Deep Learning Interview Questions

Beginner

1. What is Deep Learning, and how does it relate to Machine Learning?

Answer:

Deep learning is a branch of Artificial Intelligence that uses algorithms inspired by the human brain's structure, enabling computers to learn from large amounts of data and make intelligent decisions.

It is a subfield of statistical machine learning. The key difference is that deep learning models build layers one after another, using outputs from preceding layers as inputs, allowing them to learn features automatically, whereas most traditional machine learning algorithms are "shallow" and use features directly from the training data.

2. What is an artificial neuron in a Neural Network?

Answer:

An artificial neuron is the basic unit in a neural network, inspired by biological neurons. It computes a weighted sum of its inputs plus a bias and then applies an activation function.

Formally, a neuron outputs $f(\sum_{i=1}^{n} w_i\, x_i + b)$ where x_i are inputs, w_i weights, b bias, and f is an activation function. The neuron's role is to learn a decision boundary by adjusting weights. In effect, one neuron models a simple linear function followed by nonlinearity, enabling the network to capture complex patterns

3. Why do neural networks use activation functions like sigmoid, ReLU, or tanh?

Answer:

Activation functions are critical components that introduce non-linearity into the neural network model. Without non-linearity, a deep network would behave like a

simple linear model, severely limiting its ability to learn complex patterns and relationships in the data.

4. What is overfitting in a Machine Learning model?

Answer:

Overfitting occurs when a model learns the training data too well, including its noise and specific characteristics, leading to excellent performance on the training set but poor generalization and accuracy on new, unseen data.

5. Explain the difference between Deep Learning and traditional Machine Learning, with the perspective of feature engineering.

Answer:

Traditional Machine Learning often requires manual feature engineering, where human experts extract relevant features from raw data.

In contrast, **Deep Learning** algorithms automatically learn and extract hierarchical features through their multiple layers. Deep learning models use the outputs of preceding layers as inputs for subsequent layers, allowing them to discover increasingly abstract and complex patterns directly from the raw data, eliminating the need for explicit feature engineering.

6. What is the Softmax function and when is it used?

Answer:

The **Softmax** function converts a vector of real-valued scores into a probability distribution over classes. For an output vector$z$, Softmax gives :

$$soft\max(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

It ensures the outputs are non-negative and sum to 1, making them interpretable as probabilities. Softmax is typically used as the final activation in multi-class classification networks (with cross-entropy loss), so that the model's outputs represent class probabilities.

7. What are GPUs and TPUs, and why are they used in Deep Learning?

Answer:

GPUs (Graphics Processing Units) are hardware accelerators originally built for graphics, with thousands of cores optimized for parallel matrix and tensor operations. This makes them ideal for the heavy linear algebra in neural networks.

TPUs (Tensor Processing Units) are custom ASIC chips developed by Google specifically for accelerating AI workloads. TPUs are highly optimized for tensor computations (matrix multiplications, convolutions, etc.) and can offer higher throughput/efficiency than GPUs for large-scale models.

In short, both GPUs and TPUs speed up training and inference dramatically compared to CPUs. A concise summary: "GPUs are specialized for accelerated compute tasks (graphics and AI), and TPUs are Google's custom ASICs for AI workloads". Deep learning models are typically trained on GPUs or TPUs to reduce training time from months to days.

8. What is an epoch, a batch, and an iteration in model training?

Answer:

In training a Neural Network, a **batch** is a subset of the training data used to compute a single update of the model's parameters.

An **iteration** (or update step) refers to one forward/backward pass using that batch.

An **epoch** is one full pass over the entire training set.

For example, if you have 1,000 training examples and use a batch size of 100, then one epoch consists of 10 iterations (10 batches). During training, we often run for many epochs, meaning the model sees each example multiple times (with parameter updates along the way).

9. Why do neural networks need a bias term?

Answer:

The **bias** term in a neuron acts like an intercept in a linear model. It allows the activation function to be shifted left or right, enabling the neuron to activate even when inputs are zero. In practice, bias lets each neuron learn a threshold.

For example, without bias, a linear neuron's output is forced through the origin; with bias, it can fit data that do not pass through zero. Intuitively, bias gives the network greater flexibility.

10. What is a PyTorch Tensor and how does it differ from a NumPy array?

Answer:

A **PyTorch Tensor** is a multi-dimensional array, similar to a NumPy ndarray, but with additional capabilities.

