong predictions, the MSE would be a small positive number, which doesn't reflect the seriousness of the misclassification.

Moreover, log loss takes into account the uncertainty of your prediction based on how much it varies from the actual label. This provides a more nuanced view of the performance of our model.

Lastly, logistic regression with log loss is a convex optimization problem, which guarantees that if a global minimum exists, the algorithm will find it. This is not the case with MSE and logistic regression, where multiple local minima might exist.

So, in summary, log loss is more suitable for logistic regression because it handles binary outcomes better, provides a harsher penalty for confident and incorrect predictions, and turns the problem into a convex optimization problem.

- **[M] When should we use RMSE (Root Mean Squared Error) over MAE (Mean Absolute Error) and vice versa?**

Both RMSE (Root Mean Squared Error) and MAE (Mean Absolute Error) are popular metrics for regression models, and they quantify the difference between the predicted and actual values. However, they have different characteristics and are used in different scenarios.

RMSE is more sensitive to large errors because it squares the differences before averaging them. This means that RMSE will be larger than or equal to MAE. If large errors are particularly undesirable in your application, RMSE would be a good metric to use. For example, if you're predicting house prices, an error of \$100,000 is more than twice as bad as an error of \$50,000. In such a case, RMSE would be a better choice because it penalizes large errors more heavily.

On the other hand, MAE is less sensitive to large errors and might be a better choice if all errors are equally important. It's a more robust metric against outliers. For instance, if you're predicting the time it takes for a package to be delivered, an error of 1 hour is just as bad whether the package was supposed to arrive in 2 hours or 5 hours. In this case, MAE would be a more appropriate metric because it treats all errors the same.

In summary, the choice between RMSE and MAE depends on the problem at hand and how you want to treat large errors. If large errors are much worse than small ones, use RMSE. If all errors are equally bad, use MAE.

- **[M] Show that the negative log-likelihood and cross-entropy are the same for binary classification tasks.**

Sure, I'd be happy to explain this. Let's start with the definitions.

The negative log-likelihood (NLL) for a binary classification problem is defined as:

$$\text{NLL} = -\frac{1}{N} \sum (y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i))$$

Where:

- N is the total number of observations
- $y_i$  is the actual class label (0 or 1) for observation i
- $p_i$  is the predicted probability of the positive class for observation i

On the other hand, the cross-entropy (CE) loss for a binary classification problem is defined as:

$$\text{CE} = -\frac{1}{N} \sum (y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i))$$

As you can see, the formulas for NLL and CE are identical for binary classification problems. Therefore, the negative log-likelihood and cross-entropy are the same for binary classification tasks.

The reason we have two different names for the same thing is historical and has to do with the fields in which these terms originated. Cross-entropy comes from information theory, while negative log-likelihood comes from statistics. But in the context of binary classification, they are the same thing.

It's also worth noting that while these are the same for binary classification, they can differ for multi-class classification problems depending on how you define and calculate them.

- **[M] For classification tasks with more than two labels (e.g. MNIST with 10 labels), why is cross-entropy a better loss function than MSE?**

Cross-entropy loss is generally preferred over Mean Squared Error (MSE) for multi-class classification tasks for a few reasons:

1. **\*\*Gradient Vanishing Problem\*\***: MSE can lead to slower convergence during training. The gradients can become very small if the predicted probability is far from the actual label, which can slow down the learning process. This is known as the vanishing gradient problem. Cross-entropy loss does not suffer from this problem, as it provides larger gradients when the predicted probability is far from the actual label, leading to faster learning.
2. **\*\*Interpretability\*\***: Cross-entropy loss is more interpretable in terms of information theory. It measures the "distance" between the actual and predicted probability distributions, or in other words, it quantifies the amount of information needed to encode the labels given the predictions. This is more intuitive and meaningful for classification tasks than the squared difference used in MSE.
3. **\*\*Optimization for Probability Outputs\*\***: Cross-entropy loss is more suitable for outputs that are probabilities. MSE assumes that the output is a numerical value and tries to minimize the difference between the predicted and actual numerical values. In contrast, cross-entropy loss is designed to measure the difference between two probability distributions, making it more suitable for classification tasks where the output is a probability distribution over the classes.

For example, consider a binary classification problem where the true label is 1 (i.e., the true probability distribution is [0, 1]). If the predicted probability distribution is [0.9, 0.1], the MSE is  $(0-0.9)^2 + (1-0.1)^2 = 0.82$ . But if the predicted distribution is [0.1, 0.9], the MSE is  $(0-0.1)^2 + (1-0.9)^2 = 0.02$ . Even though the second prediction is correct and the first one is completely wrong, the MSE for the first prediction is much higher than for the second one. This shows that MSE can be misleading for classification tasks. On the

other hand, the cross-entropy loss for the first prediction is  $-\log(0.1) = 2.3$ , and for the second prediction is  $-\log(0.9) = 0.1$ , which correctly reflects that the second prediction is much better than the first one.

- **[E] Consider a language with an alphabet of 27 characters. What would be the maximal entropy of this language?**

Entropy is a measure of uncertainty or randomness. In the context of information theory, it quantifies the amount of information in a message. The maximal entropy occurs when all outcomes are equally likely.

In this case, we have a language with an alphabet of 27 characters. If each character is equally likely, the entropy is maximized. The formula for entropy is:

$$H = - \sum P(x) \log_2 P(x)$$

where  $P(x)$  is the probability of each character. If each character is equally likely,  $P(x) = 1/27$  for all characters.

So, the maximal entropy would be:

$$\begin{aligned} H_{\max} &= - \sum (1/27) \log_2 (1/27) \text{ for all 27 characters} \\ &= \log_2(27) \end{aligned}$$

So, the maximal entropy of this language would be approximately 4.755 bits per character. This means that, on average, you would need about 4.755 bits to encode a character from this language under optimal encoding.

- **[E] A lot of machine learning models aim to approximate probability distributions. Let's say  $P$  is the distribution of the data and  $Q$  is the distribution learned by our model. How do we measure how close  $Q$  is to  $P$ ?**

There are several ways to measure how close a learned distribution  $Q$  is to the true data distribution  $P$ . One of the most common methods is to use a concept called divergence, which quantifies the difference

between two probability distributions.

A widely used type of divergence is the Kullback-Leibler (KL) divergence. KL divergence measures the amount of information lost when Q is used to approximate P. It is not symmetric, meaning the divergence of P from Q is not the same as the divergence of Q from P.

The KL divergence of Q from P is calculated as follows:

$$D_{KL}(P || Q) = \sum P(x) \log (P(x) / Q(x))$$

where the sum is over all possible events x, P(x) is the probability of x according to distribution P, and Q(x) is the probability of x according to distribution Q.

However, KL divergence has its limitations. For instance, it's not defined if there exists an event which has non-zero probability under P but zero probability under Q.

To address this, other divergences such as Jensen-Shannon divergence, which is symmetric and smoother, can be used. There are also metrics like the Earth-Mover's (Wasserstein) distance that can provide a more intuitive measure of the difference between two distributions, especially for continuous distributions.

In practice, the choice of measure depends on the specific characteristics of the problem and the distributions involved.

- **MPE (Most Probable Explanation) vs. MAP (Maximum A Posteriori)**

1. **[E] How do MPE and MAP differ?**

MPE (Most Probable Explanation) and MAP (Maximum A Posteriori) are both concepts used in Bayesian statistics, but they are used for different purposes.

MAP (Maximum A Posteriori) estimation is a method used to estimate the most likely value of a parameter given some observed data. It's a form of Bayesian inference that combines the prior

distribution of the parameter (what we think about the parameter before seeing the data) with the likelihood of the data given the parameter to form a posterior distribution. The parameter value that maximizes this posterior distribution is the MAP estimate.

For example, if we're trying to estimate the mean height of a population of people, we might have a prior belief about what that mean height is (based on previous studies or other information). When we collect some height data from a sample of the population, we can use MAP estimation to update our belief about the mean height.

On the other hand, MPE (Most Probable Explanation) is a concept used in graphical models, such as Bayesian networks or Markov random fields. It's used to find the most likely assignment of all variables in the model given some observed data. In other words, it's about finding the most probable state of the entire system, not just a single parameter.

For example, in a Bayesian network that models the relationships between different symptoms and diseases, given some observed symptoms, the MPE would be the most probable assignment of all diseases (i.e., whether each disease is present or not).

In summary, while both MAP and MPE are about finding "most probable" things, MAP is about finding the most probable value of a parameter, while MPE is about finding the most probable state of an entire system.

## 2. [H] Give an example of when they would produce different results.

MPE (Most Probable Explanation) and MAP (Maximum A Posteriori) are two inference techniques used in Bayesian networks. While they are similar, they can produce different results in certain situations.

Let's first understand what each of them does:

- MAP: It finds the most probable configuration of some subset of variables, given the values of the other variables. It's used to find the most probable state of some variables of interest, regardless of

the states of the other variables.

- MPE: It finds the most probable configuration of all variables in the network. It's used to find the most probable state of all variables in the network.

The key difference between the two is that MAP considers only a subset of variables, while MPE considers all variables in the network.

Let's consider an example to illustrate when they would produce different results:

Suppose we have a Bayesian network that models the health of a person. The variables are "Diet" (healthy/unhealthy), "Exercise" (yes/no), "Weight" (normal/overweight), and "Heart Disease" (yes/no).

Let's say we observe that the person has heart disease and we want to infer the most probable explanation for this. If we use MPE, it will consider all variables in the network and might give us a result like {Diet: unhealthy, Exercise: no, Weight: overweight, Heart Disease: yes} as the most probable explanation.

However, if we are only interested in the person's lifestyle habits and use MAP to infer the most probable state of the "Diet" and "Exercise" variables given the observation, it might give us a different result like {Diet: unhealthy, Exercise: yes}. This is because MAP only considers a subset of variables and ignores the others, so the most probable state of "Diet" and "Exercise" might be different when we don't consider "Weight".

In conclusion, while both MPE and MAP are useful inference techniques in Bayesian networks, they can produce different results depending on whether we want to consider all variables in the network (MPE) or only a subset of them (MAP).

- **[E] Suppose you want to build a model to predict the price of a stock in the next 8 hours and that the predicted price should never be off more than 10% from the actual price. Which metric would you use?**

### Hint: check out MAPE.

In this case, the Mean Absolute Percentage Error (MAPE) would be a suitable metric to use. MAPE measures the average absolute percent difference between the actual and the predicted values, which aligns well with your requirement that the predicted price should never be off more than 10% from the actual price.

MAPE is calculated using the formula:

$$\text{MAPE} = (1/n) * \sum (|\text{Actual} - \text{Predicted}| / |\text{Actual}|) * 100\%$$

where n is the number of data points.

The output is a percentage, which makes it easy to interpret. For instance, a MAPE of 10% indicates that the average prediction is off by 10% from the actual value. Therefore, in your case, you would aim to train your model such that the MAPE is less than or equal to 10%.

However, it's important to note that MAPE has its limitations. It's not defined when the actual value is zero, and it can put a heavier penalty on under-predictions than over-predictions. So, while it's a good starting point, you might also want to consider other metrics depending on the specific characteristics of your data and problem.

## ▼ Chapter 8: Machine Learning Algorithms Answer

### ▼ Classical Machine Learning Questions

#### 1. [E] What are the basic assumptions to be made for linear regression?

Linear regression is a statistical method that allows us to summarize and study relationships between two continuous (quantitative) variables. When we're using linear regression, there are a few key assumptions we need to make:

1. **\*\*Linearity\*\***: The relationship between the independent and dependent variables is linear. This means that the line of best fit through the data points is a straight line, rather than a curve or some sort of grouping factor.
2. **\*\*Independence\*\***: The observations are independent of each other. This means the observations are not derived from repeated measurements or matched data.
3. **\*\*Homoscedasticity\*\***: The variance of the errors is constant across all levels of the independent variables. In other words, the spread of the residuals (errors) should be the same across all values of the independent variables.
4. **\*\*Normality\*\***: The errors of the model are normally distributed. If the errors are not normally distributed, confidence intervals may become too wide or narrow.
5. **\*\*No Multicollinearity\*\***: The independent variables are not too highly correlated with each other. Multicollinearity can lead to unstable estimates of the regression coefficients and make it difficult to determine the effect of independent variables on the dependent variable.

Violations of these assumptions can lead to inaccurate or misleading results, so it's important to check these assumptions when fitting a linear regression model.

## 2. **[E] What happens if we don't apply feature scaling to logistic regression?**

If we don't apply feature scaling to logistic regression, it can lead to slower or less efficient model training, and potentially less accurate results.

Logistic regression uses a method called gradient descent to find the optimal parameters. If the features are not on a similar scale, gradient descent can take a long time to converge because it will oscillate more extensively towards the minimum.

For example, let's say we have two features: age (ranging from 0 to 100) and income (ranging from 0 to 100,000). Without scaling, the model will be dominated by income because of its larger range. This can lead to a model that's not as accurate as it could be, because it's not properly taking into account the influence of age.

So, while logistic regression will still work without feature scaling, it's generally a good idea to apply it to ensure efficient and accurate model training.

### 3. **[E] What are the algorithms you'd use when developing the prototype of a fraud detection model?**

When developing a prototype for a fraud detection model, there are several algorithms that can be used. The choice of algorithm often depends on the specific requirements of the project, the nature of the data, and the problem at hand. However, here are a few algorithms that are commonly used in fraud detection:

1. Logistic Regression: This is a simple and fast algorithm that is often used as a baseline in binary classification problems like fraud detection. It's easy to implement and interpret, making it a good choice for a prototype.
2. Decision Trees and Random Forests: These algorithms are particularly useful when the data has many features that interact in complex ways. Random Forests, which are ensembles of Decision Trees, can capture these interactions and are less prone to overfitting.
3. Gradient Boosting Machines (like XGBoost or LightGBM): These are powerful ensemble methods that build a series of weak learners (typically decision trees) in a way that each new tree corrects the mistakes of the previous ones. They often perform very well on a wide range of problems, but they can be more computationally intensive and harder to tune than simpler methods.
4. Neural Networks: Deep learning can be very effective for fraud detection, especially when there are complex patterns in the data or when the data includes unstructured components like text or images.

However, they require a lot of data and computational resources, and they can be harder to interpret than other methods.

5. Anomaly Detection Algorithms: Since fraud is often about finding unusual patterns in the data, algorithms specifically designed for anomaly detection can be very useful. Examples include One-Class SVM, Isolation Forest, and Local Outlier Factor (LOF).

6. Autoencoders: These are a type of neural network used for anomaly detection. They are trained to reconstruct their input data, and they tend to do poorly on data that is significantly different from what they were trained on - like fraudulent transactions might be.

Remember, the choice of algorithm is just one part of building a fraud detection system. It's also important to carefully preprocess the data, engineer relevant features, and evaluate the model's performance using appropriate metrics.

#### 4. Feature selection.

##### a. [E] Why do we use feature selection?

Feature selection is a critical step in the machine learning pipeline for several reasons:

1. **\*\*Improving Model Performance\*\***: Not all features are useful. Some may be irrelevant or redundant, adding noise rather than information. By selecting only the most important features, we can improve the accuracy of our model.
2. **\*\*Reducing Overfitting\*\***: A model that uses too many features, especially ones that are not relevant, is prone to "overfitting". Overfitting means the model performs well on the training data but poorly on unseen data. By reducing the number of features, we can reduce the risk of overfitting.
3. **\*\*Speeding up Training\*\***: Training a model on fewer features can significantly reduce the computational complexity of the model, making it faster to train.

4. **\*\*Improving Interpretability\*\***: Models with fewer features are easier to understand and interpret. This is especially important in situations where we need to explain the model's decisions, such as in healthcare or finance.
5. **\*\*Reducing Data Dimensionality\*\***: High-dimensional data can be challenging to work with, and it can be difficult to visualize or find patterns in such data. Feature selection can help reduce dimensionality, making the data easier to understand.

In summary, feature selection helps us create models that are more accurate, faster to train, less prone to overfitting, and easier to understand.

b. **[M] What are some of the algorithms for feature selection? Pros and cons of each.**

There are several algorithms and techniques for feature selection in machine learning, each with its own pros and cons. Here are a few of them:

1. **\*\*Filter Methods\*\***: These methods select features based on their scores in various statistical tests for their correlation with the outcome variable. Examples include Chi-Squared test, information gain, and correlation coefficient scores.

Pros: They are very fast to compute and independent of the model.

Cons: They do not consider the interaction of features.

2. **\*\*Wrapper Methods\*\***: These methods consider the selection of a set of features as a search problem. Examples include recursive feature elimination, forward selection, and backward elimination.

Pros: They are able to detect the interaction between variables since they use a machine learning model to score the feature subsets.

Cons: They can be computationally expensive, especially when the feature space is large.

3. **\*\*Embedded Methods\*\***: These methods learn which features best contribute to the accuracy of the model while the model is being created. Examples include LASSO, Ridge Regression, and Decision Trees.

Pros: They are able to detect the interaction between variables and are less prone to overfitting. They are also generally faster than wrapper methods.

Cons: They are more complex to implement and understand. They are also model-dependent.

4. **\*\*Dimensionality Reduction\*\***: These methods reduce the dimension of the feature space. Examples include Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA).

Pros: They are effective when dealing with a large number of input variables, and they can help improve model performance by removing multicollinearity.

Cons: They create new composite features, which may be harder to interpret.

Choosing the right feature selection method depends on the problem, the nature of your data, and the machine learning algorithm you plan to use. It's often a good idea to experiment with different methods and see which one works best for your specific use case.

## 5. k-means clustering.

### a. [E] How would you choose the value of k?

Choosing the right number of clusters ( $k$ ) in k-means clustering is not always straightforward and often depends on the specific data and domain knowledge. However, there are several common methods to help determine the optimal number of clusters:

1. **\*\*Elbow Method\*\***: This is the most common method. Here, we run the k-means algorithm for a range of  $k$  values and plot the sum of squared errors (SSE) for each value of  $k$ . As the number of clusters increases, the SSE decreases. The idea is to choose the  $k$  at which the SSE decreases

abruptly, which produces an "elbow effect" in the graph. This point is considered as the optimal number of clusters.

2. **Silhouette Analysis**: This method measures how close each point in one cluster is to the points in the neighboring clusters. The silhouette score ranges from -1 to 1. A high value indicates that the point is well-matched to its own cluster and poorly matched to neighboring clusters. If most points have a high value, then the clustering configuration is appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters.

3. **Gap Statistic**: The Gap statistic compares the total intracluster variation for different values of  $k$  with their expected values under null reference distribution of the data. The value of  $k$  that yields the largest Gap statistic is considered the optimal number of clusters.

4. **Domain Knowledge**: Sometimes, the choice of  $k$  might be driven by business requirements or domain knowledge. For example, in a customer segmentation problem, the business might have a specific number of segments in mind.

Remember, the optimal number of clusters can vary depending on the method used and the specific dataset. It's often a good idea to try multiple methods and compare the results.

b. **[E] If the labels are known, how would you evaluate the performance of your k-means clustering algorithm?**

If the true labels of the data are known, there are several metrics that can be used to evaluate the performance of a k-means clustering algorithm. These are often referred to as "external" evaluation metrics, as they rely on information external to the algorithm itself. Here are a few of the most common ones:

1. **Adjusted Rand Index (ARI)**: This measure compares how pairs of data points relate to each other in the predicted and true clusters. The ARI score ranges from -1 to 1, where 1 means the predicted and

true clusters are identical, 0 means the clustering is random, and negative values mean the clustering is worse than random.

2. **Normalized Mutual Information (NMI)**: This metric measures the amount of information obtained about one variable through the other variable. It is normalized to be between 0 and 1, where 1 means the predicted and true clusters are identical, and 0 means the clustering is no better than random.
3. **Fowlkes-Mallows Index (FMI)**: This index measures the geometric mean of precision and recall, which are common metrics in classification tasks. The FMI ranges from 0 to 1, where 1 means the predicted and true clusters are identical, and 0 means the clustering is no better than random.
4. **Jaccard Index**: This measure compares the size of the intersection of predicted and true clusters to the size of their union. The Jaccard index ranges from 0 to 1, where 1 means the predicted and true clusters are identical, and 0 means there is no overlap between the clusters.

Remember, these metrics are useful when the true labels are known, which is not always the case in unsupervised learning tasks like clustering. When the true labels are not known, other "internal" evaluation metrics, like the silhouette score or the within-cluster sum of squares, can be used instead.

c. **[M] How would you do it if the labels aren't known?**

K-means clustering is actually an unsupervised learning algorithm, which means it is designed to work when labels aren't known. It's used to automatically group similar data points together and discover underlying patterns.

Here's a high-level overview of how it works:

1. **Initialization**: First, we randomly initialize 'k' points in the feature space, these are our initial centroids.
2. **Assignment**: Each data point is assigned to the nearest centroid, and they form 'k' clusters.

3. **Update**: The centroids of the newly formed clusters are calculated. This is usually done by taking the mean of all data points in the cluster.
4. **Repeat**: Steps 2 and 3 are repeated until the centroids do not change significantly, or a maximum number of iterations is reached.

In Python, using the scikit-learn library, the code would look something like this:

```
from sklearn.cluster import KMeans

# Assume we have a dataset 'X'
kmeans = KMeans(n_clusters=3, random_state=0).fit(X)

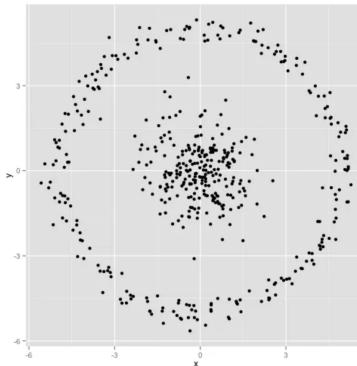
# Get cluster labels for each data point
labels = kmeans.labels_

# Get the coordinates of the cluster centers
centroids = kmeans.cluster_centers_
```

The challenge with k-means is choosing the right number of clusters 'k'. This is usually done using methods like the Elbow method, Silhouette analysis, or the Gap statistic method.

Remember, k-means works best when the data is spherical (or circular in 2D) and clusters are of similar size. If the clusters have complicated geometries or are of very different sizes, other clustering methods might be more appropriate.

