PyTorch Specialization

Complete Quiz Study Guide

12 Quizzes • 120 Questions

Course 1: Deep Learning Fundamentals
Course 2: Intermediate PyTorch
Course 3: Advanced PyTorch

All Questions with Options, Answers & Reasoning

January 20, 2026

Contents

Part I
Course 1: Deep Learning Fundamentals

1 Quiz 1: PyTorch Basics

Topics: Loss Functions, Activation Functions, Epochs, Tensors, Broadcasting

Q1

What does a loss function measure in a neural network?

Correct Answer

The error between predictions and actual values

Reasoning

Loss function quantifies the difference (error) between model predictions and ground truth labels. It provides the signal for optimization - the model adjusts weights to minimize this error.

Q2

Why do neural networks need non-linear activation functions like ReLU?

Correct Answer

To enable the network to learn non-linear patterns

Reasoning

Without non-linear activations, stacking linear layers results in just another linear transformation (composition of linear functions is linear). ReLU introduces non-linearity, enabling the network to learn complex, non-linear patterns in data.

Q3

What is an epoch in the context of training neural networks?

Correct Answer

One complete pass through the entire training dataset

Reasoning

An epoch = one full iteration over the entire training dataset. If you have 1000 samples and batch size 100, one epoch = 10 batches. Multiple epochs allow the model to learn from all data repeatedly.

Q4

What does loss.backward() do in PyTorch?

Correct Answer

Calculates gradients to reduce the loss

Reasoning

loss.backward() performs backpropagation, computing gradients of the loss with respect to all parameters that have requires_grad=True. These gradients indicate how to adjust weights to reduce the loss.

Q5

What does dtype specify in a PyTorch tensor?

Correct Answer

The type of numbers stored in the tensor

Reasoning

dtype specifies the data type (float32, float64, int32, int64, etc.) of tensor elements. This affects memory usage, numerical precision, and compatibility with operations.

Q6

What is the result of: torch.tensor([2., 4., 6.]) / torch.tensor([1., 2., 3.]) + 5?

Correct Answer

tensor([7., 7., 7.])

Reasoning

Element-wise division first: [2/1, 4/2, 6/3] = [2, 2, 2]. Then scalar addition broadcasts: [2+5, 2+5, 2+5] = [7, 7, 7].

Q7

When you perform operations like multiplication or addition on tensors, what happens?

Correct Answer

Operations apply to each element in the tensor

Reasoning

PyTorch operations are element-wise by default (vectorized operations). This enables efficient parallel computation on GPUs without explicit loops.

Q8

What makes tensors different from regular Python lists for mathematical operations?

Correct Answer

Tensors are optimized for mathematical operations

Reasoning

Tensors leverage GPU acceleration and optimized C++/CUDA backends for fast math operations. They also support automatic differentiation, broadcasting, and are the fundamental data structure for deep learning.

Q9

What is the shape of a scalar tensor created by: torch.tensor(5.0).unsqueeze(0).squeeze()?

Correct Answer

torch.Size([])

Reasoning

Starting with scalar (shape []), unsqueeze(0) adds dimension → [1], then squeeze() removes ALL size-1 dimensions → [] (0-dimensional scalar tensor).

Q10

What is broadcasting in PyTorch?

Correct Answer

Expanding tensors to compatible shapes for operations

Reasoning

Broadcasting automatically expands smaller tensors to match larger tensor shapes for compatible operations. For example, adding a scalar to a matrix broadcasts the scalar to every element.

2 Quiz 2: Training & Evaluation

Topics: Normalization, Model Evaluation, MSE Loss, GPU Setup, Optimizers

Q1

Why do we normalize input data before training a neural network?

Correct Answer

To help the model train more effectively

Reasoning

Normalized data (mean≈0, std≈1) improves gradient flow, prevents vanishing/exploding gradients, and speeds convergence. It ensures all features contribute equally to learning.

Q2

Why should you use model(x) instead of model.forward(x) for inference?

Correct Answer

Using model() handles PyTorch's internal setup properly

Reasoning

model() invokes `__call__()` which runs registered hooks, handles nested modules, and calls forward() internally. Directly calling forward() skips these important mechanisms.

Q3

Why does Mean Squared Error (MSE) square the errors?

Correct Answer

To prevent cancellation and penalize larger errors

Reasoning

Squaring ensures all errors are positive (no cancellation of +/- errors) and disproportionately penalizes larger errors (quadratic penalty). This makes the model focus more on fixing big mistakes.

Q4

Why do we evaluate models on a separate test set?

Correct Answer

To see if the model learned generalizable patterns

Reasoning

Test set measures generalization to unseen data. Training accuracy alone can mask overfitting - a model might memorize training data but fail on new examples.

Q5

What is the correct way to set up device for GPU/CPU in PyTorch?

Correct Answer

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

Reasoning

Standard PyTorch idiom: checks CUDA availability, falls back to CPU if no GPU. This makes code portable across different hardware configurations.

Q6

What does the Flatten layer do in a neural network?

Correct Answer

Reshapes 2D images into 1D vectors

Reasoning

Linear layers expect 1D input. For MNIST, images are 28×28 (2D), so Flatten converts [batch, 1, 28, 28] → [batch, 784] for the fully connected layers.

Q7

Which optimizer is generally recommended as a good default choice?

Correct Answer

Adam

Reasoning

Adam combines momentum + adaptive learning rates (per-parameter). It works well across diverse problems without much hyperparameter tuning, making it an excellent default choice.

Q8

Should you shuffle data in DataLoader for training and testing?