Tensors can be moved to GPU memory for fast computation, whereas NumPy arrays live on CPU.

Importantly, PyTorch tensors can track computations for automatic differentiation (if you set requires_grad=True).

PyTorch is tightly integrated with NumPy: you can easily convert between a numpy.ndarray and a torch.Tensor (using methods like torch.from_numpy or tensor.numpy()), and PyTorch's tensor operations are largely analogous to NumPy's.

The key difference is that PyTorch tensors support GPU acceleration and autograd, while NumPy arrays do not.

Intermediate 🚀

1. What is Autograd? Elaborate w.r.t Pytorch.

Answer:

Autograd is PyTorch's automatic differentiation engine.

It automatically computes gradients of tensor operations during backpropagation. In practice, when you perform operations on tensors with requires_grad=True, PyTorch builds a computation graph on-the-fly. Then, calling loss.backward() propagates gradients backward through this graph using the chain rule.

This gradient information is stored in each tensor's .grad attribute. As the PyTorch documentation states, torch.autograd "is the automatic differentiation engine that powers neural network training". Autograd handles the complexity of computing partial derivatives, so you don't have to derive them manually.

2. What is gradient descent and how does backpropagation work in training neural networks?

Answer:

Gradient descent is an optimization algorithm used to train neural networks by minimizing the loss function.

At each step, it computes the gradient (slope) of the loss with respect to the model's parameters and updates the parameters by moving a small step in the opposite direction of the gradient. In formula: $w \leftarrow w - \eta \nabla_w L$, where η is the learning rate. Intuitively, one starts at an initial point on the loss surface, computes the derivative (slope), and "steps downhill" towards a minimum.

Repeating this process iteratively reduces the loss. As described in literature, gradient descent "uses the slope of the loss function to iteratively update parameters" until convergence. Properly tuned, this process finds weights that minimize prediction error.

Backpropagation is the algorithm for efficiently computing gradients of the loss with respect to each model parameter.

Conceptually, you perform a forward pass to compute the loss, then a backward pass where errors are propagated back through the network.

During backpropagation, each neuron applies the chain rule to compute how the loss changes with respect to its inputs and weights. Mathematically, gradients "flow backwards" through the computational graph. As one source explains: "The backward pass then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs". These gradients are then used by an optimizer to update the weights.

3. Compare and contrast Batch Gradient Descent, Stochastic Gradient Descent, and Mini-Batch Gradient Descent.

Answer:

Batch Gradient Descent (BGD) calculates the gradient using the entire dataset for each parameter update, leading to stable convergence but being slow and memory-intensive for large datasets.

Stochastic Gradient Descent (SGD) updates parameters after processing each single data point, making it faster and memory-efficient but resulting in noisy gradients that can cause oscillations. However, this noise can help escape local minima.

Mini-Batch Gradient Descent combines aspects of both, splitting the dataset into small batches and updating parameters after each batch. It strikes a balance between computational efficiency, speed, and stability, reducing noise compared to SGD while being faster than BGD, and is often the preferred method for large datasets.

4. Explain the Vanishing Gradient problem and one common way to mitigate it.

Answer:

The **Vanishing Gradient problem** occurs when gradients propagated backward through deep neural networks become exponentially smaller in earlier layers. This causes weights in those layers to update very slowly, hindering learning.

One common way to mitigate it is by using the **ReLU** (Rectified Linear Unit) activation function. For positive inputs, ReLU has a constant gradient of 1, allowing gradients to pass through unchanged during backpropagation, which helps prevent them from vanishing.

Another method is **Batch Normalization**, which normalizes activations to stabilize gradient flow.

5. What is the purpose of Batch Normalization in deep learning?

Answer:

Batch Normalization (BN) is a technique that normalizes the activations of intermediate layers in a Deep Neural Network to have a zero mean and unit standard deviation across mini-batches. Its primary purpose is to stabilize the learning process, enabling the use of higher learning rates, which leads to faster convergence and improved accuracy. It also helps mitigate vanishing and exploding gradients and provides an implicit regularization effect.

6. How do L1 and L2 regularization differ in their effect on model coefficients?

Answer:

Both **L1** (Lasso) and **L2** (Ridge) regularization add a penalty term to the loss function to prevent overfitting.