- d. **[H] Given the following dataset, can you predict how K-means clustering works on it? Explain.**



Given the image, which appears to depict data points arranged in a circular or annular distribution, K-means clustering may not be the ideal algorithm for this dataset. K-means is likely to struggle with this kind of data structure for several reasons:

1. **Cluster Shape:** K-means assumes that clusters are globular; hence it tends to work well when clusters are roughly spherical. In this case, the data points form a ring shape, which is not globular.
2. **Cluster Centroid:** In K-means, each cluster is represented by a centroid, which is the mean of all points within the cluster. For a ring-shaped distribution, the centroid of the points would be at the center of the ring, which does not correspond to the "center" of any actual data points.
3. **Cluster Assignment:** K-means assigns points to clusters based on the nearest centroid. In this case, points on opposite sides of the ring would be equidistant from the center and could be incorrectly assigned to the same cluster.

If you apply K-means to this dataset, it would likely try to partition the ring into pie-slice-shaped sections or incorrectly place centroids outside the distribution, leading to a poor clustering outcome.

Alternative clustering algorithms like DBSCAN (Density-Based Spatial Clustering of Applications with Noise) or Spectral Clustering could be more suitable here. DBSCAN, for example, would identify the

ring as a single cluster based on point density, while Spectral Clustering would use the eigenvectors of a similarity matrix to find a clustering that respects the ring's shape.

In conclusion, for this specific dataset, K-means is not an appropriate choice due to its inherent assumptions about cluster geometry which do not align with the annular shape of the data.

## 6. k-nearest neighbor classification.

- a. [E] How would you choose the value of k?
- b. [E] What happens when you increase or decrease the value of k?

The k in k-nearest neighbors (k-NN) refers to the number of nearest neighbors to consider when making a prediction. Changing the value of k can have a significant impact on the model's performance.

1. **Decreasing k**: When you decrease the value of k, the model becomes more sensitive to the local structure of the data. This can lead to a more complex decision boundary, which may fit the training data better. However, it also increases the risk of overfitting, where the model performs well on the training data but poorly on unseen data. For example, if k=1, the model will simply assign the class of the nearest neighbor to a new instance, which can lead to high variance if your data is noisy.

2. **Increasing k**: When you increase the value of k, the model becomes more robust to outliers and noise in the data, as it considers a larger neighborhood of points. This can lead to a smoother decision boundary, which may not fit the training data as well but can generalize better to unseen data. However, if k is too large, the model may start to include points from other classes, leading to underfitting, where the model is too simple to capture the underlying structure of the data.

In practice, selecting the best value for k often involves a trade-off between bias and variance, and is typically done through cross-validation. This involves training and evaluating the model with different values of k, and selecting the value that gives the best performance on a validation set.

c. **[M] How does the value of k impact the bias and variance?**

The k-nearest neighbors (k-NN) algorithm is a type of instance-based learning where the output is classified based on the majority of the k nearest points. The value of k plays a significant role in determining the bias and variance of the model.

Bias is the simplifying assumptions made by the model to make the target function easier to approximate. Variance is the amount that the estimate of the target function will change given different training data.

1. When k is small: The model becomes more complex, leading to lower bias but higher variance. This is because the prediction in a small neighborhood is very sensitive to noise in the data. A small value of k means that noise will have a higher influence on the result, and a larger value of k reduces the noise effect. So, with a lower value of k, the decision boundary is more irregular, leading to overfitting.

2. When k is large: The model becomes less complex, leading to higher bias but lower variance. This is because as k increases, the model becomes more resistant to noise but also less sensitive to the underlying patterns in the data. So, with a larger value of k, the decision boundary becomes smoother, leading to underfitting.

In practice, selecting the optimal k for your k-NN model involves balancing bias and variance to minimize the total error. This is usually done through cross-validation: trying out different values of k and choosing the one that performs best on a validation set.

For example, in a 10-fold cross-validation, you'd divide your data into 10 'folds', or subsets. Then, you'd fit the model 10 times, each time using 9 folds for training and 1 fold for validation. You'd average the results, and the value of k that gives the lowest validation error is the one to choose. This process helps to ensure that the model generalizes well to unseen data.

7. **k-means and GMM are both powerful clustering algorithms.**

a. **[M] Compare the two.**

Sure, both k-means and Gaussian Mixture Models (GMM) are indeed powerful clustering algorithms, but they have different strengths and weaknesses and are suited to different types of data.

1. **Model Structure**: K-means is a special case of GMM, in a way that it assumes that each cluster's covariance along all dimensions approaches 0. This implies that a point will get assigned only to the cluster closest to it. On the other hand, GMM takes into account the level of certainty with which a point belongs to a given cluster. This uncertainty is also revised during each iteration making GMM a much more flexible model in terms of cluster covariance compared to k-means.
2. **Speed and Scalability**: K-means is faster and more scalable. GMM tends to be slower due to using information about the data distribution — e.g., probabilities of points belonging to clusters. This makes GMM much more computationally intensive than k-means.
3. **Cluster Assignment**: In k-means, data points are hard-assigned to one cluster or another. In GMM, data points are not assigned directly to a cluster, but rather data points are assigned to different clusters with different probabilities (soft assignment).
4. **Cluster Shape**: K-means assumes that clusters are spherical and all clusters have the same variance, which is not always the case in real-world data. GMM, on the other hand, can accommodate ellipsoidal clusters, as it incorporates information about the data covariance structure.
5. **Number of Clusters**: Both methods require the user to specify the number of clusters, but GMM provides a formal statistical method (like BIC or AIC) to estimate the best number of clusters, while k-means does not.

In summary, if you have computational resources and the data distribution is not simple (e.g., clusters of different sizes, different densities, non-spherical shapes), GMM can be a better choice. If you need to scale to a large number of samples or need speed, k-means would be the way to go.

b. [M] When would you choose one over another?

Hint: Here's an example of how K-means and GMM algorithms perform on the artificial mouse dataset.

Both k-means and Gaussian Mixture Models (GMM) are indeed powerful clustering algorithms, each with their strengths and weaknesses. The choice between them depends on the nature of the problem and the data at hand.

K-means is a simple and fast algorithm, which makes it a good choice for large datasets. It works best when the clusters are spherical and roughly of the same size. However, k-means assumes that clusters are convex and isotropic, which means they are not influenced by the covariance of the data. Therefore, it may not perform well when clusters are of different shapes or densities.

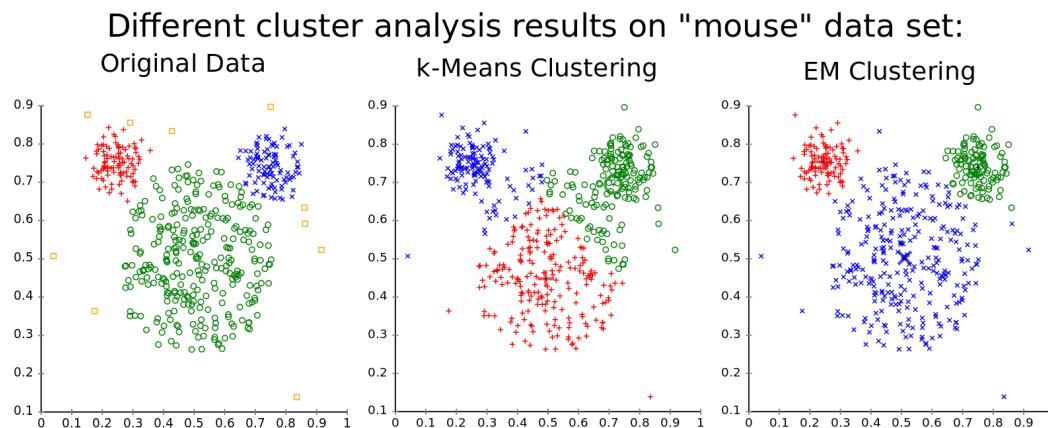
On the other hand, GMM is a more flexible approach as it incorporates information about the covariance structure of the data as well as the centers of the latent Gaussians. It allows for mixed membership of points to clusters. In other words, each point belongs to each cluster to a different degree. This degree is based on the probability of the point being generated from each cluster's (multivariate) Gaussian distribution, with cluster center as the distribution's mean and cluster covariance as its covariance. Because of this, GMM can accommodate clusters that are elongated or have other non-spherical shapes.

However, GMM tends to be slower due to its use of expectation-maximization, which can be computationally intensive, especially for large datasets. It also assumes that the data is normally distributed, which may not always be the case.

In summary, if speed and simplicity are important, or if you are dealing with large, spherical clusters, k-means might be the better choice. If the shape of the clusters is a critical factor, or if the data is normally distributed, GMM might be more suitable. It's also worth noting that these are not the only

clustering algorithms available, and other methods like DBSCAN or hierarchical clustering might be more appropriate depending on the specific problem and data.

Image from [Mohamad Ghassany's course on Machine Learning](#)



8. Bagging and boosting are two popular ensembling methods. Random forest is a bagging example while XGBoost is a boosting example.

a. [M] What are some of the fundamental differences between bagging and boosting algorithms?

Bagging and boosting are indeed two popular ensemble learning methods, each with its own unique characteristics and use cases.

1. \*\*Sampling Method\*\*: Bagging and boosting use different methods for sampling the training data. Bagging, which stands for Bootstrap Aggregating, involves creating multiple subsets of the original dataset, selecting samples at random with replacement. Each of these subsets is then used to train a separate model. In contrast, boosting builds models in a sequential manner where each new model is trained to correct the mistakes made by the previous models. The data sampling in boosting is

weighted, meaning instances that were incorrectly predicted by previous models are given more weight so they have a higher chance of being included in the training set for the next model.

2. **Model Building**: In bagging, all models are trained independently and in parallel, meaning the order in which they are built does not matter. However, in boosting, models are built sequentially, with each new model being influenced by the performance of those built before it.
3. **Prediction Aggregation**: In bagging, the final prediction is typically an unweighted vote of all models (for classification) or an average (for regression). In boosting, the final prediction is a weighted sum that takes into account the performance of each individual model.
4. **Bias-Variance Tradeoff**: Bagging aims to decrease the model's variance without increasing the bias. This makes it a good choice when dealing with complex models that tend to overfit. On the other hand, boosting can reduce both bias and variance, which makes it powerful but also more prone to overfitting if not carefully tuned.
5. **Example Algorithms**: As you mentioned, Random Forest is a classic example of a bagging algorithm, while Gradient Boosting and XGBoost are examples of boosting algorithms.

In practice, the choice between bagging and boosting would depend on the dataset, the specific problem at hand, and the performance of the individual models.

b. **[M] How are they used in deep learning?**

Bagging and boosting are indeed popular ensemble methods in machine learning, but they are typically used with decision tree-based models, like Random Forests and XGBoost, rather than deep learning models.

However, there are ways to use the concepts of bagging and boosting in the context of deep learning.

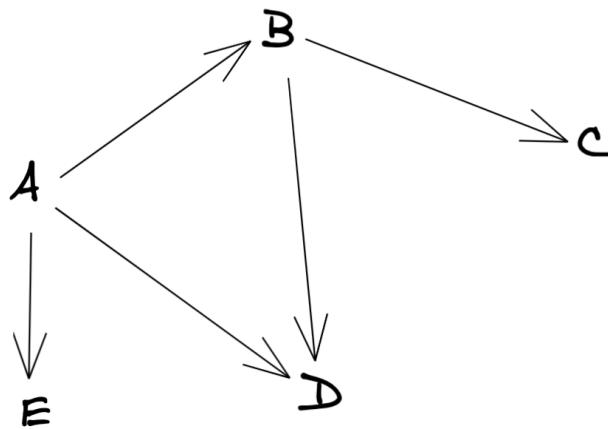
Bagging, or bootstrap aggregating, is a technique that reduces variance and helps to avoid overfitting. In deep learning, one way to apply bagging is by training multiple neural networks independently and then averaging their predictions. This is similar to how Random Forests operate, where multiple decision trees are trained on different subsets of the data and their predictions are averaged. However, due to the computational cost of training deep learning models, this is not a common practice.

Boosting, on the other hand, is a technique that reduces bias and helps to improve the predictive power of the model. It's a sequential process where each model tries to correct the mistakes of the previous one. In the context of deep learning, boosting is not typically used because deep learning models, like neural networks, are usually powerful enough to capture complex patterns in the data.

However, there are some research papers and methods that try to incorporate boosting-like techniques into deep learning. For example, the concept of "hard example mining" in object detection is somewhat similar to boosting, where the model focuses more on the examples it finds difficult to classify.

In conclusion, while bagging and boosting are not commonly used in deep learning in the same way they are in decision tree-based models, the underlying concepts of reducing variance and bias, and focusing on hard-to-classify examples, are certainly applicable and can be used to improve the performance of deep learning models.

9. Given this directed graph.



1. **[E] Construct its adjacency matrix.**

For a directed graph, the adjacency matrix is a square matrix used to represent which vertices (or nodes) of the graph are adjacent to which other vertices. Each cell  $(i, j)$  in the matrix is filled with a 1 if there is an edge from vertex  $i$  to vertex  $j$ , and a 0 otherwise.

Given the directed graph with vertices A, B, C, D, E, the adjacency matrix would be:

	A	B	C	D	E
A	0	1	0	1	1
B	0	0	1	1	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

2. **[E] How would this matrix change if the graph is now undirected?**

For an undirected graph, each edge is bidirectional. This means if there's an edge between vertex A and vertex B, it would be represented in the matrix as a 1 in both cells (A, B) and (B, A). The adjacency matrix of an undirected graph is always symmetric about the diagonal, because if vertex A is adjacent to vertex B, then vertex B must also be adjacent to vertex A.

The matrix would change as follows, to reflect the bidirectional nature of the edges:

A	B	C	D	E	
A	0	1	0	1	1
B	1	0	1	1	0
C	0	1	0	0	0
D	1	1	0	0	0
E	1	0	0	0	0

### 3. [M] What can you say about the adjacency matrices of two isomorphic graphs?

Two graphs are isomorphic if there is a one-to-one mapping between their vertices that preserves the adjacency relationship. The adjacency matrices of two isomorphic graphs will be the same up to permutation of rows and columns. In other words, if one graph's adjacency matrix can be converted into the other's by swapping rows and the corresponding columns (since the matrix must remain symmetric), the two graphs are isomorphic. This reflects the fact that isomorphic graphs have the same structure, but the labels of their nodes might be different or they might be rotated/translated versions of each other in a graphical representation.

However, it is important to note that having identical adjacency matrices does not guarantee isomorphism, because different labelings of the same graph can lead to different matrices. The

converse is true: if two graphs are isomorphic, their adjacency matrices will be identical up to permutation of rows and columns. This is a necessary but not sufficient condition for graph isomorphism.

10. Imagine we build a user-item collaborative filtering system to recommend to each user items similar to the items they've bought before.

- a. [M] You can build either a user-item matrix or an item-item matrix. What are the pros and cons of each approach?

Sure, let's discuss both approaches.

#### 1. User-Item Matrix:

Pros:

- This approach is straightforward and intuitive. It involves creating a matrix where the rows represent users and the columns represent items. The entries in the matrix are the ratings that the users have given to the items.
- It works well when there are a lot of users in the system. The more users there are, the more likely it is that a user will have rated an item that another user has also rated, allowing the system to find similarities between users.

Cons:

- It can be computationally expensive, especially when there are a lot of users and items. The matrix can become very large and sparse, which can make it difficult to find similarities between users.
- It can suffer from the cold start problem. If a new user or item enters the system, the system may not have enough information to make accurate recommendations.

## 2. Item-Item Matrix:

Pros:

- This approach involves creating a matrix where the rows and columns represent items, and the entries in the matrix are the similarities between items. It can be more efficient than the user-item approach, especially when there are fewer items than users.
- It can handle the cold start problem better than the user-item approach. If a new user enters the system, the system can still make recommendations based on the similarities between items.

Cons:

- It can be less intuitive than the user-item approach. It's not always clear how to calculate the similarity between items, especially when the items are very different from each other.
- It can be computationally expensive, especially when there are a lot of items. The matrix can become very large and sparse, which can make it difficult to find similarities between items.

In practice, the choice between these two approaches often depends on the specific characteristics of the data and the computational resources available. For example, if there are a lot more users than items, the item-item approach might be more efficient. On the other hand, if the items are very diverse and it's difficult to calculate similarities between them, the user-item approach might be more appropriate.

### b. [E] How would you handle a new user who hasn't made any purchases in the past?

This is a common problem in recommendation systems known as the "cold start" problem. There are a few strategies we can use to handle new users who haven't made any purchases in the past:

1. **Popularity-based recommendations**: For a new user, we can start by recommending the most popular items. These are items that have been bought or rated highly by a large number of users. This

approach is simple and doesn't require any information about the user.

2. **Demographic-based recommendations**: If we have demographic information about the user, such as age, gender, or location, we can recommend items that are popular among similar demographic groups.
3. **Content-based recommendations**: If the user hasn't made any purchases but has browsed or clicked on certain items, we can use this information to recommend similar items based on content features. For example, if a user has clicked on several science fiction books, we can recommend other books in the science fiction genre.
4. **Hybrid approach**: We can also use a combination of the above strategies. For example, we can start with popularity-based recommendations, then gradually incorporate demographic and content-based recommendations as we learn more about the user.
5. **Ask the user**: Another approach is to simply ask the user about their preferences when they first join. This could be in the form of a short survey or by asking the user to rate a few sample items.

Remember, the goal is to gather as much information as possible about the new user's preferences so we can start providing personalized recommendations.

#### 11. **[E] Is feature scaling necessary for kernel methods?**

Yes, feature scaling is generally necessary for kernel methods, particularly in machine learning algorithms that use distance measures such as Support Vector Machines (SVM) or k-nearest neighbors (KNN).

Kernel methods transform the data into a higher dimensional space where it becomes linearly separable. If features are on different scales, the feature with a larger scale can dominate when calculating distances, leading to inaccurate results.

For example, consider a dataset with two features: age (ranging from 0 to 100) and income (ranging from 0 to 1,000,000). Without feature scaling, the 'income' feature would dominate the 'age' feature simply because its values are larger, which could lead to incorrect classifications or predictions.

By scaling features, we ensure that no particular feature dominates others due to its scale, and each feature contributes approximately proportionately to the final distance. This can lead to better and more accurate models.

Common methods of feature scaling include normalization (scaling features to be between 0 and 1) and standardization (scaling features to have a mean of 0 and a standard deviation of 1)

## 12. Naive Bayes classifier.

### a. [E] How is Naive Bayes classifier naive?

The Naive Bayes classifier is considered "naive" because it makes a very strong assumption that the features in a dataset are mutually independent of each other. This means that the presence or absence of a particular feature does not affect the presence or absence of any other feature.

For example, if we are using a Naive Bayes classifier to predict whether an email is spam or not, and our features are the words in the email, the "naive" assumption is that each word in the email occurs independently of the others. This is obviously a simplification, because in real language, words are often dependent on each other. For instance, the word "free" might be more likely to appear in an email if the word "offer" is also present.

Despite this simplification, Naive Bayes classifiers often perform surprisingly well in practice and are particularly popular in text classification problems. They are fast and easy to implement, and their performance can be quite competitive with more sophisticated methods.

### b. [M] Let's try to construct a Naive Bayes classifier to classify whether a tweet has a positive or negative sentiment. We have four training samples:

Tweet	Label
I love this sandwich	Positive
This is an amazing place!	Positive
I feel very good about these beers.	Positive
I don't like this restaurant.	Negative

The Naive Bayes classifier works on the principle of Bayes' theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

To construct a Naive Bayes classifier, we would first need to preprocess the text data. This involves steps like tokenization (splitting the text into individual words), removing stop words (like 'is', 'an', 'the'), and stemming (reducing words to their root form).

Next, we would create a frequency table of the words in each class (positive and negative). This table would be used to calculate the likelihood of each word given a class.

For example, the likelihood of the word "love" given the positive class would be calculated as the number of times "love" appears in positive tweets divided by the total number of words in positive tweets.

The prior probability of each class would be calculated as the number of tweets in that class divided by the total number of tweets.

Finally, to classify a new tweet, we would calculate the posterior probability of each class given the words in the tweet, and assign the tweet to the class with the highest probability.

Here's a simplified Python code snippet for this process using the sklearn library:

```
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.naive_bayes import MultinomialNB

# Training data

tweets = ["I love this sandwich.", "This is an amazing place!", "I feel very

labels = ["positive", "positive", "positive", "negative"]

# Preprocessing and vectorization

vectorizer = CountVectorizer(stop_words='english')

X_train = vectorizer.fit_transform(tweets)

# Training the Naive Bayes classifier

clf = MultinomialNB()

clf.fit(X_train, labels)

# Classifying a new tweet

new_tweet = ["I love this place!"]
```

```
X_test = vectorizer.transform(new_tweet)

prediction = clf.predict(X_test)

print(prediction) # Outputs: ['positive']
```

This is a simplified example and in a real-world scenario, you would need a much larger training dataset to get accurate results. Also, you might want to use more sophisticated techniques for preprocessing and feature extraction, like TF-IDF or word embeddings.

13. Two popular algorithms for winning Kaggle solutions are Light GBM and XGBoost. They are both gradient boosting algorithms.

a. [E] What is gradient boosting?

Gradient boosting is a machine learning technique that's used for both regression and classification problems. It works by building a model in a stage-wise fashion, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

In simple terms, gradient boosting involves three elements:

1. A loss function to be optimized: The loss function depends on the type of problem you're trying to solve. For example, regression problems may use a different loss function than classification problems. The key is that the loss function must be differentiable.
2. A weak learner to make predictions: Gradient boosting uses decision trees as the weak learner. These are constructed in a greedy manner, meaning that the best split point is chosen by pure local optimization.

3. An additive model to add weak learners to minimize the loss function: Trees are added one at a time, and existing trees in the model are not changed. The gradient descent procedure is used to minimize the loss when adding trees.