Correct Answer

Shuffle training data, but not test data

Reasoning

Shuffling training data prevents learning order-dependent patterns and ensures varied batches. Test order doesn't affect metric computation, and keeping it fixed aids reproducibility.

Q9

What does torch.no_grad() do?

Correct Answer

Disables gradient tracking to save memory

Reasoning

torch.no_grad() disables autograd gradient computation, reducing memory usage and speeding up inference. Used during evaluation when gradients aren't needed.

Q10

What does model.eval() do?

Correct Answer

Changes how certain layers behave during evaluation

Reasoning

model.eval() switches layers like Dropout (disabled) and BatchNorm (uses running statistics instead of batch statistics) to evaluation behavior. Always use before inference.

3 Quiz 3: Data Pipelines

Topics: Custom Datasets, DataLoader, Transforms, Data Augmentation

Q1

In the lecture, why did three different teams building the flower identification app get such different results even though they used the same model architecture?

A. They trained for different numbers of epochs

B. **They handled and prepared the data differently** ✓

C. They used different batch sizes

D. They used different loss functions

Reasoning

Data preparation (preprocessing, augmentation, splitting) has huge impact on model performance, even with identical architectures. The same model can succeed or fail based purely on data handling.

Q2

Three categories of data pipeline problems are introduced: access, quality, and efficiency. Which category involves loading data in batches rather than one item at a time?

A. Quality problems

B. Architecture problems

C. **Efficiency problems** ✓

D. Access problems

Reasoning

Loading data in batches rather than one at a time is an efficiency optimization to speed up training. Batching enables parallel processing and better GPU utilization.

Q3

In the __init__ method of the Oxford Flowers dataset, why are image paths stored but the actual images not loaded into memory?

A. Images must be transformed before loading

B. PyTorch doesn't allow loading images in __init__

C. Loading images in __init__ would be too slow

D. **Loading all images at once would use too much memory** ✓

Reasoning

Storing paths in __init__ and loading images lazily in __getitem__ prevents memory overflow with large datasets. This is the standard lazy loading pattern for efficient data handling.

Q4

The Oxford Flowers dataset labels range from 1 to 102, but PyTorch expects labels from 0 to 101. Where should you fix this indexing issue?

A. **In the __init__ method by subtracting 1 from all labels** ✓

B. In the __getitem__ method when returning each label

C. PyTorch will automatically adjust the labels

D. In the __len__ method when reporting dataset size

Reasoning

Fix indexing in __init__ by subtracting 1 from all labels **once**, rather than in __getitem__ which runs thousands of times during training. This is more efficient.

Q5

Why does the Normalize transform help neural networks learn more effectively?

A. **It spreads pixel values for better gradient-based learning** ✓

B. It reduces the file size of images

C. It converts images from PIL format to tensors

D. It makes all images the same brightness

Reasoning

Normalize centers data around 0 with unit variance, spreading values optimally for gradient-based optimization. This improves gradient flow and training stability.

Q6

ToTensor() performs three important operations. Which of the following is NOT one of them?

A. **Normalizes pixel values using mean and standard deviation** ✓

B. Converts the image from a PIL object to a PyTorch tensor

C. Scales pixel values from 0-255 to 0-1

D. Rearranges dimensions to [channels, height, width]

Reasoning

ToTensor() does: PIL→tensor, scales 0-255→0-1, rearranges to [C,H,W]. Mean/std normalization is a **separate** Normalize() transform that must be applied explicitly.

Q7

What's wrong with loading data from a large file inside the __getitem__ method instead of in __init__?

A. It prevents the DataLoader from shuffling the data

B. It causes the data to be loaded into memory all at once and crash

C. **The entire file is read for every sample access** ✓

D. PyTorch doesn't allow file reading in __getitem__

Reasoning

Reading from a large file in __getitem__ may re-read/re-parse the entire file for each sample access—extremely inefficient. Load metadata once in __init__, access individual items in __getitem__.

Q8

When you iterate through a DataLoader, what does each iteration give you?

A. **A batch of images and a batch of labels** ✓

B. A single image and its label

C. Just the images—you need to fetch labels separately

D. The entire dataset at once

Reasoning

DataLoader returns batched tensors: a batch of inputs and a batch of corresponding labels together. This is the standard pattern for training loops.

Q9

Why should you apply data augmentation to the training set but not to the validation set?

A. PyTorch doesn't support augmentation on validation data

B. Augmentation only works on images the model has already seen

C. **Validation needs consistent data to measure improvements reliably** ✓

D. Augmentation is too slow to apply during validation

Reasoning

Augmentation adds randomness for training diversity; validation data should be consistent across epochs to reliably measure model improvement. Random validation would give noisy metrics.

Q10

In the robust dataset implementation, what happens when __getitem__ encounters a corrupted or problematic image?

A. **It logs the error and retrieves another valid image** ✓

B. It returns None and continues to the next batch

C. It raises an exception and stops training

D. It replaces the corrupted image with a blank tensor

Reasoning

A robust implementation catches exceptions and returns a different valid sample to keep training going without crashing. This handles real-world data quality issues gracefully.

4 Quiz 4: CNNs & Model Architecture

Topics: Convolutional Layers, Pooling, nn.Module, nn.Sequential

Q1

What is the key advantage of having CNNs learn their own filters rather than using hand-designed filters?

A. Hand-designed filters need extra storage for configuration metadata

B. Learned filters work for any image classification task without retraining

C. **The model discovers task-specific patterns automatically** ✓

D. Hand-designed filters are too slow to compute

Reasoning

Learned filters adapt to the specific task, automatically discovering optimal patterns for that dataset rather than relying on generic hand-designed features like edge detectors.