The key difference lies in their penalty terms and effects on coefficients:

- L1 Regularization adds the absolute value of the sum of coefficients as a penalty. It can reduce some coefficient values exactly to zero, effectively performing feature selection and creating sparse models.
- L2 Regularization adds the squared sum of coefficients as a penalty. It reduces coefficient values towards zero but never exactly to zero, meaning it doesn't perform feature selection but is effective at reducing the magnitude of all coefficients and handling multicollinearity.

7. What are the key components of a Convolutional Neural Network (CNN)? Explain convolution, kernels, pooling, padding, and stride.

Answer:

A **CNN** has several specialized layers:

Convolutional layers: Apply learnable filters (kernels) that slide over the input image or feature map. Each filter performs a convolution operation that multiplies and sums local patches of the input, detecting features like edges or textures. Stacking many filters yields multiple feature maps.

Kernels/Filters: Small matrices (e.g. 3×3) of weights in a convolution. During a pass, each filter produces one feature map by convolving over the entire input.

Stride: The step size of how the filter moves across the input. A stride of 1 moves the filter one pixel at a time; larger strides down sample the output by skipping positions.

Padding: Adding a border (usually zeros) around the input so that filters can properly scan the edges or control the output dimensions. For example, “same” padding keeps output the same size as input by padding sufficiently.

Pooling layers: Perform down sampling (e.g. max-pooling or average-pooling) to reduce spatial dimensions while keeping the most important information. For instance, max-pooling takes the maximum value in each region, providing translation invariance and reducing computation. As one resource notes, pooling “is responsible for dimensionality reduction, minimizing computational requirements while retaining essential features. Two common types are max pooling (maximum value) and average pooling”

8. What is Data Augmentation and why is it used in CNN training?

Answer:

Data Augmentation involves creating modified versions of the training images (or other data) to artificially enlarge the dataset.

Common techniques include random crops, flips, rotations, color jitter, and scaling. The goal is to expose the model to a wider variety of inputs, improving robustness and reducing overfitting.

For example, flipping an image of a cat horizontally still yields a valid cat image, so the model learns that flipping doesn't change the class.

Augmentation acts as a regularizer by introducing variability; it effectively trains the network on many "noisy" variants of the data, which helps generalization to new, unseen data.

9. What is Transfer Learning in the context of CNNs, and how are pre-trained models (ResNet, EfficientNet, MobileNet, etc.) used?

Answer:

Transfer learning means taking a model pre-trained on a large dataset (like ImageNet) and adapting it to a new task.

In CNNs, one typically takes a pre-trained network (e.g. ResNet, EfficientNet, MobileNet) and either freezes its early layers (feature extractor) and retrains the later layers for the new classes, or fine-tunes the whole network on the new data. For example, ResNet introduces residual (skip) connections to train very deep networks, and EfficientNet uses a principled scaling of width, depth, and resolution for efficiency.

These pre-trained architectures have already learned useful image features, so even with limited data you can achieve high accuracy by fine-tuning. Transfer learning accelerates training and often improves performance compared to training from scratch on the same dataset.

10. Explain RNNs (Recurrent Neural Networks) and how LSTM/GRU cells work.

Answer:

RNNs are neural networks designed for sequential data. In a standard (vanilla) RNN, the network maintains a hidden state that is updated step-by-step as it processes each element of the input sequence, allowing it to carry information across time. However, vanilla RNNs suffer from vanishing (and exploding) gradients, making it hard to learn long-range dependencies.

LSTM (Long Short-Term Memory) cells solve this by having a more complex internal structure: a memory cell and gates (input, forget, output) that control what information to write, keep, or output. This architecture lets the network learn to retain or forget information over long sequences. **GRU** (Gated Recurrent Unit) is a simpler gated variant (with update and reset gates) that also mitigates vanishing gradients. In short, LSTM and GRU are special RNN units that maintain and

Advanced

1. What is the Transformer architecture? Describe embeddings, positional encoding, attention, multi-head attention, encoder, and decoder.

Answer:

The **Transformer** is a neural architecture for sequence modelling that relies entirely on self-attention mechanisms, dispensing with recurrence. Key components:

Embeddings: Input tokens (words) are converted to continuous vectors (embeddings). Transformer also adds positional encodings (sinusoidal vectors) to these embeddings, so the model knows the order of tokens, since self-attention has no inherent notion of position.

Self-Attention: For each position in the input, attention computes a weighted sum of values at all positions, where weights come from similarity between a query (at that position) and keys (at other positions). This lets the model relate different parts of the sequence.