So, in essence, gradient boosting involves building many simple models (like decision trees), then combining them in a way that makes the overall model stronger. The aim is to reduce bias, and this is done by sequentially adding predictors to an ensemble, each one correcting its predecessor.

b. **[M] What problems is gradient boosting good for?**

Gradient boosting is a powerful machine learning technique that is particularly good at solving both regression and classification problems, but it can also be adapted for ranking and anomaly detection tasks. Here are some specific areas where gradient boosting shows its strength:

1. **Predictive Modeling**: Gradient boosting is excellent for predictive modeling tasks where the goal is to predict a numeric or categorical outcome. This includes problems such as predicting house prices, customer churn, or disease diagnosis.
2. **Large Datasets**: Gradient boosting algorithms like LightGBM and XGBoost are designed to handle large datasets efficiently. They use a form of gradient boosting called "gradient boosting decision tree" (GBDT), which can handle both numerical and categorical features directly and can naturally handle missing data.
3. **High Dimensionality**: Gradient boosting can handle datasets with high dimensionality. It builds trees one at a time, where each new tree helps to correct errors made by previously trained trees. This makes it less susceptible to overfitting in high dimensional spaces compared to other algorithms.
4. **Feature Importance**: Gradient boosting provides feature importance scores, which can be very useful for interpretability in machine learning. This allows you to understand which features are the

most influential in making the prediction, which can be critical in certain domains like healthcare or finance.

5. **Anomaly Detection**: Gradient boosting can be used for anomaly detection where data is often highly unbalanced such as DNA sequences, credit card transactions, or cybersecurity.

6. **Non-linear Relationships**: Gradient boosting can model complex non-linear relationships. Each weak learner (typically a decision tree) is built on the residuals/errors of the previous one, gradually reducing the error and improving the model's prediction.

However, it's important to note that while gradient boosting is a powerful tool, it's not a silver bullet for all problems. It can be sensitive to noisy data and outliers, and can overfit if not properly regularized or if the data is too sparse. Also, training a gradient boosting model can be computationally intensive and time-consuming, especially for larger datasets.

#### 14. SVM.

##### a. [E] What's linear separation? Why is it desirable when we use SVM?

Linear separation in the context of Support Vector Machines (SVM) refers to the ability to separate data points of different classes using a straight line (in two dimensions) or a hyperplane (in higher dimensions). In a binary classification problem, for example, if we can draw a straight line such that all points of one class are on one side and all points of the other class are on the other side, we say that the data is linearly separable.

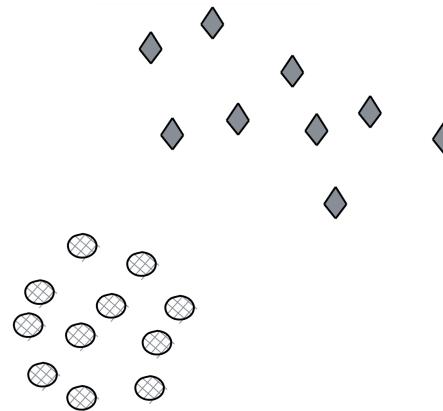
Linear separation is desirable in SVM for a couple of reasons:

1. **Simplicity**: Linear models are easier to understand and interpret. They require less computational resources and are faster to train than non-linear models.

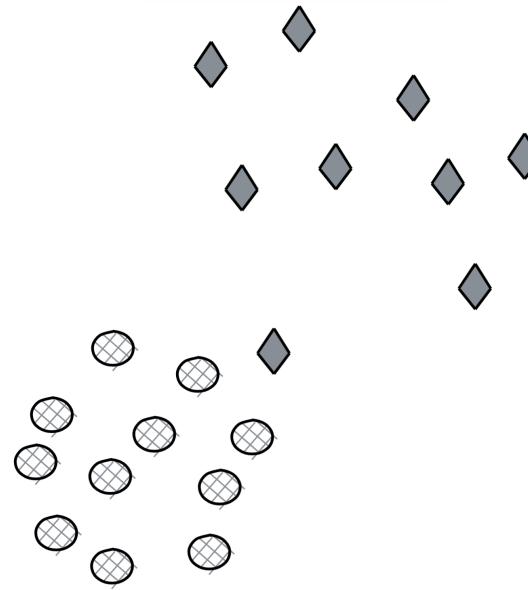
2. Overfitting: Linear models are less prone to overfitting. Overfitting happens when a model learns the noise along with the underlying pattern in the data. It performs well on the training data but poorly on unseen data. By using a linear model, we can often achieve a good balance between bias (underfitting) and variance (overfitting).

However, it's important to note that not all data is linearly separable. In such cases, SVMs can use something called the kernel trick to transform the data into a higher dimension where it is linearly separable. This allows SVMs to handle non-linear data as well.

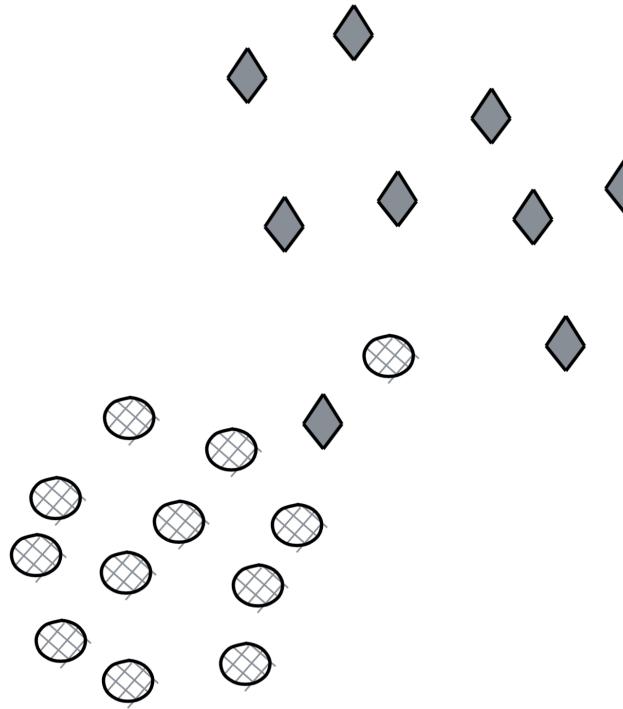
b. [M] How well would vanilla SVM work on this dataset?



c. [M] How well would vanilla SVM work on this dataset?



d. [M] How well would vanilla SVM work on this dataset?



### How to answer above questions:

A vanilla SVM (Support Vector Machine) refers to a standard SVM classifier with a linear kernel. The SVM algorithm is fundamentally a linear classifier that looks for the hyperplane that best separates different classes by maximizing the margin between the closest points of the classes (support vectors).

If the dataset is like the one with a circular or non-linear pattern, a linear kernel would not be able to find a hyperplane that correctly classifies the data points into separate categories.

In such cases, one might employ a kernel trick to transform the input space into a higher dimension where the separation between the classes is linear. The most common kernels used in SVM are:

- Polynomial kernel: Suitable for non-linearly separable data that can be separated by polynomial decision boundaries.
- Radial Basis Function (RBF) or Gaussian kernel: Very useful for non-linearly separable data, as it can handle cases where the relationship between class labels and attributes is not merely polynomial.

If the dataset consists of two clear clusters that are not linearly separable in the given feature space, using a vanilla SVM would not yield good results. Instead, an SVM with an appropriate non-linear kernel such as RBF would be a better choice.

To determine how well an SVM or any classifier would work on a given dataset, one would typically split the dataset into training and testing sets, train the classifier on the training set, and then evaluate its performance on the testing set using metrics such as accuracy, precision, recall, F1 score, etc. For non-linearly separable data, the performance metrics of a vanilla SVM would likely show poor results due to misclassification.

## ▼ Natural Language Processing

### 1. RNNs

#### a. [E] What's the motivation for RNN?

Recurrent Neural Networks (RNNs) are primarily motivated by the need to process sequential data. Traditional neural networks, such as feed-forward networks, are not capable of handling such data because they assume that inputs are independent of each other. However, in many real-world scenarios, this is not the case.

For instance, consider the task of language modeling, where we need to predict the next word in a sentence. The choice of the next word heavily depends on the words that came before it. Similarly, in

time-series prediction tasks, like predicting stock prices, the future value is highly dependent on the past values.

RNNs are designed to remember past information and use it in the current computation, which makes them ideal for such tasks. They have internal loops that allow information to be passed from one step in the sequence to the next, thereby providing a form of memory. This ability to use their internal state to process sequences of inputs makes RNNs extremely useful for tasks in natural language processing, speech recognition, and many other areas.

b. **[E] What's the motivation for LSTM?**

The motivation for Long Short-Term Memory (LSTM) networks comes from the limitations of traditional Recurrent Neural Networks (RNNs).

RNNs are designed to use their historical information to inform their predictions. This makes them great for tasks where the past information is important, such as time series prediction or natural language processing. However, RNNs suffer from a problem known as the vanishing gradient problem, where the contribution of information decays geometrically over time. This makes it difficult for RNNs to carry forward information from the distant past to the present.

LSTMs were developed to address this issue. They have a different structure that allows them to carry forward important information. They do this through a series of gates that control the flow of information. This structure allows them to decide what information to keep (or remember) and what information to discard.

This makes LSTMs much better at handling long-term dependencies in data, which is particularly useful in tasks like language modeling, where the meaning of a word can depend on the words that appeared much earlier in the sentence.

c. **[M] How would you do dropouts in an RNN?**

Dropout is a regularization technique used in neural networks to prevent overfitting. It works by randomly setting a fraction of input units to 0 at each update during training time. However, applying dropout in Recurrent Neural Networks (RNNs) is not straightforward due to the temporal connections.

In a traditional feed-forward neural network, dropout can be applied at each layer independently. But in an RNN, if we apply dropout in the standard way at each time step independently, it would be equivalent to injecting noise at each time step. This can disrupt the RNN's ability to learn from sequences as the dropped-out features would be different at each time step, making it difficult for the model to learn temporal dependencies.

A more effective way to apply dropout in RNNs is to use the same dropout mask (the same pattern of dropped out units) at each time step, also known as recurrent dropout. This allows the model to correctly propagate its learning error through time since the dropped-out units remain consistent across time steps. This method was proposed by Yarin Gal in his PhD thesis and has been adopted in various deep learning libraries.

For example, in Keras, you can specify dropout in an LSTM layer (a type of RNN layer) with the `dropout` and `recurrent\_dropout` parameters. The `dropout` parameter controls the amount of input dropout and `recurrent\_dropout` controls the recurrent dropout.

```
from keras.layers import LSTM  
  
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
```

In this example, 20% of the input units and 20% of the recurrent units would be set to 0 during training. The same dropout mask would be used at each time step for the recurrent units.

It's important to note that while dropout helps in reducing overfitting, it also makes the network more complex and thus requires more epochs to train. So, it's a trade-off that needs to be considered based on the specific use case.

## 2. [E] What's density estimation? Why do we say a language model is a density estimator?

Density estimation is a process in statistics where we construct an estimate of the probability density function of a random variable. In simpler terms, it's about finding the underlying distribution of the data.

A language model can be considered a density estimator because it estimates the probabilities of different sequences of words. For example, given a sequence of words, a language model can predict the probability of the next word or even the next sequence of words. This is essentially estimating the density of word sequences in the language.

Let's take a simple example. Suppose we have a language model trained on English text. If we give it the sequence "I am going to the", it might predict that the next word is likely to be "store" or "park", because those are common phrases in English. It's less likely to predict something like "pineapple", because "I am going to the pineapple" is not a common phrase in English. This is an example of how a language model estimates the density (or probability distribution) of word sequences.

## 3. [M] Language models are often referred to as unsupervised learning, but some say its mechanism isn't that different from supervised learning. What are your thoughts?

Language models are indeed often categorized as unsupervised learning because they don't require labeled data in the traditional sense. They learn to predict the next word in a sentence given the previous words, which is a form of unsupervised learning because the "labels" are just the subsequent words in the text itself.

However, the training process of language models does resemble supervised learning in some ways. For instance, during training, we do have a target output for each input - the next word in the sequence. This is

similar to supervised learning where we have a known target output (label) for each input during training.

In the case of language models, we're essentially supervising the learning process by providing the correct next word in the sequence. The model is then penalized based on how far its prediction is from the actual word, similar to how a supervised learning model would be penalized based on how far its prediction is from the actual label.

So, while language models are technically unsupervised because they learn from unlabeled data, the training process does bear similarities to supervised learning. It's a bit of a gray area and can be seen as a blend of both supervised and unsupervised learning paradigms.

#### 4. Word embeddings.

##### a. [M] Why do we need word embeddings?

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. They are a distributed representation for text that is perhaps one of the key breakthroughs for the impressive performance of deep learning methods on challenging natural language processing problems.

We need word embeddings for several reasons:

1. **\*\*Dimensionality Reduction\*\***: One of the main advantages of using word embeddings is that they help in reducing the dimensionality of the data. Traditional Bag-of-Words (BoW) models represent data in high-dimensional space, which can lead to issues like the curse of dimensionality. Word embeddings, on the other hand, represent words in a continuous vector space, thereby reducing the dimensionality of the data.
2. **\*\*Capturing Context and Semantics\*\***: Word embeddings are trained to capture the context in which words appear. This means that words that are used in similar contexts will have similar embeddings.

This is a powerful feature, as it allows the model to understand synonyms (words with similar meanings), and even analogies between words.

3. **\*\*Efficiency\*\***: By converting words into vectors, we can easily perform operations on them and use them as input for machine learning models. This is much more efficient than using one-hot encoded vectors or other types of representation.

4. **\*\*Handling Out of Vocabulary (OOV) words\*\***: Word embeddings provide a way to handle OOV words. If a word is not present in the training corpus, a BoW model would not be able to create a vector for it. However, with word embeddings, we can create a vector for the OOV word based on the context in which it is used.

In summary, word embeddings are a powerful tool for natural language processing tasks. They allow us to reduce the dimensionality of our data, capture the context and semantics of words, and handle OOV words efficiently.

b. **[M] What's the difference between count-based and prediction-based word embeddings?**

Count-based and prediction-based word embeddings are two different methods used to create word embeddings, which are vector representations of words that capture their meanings.

Count-based word embeddings, such as TF-IDF and Latent Semantic Analysis (LSA), are based on the idea of the "Distributional Hypothesis" in linguistics, which states that words that occur in the same contexts tend to have similar meanings. These methods create word embeddings by counting how often each word appears in the context of other words in large amounts of text. For example, in the TF-IDF method, the importance of a word in a document is measured by multiplying its frequency in the document (Term Frequency) by the inverse of its frequency in a set of documents (Inverse Document Frequency).

On the other hand, prediction-based word embeddings, such as Word2Vec and GloVe, create word embeddings by training a predictive model to guess a word given its context (or vice versa). For example, the Word2Vec model uses a shallow neural network to predict a target word given its context words (Continuous Bag of Words, CBOW) or to predict context words given a target word (Skip-Gram).

The main difference between the two methods is that count-based methods are based on counting word co-occurrences in a corpus, while prediction-based methods are based on predicting words from their contexts using machine learning models.

In terms of performance, prediction-based methods often outperform count-based methods on tasks that require capturing complex patterns and semantic relationships between words, because they can capture both the local context (nearby words) and the global context (the entire document or a large window of words). However, count-based methods can be more efficient and easier to implement, and they can perform well on tasks that require capturing simple patterns and syntactic relationships between words.

- c. **[H] Most word embedding algorithms are based on the assumption that words that appear in similar contexts have similar meanings. What are some of the problems with context-based word embeddings?**

Context-based word embeddings, such as Word2Vec, GloVe, or FastText, have revolutionized the field of natural language processing by providing a way to capture semantic relationships between words in a high-dimensional space. However, they do have some limitations:

1. **Polysemy**: A significant issue is the handling of words with multiple meanings. For example, the word "bank" can refer to a financial institution or the side of a river. However, in most word embedding models, each word is represented by a single vector, which cannot capture these multiple meanings effectively.

2. **Out-of-vocabulary words**: These models struggle with words that are not in their training corpus. For instance, if a model is trained on a corpus that does not include the word "blockchain", it will not be able to generate a meaningful vector for this word.
3. **Lack of interpretability**: Although word embeddings can capture semantic and syntactic relationships, the dimensions of the embeddings themselves do not have an interpretable meaning. This can make it difficult to understand why the model behaves the way it does.
4. **Contextual independence**: Traditional word embeddings are context-independent, meaning the representation for a word is the same regardless of its context. This can be problematic for words that have different meanings in different contexts.
5. **Bias**: Word embeddings can inadvertently capture and perpetuate biases present in the training data. For example, if the training data contains gender biases, the resulting embeddings may also reflect these biases.

To overcome some of these limitations, newer models like BERT (Bidirectional Encoder Representations from Transformers) and ELMo (Embeddings from Language Models) generate context-dependent embeddings, which can capture the meaning of a word in a specific context. These models can handle polysemy and provide more accurate representations for words based on their usage. However, they are computationally more expensive and still face challenges with out-of-vocabulary words and bias.

## 5. Given 5 documents:

```
D1: The duck loves to eat the worm
D2: The bird loves to eat the worm
D3: The early bird gets the worm
```

D4: The early duck gets the worm

D5: The bird and the duck love the early worm

1. [M] Given a query Q: "The early bird gets the worm", find the two top-ranked documents according to the TF/IDF rank using the cosine similarity measure and the term set {bird, duck, worm, early, get, love}. Are the top-ranked documents relevant to the query?

To answer this question, we first need to understand the concept of TF/IDF and cosine similarity.

TF/IDF stands for Term Frequency-Inverse Document Frequency. It's a numerical statistic used to reflect how important a word is to a document in a collection or corpus. The TF/IDF value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space. It is defined to equal the cosine of the angle between them, which is also the same as the inner product of the same vectors normalized to both have length 1.

We would first calculate the TF/IDF values for each term in each document. Then, we would calculate the cosine similarity between the query and each document.

However, without the actual frequency of each term in each document and the overall corpus, we can't calculate the exact TF/IDF values or cosine similarities. But we can make some educated guesses based on the information given.

The query "The early bird gets the worm" is most similar to D3 and D4 because they contain all the terms in the query. Therefore, they would likely have the highest cosine similarity scores.

As for relevance, D3 is clearly relevant because it's identical to the query. D4 could be considered relevant because it contains the same structure and most of the same terms as the query, but it replaces "bird" with "duck". Depending on the context, this could either be seen as a minor difference (if "bird" and "duck" are considered similar) or a major difference (if "bird" and "duck" are considered very different).

In conclusion, without more information, it's impossible to calculate the exact TF/IDF values or cosine similarities. However, based on the information given, D3 and D4 would likely be the top-ranked documents, and they would likely be considered relevant to the query.

2. **[M] Assume that document D5 goes on to tell more about the duck and the bird and mentions "bird" three times, instead of just once. What happens to the rank of D5? Is this change in the ranking of D5 a desirable property of TF/IDF? Why?**

The TF-IDF (Term Frequency-Inverse Document Frequency) algorithm ranks documents based on the importance of words in them. The importance is determined by how frequently a word appears in a document (Term Frequency) and how rare the word is across all documents (Inverse Document Frequency).

If document D5 goes on to mention "bird" three times instead of just once, the term frequency of "bird" in D5 increases. This would increase the TF-IDF score for "bird" in D5, thus potentially increasing the rank of D5 when querying for "bird".

However, whether this change in ranking is desirable or not depends on the context. If the query is about "birds", then this is a desirable property because D5 becomes more relevant due to the increased frequency of the term "bird". But if the query is about "ducks" or "worms", this change might not be as desirable because the frequency of "bird" has no relevance to these queries.

The TF-IDF algorithm is based on the assumption that words that appear frequently in a document and less frequently in other documents are important, which might not always be true. For instance, if "bird" is a common term in all documents, its increased frequency in D5 might not necessarily make D5 more relevant. This is one of the limitations of TF-IDF and why other techniques like semantic analysis or word embeddings might be used in conjunction with or instead of TF-IDF for more complex tasks.

6. **[E] Your client wants you to train a language model on their dataset but their dataset is very small with only about 10,000 tokens. Would you use an n-gram or a neural language model?**

Given the small size of the dataset, I would recommend using an n-gram model.

Neural language models, such as recurrent neural networks (RNNs) or transformers, typically require large amounts of data to effectively learn the underlying patterns. With only 10,000 tokens, a neural model might overfit the data and perform poorly on unseen data.

On the other hand, n-gram models, which predict the next word in a sequence based on the previous 'n' words, can work quite well with smaller datasets. They are simpler and faster to train than neural models. However, they do have limitations, such as the inability to capture long-range dependencies between words and the curse of dimensionality when 'n' gets large.

So, for this specific case with a small dataset, an n-gram model would be a more suitable choice.

7. **[E] For n-gram language models, does increasing the context length (n) improve the model's performance? Why or why not?**

Yes, increasing the context length (n) in n-gram language models can improve the model's performance, but only up to a certain point. The context length (n) refers to the number of previous words taken into account when predicting the next word in a sentence.

For example, in a bigram model (n=2), the model considers one previous word. In a trigram model (n=3), it considers two previous words, and so on.

Increasing the context length can improve the model's performance because it allows the model to capture more information about the structure of the language and the dependencies between words. For instance, the meaning of a word can often depend on several words that came before it, not just the immediate previous word.

However, there are two main issues with increasing n:

1. Data sparsity: As n increases, the number of possible n-grams grows exponentially, and many of them might not appear in the training data. This can lead to a lot of zero probabilities in the model, which can negatively impact its performance.
2. Computational complexity: Larger n values increase the computational complexity of the model, as it needs to store and process more n-grams.

So, while increasing n can improve the model's performance, it's important to find a balance that maximizes performance while minimizing these issues. Typically, n values of 3 (trigrams) or 4 (4-grams) are used in practice.

8. **[M] What problems might we encounter when using softmax as the last layer for word-level language models? How do we fix it?**

When using softmax as the last layer for word-level language models, we might encounter two main problems:

1. **\*\*Computational Complexity\*\***: In a large vocabulary, the softmax layer needs to compute the probability distribution over all possible words. This can be computationally expensive and slow down the training process significantly.
2. **\*\*Sparse Data Problem\*\***: In natural language processing, the distribution of words is highly skewed, with some words appearing very frequently and many words appearing very rarely. This can lead to a problem where the model overfits to the frequent words and performs poorly on the rare words.