Q2

What is the sequence of operations that happens when a filter performs a convolution at one position in an image?

A. Multiply all pixel values together, then add the filter values

B. Add filter values to pixel values, then multiply, then average

C. **Multiply corresponding values element-wise, then sum the products** ✓

D. Average the pixel values first, then multiply by the filter

Reasoning

Convolution operation: element-wise multiply filter values with corresponding pixel values in the receptive field, then sum all the products to get one output value.

Q3

A CNN has two convolutional layers, each followed by MaxPool2d with kernel_size=2. If the input image is 28×28 pixels, what size are the feature maps after both pooling layers?

A. **7×7** ✓

B. 14×14

C. 3×3

D. 28×28

Reasoning

$28 \times 28 \xrightarrow{\text{MaxPool(2)}} 14 \times 14 \xrightarrow{\text{MaxPool(2)}} 7 \times 7$. Each MaxPool with kernel=2 halves the spatial dimensions.

Q4

In a convolutional layer with in_channels=3 and out_channels=32, what does the 32 represent?

A. The layer processes 32 images at once in a batch

B. The output image will be 32×32 pixels

C. **The layer learns 32 different filters** ✓

D. The layer creates 32 copies of each input channel

Reasoning

out_channels=32 means the layer learns 32 separate filters, each producing one output feature map. Each filter detects a different pattern.

Q5

What is a key limitation of using nn.Sequential to define a model in PyTorch?

A. **Sequential doesn't support conditionals, loops, or branching** ✓

B. Sequential cannot be used with the Adam optimizer

C. Sequential models train slower than models using nn.Module

D. Sequential models cannot use convolutional layers

Reasoning

nn.Sequential is a simple linear stack of layers. For conditionals, loops, or branching architectures (like ResNet skip connections), you must subclass nn.Module and define custom forward().

Q6

What is the core difference between how static graphs and PyTorch's dynamic graphs work?

A. Static graphs require defining everything in one block while dynamic graphs split definitions across methods

B. Static graphs run faster but dynamic graphs use less memory

C. Static graphs only work with images while dynamic graphs work with any data type

D. **Static graphs are defined once before training, while dynamic graphs are built fresh eac** ✓

Reasoning

PyTorch uses dynamic graphs rebuilt on each forward pass, unlike static graphs (TensorFlow 1.x style) defined once before execution. This enables flexible architectures with conditionals and loops.

Q7

Why does PyTorch separate the __init__ and forward methods instead of combining them into a single method?

A. The __init__ method runs faster than the forward method

B. **__init__ defines the architecture, forward defines how data flows through it** ✓

C. It makes the code easier to read by splitting it into smaller sections

D. It's a Python convention that all classes must follow

Reasoning

__init__ creates and initializes layers (architecture definition), forward() defines how input data flows through those layers (computation graph). Separation enables flexible, dynamic architectures.

Q8

When defining a custom nn.Module, what is one benefit of using nn.Sequential to organize its layers?

A. Sequential automatically chooses the best layer order for your data

B. Sequential allows you to use conditionals and loops in your model

C. **You don't need to manually call each layer in forward()** ✓

D. Sequential makes the model train faster

Reasoning

nn.Sequential automatically chains layer calls in order, so you just call self.features(x) instead of manually calling each layer: conv1, relu1, pool1, conv2, etc.

Q9

What is the standard way to count the total number of parameters in a PyTorch model?

A. print(model.parameters())

B. model.count_parameters()

C. len(model.parameters())

D. **sum(p.numel() for p in model.parameters())** ✓

Reasoning

model.parameters() returns an iterator of parameter tensors; p.numel() gives element count per tensor; sum() totals all elements across all parameters.

Q10

What is the key difference between model.children() and model.modules() when inspecting a PyTorch model?

A. children() is faster but modules() is more detailed

B. **children() shows only direct children, modules() shows all nested modules recursively** ✓

C. modules() only works with Sequential blocks while children() works with all layers

D. children() shows parameters while modules() shows layers

Reasoning

children() returns only immediate child modules (one level); modules() recursively returns ALL nested modules including the model itself and deeply nested layers.

Part II
Course 2: Intermediate PyTorch

5 Quiz 5: Hyperparameter Optimization (Optuna)

Topics: Flexible CNN, Parameter Importance, Model Selection

Q1

Which of the following best describes the design principle behind parameterizing a model architecture for use with tools like Optuna?

A. It ensures Optuna trials are deterministic by locking random seeds

B. It simplifies the forward pass logic in PyTorch's nn.Module

C. **It supports flexible experimentation by testing multiple architectural designs** ✓

D. It enforces model reproducibility through fixed layer structures

Reasoning

Parameterizing architecture allows Optuna to systematically explore different configurations (number of layers, layer sizes, dropout rates) for flexible experimentation and automated architecture search.

Q2

True or False: Both dropout rate and learning rate are regularization hyperparameters.

A. **False** ✓

B. True

Reasoning

Dropout is a regularization technique (prevents overfitting by randomly dropping neurons). Learning rate is an **optimization** hyperparameter that controls step size during gradient descent—not regularization.