Multi-Head Attention: Several attention "heads" run in parallel, each learning to focus on different types of relationships. Their outputs are concatenated and projected. This allows the model to jointly attend to information from different representation subspaces.

Encoder & Decoder: A Transformer model typically has an encoder stack and a decoder stack. Each encoder layer has a self-attention sublayer and a feedforward sublayer; each decoder layer has self-attention, encoder-decoder attention (attending to encoder outputs), and feedforward. In tasks like translation, the encoder processes the source sentence, and the decoder generates the target sentence one token at a time, attending to the encoder's outputs.

In summary, Transformers use positional embeddings plus multi-headed self-attention to capture contextual relationships in parallel, followed by feedforward networks. This architecture is at the core of models like BERT and GPT.

2. Between ReLU and Sigmoid, which activation would you prefer for a hidden layer in a large MLP? Why?

Answer:

ReLU is preferred over **Sigmoid** for hidden layers in a large MLP.

Reasons:

- *Non-saturating gradients:* ReLU doesn't suffer from vanishing gradients for positive inputs, enabling deeper networks to learn effectively.
- *Computational efficiency*: ReLU is simple to compute: f(x) = max(0, x).
- *Sparse activation:* ReLU outputs zero for negative inputs, introducing sparsity which can improve generalization and computational efficiency.
- *Sigmoid issues:* Sigmoid saturates at both tails, leading to vanishing gradients; it also outputs only positive values, which can slow convergence due to non-zero-centered activations.

However, ReLU can suffer from dying units (neurons stuck at zero), for which Leaky ReLU or variants like GELU are used in practice.

3. You're building a model to detect fraudulent transactions. How would you handle imbalanced class labels during training?

Answer:

To handle imbalanced class labels in a fraud detection model, I'd use a combination of the following strategies:

Data-Level Approaches

- *Resampling Techniques:*

    - Oversampling minority class using methods like SMOTE (Synthetic Minority Oversampling Technique).
    - Undersampling majority class, possibly with techniques like Tomek links or NearMiss to avoid discarding informative samples.
    - Combine both (e.g., SMOTE + Tomek) for better balance.
- *Stratified Splitting:*
    - Ensure train/val/test splits preserve the class distribution using stratified sampling.

Algorithm-Level Approaches

- *Class Weights:*
    - Assign higher weight to the minority class in the loss function (e.g., weight parameter in CrossEntropyLoss or class_weight='balanced' in sklearn).
- *Custom Loss Functions:*
    - Use Focal Loss to down-weight easy examples and focus the model on hard, minority-class samples.

Model Evaluation

- Use appropriate metrics:
    - Avoid accuracy; instead use Precision, Recall, F1-score, ROC-AUC, and PR-AUC.
    - Focus especially on Recall (catching fraud) and Precision (minimizing false alarms).

Ensemble Methods

- Train models like Random Forest, XGBoost, or LightGBM, which handle class imbalance better and can use built-in scale_pos_weight or is_unbalance parameters.

Active Learning or Anomaly Detection

- In extreme imbalance scenarios, treat fraud as an anomaly detection problem.
- Use semi-supervised or unsupervised models like Isolation Forest, Autoencoders, or One-Class SVMs.

I'd evaluate multiple strategies experimentally and choose the combination that yields high recall with acceptable precision in production constraints.

4. You're tasked with building a real-time object detection system for autonomous drones. What trade-offs would you consider between accuracy and latency?

Answer:

In a real-time object detection system for autonomous drones, the primary trade-offs between accuracy and latency revolve around ensuring the system is both fast enough to react in real-time and accurate enough to avoid false positives/negatives. Key considerations:

- ***Model Complexity vs. Inference Speed:***
  - *Trade-off*: Deeper architectures like YOLOv7 or Faster R-CNN offer high accuracy but introduce latency. Lighter models (e.g., YOLOv5n, MobileNet-SSD) reduce latency but at a potential cost to precision and recall.
  - *Decision*: Choose the lightest model that meets the minimum acceptable accuracy threshold for the application.
- ***Input Resolution vs. Frame Rate:***
  - *Trade-off*: Higher-resolution inputs improve detection accuracy (especially for small objects), but slow down inference.
  - *Decision*: Optimize resolution dynamically based on the drone's speed and proximity to objects.
- ***Precision vs. Recall:***
  - *Trade-off*: High precision avoids false alarms (fewer unnecessary evasive maneuvers), while high recall ensures fewer missed detections (better safety).
  - *Decision*: Prioritize high recall for safety-critical tasks (e.g., obstacle detection), while tuning precision for less critical contexts.
- ***Hardware Constraints:***
  - *Trade-off*: Edge devices on drones have limited compute power and thermal budgets.
  - *Decision*: Use hardware-aware NAS (e.g., FBNet, Once-for-All) or quantized models (INT8) to optimize for the onboard accelerator (e.g., NVIDIA Jetson or Coral TPU).
- ***Latency Budget:***
  - Consider total system latency including sensor capture, pre-processing, inference, and post-processing.
  - Enforce a hard upper limit (e.g., <50ms) to maintain real-time performance.
- ***Failover Strategy:***