To address these problems, we can use the following techniques:

1. **Hierarchical Softmax**: Instead of computing the softmax over all words in the vocabulary, hierarchical softmax uses a binary tree representation of the output layer, where each leaf node of the tree corresponds to one word in the vocabulary. This reduces the computational complexity from linear to logarithmic with respect to the size of the vocabulary.
2. **Negative Sampling**: This technique simplifies the problem by transforming it into a binary classification problem. For each training step, instead of updating all the word vectors, it updates only a small percentage of them. This is done by selecting one true pair (center word and a word in its context) and several noise pairs (the center word paired with random words) and then training the model to distinguish the true pair from the noise pairs.
3. **Subsampling Frequent Words**: This technique reduces the number of high-frequency words in the training data. Each word in the training set is discarded with a probability that is related to its frequency. This can help to mitigate the problem of the model overfitting to the frequent words.
4. **Smoothing Techniques**: Techniques like Laplace smoothing or Add-k smoothing can be used to handle the problem of rare words. These techniques assign a small but non-zero probability to the words that do not appear in the training data.

By using these techniques, we can mitigate the problems associated with using softmax as the last layer for word-level language models.

#### 9. [E] What's the Levenshtein distance of the two words "doctor" and "bottle"?

The Levenshtein distance, also known as the edit distance, is a measure of the difference between two strings. It is defined as the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other.

Let's calculate the Levenshtein distance between "doctor" and "bottle".

1. "doctor" -> "boctor" (substitute 'd' with 'b', cost = 1)
2. "boctor" -> "botor" (substitute 'c' with 't', cost = 1)
3. "botor" -> "bottle" (delete 'o', cost = 1)
4. "bottle" -> "bottle" (insert 't', cost = 1)

So, the Levenshtein distance between "doctor" and "bottle" is 4.

10. **[M] BLEU is a popular metric for machine translation. What are the pros and cons of BLEU?**

BLEU, or Bilingual Evaluation Understudy, is a score for comparing a candidate translation of text to one or more reference translations. It's a very popular metric in the field of machine translation due to its simplicity and effectiveness. However, like any metric, it has its pros and cons.

Pros:

1. **\*\*Simplicity and Efficiency\*\***: BLEU is relatively simple to understand and implement. It's based on the concept of n-gram precision, which is a straightforward and computationally efficient measure.
2. **\*\*Correlation with Human Judgment\*\***: Studies have shown that BLEU scores often correlate well with human judgment of translation quality. This makes it a useful tool for evaluating and comparing different machine translation models.
3. **\*\*Scalability\*\***: BLEU can be used to evaluate translations of any length, from single sentences to entire books. This makes it a versatile tool for a wide range of machine translation tasks.

Cons:

1. **\*\*Lack of Semantic Understanding\*\***: BLEU focuses on surface-level text matching and doesn't account for meaning or semantics. Two sentences can express the same idea using different words, and a human

would consider both translations accurate, but BLEU would penalize the one that doesn't match the reference at the word level.

2. **\*\*Inadequate Handling of Morphologically Rich Languages\*\***: BLEU may not perform well with languages that have complex morphology, as it relies on exact word matching. This can lead to lower scores for accurate translations that use different but valid forms of a word.
3. **\*\*Sensitivity to Reference Translation Quality\*\***: BLEU scores are heavily dependent on the quality of the reference translations. If the reference translations are not of high quality, the BLEU scores may not accurately reflect the quality of the candidate translations.
4. **\*\*Inability to Evaluate Single Sentences Reliably\*\***: While BLEU can technically evaluate single sentences, it was designed for and is most reliable when evaluating larger documents. This is because it uses statistical methods that are more accurate with larger sample sizes.

Despite these limitations, BLEU remains a widely used metric in machine translation due to its simplicity and general effectiveness. However, it's important to use it in conjunction with other evaluation methods and human judgment to ensure a comprehensive assessment of translation quality.

11. **[H] On the same test set, LM model A has a character-level entropy of 2 while LM model A has a word-level entropy of 6. Which model would you choose to deploy?**

It seems there might be a slight confusion in your question as you mentioned "LM model A" twice. I'll assume you meant "LM model A has a character-level entropy of 2 while LM model B has a word-level entropy of 6."

Choosing between these two models would depend on the specific application and the level at which you want your language model to operate.

Entropy in the context of language models is a measure of uncertainty or randomness. Lower entropy means the model is more certain about its predictions. So, a character-level language model (LM) with an

entropy of 2 is more certain about its next character predictions compared to a word-level LM with an entropy of 6 for its next word predictions.

If your application involves tasks like text generation, spelling correction, or other tasks that require fine-grained control at the character level, then LM model A with a character-level entropy of 2 would be a better choice.

On the other hand, if your application involves tasks like text summarization, machine translation, or other tasks that operate at the word level, then despite its higher entropy, LM model B might still be a better choice because it operates at the word level.

It's also worth noting that comparing character-level entropy to word-level entropy directly is not entirely apples-to-apples, because the entropy for a character-level model is calculated based on a much smaller set of possible next steps (the character set) compared to a word-level model (the vocabulary of the language).

So, in conclusion, the choice between these two models would largely depend on the specific requirements of your application.

12. **[M] Imagine you have to train a NER model on the text corpus A. Would you make A case-sensitive or case-insensitive?**

The decision to make the text corpus case-sensitive or case-insensitive when training a Named Entity Recognition (NER) model depends on the specific requirements of the project and the nature of the data.

In general, case sensitivity can provide valuable information. For example, in English, proper nouns such as names of people, places, organizations, etc., usually start with a capital letter. This can be a useful feature for an NER model to identify named entities. So, in this case, you might want to keep the text case-sensitive.

However, there are also situations where case-insensitivity might be beneficial. For example, if your corpus contains a lot of informal text (like social media posts or text messages) where users might not strictly follow capitalization rules, then making the text case-insensitive could help the model generalize better and avoid treating the same word in different cases as different words.

In practice, you might want to experiment with both approaches and see which one gives better results on a validation set. You could also consider using a hybrid approach, where you keep both the original case-sensitive text and a case-insensitive version, and let the model learn to use the information that's most useful for the task.

In conclusion, there's no one-size-fits-all answer to this question, and the best approach can vary depending on the specific task and data.

### 13. **[M] Why does removing stop words sometimes hurt a sentiment analysis model?**

Stop words are commonly occurring words in a language like 'the', 'is', 'in', 'for', 'where', 'when', etc. In many natural language processing tasks, these words are often removed because they occur so frequently that they may not carry a lot of meaningful information that can distinguish different texts.

However, in sentiment analysis, removing stop words can sometimes hurt the performance of the model. This is because stop words can carry sentiment and context, especially in relation to other words in the sentence. For example, consider the phrases "not good" and "not bad". The word "not" is a stop word, but it's crucial in determining the sentiment of these phrases. If we remove "not", we lose the intended sentiment completely.

Another example is the word "very", which is also considered a stop word. But it can intensify the sentiment of a phrase, as in "good" versus "very good".

So, while stop words often don't carry much meaning on their own, they can significantly contribute to the sentiment of a phrase or sentence when combined with other words. Therefore, blindly removing all stop

words for sentiment analysis tasks might not be the best strategy and could potentially degrade the performance of the model. It's important to carefully consider the role of stop words in the specific task at hand.

14. **[M] Many models use relative position embedding instead of absolute position embedding. Why is that?**

Relative position embedding is often preferred over absolute position embedding in many models due to its ability to capture the relationship between elements based on their relative positions rather than their absolute positions.

In many tasks, such as natural language processing, the relationship between elements (like words in a sentence) often depends more on their relative positions than their absolute positions. For example, in the sentence "I am going to the market", the meaning of "market" is more closely related to "the" (which is immediately before it) than "I" (which is far before it), even though "I" is at an earlier absolute position in the sentence.

Relative position embedding allows the model to capture these kinds of relationships more effectively. It encodes the position of an element relative to other elements, which can provide more meaningful and context-specific information to the model.

In contrast, absolute position embedding encodes the exact position of each element in the sequence. While this can be useful in some cases, it may not capture the contextual relationships between elements as effectively as relative position embedding.

Moreover, relative position embeddings are more flexible and generalizable. They can handle sequences of varying lengths and can be applied to unseen positions during training, which is not always the case with absolute position embeddings. This makes them more suitable for tasks with dynamic input lengths, like text processing or time series analysis.

In summary, while both relative and absolute position embeddings have their uses, the choice between them depends on the specific task and the nature of the data. For tasks where the relationship between elements is more dependent on their relative positions, relative position embedding is usually the better choice.

15. **[H] Some NLP models use the same weights for both the embedding layer and the layer just before softmax. What's the purpose of this?**

This technique is known as "weight tying" or "shared embedding and softmax weights". The main purpose of this approach is to reduce the number of parameters in the model, which can help to mitigate overfitting and improve generalization.

In a typical NLP model, the embedding layer and the softmax layer are the two largest layers in terms of the number of parameters. By sharing weights between these two layers, we can significantly reduce the total number of parameters in the model.

This technique is based on the observation that these two layers are performing similar tasks. Both layers are mapping words (or more generally, tokens) to a high-dimensional space where semantically similar words are close to each other. The embedding layer does this mapping in the forward direction (from words to embeddings), while the softmax layer does it in the reverse direction (from embeddings to words). Therefore, it makes sense to use the same weights for both mappings.

In practice, weight tying can lead to better or comparable performance with fewer parameters. For example, in the Transformer model used in the original "Attention is All You Need" paper, weight tying is used and the model achieves state-of-the-art performance on several benchmark tasks.

Here's a simple example of how you might implement weight tying in PyTorch:

```

class TiedEmbeddingsText(nn.Module):
    def __init__(self, vocab_size, embed_size):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.linear = nn.Linear(embed_size, vocab_size, bias=False)
        self.linear.weight = self.embed.weight # Tie weights

    def forward(self, x):
        embed = self.embed(x)
        output = self.linear(embed)
        return output

```

In this code, the `nn.Embedding` layer and the `nn.Linear` layer share the same weight matrix. The `bias=False` argument in the `nn.Linear` layer is important, because the `nn.Embedding` layer doesn't have a bias term.

## ▼ Computer Vision

1. **[M] For neural networks that work with images like VGG-19, InceptionNet, you often see a visualization of what type of features each filter captures. How are these visualizations created?**

**Hint: check out this Distill post on Feature Visualization.**

Visualizations of what features each filter in a neural network captures are created using a variety of techniques. The purpose of these visualizations is to understand what kind of patterns, textures, or parts of the image are detected by each convolutional filter in the network. Here are some common methods used to create these visualizations:

1. **Feature Map Visualization:** After passing an image through the network, the output of the individual filters at each convolutional layer (feature maps) is visualized. This gives a direct view of what the filters are activating on. These are usually the first layer features that are easy to interpret, as they often detect edges, colors, or textures.
2. **Filter Visualization by Optimization:** To understand what activates a particular filter, an input image is optimized to maximally activate the filter. This is done by starting with a random image and using gradient ascent to change the image so that the filter's response is maximized. The resulting image often highlights the features that the filter is looking for.
3. **DeepDream:** Originally developed by Google, DeepDream modifies an image to amplify the features that are detected by the network. It works similarly to the optimization approach, but instead of starting with a random image, it starts with an existing image and enhances the patterns that the network detects.
4. **Class Activation Maps (CAM):** CAMs are useful for understanding which parts of the image were important for a particular classification decision. They use the global average pooling layers in networks like Inception to generate a heatmap over the input image, highlighting the important regions for predicting a class.
5. **Guided Backpropagation:** This method aims to visualize the gradients that flow back to the input image, which can be interpreted as the importance of each pixel for a given filter. This is often used to see which parts of an input image contribute most to the activation of a particular filter.
6. **Layer-wise Relevance Propagation (LRP):** LRP redistributes the prediction of a neural network back through its layers to the input layer, providing a heatmap of relevance that shows which pixels contribute to the final decision.

**7. t-SNE and PCA Visualization:** Although not about individual filters, t-SNE (t-distributed Stochastic Neighbor Embedding) and PCA (Principal Component Analysis) are used to visualize high-dimensional data in lower dimensions. When applied to the activations of a neural network layer, they can show clustering of features and how the network separates different classes.

To implement some of these techniques, especially optimization-based visualizations, you usually need to access the gradients with respect to the input image, which requires a deep learning framework that supports automatic differentiation like TensorFlow or PyTorch.

Here is a simple example of how you might visualize a filter from the first convolutional layer using TensorFlow and Keras:

```
from tensorflow.keras.applications import VGG19
from tensorflow.keras import Model
import numpy as np
import matplotlib.pyplot as plt

# Load a pre-trained VGG19 model
model = VGG19(weights='imagenet', include_top=False)

# Choose a layer to visualize
layer_name = 'block1_conv1'
layer = model.get_layer(name=layer_name)

# Create a model that will return the output of the layer
activation_model = Model(inputs=model.input, outputs=layer.output)
```

```

# Start with a random image
input_img = np.random.random((1, 224, 224, 3)) * 20 + 128.

# Define the loss as the mean activation of a specific filter
filter_index = 0 # Choose a filter to visualize
layer_output = layer.output
loss = K.mean(layer_output[:, :, :, filter_index])

# Compute the gradient of the input image wrt this loss
grads = K.gradients(loss, model.input)[0]

# Normalization trick: we normalize the gradient
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

# This function returns the loss and grads given the input picture
iterate = K.function([model.input], [loss, grads])

# Run gradient ascent for a number of steps
for i in range(50):
    loss_value, grads_value = iterate([input_img])
    input_img += grads_value * 0.1 # Multiply by step size

# Convert the tensor to a valid image
img = input_img[0]
img -= img.mean()

```

```
img /= (img.std() + 1e-5)
img *= 0.1
img += 0.5
img = np.clip(img, 0, 1)

# Display the image
plt.imshow(img)
```

This code will attempt to generate an image that maximizes the activation of the first filter in the `block1_conv1` layer of the VGG19 network. It uses gradient ascent in the input space, modifying a random image in the direction that increases the filter's activation. The resulting image should highlight the kind of features that the first filter is looking for.

## 2. Filter size.

### a. [M] How are your model's accuracy and computational efficiency affected when you decrease or increase its filter size?

The filter size in a convolutional neural network (CNN) is a critical parameter that can significantly impact both the model's accuracy and computational efficiency.

1. Accuracy: The filter size determines the field of view for the convolution operation. A smaller filter size means the model can capture finer details in the input data, which can be beneficial for tasks that require high granularity. However, it might also lead to overfitting if the model starts learning noise in the data. On the other hand, a larger filter size allows the model to capture more global and abstract features, but it might miss out on some finer details. Therefore, the optimal filter size is often a trade-off and depends on the specific task and data.

2. Computational Efficiency: Smaller filters require less computational resources as fewer parameters need to be learned. This can make the training process faster and more memory-efficient. However, smaller filters might require more layers to capture the same level of abstraction as larger filters, which could negate the computational benefits. Larger filters, while having more parameters and thus requiring more computational resources, can capture more information in fewer layers.

In practice, it's common to start with smaller filters (e.g., 3x3 or 5x5) and increase the filter size if necessary based on the model's performance. It's also important to note that filter size is just one of many hyperparameters that need to be tuned in a CNN, and its optimal value often depends on the other hyperparameters and the specific task at hand.

b. **[E] How do you choose the ideal filter size?**

Choosing the ideal filter size in a convolutional neural network (CNN) is a bit of an art and science. It depends on several factors, including the complexity of the task, the size of the input images, and the computational resources available.

One common practice is to start with small filter sizes such as 3x3 or 5x5. This is because smaller filters can capture local features in the image, and stacking multiple layers of small filters can effectively represent larger receptive fields. For instance, a 3x3 filter applied twice results in a 5x5 receptive field. This approach can reduce the number of parameters in the model and make the network computationally efficient.

However, if the input images are large and contain high-level features that span larger regions, you might need larger filters like 7x7 or 11x11.

It's also important to consider the depth of the network. If you're using a deep network, smaller filters can be beneficial because they allow the network to learn more complex hierarchies of features.

Ultimately, the choice of filter size often comes down to empirical testing. You might need to experiment with different sizes and see which one works best for your specific task. This process is often facilitated by techniques like cross-validation or grid search.

Remember, though, that the choice of filter size is just one aspect of designing a CNN. Other factors, like the number of filters, the stride, and the padding, also play a crucial role in the performance of the network.

3. **[M] Convolutional layers are also known as “locally connected.” Explain what it means.**

In the context of convolutional neural networks (CNNs), the term "locally connected" refers to how each neuron in a convolutional layer is connected to only a small region of the input volume. This region is defined by the size of the receptive field or the filter size.

For instance, if we have an input image of size 32x32x3 (32 pixels wide, 32 pixels high, and 3 color channels), and we use a filter size of 5x5, each neuron in the convolutional layer would be connected to a 5x5x3 region of the input image. This is in contrast to fully connected layers, where each neuron is connected to every neuron in the previous layer.

The concept of local connectivity in CNNs is inspired by the organization of the animal visual cortex, where it was observed that neurons in the visual cortex have a small receptive field and respond to stimuli only in a limited region of the visual field.

This local connectivity property of convolutional layers helps in reducing the number of parameters in the model, making it less prone to overfitting. It also helps in capturing local features in the input, which is particularly useful in image processing tasks where the spatial relationship between pixels is important.

4. **[M] When we use CNNs for text data, what would the number of channels be for the first conv layer?**

In the context of Convolutional Neural Networks (CNNs), channels refer to the depth of an input or output. For image data, the number of channels usually refers to the color channels of the image, such as red, green, and blue (RGB) for a color image, which would have 3 channels.

However, for text data, the concept of channels is a bit different. In most cases, we represent text data as a 2D matrix where each row corresponds to a word or a character, and each column corresponds to a dimension in the word or character's vector representation. These vector representations are often obtained through techniques like one-hot encoding, TF-IDF, or word embeddings like Word2Vec or GloVe.

So, for text data, the number of channels in the first conv layer would typically be 1, because we're dealing with a 2D matrix (words/characters x their vector representations) rather than a 3D tensor like in image data. However, if you have multiple different vector representations for each word or character (for example, one Word2Vec and one GloVe), then each different representation could be considered a separate channel. In this case, the number of channels would be equal to the number of different representations you have for each word or character.

But in most common scenarios, we stick to one type of vector representation, so the number of channels would be 1.

## 5. **[E] What is the role of zero padding?**

Zero padding is a technique used in convolutional neural networks (CNNs) while performing the convolution operation. The main role of zero padding is to preserve the spatial dimensions (height and width) of the input volume so that the output volume is the same size as the input.

Without padding, every time we apply a convolution layer, the size of the input shrinks. For example, if we apply a 3x3 filter to a 32x32 image, the output will be a 30x30 image. If we continue to apply convolution layers, the size of the image will keep shrinking until it's too small to continue. This could cause us to lose some important information at the borders of the image.

By applying zero padding, we add a border of zeros around the input image. This allows the convolutional filter to process the corners and edges of the image properly, preserving the spatial dimensions and ensuring that we don't lose information at the borders.

For instance, if we apply zero padding of 1 pixel around our 32x32 image, the padded image will be 34x34. Now, if we apply a 3x3 filter, the output will be a 32x32 image, the same size as the original input.

So, zero padding plays a crucial role in controlling the dimensionality of the output volumes, preserving spatial information, and allowing deeper layers in the network.

## 6. **[E] Why do we need upsampling? How to do it?**

Upsampling is a technique used in machine learning when we have imbalanced datasets. An imbalanced dataset is one where the classes are not represented equally. For example, in a binary classification problem, if we have 100 samples and 95 samples belong to class 1 and only 5 samples belong to class 2, then our dataset is highly imbalanced. This can lead to a model that is biased towards the majority class, resulting in poor predictive performance for the minority class.

Upsampling is a technique to correct this imbalance by increasing the number of samples in the minority class. There are several ways to do this:

1. **\*\*Random Oversampling\*\***: This is the simplest form of upsampling. It involves randomly duplicating observations from the minority class in order to reinforce its signal.
2. **\*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\***: This is a more sophisticated method of upsampling. It works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line.

Here is a simple example of how to do random oversampling in Python using the `resample` function from `sklearn.utils`:

```
from sklearn.utils import resample

# Separate majority and minority classes
df_majority = df[df.Class==0]
df_minority = df[df.Class==1]

# Upsample minority class
df_minority_upsampled = resample(df_minority,
                                 replace=True,        # sample with replacement
                                 n_samples=100,       # to match majority class
                                 random_state=123)   # reproducible results

# Combine majority class with upsampled minority class
df_upsampled = pd.concat([df_majority, df_minority_upsampled])

# Display new class counts
df_upsampled.Class.value_counts()
```

In this example, we first separate the majority and minority classes. Then, we upsample the minority class to match the number of samples in the majority class. Finally, we combine the upsampled minority class with the majority class to create a new, balanced dataset.

## 7. **[M] What does a 1x1 convolutional layer do?**

A 1x1 convolutional layer, also known as a network-in-network layer, is a convolutional layer with a kernel size of 1x1. While it might seem counterintuitive at first, it's actually a powerful tool in convolutional neural

networks (CNNs).

In a typical convolutional layer, we use filters of size larger than 1x1 (like 3x3 or 5x5) to scan the input image and extract features. These features are then used for tasks like image classification, object detection, etc.

A 1x1 convolutional layer, on the other hand, doesn't look at the neighborhood of pixels. Instead, it looks at one pixel at a time. But remember, in a CNN, each "pixel" is actually a stack of values, one from each feature map in the previous layer. So a 1x1 convolution is actually looking at all the features at a specific x, y location.

There are two main uses for 1x1 convolutions:

1. **Dimensionality Reduction**: By using a 1x1 convolutional layer with fewer filters than the input depth, we can reduce the dimensionality of our input. This is a computationally cheaper way to shrink the depth of our feature maps, compared to using a fully connected layer.
2. **Increasing Non-linearity**: 1x1 convolutions introduce additional non-linearity to the model without affecting the receptive fields of the convolution layers. This can help the model learn more complex patterns.

Here's an example of how a 1x1 convolution might be used in Keras:

```
from keras.models import Sequential
from keras.layers import Conv2D

model = Sequential()
model.add(Conv2D(64, (1,1), activation='relu', input_shape=(224, 224, 3)))
```

In this example, we're applying a 1x1 convolution to an input with a depth of 3 (a color image), and we're outputting 64 feature maps. So each "pixel" in the output is a linear combination of the RGB values at the same location in the input, passed through a ReLU activation function.