Q3

Complete the code to allow dynamic dropout selection between 0.1 and 0.5:

```
def objective(trial):
    n_layers = trial.suggest_int("n_layers", 1, 3)
    fc_size = trial.suggest_int("fc_size", 64, 512, step=64)
    # What goes here?
```

A. dropout_rate = trial.sample_float("dropout_rate", 0.1, 0.5)

B. dropout_rate = FlexibleCNN.suggest("dropout_rate", 0.1, 0.5)

C. **dropout_rate = trial.suggest_float("dropout_rate", 0.1, 0.5)** ✓

D. dropout_rate = trial.suggest_categorical("dropout_rate", [0.1, 0.2, 0.3, 0.4, 0.5])

Reasoning

suggest_float() is for continuous float ranges in Optuna. suggest_categorical would only give discrete predefined values; sample_float is not a valid method.

Q4

Which of the following is a best practice when designing an optimization pipeline using Optuna?

A. Define all hyperparameters regardless of their dependencies

B. **Structure the search space to reflect dependencies between architectural components** ✓

C. Include only training hyperparameters like learning rate and batch size

D. Test all possible combinations of hyperparameters exhaustively

Reasoning

Best practice: only define hyperparameters for components that exist in that trial configuration (conditional search space). For example, don't tune layer 3's size if n_layers=2.

Q5

Why might you prioritize evaluating model size and inference time in addition to accuracy?

A. To better understand the GPU temperature behavior

B. These values are easier to measure than accuracy

C. **To ensure your model meets real-world deployment constraints like speed and memory** ✓

D. Accuracy always guarantees generalization anyway

Reasoning

Real-world deployment has practical constraints: latency requirements, memory limits, edge device capabilities. A highly accurate but slow/large model may be unusable in production.

Q6

Why is the classifier initialized inside the forward method in a flexible CNN architecture?

A. To reset the classifier weights every forward pass

B. To delay initialization until the model is fully trained

C. **Because x.size(1) gives the dynamic flattened feature size, which depends on earlier lay** ✓

D. To prevent the classifier from being saved in the model checkpoint

Reasoning

Classifier input size depends on the dynamic architecture (number of conv layers, kernel sizes, pooling). The flattened feature size is only known after running the convolutional layers, so lazy initialization is needed.

Q7

What insight can you gain by analyzing parameter importances after an Optuna optimization run?

A. You can identify which training samples contributed most to performance

B. **You can determine which hyperparameters had the most influence on the objective** ✓

C. You can determine the exact optimal hyperparameter combination

D. You can identify which hyperparameters caused underfitting

Reasoning

Parameter importance analysis reveals which hyperparameters most influence the objective function. This helps prioritize future search efforts on the most impactful parameters.

Q8

How does the parallel coordinate plot help in understanding the results of a hyperparameter search?

A. It shows epoch-wise performance trends

B. It plots validation loss across models

C. **It highlights optimal combinations of hyperparameter values using color intensity** ✓

D. It ranks models based on final test accuracy

Reasoning

Parallel coordinate plots show lines connecting hyperparameter values across axes, with color intensity indicating performance quality. You can visually identify which value combinations lead to good results.

Q9

In a hyperparameter tuning process, what is the primary role of a baseline model?

A. To automate model selection based on predefined criteria

B. **To serve as a reference point against which performance improvements can be measure** ✓

C. To maximize model complexity during early experimentation

D. To confirm that your model is generalizing well to new data

Reasoning

A baseline model provides a reference point to quantify whether optimized models actually improve performance. Without a baseline, you can't tell if your tuning efforts are helping.

Q10

Given the formula Score = 0.5 × Accuracy + 0.3 × Memory + 0.2 × Time, and these normalized metrics:

Model	Accuracy	Memory	Time
A	0.8	0.7	0.3
B	0.6	0.7	0.9
C	0.7	0.6	0.95
D	0.75	0.8	0.2

Which model wins?

A. **C** ✓

B. D

C. B

D. A

Reasoning

Calculate scores: A = 0.5(0.8)+0.3(0.7)+0.2(0.3) = 0.67. B = 0.5(0.6)+0.3(0.7)+0.2(0.9) = 0.69. **C = 0.5(0.7)+0.3(0.6)+0.2(0.95) = 0.72**. D = 0.5(0.75)+0.3(0.8)+0.2(0.2) = 0.655. Model C has the highest score.

6 Quiz 6: TorchVision & Transfer Learning

Topics: PIL, Transforms, Pretrained Models, Semantic Segmentation

Q1

What is one key advantage of using domain-specific libraries like TorchVision for vision tasks?

A. **They offer optimized tools, datasets, and pretrained models tailored for vision workflow** ✓

B. They provide generic tools applicable to any data type

C. They eliminate the need for training models entirely

D. They automatically improve accuracy of any model

Reasoning

Domain-specific libraries like TorchVision provide specialized, optimized tools (transforms, datasets, pretrained models) specifically designed for vision tasks, saving development time and ensuring best practices.

Q2

Which library does TorchVision rely on for image file I/O and manipulation before tensor conversion?

A. seaborn

B. NumPy

C. **PIL (Pillow)** ✓

D. OpenCV

Reasoning

TorchVision relies on PIL (Pillow) for image I/O and manipulation before converting images to tensors. PIL is the standard Python imaging library integrated with TorchVision transforms.

Q3

What is the primary goal of transfer learning?

A. To combine multiple model architectures into a single ensemble

B. To reduce the computational cost of inference only

C. **To leverage knowledge gained from pre-training on large datasets** ✓

D. To train models from scratch more efficiently

Reasoning

Transfer learning leverages pretrained weights from large datasets (like ImageNet) to jumpstart learning on new tasks with less data. The model transfers learned features to the new domain.

Q4

What will draw_bounding_boxes() do when called?