- Incorporate temporal smoothing or tracking (e.g., SORT, Deep SORT) to compensate for occasional missed detections while maintaining low inference latency.

In summary, I would deploy a hybrid pipeline: lightweight real-time detection on-device for immediate reaction, augmented with higher-accuracy models running asynchronously on ground stations or larger processors when available.

5. How does attention in transformers help capture long-range dependencies better than RNNs or CNNs in NLP? Give a use-case example.

Answer:

Attention in transformers enables direct access to all tokens in the input sequence at each layer, allowing the model to compute dependencies between distant words without sequential processing. Unlike RNNs, which process inputs step-by-step and struggle with vanishing gradients over long sequences, and CNNs, which rely on fixed-size kernels limiting context size, attention mechanisms compute pairwise relevance scores globally.

For example, in the sentence: *"The book that the professor who the student admired wrote was well-received."* Understanding that *"book"* is the subject of *"was well-received"* requires capturing dependencies across multiple clauses. Transformers can compute attention between "book" and "was" directly, regardless of how many tokens lie in between.

Use-case: In machine translation, when translating from German to English, verbs often appear at the end in German. Transformers allow the decoder to attend to relevant verbs that are far from the current token, preserving grammatical correctness better than RNN-based seq2seq models.

6. Design a model pipeline to classify medical X-rays, and include steps for ensuring regulatory compliance, reproducibility, and fairness.

Answer:

To classify medical X-rays, I'd design a pipeline that starts with de-identified, diverse datasets, ensuring HIPAA compliance and balanced representation across age, gender, and ethnicity. I'd preprocess the images (normalize, augment) and split data by patient ID to prevent leakage.

For modeling, I'd fine-tune a pretrained CNN like **DenseNet-121** and use **Grad-CAM** for explainability. I'd incorporate uncertainty estimation (e.g., MC Dropout) to flag ambiguous cases for radiologists. Model performance would be validated with stratified cross-validation and fairness metrics (e.g., per-group AUC).

To ensure reproducibility, I'd use **MLflow** for experiment tracking, **Docker** for containerization, and fix random seeds. Regulatory compliance would involve maintaining audit logs, generating model cards, and aligning with FDA SaMD practices. The deployed model would be monitored for drift, latency, and fairness, with a feedback loop for periodic retraining.

7. You're seeing overfitting despite using dropout and L2 regularization. What advanced techniques would you explore next?

Answer:

- ***Data Augmentation:*** For image, text, or audio data, apply advanced augmentation techniques to increase effective dataset diversity:
  - **Image**: Mixup, CutMix, RandAugment.
  - **Text**: Back-translation, contextual word replacement.
  - **Tabular**: SMOTE, conditional GAN-based synthesis.
- ***Label Smoothing:*** Prevent the model from becoming too confident by assigning soft targets (e.g., 0.9 for true class, 0.1 distributed across others). This acts as a regularizer on the output distribution.
- ***Early Stopping with Checkpoint Averaging:*** Beyond just halting training early, average model weights across several top-performing checkpoints (Stochastic Weight Averaging) to generalize better.
- ***Ensembling:*** Combine predictions from multiple models (or checkpoints) using techniques like bagging, snapshot ensembling, or test-time augmentation to reduce variance.

- ***Bayesian Approaches:*** Use **Bayesian Neural Networks** or **MC Dropout** at inference to model epistemic uncertainty, which can regularize the training process indirectly.
- ***Adversarial Training:*** Introduce adversarial examples during training (FGSM, PGD) to improve robustness, which often improves generalization.
- **Gradient Clipping / Sharpness-Aware Minimization (SAM):** SAM explicitly penalizes sharp minima in the loss landscape, encouraging flatter solutions that generalize better.
- ***Noise Injection:*** Add Gaussian noise to weights, activations, or inputs during training to regularize the model.
- ***Reduce Model Capacity or Prune:*** If the model is still overfitting, reduce the number of parameters or apply structured/unstructured pruning post-training.
- ***Cross-validation Monitoring:*** Use cross-validation instead of a static validation split to make the generalization metric more robust and representative.