## 8. Pooling.

### a. [E] What happens when you use max-pooling instead of average pooling?

Max pooling and average pooling are two different types of pooling operations that you can use in a convolutional neural network. They're both used to reduce the spatial dimensions (width and height) of the input volume for the next convolutional layer.

When you use max pooling, the output is the maximum (largest) value of the input values in the region defined by the filter. This has the effect of preserving the strongest features, since they're likely to activate more strongly than other features. For example, if the feature is an edge in a certain direction, max pooling will preserve the strongest edge.

On the other hand, average pooling calculates the average of the input values in the region defined by the filter. This has the effect of smoothing over the features, which can sometimes help with removing noise in the input.

In practice, max pooling tends to be used more often than average pooling in convolutional neural networks, because it has been found to work better for tasks like image classification. However, the choice between max pooling and average pooling can depend on the specific task and dataset.

### b. [E] When should we use one instead of the other?

Pooling is a technique used in Convolutional Neural Networks (CNNs) to reduce the spatial dimensions (width and height) of the input volume for the next convolutional layer. It helps to reduce overfitting by providing an abstracted form of the representation. There are several types of pooling, including max pooling, average pooling, and global average pooling.

1. Max Pooling: This is the most commonly used pooling technique. It selects the maximum value from the portion of the image covered by the filter. Max pooling effectively retains the most significant features and discards the rest. You should use max pooling when you want to capture the presence of a particular feature in various locations in the input.
2. Average Pooling: This calculates the average value of the portion of the image covered by the filter. It helps to smooth the image, reducing high-frequency noise. You should use average pooling when you want to capture the overall summary of features in the input.
3. Global Average Pooling: This calculates the average value of the entire feature map, reducing it to a single value. It's used to reduce the dimensions drastically, often before the final classification layer. You should use global average pooling when you want to summarize the entire feature map.

The choice between these pooling methods depends on the specific requirements of your task. If you want to preserve the maximum activation of features, max pooling is a good choice. If you want to preserve all features equally by taking an average, then average pooling is more suitable. If you want to drastically reduce dimensions, global average pooling can be used.

c. [E] What happens when pooling is removed completely?

Pooling is a technique used in convolutional neural networks (CNNs) to reduce the spatial dimensions (width and height) of the input volume for the next convolutional layer. It helps to make the model more invariant to small translations and distortions, and reduces the computational complexity by reducing the number of parameters.

If pooling is removed completely from a CNN, several things could happen:

1. **Increased computational load**: The number of parameters that the network needs to learn would increase significantly. This could lead to increased training time and computational load, which might not be feasible with limited resources.

2. \*\*Overfitting\*\*: Without pooling, the network might become too complex and could start to memorize the training data instead of learning to generalize from it. This could lead to overfitting, where the model performs well on the training data but poorly on unseen data.
3. \*\*Loss of spatial invariance\*\*: Pooling provides a form of translation invariance, meaning the network can recognize an object as the same even if its position in the image changes. Without pooling, the network might lose this ability to a certain extent.

However, it's worth noting that the necessity of pooling in all scenarios is a topic of ongoing research. Some recent architectures, like ResNet and DenseNet, have shown good performance even without explicit pooling layers, instead using strided convolutions to reduce spatial dimensions.

d. **[M] What happens if we replace a  $2 \times 2$  max pool layer with a conv layer of stride 2?**

Replacing a  $2 \times 2$  max pooling layer with a convolutional layer of stride 2 would change the way the network processes information.

Max pooling layers are used to reduce the spatial dimensions (width and height) of the input volume for the next convolutional layer. They do this by taking the maximum value of the area they are looking at. This is a form of non-linear down-sampling. Max pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that once a feature has been detected, its exact location isn't as important as its relative location to the other features. This helps to control overfitting.

On the other hand, a convolutional layer with a stride of 2 will also reduce the spatial dimensions, but it does so in a different way. Instead of taking the maximum value in a certain area, it applies a filter to the input and uses a dot product operation to compute the output. This is a more complex operation and can learn more intricate patterns than max pooling.

However, this replacement could have several effects:

1. **Parameter Increase**: Convolutional layers have weights and biases that are learned during training, whereas max pooling layers do not. This means that replacing max pooling with a convolutional layer would increase the number of parameters in the model, potentially making it more complex and harder to train.
2. **Computational Complexity**: Convolution operation is more computationally expensive than max pooling. Thus, the training and inference time might increase.
3. **Loss of Invariance Properties**: Max pooling provides some level of translation invariance, meaning the network can recognize an object as the same even if it appears in different places in the image. A convolutional layer does not inherently provide this property.
4. **Learning Hierarchical Features**: Convolutional layers can learn hierarchical features, which might be beneficial for the performance of the model.

In conclusion, while replacing a max pooling layer with a convolutional layer could potentially increase the learning capacity of the model, it could also make the model more complex, slower to train, and more prone to overfitting. It's a trade-off that needs to be considered carefully.

9. **[M] When we replace a normal convolutional layer with a depthwise separable convolutional layer, the number of parameters can go down. How does this happen? Give an example to illustrate this.**

Depthwise separable convolution is a variant of the standard convolution operation and it's used to reduce the computational cost. It does this by separating the convolution operation into two simpler operations: depthwise convolution and pointwise convolution.

In a standard convolutional layer, if we have an input with `Df` channels, a kernel size of `KxK`, and we want `Nf` output channels (filters), the number of parameters is `Df \* K \* K \* Nf`.

In a depthwise separable convolution, the operation is split into two parts:

1. Depthwise convolution: This applies a single filter per input channel (input depth). If we have `Df` input channels and a kernel size of `KxK`, the number of parameters for this step is `Df \* K \* K`.
2. Pointwise convolution: This is a 1x1 convolution used to create a linear combination of the output of the depthwise layer to get `Nf` output channels. The number of parameters for this step is `Df \* Nf`.

So, the total number of parameters in depthwise separable convolution is `Df \* K \* K + Df \* Nf`, which is significantly less than the standard convolution if `K > 1` and `Nf > 1`.

Let's take an example:

Suppose we have an input with 96 channels, we use a 3x3 kernel, and we want 256 output channels.

For a standard convolutional layer, the number of parameters would be `96 \* 3 \* 3 \* 256 = 663,552`.

For a depthwise separable convolutional layer, the number of parameters would be `96 \* 3 \* 3 + 96 \* 256 = 27,648`.

As you can see, the depthwise separable convolutional layer uses significantly fewer parameters, which means it's computationally more efficient. This is one of the reasons why depthwise separable convolutions are used in mobile models like MobileNet.

10. **[M] Can you use a base model trained on ImageNet (image size 256 x 256) for an object classification task on images of size 320 x 360? How?**

Yes, you can use a base model trained on ImageNet for an object classification task on images of a different size. This is a common practice in transfer learning, where a pre-trained model is used as a starting point for a different task.

The ImageNet-trained model has learned a variety of features from the large ImageNet dataset, and these features can be useful for many different image classification tasks. However, the input size of the images needs to be the same as what the model was originally trained on.

In your case, the original model was trained on 256x256 images, but your images are 320x360. Here's how you can handle this:

1. **\*\*Resize the images\*\***: The simplest approach is to resize your images to 256x256. However, this might result in loss of information if the aspect ratio is not preserved.

```
from keras.preprocessing.image import load_img, img_to_array

def preprocess_image(image_path):
    img = load_img(image_path, target_size=(256, 256))
    img = img_to_array(img)
    return img
```

2. **\*\*Adapt the model\*\***: Another approach is to adapt the model to accept the new input size. This is more complex and might not always be possible, depending on the architecture of the model. For example, fully connected layers expect a fixed input size and cannot be adapted to a different size. However, convolutional layers can handle different input sizes, so if your model only contains convolutional layers (like many modern architectures), you can simply change the input size.

```
from keras.applications import VGG16

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(320, 360, 3))
```

3. **\*\*Use a region proposal network\*\***: If the objects you're trying to classify are smaller and located in different parts of the image, you might use a region proposal network like R-CNN or its faster versions

(Fast R-CNN, Faster R-CNN). These models can handle different input sizes and are designed to identify and classify objects in different parts of the image.

Remember, when using transfer learning, it's also common to fine-tune the model on the new task. This involves unfreezing some or all of the layers of the model and continuing training on the new task. This allows the model to adapt to the specifics of the new task, while still benefiting from the pre-training on ImageNet.

## 11. **[H] How can a fully-connected layer be converted to a convolutional layer?**

Converting a fully-connected layer to a convolutional layer is a concept often used in the context of Convolutional Neural Networks (CNNs). This is particularly useful when we want to apply the network to inputs of varying sizes or when we want to use the network for fully convolutional inference.

A fully connected layer takes all neurons in the previous layer and connects it to every single neuron it has. A convolutional layer, on the other hand, does not fully connect but instead uses a convolution operation on the input layer to produce its output.

The conversion process can be done as follows:

1. **\*\*Reshaping:\*\*** The first step is to reshape the output of the layer preceding the fully connected layer into a 3D volume (height x width x depth). This is because convolutional layers work with 3D volumes of neurons.
2. **\*\*Replacing the Fully Connected layer with a Convolutional layer:\*\*** The fully connected layer can be replaced with a convolutional layer where the filters are of the same size as the reshaped output volume. For example, if the output volume is of size  $7 \times 7 \times 512$  and the fully connected layer has 4096 neurons, we can replace the fully connected layer with a convolutional layer with 4096 filters each of size  $7 \times 7$ .

The reason this works is because a fully connected layer is essentially a convolutional layer where the filter size is equal to the input size. So, when we replace the fully connected layer with a convolutional layer,

we're just using a smaller filter that slides over the input volume, computing dot products, much like in a fully connected layer.

This conversion allows us to apply the network to inputs of different sizes, not just the size it was trained on. This is particularly useful in tasks like object detection and segmentation where the input can be of varying sizes.

Here is a Python code snippet that demonstrates this conversion:

```
# Assume we have a fully connected layer 'fc' in PyTorch
fc = nn.Linear(512*7*7, 4096)

# We can convert it to a convolutional layer like this:
conv = nn.Conv2d(512, 4096, kernel_size=7, stride=1, padding=0)

# Copying the weights from the fully connected layer to the convolutional layer
conv.weight.data.copy_(fc.weight.data.view(4096, 512, 7, 7))
conv.bias.data.copy_(fc.bias.data)
```

In this code, we first define a fully connected layer 'fc' and a convolutional layer 'conv'. We then copy the weights and biases from the fully connected layer to the convolutional layer after reshaping the weights to match the dimensions of the convolutional layer's weights.

## 12. [\[H\] Pros and cons of FFT-based convolution and Winograd-based convolution.](#)

**Hint: Read Fast Algorithms for Convolutional Neural Networks (Andrew Lavin and Scott Gray, 2015)**

Fast Fourier Transform (FFT)-based convolution and Winograd-based convolution are two different methods used to speed up the convolution operation in Convolutional Neural Networks (CNNs). Each

method has its own advantages and disadvantages.

## 1. FFT-based Convolution:

Pros:

- FFT-based convolution can be highly efficient for large input sizes and large filter sizes because it transforms the convolution operation into an element-wise multiplication in the frequency domain, which can be computed quickly.
- It can leverage the computational efficiency of FFT algorithms, which are available in many software libraries and hardware architectures.
- It also allows for parallel computation, which can further speed up the process.

Cons:

- FFT-based convolution can be less efficient for small input sizes and small filter sizes because the overhead of the FFT and inverse FFT operations can outweigh the benefits.
- It requires zero-padding of the input data to match the size of the filter, which can increase memory usage.
- It can introduce numerical precision errors due to the transformation between the spatial domain and the frequency domain.

## 2. Winograd-based Convolution:

Pros:

- Winograd-based convolution can be more efficient for small input sizes and small filter sizes because it reduces the number of multiplications in the convolution operation.
- It does not require zero-padding of the input data, which can save memory.

- It does not involve transformation between the spatial domain and the frequency domain, so it can avoid numerical precision errors.

Cons:

- Winograd-based convolution can be less efficient for large input sizes and large filter sizes because it does not reduce the number of additions in the convolution operation.
- It requires a more complex implementation than FFT-based convolution, which can be a challenge for software developers.
- It may not be as well supported by existing software libraries and hardware architectures as FFT-based convolution.

In practice, a hybrid approach that combines FFT-based convolution and Winograd-based convolution can often achieve the best performance. For example, FFT-based convolution can be used for the initial layers of a CNN where the input size is large, and Winograd-based convolution can be used for the later layers where the input size is small. This approach can take advantage of the strengths of both methods and mitigate their weaknesses.

## ▼ Reinforcement Learning

### 1. **[E] Explain the explore vs exploit tradeoff with examples.**

The explore vs exploit tradeoff is a fundamental concept in reinforcement learning and decision-making scenarios. It's a dilemma about whether to stick with what you know and exploit it, or to try new options and explore them.

Let's consider an example. Suppose you're at a restaurant you visit frequently, and you really like the pasta there. Now, you have two choices:

1. Exploit: You can order the pasta again, exploiting your knowledge that it's a good choice.

2. Explore: You can try a new dish, exploring the menu to potentially find a new favorite.

Choosing the pasta is the "exploit" option. You're using your existing knowledge to make a decision that you're confident will result in a good outcome. However, this may prevent you from discovering a dish you might like even more.

Choosing a new dish is the "explore" option. You're taking a risk to gather new information, which could lead to a better outcome in the future (finding a new favorite dish), but it could also lead to a worse outcome (disliking the new dish).

In machine learning, particularly in reinforcement learning, this tradeoff is crucial. For example, in a multi-armed bandit problem, where a gambler must decide which slot machine (or "arm") to play, the choice is between playing a machine that has paid off well in the past (exploit) or trying a different machine in hopes it might pay off better (explore).

Balancing exploration and exploitation is key to achieving optimal results. Too much exploitation can lead to suboptimal results (missing out on better options), while too much exploration can waste resources on inferior options. The optimal balance often depends on the specific context and goals.

## 2. **[E] How would a finite or infinite horizon affect our algorithms?**

In the context of reinforcement learning, the horizon refers to the number of steps an agent can look ahead in the future. A finite horizon means the agent only considers a certain number of future steps, while an infinite horizon means the agent considers all future steps.

The choice between finite and infinite horizons can significantly affect our algorithms.

1. **Computational Complexity**: Finite horizon models often have lower computational complexity because they only need to consider a limited number of future steps. This can make them faster and more efficient, especially in situations where computational resources are limited.

2. **Optimality**: Infinite horizon models, on the other hand, can potentially lead to more optimal decisions because they consider all future consequences. However, they can be computationally intensive and may require more sophisticated algorithms, such as dynamic programming or Monte Carlo methods.
3. **Discounting**: In infinite horizon problems, it's common to use a discount factor to ensure that the sum of future rewards is finite. This can introduce a bias towards immediate rewards, which may or may not be desirable depending on the specific problem.
4. **Uncertainty**: Finite horizons can be more appropriate when there's a lot of uncertainty about the future. If the environment is changing rapidly, it might not make sense to plan too far ahead.

In summary, the choice between finite and infinite horizons depends on the specific problem and the available resources. It's a trade-off between computational efficiency and optimality.

### 3. **[E] Why do we need the discount term for objective functions?**

The discount term in objective functions, often used in reinforcement learning, is important for a couple of reasons.

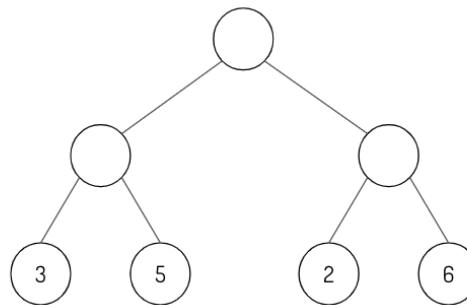
First, it helps to balance immediate and future rewards. In many situations, it's beneficial to take an action that may not have the highest immediate reward, but leads to higher overall rewards in the future. The discount factor allows us to quantify this trade-off. A high discount factor means we value future rewards more, while a low discount factor means we're more focused on immediate rewards.

Second, the discount factor helps to ensure the convergence of the value iteration algorithm in reinforcement learning. Without the discount factor, the sum of rewards could potentially become infinite, making it impossible for the algorithm to find an optimal policy. The discount factor ensures that future rewards contribute less and less, making the sum finite and allowing the algorithm to converge.

For example, consider a robot navigating a maze to find a reward. If the discount factor is high, the robot might take a longer path that it knows will lead to the reward. If the discount factor is low, the robot might

take a shorter path that it hopes will lead to the reward, but it's less certain. The discount factor allows us to control this trade-off between certainty and reward.

4. **[E] Fill in the empty circles using the minimax algorithm.**



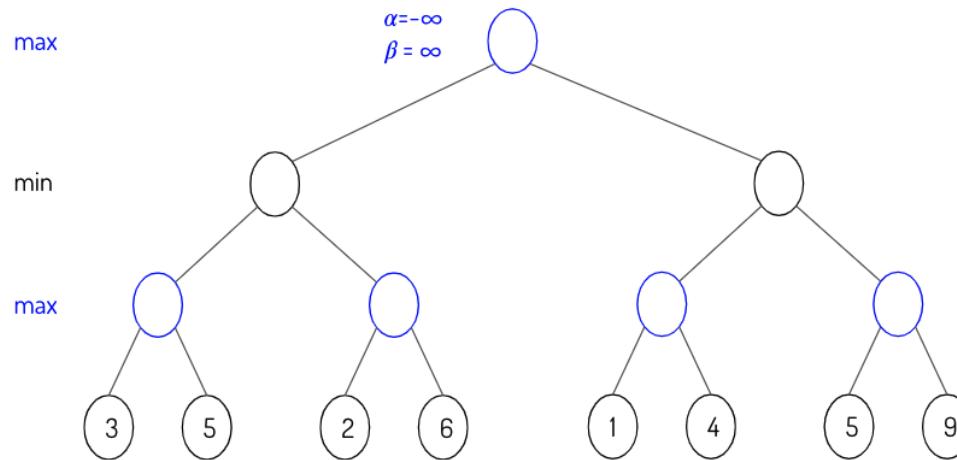
The minimax algorithm is used in decision-making and game theory to minimize the possible loss for a worst-case scenario. When dealing with algorithms that make decisions based on a tree structure, like in many board games, the minimax algorithm is often used.

In the provided tree, the leaves are the end states with their corresponding values. To fill in the empty circles using the minimax algorithm, we'll work from the bottom up:

1. For the left subtree, the algorithm will choose the minimum value of the leaf nodes because the opponent aims to minimize our score. Therefore, it will choose 3 (the smallest value between 3 and 5).
2. For the right subtree, the algorithm will also choose the minimum value of the leaf nodes, hence it will choose 2 (the smallest value between 2 and 6).
3. Finally, for the root node, the algorithm will choose the maximum value between the two subtree results since it's our turn and we want to maximize our score. So, it will choose the greater value between 3 (left subtree result) and 2 (right subtree result), which is 3.

Thus, the empty circles will be filled with 3 for the left middle circle, 2 for the right middle circle, and then the root circle will choose the maximum of its two children, which is 3.

5. **[M] Fill in the alpha and beta values as you traverse the minimax tree from left to right.**



The alpha-beta pruning in the minimax algorithm is a search algorithm that seeks to decrease the number of nodes evaluated in the minimax algorithm. It's used in game theory to make decisions in two-player games. Alpha represents the best value that the maximizer currently can guarantee at that level or above, while beta represents the best value that the minimizer currently can guarantee at that level or above.

Let's say we have a minimax tree and we're traversing it from left to right. We start with alpha as negative infinity and beta as positive infinity because we have no information about the tree yet.

Here's a step-by-step process:

1. The first node we visit is a maximizing node. We traverse its children from left to right. The first child returns a value. We set this value as our new alpha since it's the best value we've found so far.

2. We move to the next child. If this child returns a value greater than our current alpha, we update alpha. If not, alpha stays the same.
3. If at any point, our alpha is greater than or equal to beta, we can stop evaluating the remaining children of the current node (this is the "pruning" part of alpha-beta pruning). The reasoning is that the minimizing parent node will never choose this node because it has found a better (i.e., lower) value already (the beta value).
4. Once we've evaluated all children of the maximizing node, we return the alpha value to the parent node.
5. If the parent node is a minimizing node, it will compare this value with its current beta value. If it's lower, it will update beta. If not, beta stays the same.
6. The process continues as we move through the tree, updating alpha and beta values as we go and pruning branches where possible.

Remember, the goal of the alpha-beta pruning is to ignore parts of the tree that we know will not be selected by the minimax algorithm, thus saving computational resources.

6. **[E] Given a policy, derive the reward function.**

In reinforcement learning, a policy is a strategy that the agent employs to determine the next action based on the current state. The reward function, on the other hand, is a way to measure the success of an action in a given state.

However, it's important to note that typically in reinforcement learning, we don't derive the reward function from the policy. Instead, the reward function is usually a part of the environment and is used to guide the learning of the policy. The reward function provides feedback to the agent about the quality of its actions, and the agent's goal is to learn a policy that maximizes the expected cumulative reward.

In other words, the reward function is not something that is typically derived or learned; it's usually defined based on the problem at hand. For example, in a game of chess, the reward might be +1 for winning, -1 for losing, and 0 for a draw or for each move.

If you're trying to infer a reward function given a policy (a problem known as inverse reinforcement learning), the process is more complex and involves assumptions about the optimality of the observed policy and the structure of the reward function. This is an active area of research in machine learning.

## 7. [M] Pros and cons of on-policy vs. off-policy.

On-policy and off-policy are two types of learning methods in reinforcement learning. They differ in how they generate and use data for learning.

On-policy methods, like SARSA (State-Action-Reward-State-Action), use the current policy to generate data and update the same policy. This means they learn from the actions they are currently taking.

Off-policy methods, like Q-learning, generate data using one policy (behavior policy) but update a different policy (target policy). This means they can learn from actions taken by other policies, not just the current one.

Here are some pros and cons of each:

On-Policy:

**Pros:**

1. On-policy methods are generally simpler to understand and implement.
2. They can be more sample efficient because they are learning directly from the current policy's actions.