A. It runs object detection and returns the predicted boxes

B. **It returns a new image tensor with boxes and labels drawn on it** ✓

C. It modifies the original image tensor in place

D. It saves the image with bounding boxes to disk

Reasoning

draw_bounding_boxes() returns a NEW tensor with boxes/labels drawn; it doesn't modify in-place, run detection, or save to disk. It's purely a visualization utility.

Q5

Why would you apply RandomResizedCrop() instead of just RandomCrop()?

A. To ensure all crops are center-aligned after resizing

B. To crop images at a random location but with a fixed aspect ratio

C. To normalize pixel intensity ranges after cropping

D. **To introduce random cropping along with random resizing, simulating scale variation** ✓

Reasoning

RandomResizedCrop applies random scale changes AND random cropping, simulating objects appearing at different sizes/distances. This is powerful data augmentation for scale invariance.

Q6

Why is it important to apply Resize() before CenterCrop() or RandomCrop()?

A. To convert the image to grayscale first

B. **To ensure the image is large enough to safely crop to the target size** ✓

C. To change the image color depth

D. To remove background artifacts before cropping

Reasoning

Resize ensures image dimensions are $\geq$ crop size. Cropping an image smaller than the target size would fail or produce unexpected results.

Q7

Where should Normalize() typically be applied in an image transform pipeline?

A. Before converting the image to a tensor

B. Before loading the image from disk

C. **Immediately after converting to a tensor with ToTensor()** ✓

D. After training is complete

Reasoning

Normalize requires tensor input (operates on float tensors), so it must come after ToTensor() in the transform pipeline. ToTensor first, then Normalize.

Q8

What output does a pretrained DeepLabV3 semantic segmentation model produce?

A. A tensor of class scores for the entire image

B. A single prediction of what class is in the image

C. A list of bounding boxes around detected objects

D. **A per-pixel tensor indicating the predicted class for each pixel** ✓

Reasoning

DeepLabV3 is a semantic segmentation model outputting per-pixel class predictions—a dense prediction where every pixel gets classified (not bounding boxes or single class).

Q9

Fill in the missing transform for using a pretrained segmentation model:

```
transform = transforms.Compose([
    transforms.ToTensor(),
    -----------
])
```

A. **transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])** ✓

B. transforms.CenterCrop(224)

C. transforms.Grayscale()

D. transforms.Resize((32, 32))

Reasoning

Pretrained models expect ImageNet normalization: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]. This matches the preprocessing used during original training.

Q10

Why is unsqueeze(0) often used after converting an image to a tensor for model input?

A. **It adds a batch dimension so the input shape is [1, C, H, W]** ✓

B. It adds a color channel to grayscale images

C. It flattens the image for matrix multiplication

D. It ensures the pixel values are between 0 and 1

Reasoning

Models expect 4D input [batch, channels, height, width]. unsqueeze(0) adds the batch dimension to transform a single image from [C,H,W] to [1,C,H,W].

7 Quiz 7: NLP Fundamentals

Topics: Tokenization, Embeddings, BERT, Padding, OOV, NER

Q1

Why is lowercasing commonly applied during manual tokenization in NLP preprocessing?

A. **To reduce vocabulary size by treating "Word" and "word" as the same token** ✓

B. To improve model precision on named entities

C. To encode sentiment information in the text

D. To truncate long tokens to a maximum length

Reasoning

Lowercasing merges "Word", "WORD", "word" into a single token, reducing vocabulary size and sparsity. This simplifies the model but may lose case-based information (proper nouns, acronyms).

Q2

When using nn.EmbeddingBag instead of nn.Embedding, what key advantage does it provide?

A. **It automatically pools (sums/means) embeddings across tokens in one operation** ✓

B. It eliminates the need for tokenization entirely

C. It supports static but not dynamic embeddings

D. It automatically generates attention masks

Reasoning

nn.EmbeddingBag computes sum/mean of embeddings in one efficient operation, vs separate embedding lookup + pooling steps. Useful for bag-of-words style representations.

Q3

What is the main purpose of padding sequences in a batch for NLP?

A. **To ensure all sequences in a batch have the same length for tensor operations** ✓

B. To convert words into one-hot vectors

C. To remove irrelevant tokens from sequences

D. To truncate sequences that are too long

Reasoning

Batched tensor operations require uniform dimensions. Padding ensures all sequences have the same length so they can be stacked into a single tensor for parallel processing.

Q4

When using pretrained embeddings, why would you "freeze" the embedding weights?

A. To improve tokenizer efficiency

B. To automatically generate attention masks

C. **To prevent gradient updates and retain the pretrained semantic representations** ✓

D. To reduce GPU memory usage during inference only

Reasoning

Freezing embedding weights (requires_grad=False) prevents gradient updates, preserving the learned semantic knowledge from large-scale pretraining. Useful when you have limited data.

Q5

Which is NOT a typical preprocessing step before sending input to a pretrained BERT model?

A. Lowercasing text (for uncased models)

B. Tokenizing into subwords or word pieces

C. Padding sequences to a uniform length

D. **Converting token IDs back into strings** ✓

Reasoning

Preprocessing converts strings TO tokens/IDs for model input. Converting tokens back to strings is the reverse operation (decoding output), not preprocessing.

Q6

Which method is commonly used to handle Out Of Vocabulary (OOV) words?

A. Dropping all OOV words from the dataset entirely

B. **Using subword tokenization (BPE, WordPiece) to break unknown words into known pi** ✓

C. Using rule-based systems to manually define meanings

D. Simply ignoring the issue during training

Reasoning

Subword tokenization (BPE, WordPiece) handles OOV by breaking unknown words into known subword units. Even unseen words can be represented as combinations of known pieces, plus [UNK] as fallback.