These techniques are context-dependent, and I would evaluate them based on the data type, training dynamics, and failure modes observed in metrics.

8. Why is positional encoding essential in transformer architectures, and how does it affect context awareness in sequence models?

Answer:

Positional encoding is essential in transformer architectures because transformers are inherently permutation-invariant i.e. they have no built-in notion of sequence order due to the absence of recurrence or convolution. Without positional encoding, the model treats input tokens as a set, not a sequence, making it impossible to learn order-sensitive tasks like language modeling or translation.

Why It's Essential:

1. Injects Order Information: Positional encodings add unique position-based signals to input embeddings, allowing the model to distinguish between sequences like:
    a. "He hit the ball."
    b. "The ball hit him."
2. Enables Attention to Be Position-Aware: Self-attention computes interactions between tokens irrespective of position. Positional encoding modulates these interactions based on relative or absolute positions, helping the model to weigh nearby vs distant tokens appropriately.

How It Affects Context Awareness:

1. Absolute Position Awareness (e.g., Sinusoidal Encoding in the original Transformer):
    a. Encodes each token's position using a deterministic function.
    b. Allows the model to generalize to longer sequences (since sin/cos functions are periodic and continuous).
2. Learned Positional Embeddings (e.g., BERT, GPT):
    a. The model learns position vectors during training, which can be more flexible but may not extrapolate to longer sequences unless explicitly trained to do so.
3. Relative Positional Encodings (e.g., Transformer-XL, T5):
    a. Encode the relative distance between tokens instead of absolute positions.
    b. Helps with long-range dependencies and better generalization in contexts where relative order matters more than absolute position.

9. Describe how you'd evaluate the performance of a binary classification model in a high-stakes healthcare setting.

Answer:

To evaluate the performance of a binary classification model in a high-stakes healthcare setting, I would take a multi-dimensional approach focused on clinical impact, safety, and fairness—far beyond just accuracy.

First, I would prioritize **recall (sensitivity)** to ensure we minimize false negatives, which in healthcare could mean missing a critical diagnosis. At the same time, I'd monitor **precision** to avoid unnecessary treatments caused by false positives. The **F1 score** can help balance these two when appropriate. However, due to class imbalance often seen in medical data (e.g., rare diseases), I'd lean on **PR-AUC** over ROC-AUC to better reflect the model's effectiveness on the minority (positive) class.

Calibration is equally important—if the model predicts a 90% chance of disease, that should correspond to an actual 90% prevalence in similar patients. I'd use calibration curves and Brier scores to assess this. In addition, I'd run **threshold analysis**, tuning the decision boundary based on utility functions or clinical cost matrices, not arbitrary cutoffs like 0.5.

Beyond performance metrics, I'd conduct detailed **error analysis**, especially on false negatives, to identify patterns or biases in the model's failure modes. This leads into

fairness analysis—I'd measure group-wise metrics (e.g., TPR, FPR) across demographic slices to catch any equity issues, using fairness constraints if necessary.

Since clinical deployment involves unknowns, I'd assess **robustness to distribution shift** by validating on external datasets from different hospitals or regions. I'd also simulate **real-world human-AI interaction**, testing how clinicians use the model and whether it genuinely improves their decisions without causing alert fatigue or overtrust.

Finally, I'd ensure **explainability and auditability** using SHAP or similar tools, especially to meet regulatory standards and maintain clinician trust. In sum, evaluation in healthcare is about ensuring safe generalization, interpretability, and fairness—not just scoring well on a test set.

10. Suppose you're training a CNN on CIFAR-10 and your training accuracy improves, but test accuracy drops. What could be going wrong?

Answer:

This indicates overfitting—the model is learning patterns specific to the training data but failing to generalize.

Possible causes:

- Insufficient regularization (e.g., weak dropout, no weight decay).
- Lack of data augmentation—not exposing the model to varied inputs.
- Model too complex—too many parameters relative to dataset size.
- Early stopping not used—training continues beyond optimal generalization point.
- Data leakage—preprocessing applied differently on train/test sets.
- To fix: increase regularization, add data augmentation, reduce model size, or use early stopping.