**Cons:**

1. They can be slower to converge because they need to explore the environment while also trying to exploit learned knowledge.
2. They can't easily reuse past experiences (no experience replay), as the data collected is tightly coupled with the policy that generated it.

Off-Policy:

**Pros:**

1. Off-policy methods can learn from a wider range of data, not just the current policy's actions. This can lead to faster learning in some cases.
2. They can use experience replay to learn from past experiences, which can improve sample efficiency.

**Cons:**

1. Off-policy methods can be more complex to implement and understand.
2. They can suffer from high variance and instability due to the discrepancy between the behavior policy and the target policy.

In practice, the choice between on-policy and off-policy depends on the specific requirements of your reinforcement learning problem. For instance, if you have a lot of past data that you want to leverage, off-policy might be a better choice. On the other hand, if you want a simpler approach and are okay with potentially slower convergence, on-policy could be the way to go.

8. **[M] What's the difference between model-based and model-free? Which one is more data-efficient?**

Model-based and model-free are two types of reinforcement learning methods.

Model-based reinforcement learning methods build a model of the environment and use it to make decisions. They try to predict what the next state and reward will be given the current state and action.

These predictions are then used to decide on the best action to take. This approach is beneficial because it allows for planning ahead and can be more data-efficient, as it uses the gathered data to build a model of the environment, which can then be used to simulate and learn from many possible scenarios.

On the other hand, model-free reinforcement learning methods directly learn the value function or policy from the interactions with the environment without building a model of the environment. Examples of model-free methods include Q-learning and policy gradients. These methods can be simpler to implement and can work well in complex environments where building an accurate model is difficult.

In terms of data efficiency, model-based methods are generally more data-efficient because they can leverage a model to generate more data. However, they rely heavily on the accuracy of the model, and if the model is not accurate, the performance can suffer. Model-free methods, while less data-efficient, do not have this dependency and can be more robust in situations where the model is hard to learn.

In practice, the choice between model-based and model-free methods depends on the specific problem and the available resources. If the environment is simple enough to model accurately and data is scarce, a model-based method might be the best choice. If the environment is complex and hard to model, or if there is plenty of data available, a model-free method might be more suitable.

#### ▼ Other

1. **[M] An autoencoder is a neural network that learns to copy its input to its output. When would this be useful?**

Autoencoders are indeed a type of neural network that aim to copy their input to their output. However, they are not just simple copy machines - they are designed to learn a compressed representation of the input data, often for the purpose of dimensionality reduction or feature learning. Here are a few scenarios where autoencoders can be particularly useful:

1. **Dimensionality Reduction**: Autoencoders can be used to reduce the dimensionality of input data, similar to techniques like Principal Component Analysis (PCA). However, unlike PCA, autoencoders are capable of learning non-linear transformations with their non-linear activation function and multiple layers. This can provide a more powerful and flexible model for complex data.
2. **Anomaly Detection**: Autoencoders can be used for anomaly detection by training the model on normal data and using it to reconstruct new data. If the reconstruction error for a new piece of data is above a certain threshold, it can be classified as an anomaly. This can be useful in various fields like fraud detection, fault detection in manufacturing, or detecting abnormal behavior in user data.
3. **Denoising Data**: Autoencoders can be used to remove noise from data. The idea is to train the autoencoder on noisy data as input and clean data as output. Once trained, the autoencoder can be used to clean up noisy data, which can be particularly useful in image processing or audio processing.
4. **Feature Learning**: Autoencoders can be used to learn features from data, which can then be used for other machine learning tasks. For example, an autoencoder could be used to learn features from raw pixel data of images, which could then be used as input to a classifier.
5. **Generative Models**: Variational autoencoders (a type of autoencoder with added constraints on the encoded representations) are used for generating new data. They are one of the most popular types of generative models.

In summary, while the basic operation of an autoencoder might seem trivial (copying its input to its output), the way it does this - by learning a compressed representation of the data - makes it a powerful tool for a variety of machine learning tasks.

## 2. **Self-attention.**

### a. **[E] What's the motivation for self-attention?**

Self-attention, also known as attention mechanism, is a concept that was introduced to improve the performance of neural networks in tasks such as machine translation, text summarization, and image recognition. The motivation behind self-attention is to allow the model to focus on the most relevant parts of the input when producing an output.

In traditional neural networks, all parts of the input contribute equally to the final output. However, in many real-world tasks, some parts of the input are more relevant than others. For example, in machine translation, the meaning of a word often depends on a few other words in the sentence, not all of them.

Self-attention allows the model to weigh the importance of each part of the input differently. It does this by creating a weighted sum of the input features, where the weights are determined by the model itself during training. This allows the model to focus more on the relevant parts of the input and less on the irrelevant parts, leading to better performance.

In addition, self-attention also helps in handling long-range dependencies in sequences, which is a common issue in tasks such as text translation or text generation. Traditional recurrent neural networks (RNNs) often struggle with long sequences due to the vanishing gradient problem. Self-attention alleviates this issue by allowing direct dependencies between all pairs of input and output, regardless of their distance in the sequence.

b. **[E] Why would you choose a self-attention architecture over RNNs or CNNs?**

Self-attention, also known as Transformer models, have several advantages over Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), particularly in the field of natural language processing.

1. **Parallelization**: Unlike RNNs, which process sequence data in a serial manner, self-attention models can process all elements in the sequence in parallel. This makes them much faster and more efficient to train on modern hardware, such as GPUs.

2. **Long-range dependencies**: Self-attention models are better at capturing long-range dependencies in sequence data. In RNNs, the ability to capture such dependencies often diminishes with distance due to the vanishing gradient problem. While CNNs can mitigate this to some extent with dilated convolutions, self-attention models can, in theory, attend to any position in the input sequence, regardless of distance.
3. **Interpretability**: Self-attention models generate attention maps, which can provide insights into how the model is making its predictions. This can be very useful for understanding and debugging the model.
4. **Flexibility**: Self-attention is a more flexible operation than convolution or recurrence because it can model various types of structured dependencies, without the need for specifying them explicitly in the architecture.
5. **Less computation for longer sequences**: Self-attention requires less computation than RNNs for longer sequences, making it more efficient for tasks like document classification, text generation, and translation.

However, it's important to note that self-attention models can be more memory-intensive than RNNs or CNNs, and they might not always be the best choice for every problem. The choice of architecture should always be guided by the specific requirements and constraints of the task at hand.

c. **[M] Why would you need multi-headed attention instead of just one head for attention?**

Multi-headed attention is a concept used in the Transformer model, which is a popular model for tasks such as translation, text summarization, and sentiment analysis. The idea behind multi-headed attention is to allow the model to focus on different parts of the input sequence in parallel, capturing various aspects of the information.

Here are a few reasons why we might want to use multi-headed attention instead of just one head:

1. **\*\*Diversity of Attention\*\***: Each attention head can potentially learn to focus on a different type of connection in the data. For example, one head might learn to pay attention to the subject-object relationship in a sentence, while another might focus on the verb-adverb relationship. This allows the model to capture a richer set of dependencies.
2. **\*\*Increased Capacity\*\***: Multi-headed attention increases the capacity of the model without increasing the computational complexity too much. Each head adds a relatively small computational cost, but it can significantly increase the expressive power of the model.
3. **\*\*Improved Performance\*\***: Empirically, it has been observed that multi-headed attention improves the performance of Transformer models. For example, in the original "Attention is All You Need" paper by Vaswani et al., they found that using multiple attention heads improved the translation quality.

Here's a simplified example of how multi-headed attention might work in practice. Suppose we have a sentence "The cat sat on the mat." A single attention head might focus on the relationship between "cat" and "sat", but it might miss the relationship between "sat" and "mat". With multi-headed attention, one head could focus on "cat" and "sat", while another could focus on "sat" and "mat", capturing more of the sentence's structure.

In conclusion, while a single attention head can be useful, using multiple heads allows the model to capture a wider range of information and dependencies in the data, leading to better performance on complex tasks.

d. **[M] How would changing the number of heads in multi-headed attention affect the model's performance?**

Multi-headed attention is a key component of the Transformer model, which is widely used in natural language processing tasks such as machine translation and text summarization. The idea behind multi-

headed attention is to allow the model to focus on different parts of the input sequence for each "head". Each head learns a different attention pattern, which can capture various aspects of the input data.

Changing the number of heads in multi-headed attention can have several effects on the model's performance:

1. **Model Complexity**: Increasing the number of heads increases the complexity of the model. This can lead to better performance if the additional complexity is warranted by the data. However, it can also lead to overfitting if the data does not have enough complexity to justify the additional model complexity.
2. **Computational Cost**: More heads mean more parameters to learn and thus more computational cost. This can slow down training and inference, and require more memory.
3. **Diversity of Representations**: More heads can potentially capture a wider range of information from the input data. For example, in a language translation task, one head might focus on syntactic information, another on semantic information, and another on lexical information. This can lead to better performance if the task benefits from these diverse representations.
4. **Diminishing Returns**: However, there can be diminishing returns to adding more heads. After a certain point, adding more heads might not lead to a significant improvement in performance, and might even hurt performance due to overfitting or increased computational cost.

In practice, the optimal number of heads is a hyperparameter that needs to be tuned for each specific task and dataset. It's also worth noting that the dimensionality of the model (i.e., the size of the hidden layers) is another important factor that interacts with the number of heads. For example, if the model is very high-dimensional, it might be able to capture a wide range of information even with a small number of heads.

### 3. Transfer learning

a. **[E] You want to build a classifier to predict sentiment in tweets but you have very little labeled data (say 1000). What do you do?**

Transfer learning is a great approach when you have limited labeled data. The idea is to leverage the knowledge gained from a related task where abundant data is available, and apply it to the task at hand.

In this case, we can use a pre-trained model that has been trained on a large text corpus, such as the BERT model from Google or GPT-2 from OpenAI. These models have been trained on tasks like language modeling or next sentence prediction, and they have learned a lot about the structure and semantics of the language.

Here's how we can proceed:

1. **Select a Pre-trained Model**: We can start by selecting a pre-trained model that is suitable for our task. For text classification tasks like sentiment analysis, BERT is a good choice.
2. **Fine-tuning**: We then fine-tune this pre-trained model on our specific task. In our case, we would fine-tune the model on our small labeled dataset of tweets for sentiment analysis. Fine-tuning involves training the model on our data, but instead of starting the learning from scratch, the model starts from the learned knowledge of the pre-trained model. This often leads to good results even with small amounts of data.
3. **Evaluation and Adjustment**: Finally, we evaluate the performance of our model on a validation set. If the performance is not satisfactory, we can adjust the model architecture, learning rate, or other hyperparameters, and repeat the fine-tuning.

In Python, using the Hugging Face's Transformers library, the code for fine-tuning a BERT model could look something like this:

```
from transformers import BertForSequenceClassification, AdamW

# Load BERT model for sequence classification, with sentiment labels being
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels = 2,
    output_attentions = False,
    output_hidden_states = False,
)

# Define the AdamW optimizer
optimizer = AdamW(model.parameters(), lr = 2e-5)

# Fine-tune the model on our data
for epoch in range(epochs):
    for batch in train_dataloader:
        model.train()
        optimizer.zero_grad()
        outputs = model(batch['input_ids'], labels=batch['labels'])
        loss = outputs[0]
        loss.backward()
        optimizer.step()
```

This approach allows us to build a sentiment classifier with little labeled data by leveraging the power of transfer learning.

b. **[M] What's gradual unfreezing? How might it help with transfer learning?**

Gradual unfreezing is a technique used in transfer learning where layers of a pre-trained model are "unfrozen" or made trainable in a gradual manner, rather than all at once. This is typically done from the top layers downwards, as the lower layers of a deep learning model usually contain more generic features (like edges or textures in a convolutional neural network for image tasks), while the top layers contain more task-specific features.

The idea behind gradual unfreezing is to allow the model to adjust to the new task in a more controlled manner. If all layers are unfrozen at once, the model might experience a significant shock during the initial stages of fine-tuning, as the gradients can change dramatically. This could potentially lead to catastrophic forgetting, where the model forgets the useful features it had learned from the pre-training task.

By gradually unfreezing the layers, we allow the model to adjust the more task-specific features first, before slowly adjusting the more generic features. This can lead to better performance and faster convergence.

Here's a simple example of how you might implement gradual unfreezing in PyTorch:

```
def unfreeze(model, layer):
    for param in model[layer].parameters():
        param.requires_grad = True

# Assume we have a pre-trained model with 5 layers
model = ...

# Initially, freeze all layers
```

```
for param in model.parameters():
    param.requires_grad = False

# Then gradually unfreeze each layer
for i in range(5):
    unfreeze(model, i)
    # Continue with training here...
```

In this example, we start by freezing all layers of the model. Then, we gradually unfreeze each layer one by one, allowing the model to adjust to the new task in a more controlled manner.

#### 4. Bayesian methods.

##### a. [M] How do Bayesian methods differ from the mainstream deep learning approach?

Bayesian methods and deep learning approaches are both powerful tools in the field of machine learning, but they differ in several key ways.

1. **\*\*Probabilistic Nature\*\***: Bayesian methods are fundamentally probabilistic. They provide a statistical framework for updating beliefs in the face of new data. This is done by calculating the posterior distribution, which combines prior beliefs with the likelihood of the observed data. This probabilistic nature allows Bayesian methods to quantify uncertainty, which can be crucial in many real-world applications. On the other hand, deep learning methods are typically deterministic and do not inherently provide a measure of uncertainty.
2. **\*\*Data Efficiency\*\***: Bayesian methods are generally more data-efficient than deep learning. They can make reasonable inferences with smaller datasets because they incorporate prior knowledge. Deep learning, on the other hand, often requires large amounts of data to train effectively and avoid overfitting.

3. **Interpretability**: Bayesian models are often more interpretable than deep learning models. They provide a full posterior distribution over model parameters, which can be inspected to understand the model's behavior. Deep learning models, on the other hand, are often criticized as being "black boxes" because their internal workings can be difficult to interpret.
4. **Scalability**: Deep learning methods are typically more scalable than Bayesian methods. They can handle large datasets and high-dimensional input spaces more effectively. Bayesian methods, on the other hand, often require computationally expensive procedures like Markov Chain Monte Carlo (MCMC) to compute the posterior distribution, which can be challenging to scale.
5. **Model Complexity**: Deep learning models can capture complex, non-linear relationships in the data due to their multiple layers and non-linear activation functions. Bayesian methods, while they can model complex phenomena, often do so with simpler models, such as linear or generalized linear models.

In summary, while both Bayesian methods and deep learning have their strengths and weaknesses, they are not mutually exclusive. In fact, there's a growing interest in Bayesian deep learning, which aims to combine the strengths of both approaches.

b. **[M] How are the pros and cons of Bayesian neural networks compared to the mainstream neural networks?**

Bayesian Neural Networks (BNNs) are a type of neural network that uses Bayesian methods to estimate the uncertainty in the model's predictions. This is in contrast to traditional, or "mainstream", neural networks, which typically provide point estimates without any measure of uncertainty.

Here are some pros and cons of Bayesian Neural Networks compared to mainstream neural networks:

Pros:

1. **Uncertainty Estimation**: One of the biggest advantages of BNNs is that they provide a measure of uncertainty in their predictions. This can be incredibly useful in many applications, such as healthcare or finance, where understanding the level of confidence in a prediction can be as important as the prediction itself.
2. **Regularization**: BNNs naturally incorporate a form of regularization due to the priors placed on the weights. This can help to prevent overfitting and improve generalization performance.
3. **Data Efficiency**: BNNs can often make good predictions with less data than traditional neural networks, because they are able to incorporate prior knowledge about the weights into the learning process.

Cons:

1. **Computational Complexity**: BNNs are generally more computationally intensive than traditional neural networks. This is because they require sampling from a posterior distribution over the weights, rather than simply optimizing the weights directly.
2. **Difficulty in Training**: BNNs can be more difficult to train than traditional neural networks. The reason is that the optimization problem in BNNs involves integrating over all possible weight configurations, which is a much more complex task than the simple optimization problem in traditional neural networks.
3. **Model Complexity**: BNNs often require more complex models to accurately capture the uncertainty in the data. This can make them more difficult to implement and understand than traditional neural networks.

In conclusion, while Bayesian Neural Networks offer some significant advantages over traditional neural networks, particularly in terms of uncertainty estimation, they also come with some notable drawbacks, especially in terms of computational complexity and difficulty in training. Therefore, the

choice between BNNs and traditional neural networks should be made based on the specific requirements of the task at hand.

c. **[M] Why do we say that Bayesian neural networks are natural ensembles?**

Bayesian neural networks are often referred to as natural ensembles because of the way they handle uncertainty and make predictions.

In a Bayesian neural network, instead of having a single set of weights as in a traditional neural network, we have a distribution over weights. This means that for each prediction, we're effectively using a different model, each with a different set of weights sampled from this distribution.

This is similar to the concept of an ensemble method, where multiple models are trained and their predictions are combined, often leading to better performance than any individual model. In the case of Bayesian neural networks, the ensemble is implicit in the distribution over the weights, hence the term "natural ensemble".

The key difference is that while traditional ensemble methods like bagging or boosting require separate training for each model, a Bayesian neural network learns the distribution over weights in a single training process. This makes Bayesian neural networks more computationally efficient than traditional ensemble methods, while still benefiting from the robustness and improved generalization that comes from ensemble learning.

In terms of real-world application, this makes Bayesian neural networks particularly useful in scenarios where we need to quantify our uncertainty about predictions, such as in medical diagnosis or financial forecasting, where it's important not just to make accurate predictions, but also to understand the level of confidence in those predictions.

5. **GANs.**

a. **[E] What do GANs converge to?**

Generative Adversarial Networks (GANs) ideally converge to a Nash equilibrium of a two-player non-cooperative game. In this game, one player is the generator network, which creates new data instances, and the other player is the discriminator network, which evaluates the authenticity of the generated instances.

The generator network tries to produce data that is as close as possible to the real data, while the discriminator network tries to distinguish between real and fake data. The game continues until a point is reached where the generator produces perfect fakes, and the discriminator can't distinguish between real and fake data. This point is the Nash equilibrium.

However, in practice, reaching this equilibrium can be quite challenging due to various issues like mode collapse, vanishing gradients, and non-convergence. These issues often require careful design of the network architecture, loss functions, and training procedures.

b. **[M] Why are GANs so hard to train?**

Training Generative Adversarial Networks (GANs) can indeed be challenging for several reasons:

1. **\*\*Mode Collapse:\*\*** This is a situation where the generator produces limited varieties of samples, or even the same sample, regardless of the input noise vector. It happens when the generator finds a particular input that can fool the discriminator and starts producing only that output. This leads to a lack of diversity in the generated samples.
2. **\*\*Vanishing Gradients:\*\*** During the training process, if the discriminator becomes too good, it can cause the generator's gradients to vanish, making the generator unable to learn.
3. **\*\*Instability:\*\*** The simultaneous training of two networks (the generator and the discriminator) that are trying to beat each other can lead to instability. The generator is trying to fool the discriminator, and the discriminator is trying to not get fooled. This two-player game can result in oscillations and divergence of the loss function.

4. \*\*Non-convergence:\*\* Unlike many other models, GANs don't always converge. The training process involves finding a Nash equilibrium to a two-player game, which is a more complex problem than simple minimization of a loss function. This complexity can lead to non-convergence issues.
5. \*\*Hyperparameter Sensitivity:\*\* GANs are notoriously sensitive to the choice of hyperparameters. Small changes in the learning rate, type of optimizer, or model architecture can lead to vastly different results.

To mitigate these issues, various strategies have been proposed, such as using different loss functions (Wasserstein loss), different architectures (like DCGAN, WGAN), gradient penalty methods, or techniques like batch normalization and label smoothing. However, despite these improvements, training GANs remains a challenging task that requires careful tuning and experimentation.

## ▼ Training neural networks

### 1. [E] When building a neural network, should you overfit or underfit it first?

When building a neural network, it's a common strategy to initially aim to overfit on the training data. The rationale behind this approach is as follows:

1. **Ensuring Model Capacity:** Overfitting to the training data confirms that the model has enough capacity, complexity, and flexibility to learn the underlying patterns. If a model cannot overfit, it's likely that it's too simple and doesn't have the capability to capture the nuances in the data.
2. **Debugging Model:** If a model can't overfit, it may indicate issues with the model architecture, data preprocessing, learning rate, or other hyperparameters. It's a way to ensure that the data pipeline is set up correctly and the model is able to learn from the data.
3. **Refinement:** Once you have a model that can overfit, you can then take steps to mitigate overfitting, such as adding dropout, L1/L2 regularization, or using techniques like data augmentation. You can also

refine the complexity of the model by reducing the number of layers or units, and use early stopping based on validation loss to find a good balance.

4. **Generalization:** After addressing overfitting, the focus shifts to improving the model's generalization to new data by tuning hyperparameters based on performance on a validation set.

In summary, starting by overfitting ensures that the model is capable of learning, and then you can apply regularization techniques to achieve a model that generalizes well to new, unseen data.

2. [\[E\] Write the vanilla gradient update.](#)

3. [Neural network in simple Numpy.](#)

- a. [\[E\] Write in plain NumPy the forward and backward pass for a two-layer feed-forward neural network with a ReLU layer in between.](#)

The vanilla gradient update, also known as the basic gradient descent update, is performed on the parameters (weights) of the model to minimize the loss function. Here's how it's typically expressed:

$$\theta = \theta - \alpha \cdot \nabla_{\theta} J(\theta)$$

Where:

- $\theta$  represents the parameters of the model.
- $\alpha$  is the learning rate, a hyperparameter that determines the step size at each iteration while moving toward a minimum of the loss function.
- $\nabla$  is the gradient of the loss function  $J(\theta)$  with respect to the parameters  $\theta$ .

b. [M] Implement vanilla dropout for the forward and backward pass in NumPy.