Q7

What happens when you apply a BERT tokenizer with padding=True, truncation=True, return_tensors="pt"?

A. It converts text into character-level tensors

B. It automatically trains a model on the input

C. **It outputs PyTorch tensors with padded/truncated input_ids and attention_mask** ✓

D. It removes all unknown tokens entirely from the input

Reasoning

BERT tokenizer with those arguments returns a dictionary with input_ids and attention_mask as PyTorch tensors, properly padded to uniform length and truncated if too long.

Q8

What is the purpose of nn.Embedding in a text classifier?

A. It directly predicts class probabilities from token indices

B. **It maps discrete token indices to dense vectors that capture semantic similarity** ✓

C. It normalizes embeddings to unit vectors automatically

D. It applies padding and truncation to input sequences

Reasoning

nn.Embedding maps discrete token indices to continuous dense vectors in a semantic space where similar words are closer together. It's a learnable lookup table.

Q9

What is Named Entity Recognition (NER)?

A. A system that evaluates sentiment polarity in text

B. A method to generate abstractive text summaries

C. A technique for translating text between languages

D. **A process that identifies and classifies entities like people, organizations, and locations** ✓

Reasoning

NER (Named Entity Recognition) identifies and classifies named entities in text into predefined categories: PERSON, ORGANIZATION, LOCATION, DATE, etc.

Q10

What key aspect should you consider when selecting pretrained word embeddings for your task?

A. **Vocabulary size and domain coverage** ✓

B. Training batch size used originally

C. Number of epochs in original training

D. Total training duration in hours

Reasoning

Vocabulary coverage is critical for pretrained embeddings. Domain mismatch (e.g., medical text with general embeddings) leads to high OOV rates and poor performance.

8 Quiz 8: Performance Optimization

Topics: DataLoader, Profiler, Mixed Precision, Gradient Accumulation, Lightning

Q1

What is the main goal when optimizing batch size in DataLoader?

A. Always use the largest possible batch size

B. Set batch size equal to the number of DataLoader workers

C. **Find a balance that maximizes GPU utilization while fitting within memory limits** ✓

D. Use the smallest batch size to reduce memory usage

Reasoning

Batch size optimization balances GPU utilization (larger = better parallelism) against memory limits (too large = OOM error). Find the sweet spot for your hardware.

Q2

What is pinned memory in PyTorch DataLoader?

A. Memory that is shared between CPU and GPU simultaneously

B. **A special region of RAM that stays fixed in place, enabling faster GPU data transfer** ✓

C. Memory that is automatically compressed to save space

D. Memory that is automatically freed after each batch

Reasoning

Pinned (page-locked) memory stays fixed in RAM, enabling faster DMA (Direct Memory Access) transfers to GPU without paging overhead. Set pin_memory=True for speedup.

Q3

What does the prefetch_factor parameter control in PyTorch DataLoader?

A. **How many batches each worker preloads into memory ahead of training consumption** ✓

B. The size of each batch in number of samples

C. Whether to use pinned memory or regular memory

D. The total number of worker processes to spawn

Reasoning

prefetch_factor controls how many batches each DataLoader worker prepares ahead of time. Higher values mean more data ready when training needs it, reducing wait time.

Q4

What is the default value of prefetch_factor in PyTorch DataLoader?

A. Equal to the number of workers

B. **2** ✓

C. 4

D. 1

Reasoning

PyTorch DataLoader default prefetch_factor is 2, meaning each worker preloads 2 batches ahead. This is a reasonable default for most use cases.

Q5

What is the primary goal when optimizing training loops?

A. Reduce the number of training epochs needed

B. **Make models train faster and more efficiently without sacrificing accuracy** ✓

C. Use as much GPU memory as possible

D. Increase model accuracy at any computational cost

Reasoning

Training optimization aims for faster/more efficient training while maintaining model accuracy and quality. Speed without quality loss is the goal.

Q6

Which question can the PyTorch Profiler help answer?

A. What batch size will give the best accuracy?

B. What is the optimal learning rate for my model?

C. **Which operations are consuming the most time during training?** ✓

D. How many epochs should I train for?

Reasoning

PyTorch Profiler identifies bottlenecks by showing which operations consume the most time—CPU ops, GPU kernels, data loading, memory transfers. Essential for optimization.

Q7

In the Lightning profiler configuration, what does wait=2, warmup=2, active=10 mean?

A. Wait 2 epochs, warm up for 2 epochs, then profile for 10 epochs

B. Use 2 workers, 2 warmup steps, and profile 10 different models

C. **Skip 2 batches, warm up profiler for 2 batches, then actively record for 10 batches** ✓

D. Profile every 2nd batch for a total of 10 measurements

Reasoning

Profiler schedule operates per-batch: wait=skip first N batches, warmup=start profiler but don't record, active=record detailed traces. This avoids profiling cold-start overhead.

Q8

What does mixed precision training primarily aim to achieve?

A. Increase model accuracy by using higher numerical precision

B. **Lower memory usage and faster computation while maintaining training stability** ✓

C. Eliminate the need for gradient computation entirely

D. Reduce the total number of model parameters

Reasoning

Mixed precision (FP16 compute, FP32 master weights) reduces memory footprint and leverages tensor cores for faster computation, while maintaining numerical stability through loss scaling.

Q9

How does gradient accumulation work?

A. It automatically adjusts the learning rate based on gradient magnitude

B. It reduces the number of gradients computed per batch

C. It stores gradients from previous epochs for comparison

D. **It sums gradients over multiple small batches before performing one optimizer step** ✓

Reasoning

Gradient accumulation sums gradients over N small batches, then updates weights once—simulating a larger effective batch size without requiring more GPU memory per batch.

Q10

What is the main advantage of PyTorch Lightning's architectural approach?

A. Lightning uses less memory than vanilla PyTorch

B. Lightning only works with specific predefined model types

C. **Lightning enforces clean separation between model logic and training infrastructure** ✓

D. Lightning always runs faster than equivalent PyTorch code

Reasoning

Lightning architecture separates model logic (LightningModule: forward, training_step) from training infrastructure (Trainer: loops, logging, checkpointing), improving code organization and reusability.

Part III
Course 3: Advanced PyTorch

9 Quiz 9: Advanced Architectures

Topics: ModuleList, Siamese Networks, ResNet, DenseNet

Q1

You want to create a model with a variable number of layers, so you store them in a Python list. What is the primary problem with this approach?

A. Layers stored in Python lists can't be moved to GPU with .to(device)

B. **The layers won't be registered, so they won't train or save properly** ✓

C. The model will run too slowly due to Python overhead

D. You can't iterate through Python lists in the forward method

Reasoning

Python lists aren't registered by PyTorch's module system. Layers won't be tracked for training (no gradients), saving (state_dict), or device transfer. Use nn.ModuleList instead to properly register dynamic layers.

Q2

How does PyTorch handle backpropagation when a model uses conditional execution (if-statements in forward)?

A. Conditional execution isn't possible in PyTorch models

B. **It builds a fresh computation graph for whichever branch was actually taken** ✓

C. It requires you to manually specify which branch was taken for gradients

D. It averages the gradients from all possible branches

Reasoning

PyTorch's dynamic computation graph is built on-the-fly during each forward pass, only including operations that were actually executed. This naturally handles conditionals—only the taken branch is in the graph.

Q3

In a Siamese network, parameter sharing means both branches use the same layers. How does this affect training?

A. It prevents back-propagation through the second branch entirely

B. **Gradients from both branches accumulate and update the same shared weights**

✓

C. It disables the contrastive loss margin calculation

D. It doubles the total number of learnable parameters

Reasoning

Siamese networks share weights between branches. During backprop, gradients from both forward passes accumulate on the same parameters, effectively doubling the gradient signal from each pair.

Q4

What is a residual connection (skip connection) in a ResNet block?

A. A connection that only activates when gradients become too small

B. A gating mechanism that selectively keeps or discards features

C. **A direct addition of the original input to the block's transformed output** ✓

D. A separate pathway that processes input through additional layers

Reasoning

Residual/skip connection adds input directly to block output: $y = F(x) + x$. This enables identity mapping and helps gradients flow through very deep networks.

Q5

What does the growth rate hyperparameter control in a DenseNet?

A. The size of spatial dimensions after each dense layer

B. The total number of dense blocks in the network

C. The compression factor used in transition layers

D. **How many new feature map channels each dense layer adds** ✓

Reasoning

DenseNet growth rate k specifies how many new feature map channels each dense layer contributes to the concatenated output. If $k = 32$, each layer adds 32 new channels.

Q6

What architectural pattern does a SelectiveForward module (with "if hard_example") illustrate?

A. Parameter sharing between branches

B. Skip connections for gradient flow

C. Dynamic model creation at runtime

D. **Conditional execution—different paths based on input conditions** ✓

Reasoning

The if-statement selects different network paths at runtime based on input conditions—this is conditional execution, a key pattern enabled by PyTorch's dynamic graphs.

Q7

During backpropagation through a residual block, why does the gradient include an extra additive term?

A. It amplifies small gradients to speed up early layer training

B. It allows the network to dynamically skip poorly performing layers

C. **The skip connection provides a direct gradient path, preventing vanishing gradients** ✓

D. It helps balance gradient magnitudes across layers of different depths

Reasoning

Skip connection gradient: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot (\frac{\partial F}{\partial x} + 1)$. The +1 term provides a direct gradient highway, preventing vanishing gradients in deep networks.

Q8

Which line correctly initializes the convolution for a DenseNet TransitionLayer that compresses channels?

A. self.conv = nn.Linear(in_channels, out_channels)

B. **self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, bias=Fals** ✓

C. self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=2, padding=1)

D. self.conv = nn.Conv2d(out_channels, in_channels, kernel_size=1)

Reasoning

TransitionLayer uses 1×1 convolution to reduce (compress) channels without changing spatial dimensions. AvgPool handles spatial downsampling separately. Bias=False is typical with BatchNorm.

Q9

In a ResidualBlock, when must a downsample layer be supplied to match dimensions?

A. Only when in_channels < out_channels and stride equals 1

B. Only when stride == 1 and in_channels == out_channels

C. **Whenever either stride ≠ 1 or in_channels ≠ out_channels** ✓

D. Never—the element-wise addition works for any tensor combination

Reasoning

For addition $x + F(x)$, tensor shapes must match exactly. A downsample layer (1×1 conv with stride) adjusts dimensions when stride≠1 OR channel counts differ.

Q10

DenseNet concatenates outputs from all previous layers, causing channels to grow. What architectural component prevents this from exploding?