Below is a basic implementation of dropout for a single layer in NumPy, with separate functions for the forward and backward pass:

```
import numpy as np

def forward_pass_with_dropout(X, dropout_rate):
    """
        Apply dropout to the input layer X during the forward pass.

    :param X: Input data for the layer, numpy array of shape (n_features, r
    :param dropout_rate: The probability of setting a neuron's output to 0
    :return: A tuple (output after applying dropout, dropout mask used)
    """

    # Create a mask using the dropout rate, setting to 0 for the dropped un
    dropout_mask = np.random.rand(*X.shape) > dropout_rate

    # Apply the mask to the input data
    dropped_out_X = np.multiply(X, dropout_mask)

    # During training, we'll scale data to not change the expected value
    dropped_out_X /= (1 - dropout_rate)

    return dropped_out_X, dropout_mask
```

```
def backward_pass_with_dropout(dA, dropout_mask, dropout_rate):
    """
    Apply the stored mask to the gradient during the backward pass.

    :param dA: Gradient of the loss with respect to the activations, numpy
    :param dropout_mask: The dropout mask that was used during the forward
    :param dropout_rate: The probability of setting a neuron's output to 0
    :return: Gradient after applying dropout mask
    """

    # Apply the dropout mask to the gradients
    dA_with_dropout = np.multiply(dA, dropout_mask)

    # Scale the gradients as we did during the forward pass
    dA_with_dropout /= (1 - dropout_rate)

    return dA_with_dropout

# Example usage:
np.random.seed(0) # for reproducibility
X = np.random.randn(5, 3) # 5 features, 3 samples
dropout_rate = 0.2 # 20% dropout rate

# Forward pass
X_dropped, mask = forward_pass_with_dropout(X, dropout_rate)
```

```

# Suppose we have some gradient dA from the backward pass
dA = np.random.randn(5, 3)

# Backward pass
dA_dropped = backward_pass_with_dropout(dA, mask, dropout_rate)

```

This code snippet demonstrates how to implement dropout in a neural network's forward and backward propagation. Note that the dropout mask is created during the forward pass and then applied to the input `x` as well as to the gradient `dA` during the backward pass. It's important to scale the activations and gradients by `1 / (1 - dropout_rate)` to account for the reduced number of units during training, which ensures that the expected value of the outputs remains the same.

#### 4. Activation functions.

- a. [E] Draw the graphs for sigmoid, tanh, ReLU, and leaky ReLU.
- b. [E] Pros and cons of each activation function.

##### **Sigmoid Function:**

- Pros:
  - Smooth gradient, preventing "jumps" in output values.
  - Output values bound between 0 and 1, normalizing the output of each neuron.
  - Historically popular since they have a clear probabilistic interpretation.
- Cons:
  - Prone to vanishing gradient problem, which slows down the training and convergence.

- Outputs are not zero-centered, which can lead to the gradients being all positive or all negative, and this can lead to inefficient gradient updates (zig-zagging).
- Saturates and kills gradients when the input is very large or small.

### **Hyperbolic Tangent (tanh) Function:**

- Pros:
  - Zero-centered, leading to more efficient gradient propagation.
  - Scaled version of the sigmoid function, and like sigmoid, has a smooth gradient.
- Cons:
  - Still prone to the vanishing gradient problem, although to a lesser extent than the sigmoid function.

### **Rectified Linear Unit (ReLU) Function:**

- Pros:
  - Computationally efficient, allowing the network to converge faster.
  - Does not saturate in the positive region, reducing the likelihood of the vanishing gradient problem.
  - Provides a sparse representation which can be advantageous.
- Cons:
  - Not zero-centered output.
  - Can suffer from the "dying ReLU" problem, where neurons can become inactive and never activate again because the gradient at 0 is 0.

### **Leaky ReLU Function:**

- Pros:
  - Addresses the "dying ReLU" problem by allowing a small, non-zero gradient when the unit is not active.
  - Retains most of the benefits of ReLU.
- Cons:
  - The benefit over ReLU is not always clear and can be problem-dependent.
  - Introduces a new hyperparameter (the slope of the leak), which needs to be defined and, in some cases, tuned.

#### **c. [E] Is ReLU differentiable? What to do when it's not differentiable?**

- The ReLU activation function is differentiable everywhere except at 0. It has a derivative of 1 for positive inputs and 0 for negative inputs. At 0, the ReLU function is not differentiable because there is a sharp point where the gradient changes abruptly.
- In practice, when training neural networks using gradient-based methods, the non-differentiability at a single point (0 in the case of ReLU) is usually not a problem. This is because the chance of the input to a ReLU being exactly 0 is low, and during backpropagation, if the input to a ReLU is 0, you can choose either the left-hand derivative (0) or the right-hand derivative (1), and the algorithm will still perform well.
- Some deep learning frameworks handle the non-differentiability at 0 by defining the gradient to be 0 at that point. This is a practical solution that works well in most cases.

#### **d. [M] Derive derivatives for sigmoid function when is a vector.**

- The sigmoid function is given by  $\sigma(z) = \frac{1}{1+e^{-z}}$ .
- The derivative of the sigmoid function with respect to  $z$  is  $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ .
- When  $\mathbf{z}$  is a vector, we can compute the derivative element-wise. If  $\mathbf{z} = [z_1, z_2, \dots, z_n]^T$ , then the derivative of the sigmoid function applied to  $\mathbf{z}$  is a vector  $\sigma'(\mathbf{z})$  where each element is  $\sigma'(z_i) = \sigma(z_i)(1 - \sigma(z_i))$ .

## 5. [E] What's the motivation for skip connection in neural works?

Skip connections, also known as residual connections, are used in deep neural networks to allow gradients to flow through a network directly, without passing through non-linear activation functions. The motivations for using skip connections are:

- **Alleviating Vanishing Gradient Problem:** By providing an alternative pathway for the gradient to flow through, they help mitigate the vanishing gradient problem that can occur in very deep networks.
- **Facilitating Deeper Networks:** Skip connections help in training much deeper networks by allowing information to propagate well in both forward and backward passes.
- **Encouraging Feature Reuse:** They enable the network to reuse features from previous layers, adding them to the features of the current layer.
- **Improving Convergence:** Networks with skip connections tend to converge faster and perform better, as the added gradients help the optimization process.

## 6. Vanishing and exploding gradients.

### a. [E] How do we know that gradients are exploding? How do we prevent it?

#### i. Exploding Gradients:

- **Detection:** Exploding gradients can be detected when the weights or the gradient of the weights become excessively large, often resulting in numerical overflow or NaN errors in the model's loss or outputs.
- **Prevention:** To prevent exploding gradients, the following techniques can be used:
  - **Gradient Clipping:** This technique involves scaling the gradients when they exceed a certain threshold to keep them within a manageable range.
  - **Weight Regularization:** Applying regularization techniques, such as L1 or L2 regularization, can keep the weights small.
  - **Proper Initialization:** Using a suitable initialization strategy for the network weights can also help prevent gradients from exploding.
  - **Using Batch Normalization:** This can help maintain stable gradients across different layers by normalizing the output of each layer.

b. **[E] Why are RNNs especially susceptible to vanishing and exploding gradients?**

- **Susceptibility:** RNNs are especially susceptible to the vanishing gradient problem due to their recurrent nature. In RNNs, gradients are computed through time using the chain rule, and this involves taking the product of several small numbers (the derivatives of the activation functions), which gets exponentially smaller the more steps back in time you go. This makes it difficult for RNNs to learn long-range dependencies within the data.
- **Cause:** The main reason for this susceptibility is the repeated multiplication of gradients across timesteps during backpropagation through time (BPTT). If the gradients are small (less than 1), their products over many timesteps will tend toward zero, leading to vanishing gradients.
- **Mitigation:** To mitigate vanishing gradients in RNNs, several strategies can be employed:

- **Gated Units:** Using RNN variants with gating mechanisms, such as LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Units), which are specifically designed to combat this issue.
- **Gradient Clipping:** Similar to exploding gradients, clipping can also be applied to prevent vanishing gradients by ensuring that gradients do not become too small.
- **Using Skip Connections:** Skip connections can also be used in RNNs to provide alternative paths for gradient flow.
- **Proper Initialization and Activation Functions:** Choosing the right initialization strategy and activation functions that do not saturate can help maintain gradient magnitudes.

7. **[M] Weight normalization separates a weight vector's norm from its gradient. How would it help with training?**

**Weight Normalization:**

Weight normalization is a technique that decouples the length of the weight vectors from their direction. By reparameterizing the weight vectors in terms of their norm and direction, this approach can speed up the convergence of stochastic gradient descent optimization by improving the conditioning of the optimization problem.

• **Training Benefits:**

- **Improved Optimization:** It helps in the optimization process by ensuring that gradients are more isotropic, meaning they're the same in all directions. This is particularly useful when dealing with complex, high-dimensional spaces where certain directions might not be updated efficiently due to poor conditioning.

- **Faster Convergence:** Since weight normalization can lead to a smoother loss landscape and better-conditioned optimization problem, it often results in faster convergence.
- **Stabilizes Training:** By normalizing the weights, the scale of the gradients across different layers becomes more uniform, which can help stabilize the training process.
- **Robust to Initialization:** Weight normalization makes the model less sensitive to the initialization of the weights.

In practice, weight normalization tends to be particularly useful for recurrent neural networks and GANs (Generative Adversarial Networks), where the conditioning of the optimization problem can significantly affect training dynamics.

8. **[M] When training a large neural network, say a language model with a billion parameters, you evaluate your model on a validation set at the end of every epoch. You realize that your validation loss is often lower than your train loss. What might be happening?**

When the validation loss is lower than the training loss in a large neural network, several factors could be contributing to this phenomenon:

- **Regularization:** If you are using dropout, weight regularization (like L1/L2), or other regularization techniques during training, these are not active when evaluating the model on the validation set. This can lead to better performance on the validation set since the network can utilize all its neurons and weights without the regularization penalty.
- **Batch Normalization in Training Mode:** If batch normalization is used and it's incorrectly left in training mode during validation, the model will use the statistics of the current batch, rather than the accumulated statistics from the training to normalize the layer's inputs. This can sometimes lead to better performance if the validation batch statistics happen to be more favorable.

- **Training Batch Size:** If you're using a small batch size during training, the stochastic nature of the gradient updates can make the training loss appear higher due to the noise in gradient estimation. During validation, the entire dataset is usually evaluated in a deterministic way, which can result in a lower loss.
- **Data Preprocessing:** Differences in data preprocessing or augmentation between the training and validation datasets can also lead to discrepancies in loss. For instance, if aggressive augmentation is used in training but not in validation, the model may perform better on the less perturbed validation data.
- **Early in Training:** Especially early in the training process, it's common to see the validation loss lower than the training loss because the training loss is calculated as an average over the entire epoch, while the validation loss is calculated after the epoch, thus benefiting from the entire learning that took place.
- **Model Capacity:** A very large model with a huge number of parameters (like a billion-parameter language model) might still be underfitting the training data, despite its size. In this case, the model is complex enough to perform well on both the training and validation sets but hasn't trained for enough epochs to fit and possibly overfit the training data.

It's important to carefully examine the training process and consider these factors to accurately diagnose why this behavior is occurring. If this pattern persists towards the end of training, it might suggest that the model could be further improved, possibly by training for more epochs, tuning hyperparameters, or using less aggressive regularization.

#### 9. **[E] What criteria would you use for early stopping?**

Early stopping is a form of regularization used to avoid overfitting when training a learner with an iterative method, such as gradient descent. The following criteria are commonly used for early stopping:

- **Validation Loss:** The most common criterion is to monitor the validation loss. If the validation loss stops decreasing and begins to increase, this is a sign that the model is starting to overfit. Early stopping is typically implemented by stopping training if the validation loss does not improve for a specified number of iterations or epochs, known as the "patience" parameter.
- **Validation Metric Plateau:** Instead of, or in addition to, the loss, you can monitor a performance metric that matters for your problem, such as accuracy, F1 score, precision, recall, etc. If this metric stops improving, it may be time to stop training.
- **Performance Relative to a Benchmark:** If the model's performance on a validation set reaches or surpasses a predefined benchmark, training can be stopped.
- **Computational Constraints:** You might also consider computational constraints, such as time or resources. If training isn't yielding substantial improvements and you're limited by these constraints, you might opt for early stopping.

## 10. [E] Gradient descent vs SGD vs mini-batch SGD.

- **Gradient Descent:**
  - It computes the gradient of the cost function with respect to the parameters for the entire training dataset.
  - It's guaranteed to converge to the global minimum for convex loss surfaces and to a local minimum for non-convex surfaces.
  - It can be very slow and is infeasible for datasets that don't fit in memory.
  - It may get stuck in local minima when dealing with non-convex loss functions.
- **Stochastic Gradient Descent (SGD):**

- It computes the gradient of the cost function with respect to the parameters for a single training example at a time.
  - It's much faster than gradient descent and can be used for online learning.
  - It's more likely to escape local minima due to its stochastic nature but has a higher variance in the updates leading to fluctuation in the loss over iterations.
  - The convergence is usually noisier compared to gradient descent.
- **Mini-batch Stochastic Gradient Descent (Mini-batch SGD):**
    - It computes the gradient of the cost function with respect to the parameters for a small subset of the training data, known as a mini-batch.
    - It's a compromise between the robustness of SGD and the efficiency of batch gradient descent.
    - It can take advantage of vectorized operations which make computations faster and typically leads to a smoother convergence than SGD.
    - The size of the mini-batch is a hyperparameter that can be tuned for better performance. A well-chosen mini-batch size will often converge more quickly than SGD or batch gradient descent.

In summary, batch gradient descent uses the entire dataset to compute gradients, which can be slow and resource-intensive; stochastic gradient descent uses one example at a time, which is fast but noisy; and mini-batch gradient descent strikes a balance between the two, using subsets of the dataset to compute gradients more efficiently and with less noise in the updates than pure SGD.

11. **[H] It's a common practice to train deep learning models using epochs: we sample batches from data without replacement. Why would we use epochs instead of just sampling data with replacement?**

Training deep learning models using epochs, where each epoch goes through the entire dataset exactly once without replacement, is a standard practice primarily for the following reasons:

1. **Coverage of the Dataset:** By using epochs, we ensure that every data point is used for training exactly once per epoch. This complete coverage guarantees that the learning process takes into account the entire distribution of the data.
2. **Convergence Properties:** Theoretical and empirical evidence suggests that training with epochs helps in achieving stable convergence. Algorithms that traverse datasets using epochs often have guarantees for convergence to a local minimum or a saddle point under certain conditions.
3. **Cyclic Learning Rate Schedules:** Epochs provide a natural unit of measurement for cyclic learning rates and other learning rate schedules. For example, one might decrease the learning rate after each epoch or after a fixed number of epochs, which can help the model to settle into a minimum of the loss function.
4. **Regularization Effect:** When you shuffle the dataset at the beginning of each epoch, it introduces randomness in the order of data points. This can have a regularizing effect, as it prevents the model from memorizing the sequence of training examples.
5. **Resource Utilization:** Training with epochs allows for better resource allocation and utilization. If you have an estimate of how long an epoch takes, you can plan your compute resources accordingly.
6. **Performance Measurement:** Epochs serve as a useful measure for tracking and reporting the performance of the model. It's standard to evaluate the model on a validation set after each epoch to monitor its generalization performance.

When sampling with replacement (as in pure stochastic gradient descent), there is no guarantee that all examples will be seen after a certain number of samples. This could lead to a situation where some

examples are seen multiple times while others are not seen at all, leading to potential overfitting on certain examples and poor generalization from the training data.

However, it's worth noting that there are situations where sampling with replacement is beneficial or necessary, such as when the dataset is too large to fit into memory or when training a model using a stream of data.

In practice, the choice between using epochs and sampling with replacement may depend on the specific context and constraints of the problem you're trying to solve, but the use of epochs has become a standard practice because of its benefits in terms of convergence and performance measurement.

12. **[M] Your model's weights fluctuate a lot during training. How does that affect your model's performance? What to do about it?**

When a model's weights fluctuate significantly during training, it may indicate that the learning process is unstable. This instability can affect the model's performance in several ways:

1. **Difficulty in Convergence:** If the weights are changing drastically between iterations, the model may have trouble converging to a stable solution. This can lead to suboptimal training and poor generalization on unseen data.
2. **Overfitting Risk:** Large fluctuations might cause the model to overfit, as it might be reacting too strongly to individual training examples rather than learning the underlying pattern.
3. **Escaping Local Minima:** On the positive side, some fluctuation can be beneficial in escaping local minima, but excessive fluctuation is generally undesirable.

**Actions to Stabilize Training:**

To address the issue of fluctuating weights:

- **Decrease Learning Rate:** A high learning rate can cause large updates to the weights, leading to instability. Reducing the learning rate can help the model make smaller, more controlled steps during training.
- **Use Learning Rate Schedulers:** Implement learning rate schedules that adapt the learning rate during training, such as reducing it gradually as the training progresses.
- **Implement Weight Regularization:** Techniques like L1 or L2 regularization can constrain the size of the weights, thereby stabilizing the training.
- **Gradient Clipping:** This involves limiting the size of the gradients during backpropagation, which can prevent large updates to the weights.
- **Use Momentum or Adaptive Learning Rate Optimizers:** Algorithms like Adam, RMSprop, or SGD with momentum can help smooth out the updates to the weights.

13. Learning rate.

a. [E] Draw a graph number of training epochs vs training error for when the learning rate is:

- i. too high
- ii. too low
- iii. acceptable.

b. [E] What's learning rate warmup? Why do we need it?

Learning rate warmup is a technique where the learning rate is gradually increased from a small to a large value during the initial phase of training. The purpose of this technique is to stabilize the training process and improve convergence. Warmup is particularly useful in the following scenarios:

- **Training Deep Networks:** Deep networks are often sensitive to the choice of the initial learning rate. Starting with a too high learning rate can lead to instability, while starting with a too low learning rate can lead to slow convergence.
- **Training with Large Batch Sizes:** When using large batch sizes, the noise in the gradient estimate is reduced, which can lead to instability if the learning rate is not adjusted. Warmup helps to mitigate this issue by allowing the model to start learning with small steps and then gradually increasing the step size.
- **Adaptive Optimizers:** Even when using adaptive learning rate optimizers like Adam, warmup can help in the initial stages where the moving averages of the gradients are not yet stable.

In practice, learning rate warmup involves starting with a low learning rate and then increasing it linearly or according to some schedule for a certain number of epochs or steps before switching to the main learning rate for the remainder of the training. This approach helps to gently nudge the model towards the right direction in the loss landscape, thus improving the stability and performance of the training process.

#### 14. **[E] Compare batch norm and layer norm.**

Batch normalization (BatchNorm) and layer normalization (LayerNorm) are both techniques used to stabilize and accelerate the training of deep neural networks, but they differ in how they normalize the input data.

##### **Batch Normalization (BatchNorm):**

- **Mechanism:** BatchNorm normalizes the activations of the previous layer for each batch. It computes the mean and variance for the batch, and then normalizes the activations. After normalization, it applies a scale and shift operation, allowing the network to undo the normalization if it is beneficial.

- **Usage:** Commonly used in convolutional neural networks (CNNs) and fully connected layers.
- **Benefits:**
  - Can lead to faster convergence and higher overall performance.
  - Helps to mitigate the internal covariate shift problem, where the distribution of each layer's inputs changes during training.
- **Limitations:**
  - Dependency on batch size – doesn't work well with small batch sizes.
  - Performance can degrade with non-i.i.d data (e.g., in recurrent neural networks).
  - Computationally intensive and can slow down training, especially for small batches.
- **Application:** More effective in scenarios where the batch size is large enough and the input data is i.i.d.

### **Layer Normalization (LayerNorm):**

- **Mechanism:** LayerNorm normalizes the activations across the features in a layer rather than across a batch. For each data point, it computes the mean and variance used for normalization over all the features.
- **Usage:** Commonly used in recurrent neural networks (RNNs), including LSTM and Transformer models.
- **Benefits:**
  - Independent of batch size – can be used with a batch size of 1.
  - Can stabilize the hidden state dynamics in RNNs.

- Useful for tasks with non-i.i.d data and in online learning scenarios.
- **Limitations:**
  - It may not lead to as fast convergence as BatchNorm in some cases (especially in CNNs).
  - The normalization is done per data point, which may not capture some batch-level information.
- **Application:** Particularly effective in sequence models and in scenarios where the batch size is limited or the data is not i.i.d.

In summary, BatchNorm and LayerNorm are both normalization techniques to stabilize training, but they differ in their approach: BatchNorm normalizes across the batch dimension and is more effective for larger batches in CNNs, while LayerNorm normalizes across features for each data point, making it suitable for RNNs and scenarios with small or dynamic batch sizes.

## 15. [M] Why is squared L2 norm sometimes preferred to L2 norm for regularizing neural networks?

The squared L2 norm is often preferred over the L2 norm for regularizing neural networks due to several reasons:

- **Analytical Convenience:** The derivative of the squared L2 norm is linear with respect to the weights, which simplifies the gradient calculations during backpropagation. Specifically, the gradient of the squared L2 norm of weights is just 2 times the weights themselves, making the gradient update rule straightforward and computationally efficient.
- **Stronger Penalty for Larger Weights:** The squared L2 norm increases quadratically with the weight values, imposing a heavier penalty on larger weights compared to the L2 norm. This characteristic helps in more effectively preventing the weights from growing too large, which is a key goal of regularization to avoid overfitting.

- **Stable Gradients:** The gradient of the squared L2 norm is more stable and smooth compared to the L1 norm. The L2 norm's gradient can become very small when the weights are close to zero, potentially slowing down learning. The squared term ensures that the penalty's gradient is significant enough for effective learning.
- **Convexity:** The squared L2 norm is a convex function, which is beneficial in optimization, ensuring there are no local minima other than the global minimum in the regularization term.

In summary, the squared L2 norm is often preferred in neural network regularization due to its computational efficiency, effectiveness in penalizing large weights, and desirable properties in optimization.

16. **[E] Some models use weight decay: after each gradient update, the weights are multiplied by a factor slightly less than 1. What is this useful for?**

Weight decay is a regularization technique where, after each gradient update, the weights of the neural network are multiplied by a factor slightly less than 1. This is useful for:

- **Preventing Overfitting:** By shrinking the weights towards zero, weight decay reduces the complexity of the model, helping to prevent overfitting. Smaller weights tend to lead to simpler models, which generalize better to new, unseen data.
- **Encouraging Sparse Weights:** Weight decay can encourage the network to maintain only the most important weights, effectively leading to a sparser model. Sparse models are often more efficient and can be more interpretable.
- **Stabilizing Training:** Weight decay can improve the stability of the training process by ensuring that the weights do not grow too large. Large weights can lead to numerical instability and exacerbate issues like exploding gradients.

- **Simulating L2 Regularization:** Mathematically, weight decay is similar to applying an L2 regularization term to the loss function, especially under certain conditions (e.g., with small learning rates and simple optimization algorithms like stochastic gradient descent).

Weight decay is particularly useful when training deep and complex neural networks, as it provides a simple yet effective way to control overfitting and improve the generalization capability of the model.

17. It's a common practice for the learning rate to be reduced throughout the training.

a. [E] What's the motivation?

- **Convergence to Minima:** Initially, a higher learning rate is beneficial as it helps the model to quickly converge towards the minima. However, as training progresses, a high learning rate might cause the model to oscillate around the minima without settling. Reducing the learning rate helps the model to take smaller steps, thereby allowing it to fine-tune its parameters and settle into the minima.
- **Avoiding Overshooting:** In the initial phases of training, the gradients are usually large, and a high learning rate helps in making significant updates. But as we approach a minimum, gradients become smaller, and a high learning rate could cause overshooting, preventing the model from finding the lowest point of the loss function.
- **Adapting to the Loss Landscape:** The loss landscape can be more complex as it gets closer to the minimum. A smaller learning rate gives the model the sensitivity it needs to navigate through intricate contours in the loss landscape.

b. [M] What might be the exceptions?

While reducing the learning rate during training is a widely used and effective technique, there are exceptions and scenarios where this approach might not be optimal:

- **Non-Stationary Problems:** In scenarios where the data distribution changes over time (non-stationary environments), keeping a higher learning rate might be beneficial as it allows the model to adapt more quickly to the changing data.
- **Large Batch Training:** When training with very large batch sizes, the noise in the gradient estimates is reduced. In such cases, a constant or even an increasing learning rate schedule might be more effective to maintain a level of exploration.
- **Complex Loss Landscapes with Many Local Minima:** In cases where the loss landscape is highly complex with many local minima or saddle points, a constant or cyclical learning rate might help the model to escape these and find better minima.
- **Use of Advanced Optimizers:** Some advanced optimization algorithms, like Adam or RMSprop, adapt the learning rate for each parameter dynamically based on estimates of first and second moments of the gradients. With these optimizers, the need for manual learning rate reduction might be less critical, though it can still be beneficial.
- **Exploration-Exploitation Tradeoff:** In reinforcement learning and other areas where there's a tradeoff between exploration and exploitation, a constant or cyclical learning rate might be used to maintain a balance between exploring new strategies and exploiting known good strategies.

In summary, while reducing the learning rate during training is generally beneficial for convergence and stability, the optimal learning rate schedule can depend on the specific characteristics of the problem, the data, the model, and the training method.

18. Batch size.

- a. [E] What happens to your model training when you decrease the batch size to 1?

When you decrease the batch size to 1 (which essentially turns the training into stochastic gradient descent), the following effects are observed:

- **Increased Variance in Updates:** Each update is based on a single example, so the gradient estimates have high variance. This can lead to a more irregular training path, potentially escaping local minima but also potentially causing instability in convergence.
- **Frequent Updates:** The model weights are updated after every example, which can make training faster in terms of convergence per iteration but computationally more expensive overall due to frequent updates and less efficient matrix operations.
- **Higher Generalization:** Training with a batch size of 1 can sometimes lead to better generalization, as the model is less likely to overfit to the noise in the data.
- **Less Utilization of Hardware Acceleration:** Small batch sizes don't utilize the full parallelization capabilities of GPUs and other hardware accelerators as effectively as larger batches

b. **[E] What happens when you use the entire training data in a batch?**

Using the entire training dataset as a batch, also known as batch gradient descent, has the following effects:

- **Stable Gradient Estimates:** The gradient is calculated over the entire dataset, providing a very stable and consistent direction for the updates. This stability can lead to more predictable and smooth convergence.
- **Memory Limitations:** Requires significant memory, which might not be feasible with large datasets.
- **Less Frequent Updates:** Weights are updated only once per epoch, which could lead to slower convergence in terms of epochs.

- **Potential for Poorer Generalization:** Training with very large batches can sometimes lead to poorer generalization performance compared to smaller batches.
- **Efficient Use of Hardware:** Large batches can make more efficient use of the computational power of GPUs and other hardware accelerators.

c. **[M] How should we adjust the learning rate as we increase or decrease the batch size?**

The learning rate should ideally be adjusted when changing the batch size:

- **Increase Batch Size:** When increasing the batch size, you can typically increase the learning rate. Larger batches provide more stable and accurate gradient estimates. Therefore, the model can tolerate larger steps without the risk of diverging. A common heuristic is to scale the learning rate linearly with the batch size.
- **Decrease Batch Size:** Conversely, when decreasing the batch size, you should generally decrease the learning rate to account for the higher variance in the gradient estimates. Smaller batches mean noisier gradients, so a smaller learning rate helps in taking more cautious steps during training.
- **Rule of Thumb:** A rule of thumb is to increase the learning rate in proportion to the increase in batch size, and vice versa. For example, if you double the batch size, you might also double the learning rate, keeping in mind the need to tune and validate this adjustment based on the specific model and data.

In practice, the optimal adjustment of learning rate with respect to batch size is problem-specific and should be empirically determined through experimentation and validation.

19. **[M] Why is Adagrad sometimes favored in problems with sparse gradients?**

Adagrad (Adaptive Gradient Algorithm) is often favored for problems with sparse gradients due to its unique approach to adapting the learning rate:

- **Individual Learning Rates:** Adagrad adjusts the learning rate for each parameter individually, based on the historical squared gradients. This means that parameters with sparse gradients (i.e., gradients that don't change much) get larger updates, and those with frequent large updates get smaller learning rates.
- **Suitability for Sparse Data:** In problems like natural language processing and recommender systems, where the data is typically sparse, Adagrad's feature of adapting learning rates to individual parameters helps in making more significant updates to infrequently occurring features.
- **Compensating for Imbalance:** Sparse data often leads to an imbalance in the frequency of feature updates. Adagrad compensates for this by making larger updates to infrequent features, which might be critical for the model's performance.
- **Preventing Vanishing Learning Rate:** While Adagrad's learning rate tends to decrease over time (which can be a limitation in some cases), its ability to perform larger updates for infrequent parameters makes it suitable for sparse settings where these parameters play a significant role.

20. [Adam vs. SGD.](#)

a. [\[M\] What can you say about the ability to converge and generalize of Adam vs. SGD?](#)

- **Convergence Speed:** Adam usually converges faster than SGD because it computes adaptive learning rates for each parameter. Adam combines the advantages of two other extensions of stochastic gradient descent – AdaGrad and RMSProp.
- **Generalization:** While Adam often converges faster, SGD might achieve better generalization in some cases, especially with proper tuning of the learning rate and momentum. SGD's slower and

more steady convergence can sometimes lead to a more robust model at convergence.

- **Sensitivity to Hyperparameters:** Adam is often praised for being less sensitive to hyperparameter tuning (especially the learning rate) than SGD. However, for achieving the best results, especially in terms of generalization, careful tuning of Adam's hyperparameters is still important.

b. [M] What else can you say about the difference between these two optimizers?

- **Learning Rate Adaptation:** Adam adjusts the learning rate for each parameter based on the first and second moments of the gradients. In contrast, standard SGD uses the same learning rate for all parameters, which does not adapt during training (unless manually scheduled).
- **Momentum:** Adam incorporates momentum by keeping track of the exponentially decaying average of past gradients (first moment). This helps Adam in navigating the loss landscape more smoothly compared to SGD.
- **Hyperparameters:** Adam introduces additional hyperparameters (such as the decay rates for the moment estimates) that are not present in standard SGD. These additional parameters often come with default values that work reasonably well in practice.
- **Robustness in Non-Stationary Environments and Sparse Gradients:** Adam is generally more robust in non-stationary objectives and problems with sparse gradients compared to SGD.

In summary, Adam is known for its fast convergence and less sensitivity to hyperparameter tuning, making it a go-to choice for many practitioners, especially in the initial stages of model development. SGD, on the other hand, with the right learning rate scheduling and momentum, might offer better generalization in certain scenarios. The choice between Adam and SGD often depends on the specific problem, dataset, and even empirical results obtained during the model development phase.

21. **[M] With model parallelism, you might update your model weights using the gradients from each machine asynchronously or synchronously. What are the pros and cons of asynchronous SGD vs. synchronous SGD?**

In model parallelism, different parts of a neural network model are distributed across multiple machines or processors. The weights can be updated using gradients from each machine either asynchronously (Asynchronous SGD) or synchronously (Synchronous SGD). Each approach has its advantages and disadvantages:

#### **Asynchronous SGD:**

- Pros:
  - **Speed:** As updates are made asynchronously, the training process can be faster as it doesn't have to wait for the slowest worker to complete its task.
  - **Resource Utilization:** It can lead to better resource utilization as all workers are used effectively without idle time.
- Cons:
  - **Stale Gradients:** As updates are made asynchronously, there is a possibility of using outdated (stale) gradients for updating the weights, which can potentially lead to convergence issues or suboptimal training.
  - **Harder to Tune:** The learning rate and other hyperparameters might need more careful tuning to account for the noise introduced by stale gradients.

#### **Synchronous SGD:**

- Pros:

- **Stable Convergence:** As all gradients are computed from the current version of the model, this can lead to more stable and reliable convergence.
- **Consistency:** It ensures that all updates are made using the latest data, reducing the impact of stale gradients.
- Cons:
  - **Speed:** Synchronous SGD can be slower than asynchronous SGD, as it requires waiting for the slowest worker to complete its task before proceeding with an update.
  - **Resource Utilization:** There may be idle time for some workers, leading to less efficient use of resources.

In summary, the choice between asynchronous and synchronous SGD in a distributed training environment depends on the specific requirements of the training process and the trade-offs between speed and stability. Asynchronous SGD offers faster training but can suffer from issues due to stale gradients, while synchronous SGD provides more stable convergence at the cost of potentially slower training.

## 22. **[M] Why shouldn't we have two consecutive linear layers in a neural network?**

Having two consecutive linear layers in a neural network is generally considered ineffective due to the following reason:

- **Lack of Increased Expressiveness:** The composition of two linear transformations is still a linear transformation. This means that if you have two consecutive linear layers (without a non-linear activation function in between), they could be replaced by a single linear layer without loss of any expressiveness or capacity of the model.
- **Redundancy:** Since the two linear layers can effectively be combined into one, having two separate layers only adds redundancy to the model, increasing the number of parameters without increasing the

model's ability to capture complex patterns in the data.

- **Inefficiency:** More parameters mean more computation and more memory usage, which is inefficient, especially when these additional parameters do not contribute to a more powerful model.

To make each layer in a neural network contribute to its ability to capture and model more complex patterns, it's important to introduce non-linearities between linear layers. This is usually done by using non-linear activation functions (like ReLU, sigmoid, tanh, etc.) between linear (or dense) layers. These non-linear activation functions allow the neural network to learn more complex functions and provide the capacity to model complex relationships in the data.

23. **[M] Can a neural network with only RELU (non-linearity) act as a linear classifier?**

A neural network that uses only ReLU (Rectified Linear Unit) activation functions cannot act as a linear classifier. The key reasons are:

- **Non-linear Transformation:** Although each individual ReLU function performs a piecewise linear transformation, it introduces a non-linearity at zero (i.e., it outputs zero for negative inputs and linearly scales positive inputs). This non-linearity, when applied across a network with multiple layers and units, allows the network to model non-linear relationships.
- **Capability Beyond Linear Classification:** Due to the composition of multiple ReLU functions in a multi-layer network, the overall transformation that the network applies to the input data is non-linear. This allows the network to learn complex patterns that linear classifiers cannot.

In essence, while ReLU itself performs a simple operation, the combination of multiple ReLU units in a network, especially across layers, enables the modeling of non-linear decision boundaries, making the network capable of more than just linear classification.

24. **[M] Design the smallest neural network that can function as an XOR gate.**

An XOR (exclusive OR) gate is a fundamental digital logic gate that outputs true (or 1) only when the inputs differ. To design a neural network that functions as an XOR gate, you need at least a two-layer network (one hidden layer and one output layer) with non-linear activation functions. Here's a simple design:

- **Input Layer:** 2 neurons (for the two binary inputs).
- **Hidden Layer:** 2 neurons with non-linear activation functions (e.g., ReLU or Sigmoid).
- **Output Layer:** 1 neuron with a non-linear activation function (e.g., Sigmoid to output a value between 0 and 1).

Here's a simple implementation in Python:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# XOR Neural Network
class XORNeuralNetwork:
    def __init__(self):
        # Weights and biases initialized to emulate XOR
        self.hidden_weights = np.array([[20, 20], [-20, -20]])
        self.hidden_bias = np.array([-10, 30])
        self.output_weights = np.array([20, 20])
        self.output_bias = -30
```

```

def forward(self, x):
    hidden_layer_output = sigmoid(np.dot(x, self.hidden_weights) + self.h:
    output = sigmoid(np.dot(hidden_layer_output, self.output_weights) + se
    return output

# Testing the XOR Neural Network
xor_nn = XORNeuralNetwork()

# XOR Inputs and Outputs
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for x in inputs:
    print(f"Input: {x}, XOR Output: {xor_nn.forward(x)}")

```

This network effectively models the XOR function by explicitly setting weights and biases to create the necessary non-linear decision boundaries.

## 25. **[E] Why don't we just initialize all weights in a neural network to zero?**

Initializing all weights to zero in a neural network is not recommended for the following reasons:

- **Symmetry Problem:** If all weights are initialized to zero, all neurons in a given layer will perform the same computation, producing the same output. Consequently, their gradients will also be the same during backpropagation. This symmetry does not break during training, meaning that all neurons will learn the same features, which is ineffective.
- **No Gradient Descent Learning:** With zero initialization, the gradients calculated during backpropagation will also be zero (due to the derivative of the loss with respect to the weights being

zero). This means the weights will not be updated during training, rendering the learning process ineffective.

- **Breaking Symmetry:** Effective training of neural networks requires breaking the symmetry of neurons' weights. This is usually done by initializing weights to small random values, ensuring that each neuron learns different features and patterns in the input data.

In summary, initializing weights to non-zero values allows for effective learning and model differentiation, essential for training neural networks.

## 26. Stochasticity.

### a. [M] What are some sources of randomness in a neural network?

Several sources contribute to the randomness (stochasticity) in neural networks:

- **Initialization of Weights:** Typically, weights in a neural network are initialized to random values drawn from a specific distribution (like a normal or uniform distribution). This randomness breaks the symmetry and allows the network to learn effectively.
- **Data Shuffling:** During training, the dataset is often shuffled before each epoch. This prevents the model from learning the order of the data and helps in generalizing better.
- **Stochastic Gradient Descent and Variants:** In stochastic gradient descent (SGD) and its variants (like mini-batch SGD), a random subset of the data is used to compute the gradient and update the weights. The randomness in the selection of these subsets introduces variability in the gradient estimates.
- **Dropout:** Dropout is a regularization technique where randomly selected neurons are ignored during training. This randomness helps the network to learn more robust features.

- **Data Augmentation:** Random transformations (like rotation, scaling, cropping, etc.) are applied to the training data, which can introduce variability and help the model generalize better.
- **Ensemble Methods:** Techniques like bagging involve training multiple models on random subsets of the data or with random initializations, introducing randomness in the overall training process.

b. **[M] Sometimes stochasticity is desirable when training neural networks. Why is that?**

Stochasticity is often desirable in training neural networks due to several reasons:

- **Avoiding Local Minima:** The randomness in gradient estimates, due to either SGD or data shuffling, helps the model to avoid getting stuck in local minima, potentially leading to better solutions in the global loss landscape.
- **Regularization Effect:** Randomness in data augmentation, dropout, and weight initialization acts as a form of regularization, preventing the network from overfitting to the noise in the training data and thereby improving generalization to unseen data.
- **Exploration:** In the context of optimization, stochasticity encourages exploration of the loss landscape, which can be crucial for finding good minima.
- **Model Robustness:** Randomness introduced by techniques like dropout or data augmentation forces the network to not rely on any single neuron or feature too heavily, thus encouraging the learning of more robust and generalized features.

In summary, stochasticity in neural network training plays a critical role in achieving good generalization, robustness, and effective exploration of the loss landscape. It helps to prevent overfitting and ensures that the network learns to perform well across a variety of different inputs.

27. **Dead neuron.**

a. **[E] What's a dead neuron?**

A dead neuron in a neural network is a neuron that always outputs the same value, regardless of the input. This typically happens when the neuron's activation function is non-linear, like ReLU (Rectified Linear Unit), and the input to the neuron is such that the activation function always outputs the minimum value (which is 0 in the case of ReLU). For ReLU, this happens when the weights and biases of the neuron are adjusted during training in such a way that the weighted sum of the inputs is always negative, leading to an output of 0.

b. **[E] How do we detect them in our neural network?**

To detect dead neurons, monitor the activations of each neuron over a batch of inputs:

- For neurons using ReLU activation, check if the output is constantly zero for all the inputs in a batch or over multiple batches.
- A more systematic approach is to calculate the percentage of times each neuron activates (outputs non-zero) over many batches. Neurons that activate very rarely or never can be considered dead.

c. **[M] How to prevent them?**

- **Proper Initialization:** Use initialization methods like He or Xavier initialization that consider the size of the previous layer to set the initial weights.
- **Leaky ReLU:** Use variants of ReLU like Leaky ReLU or Parametric ReLU (PReLU) that allow small gradients when the unit is not active, thereby keeping the neurons "alive".
- **Regularization:** Apply regularization techniques like L1 or L2 to prevent the weights from becoming too large or too small.
- **Careful Learning Rate:** Use an appropriate learning rate. A very high learning rate can cause the weights to update too drastically, leading to dead neurons.

- **Batch Normalization:** Implement batch normalization, which can help maintain the mean and variance of input distributions within a desirable range.

28. Pruning.

a. [M] Pruning is a popular technique where certain weights of a neural network are set to 0. Why is it desirable?

- **Reduced Model Size:** Pruning reduces the number of parameters in a model, decreasing its storage requirements.
- **Improved Inference Speed:** A pruned model often requires fewer computations, leading to faster inference times, which is crucial for deployment in resource-constrained environments.
- **Energy Efficiency:** Fewer computations also mean less energy consumption, making the model more eco-friendly and suitable for mobile devices.
- **Regularization Effect:** Pruning can have a regularization effect on the network, reducing overfitting and potentially improving the model's generalization ability.

b. [M] How do you choose what to prune from a neural network?

- **Low-Magnitude Weights:** A common approach is to prune weights with small magnitudes, under the assumption that they contribute less to the output.
- **Importance Measures:** Evaluate the importance of weights or neurons using criteria like the effect of their removal on the loss function or their contribution to the output variance.
- **Structured vs. Unstructured Pruning:** Decide between unstructured pruning (removing individual weights) and structured pruning (removing entire neurons or filters), depending on the requirements. Structured pruning is more likely to result in computationally efficient models.

- **Iterative Pruning and Retraining:** Prune a small portion of the network, retrain to recover performance, and repeat the process. This gradual approach often yields better results than pruning a large portion of the network at once.
- **Data-Driven Pruning:** Use data or task-specific criteria to determine which parts of the network are less critical for performance and can be pruned without significant loss in accuracy.

Pruning is a balance between reducing the size and computational requirements of a model and maintaining its accuracy. The specific approach depends on the model architecture, the application, and the computational constraints.

29. **[H] Under what conditions would it be possible to recover training data from the weight checkpoints?**

Recovering actual training data from model weights is generally a challenging task and is often not feasible, especially with complex models and large datasets. However, under certain conditions, it might be possible to infer some characteristics of the training data or even reconstruct aspects of it:

- **Simple Models and Overfitting:** If the model is relatively simple (e.g., a small neural network) and has overfitted to a small dataset, it might memorize aspects of the training data. In such cases, aspects of the training data might be inferred from the weights.
- **Leakage through Gradients:** In settings where individual gradients are stored or shared (as in some federated learning scenarios), there might be a risk of data leakage, as gradients can sometimes reveal information about the training data.
- **Inversion Attacks:** Under certain conditions, sophisticated inversion attacks using optimization techniques can reconstruct training examples from model weights, especially if the model is poorly regularized and the attacker has access to auxiliary information.

- **Small, Unique Datasets:** If the model is trained on a small, unique dataset, the weights might capture specific characteristics of this dataset, making it possible to infer certain properties of the training data.

In general, the ability to recover training data from model weights is limited and depends heavily on the complexity of the model, the nature of the dataset, and the extent of overfitting. The risk is typically higher for simpler models and smaller, more unique datasets.

30. **[H] Why do we try to reduce the size of a big trained model through techniques such as knowledge distillation instead of just training a small model from the beginning?**

Reducing the size of a large, trained model through techniques like knowledge distillation, rather than just training a small model from the beginning, offers several advantages:

- **Transfer of "Dark Knowledge":** Large models often learn nuanced representations and implicit data distributions (referred to as "dark knowledge"). Knowledge distillation allows this intricate knowledge to be transferred to a smaller model, which might not be achievable through direct training.
- **Performance Benefits:** Large models generally have better performance due to their greater capacity. Distilling their knowledge into a smaller model can often result in better performance than training the smaller model directly.
- **Balancing Size and Accuracy:** Knowledge distillation provides a way to balance the trade-off between model size and accuracy. It allows the creation of a more compact model that retains much of the accuracy of the larger model.
- **Model Architecture Constraints:** Sometimes, the architecture of small models is constrained by the requirements of the deployment environment (like mobile devices). Knowledge distillation allows these constrained models to learn effectively from larger, more capable models.

- **Resource Efficiency in Deployment:** Training a large model once and distilling its knowledge into smaller models is resource-efficient for deployment scenarios where you need different models for different resource-constrained environments.

In essence, knowledge distillation leverages the capacity of large models to capture complex relationships in the data, and then transfers this understanding to smaller models, which are more practical for deployment in resource-constrained environments. This approach often yields smaller models that perform better than if they were trained directly on the training data.