A. **TransitionLayers with 1×1 convolution to compress/reduce channels** ✓

B. Skip connections that use element-wise addition instead

C. Shared weights that reuse channels from earlier layers

D. Batch normalization that drops 50% of channels randomly

Reasoning

DenseNet TransitionLayers use 1×1 convolution with compression factor (e.g., 0.5) to reduce accumulated channels between dense blocks, keeping the network manageable.

10 Quiz 10: Diffusion Models & Interpretability

Topics: CNN Receptive Fields, Grad-CAM, Saliency Maps, Stable Diffusion

Q1

In the forward diffusion process, what happens to an image as timestep t increases toward the final step T?

A. Noise is gradually removed so the image becomes progressively sharper

B. Pixels are randomly shuffled but their intensity values remain unchanged

C. All noise is added at once in a single step at timestep T

D. **Noise is added incrementally until the image becomes nearly pure Gaussian noise** ✓

Reasoning

Forward diffusion adds Gaussian noise incrementally over T timesteps. At $t = T$, the image is approximately $\mathcal{N}(0, 1)$ pure noise. The reverse process learns to denoise.

Q2

A max-pooling layer with kernel=2 and stride=2 is inserted after a convolution. What effect does this have on the feature maps?

A. Halves the number of channels and doubles both height and width

B. Keeps spatial dimensions unchanged but reduces information density

C. Doubles the number of channels and halves the height×width product

D. **Keeps channel count unchanged and halves both height and width dimensions** ✓

Reasoning

MaxPool(kernel=2, stride=2) halves height and width by taking max in each 2×2 window. Channel count is unaffected—pooling operates spatially, not across channels.

Q3

Saliency maps and Grad-CAM both rely on gradients, yet they often highlight different regions of an image. Why?

A. Grad-CAM filters out all negative gradient contributions

B. **Saliency uses pixel-level input gradients; Grad-CAM aggregates over feature map regio** ✓

C. Saliency uses gradients from all layers; Grad-CAM only from early layers

D. They compute gradients with respect to different loss functions

Reasoning

Saliency maps compute gradients w.r.t. input pixels (fine-grained, noisy). Grad-CAM uses gradient-weighted activations from a conv layer (coarse regions, more interpretable).

Q4

If an engineer swaps the order to MaxPool first, then Conv (instead of Conv first, MaxPool second), what happens to the effective receptive field?

A. It becomes undefined because convolution cannot follow pooling

B. It is cut in half because pooling discarded spatial information

C. It stays the same (e.g., 3×3) since kernel size is unchanged

D. **It effectively doubles in each spatial dimension because pooling downsampled first** ✓

Reasoning

Pooling before conv means the convolution operates on a downsampled feature map. Each position in the pooled map already represents a 2×2 region, so the conv's receptive field in the original image is larger.

Q5

When generating images with Stable Diffusion, increasing num_inference_steps from 20 to 50 will most likely:

A. Cause the image to become over-processed and lose natural details

B. Increase variety and randomness in the generated output

C. **Produce sharper, higher-fidelity images at the cost of longer generation time** ✓

D. Make the model follow the text prompt more literally

Reasoning

More inference steps = finer denoising trajectory = higher quality/sharper images. The tradeoff is proportionally longer generation time. Quality saturates eventually.

Q6

What is the purpose of the negative_prompt argument in Stable Diffusion pipelines?

A. **It tells the model what content or styles to avoid in the generated image** ✓

B. It generates the exact opposite of what the main prompt describes

C. It provides a built-in safety filter that blocks inappropriate content

D. It reduces the overall influence of the main positive prompt

Reasoning

negative_prompt steers generation away from specified content/styles, acting as "what to avoid" guidance. Common uses: "blurry, low quality, distorted" to improve quality.

Q7

A diffusion model takes a noisy image and a timestep as input. What does it predict as output?

A. The image as it looked exactly one denoising step earlier

B. **The total noise (ϵ) that was added to create the current noisy image** ✓

C. A small incremental amount of noise to remove from the current image

D. The final completely clean image directly

Reasoning

Most diffusion models use ϵ-prediction: they predict the total noise ϵ added at timestep t. The denoising formula then removes a calibrated portion of this predicted noise.

Q8

Why do diffusion models generate images through many small denoising steps instead of removing all noise at once?

A. Removing all noise at once would require too much computational power

B. The model architecture can only predict a small amount of noise at a time

C. **Single-step denoising amplifies prediction errors; iterative steps allow gradual refinemer** ✓

D. Multiple steps are mathematically required for the diffusion equations to work

Reasoning

Single-step denoising would amplify prediction errors catastrophically. Iterative small steps allow the model to refine and correct mistakes gradually, producing high-quality results.

Q9

A CNN uses 3×3 filters throughout all layers. How can it recognize both tiny local edges and large overall shapes?

A. Deeper layers secretly use larger filters even though they're defined as 3×3

B. **Each 3×3 filter in deeper layers has a larger effective receptive field in the input** ✓

C. The network learns to completely ignore small details in deeper layers

D. Different 3×3 filters are applied to different regions of the image

Reasoning

Receptive field grows with network depth. A 3×3 filter in layer 5 effectively "sees" a much larger region of the original input than the same size filter in layer 1, due to cascaded spatial coverage.

Q10

A Grad-CAM heatmap for class "zebra" highlights background grass instead of the striped animal. Which diagnosis is most likely?

A. The input image needs to be cropped so the zebra is centered

B. The gradient calculation needs more iterations to properly converge

C. The last conv layer's receptive field is too small to see the whole zebra

D. **The classifier learned a spurious correlation between grass and zebra predictions** ✓

Reasoning

Spurious correlation: the model learned a dataset bias where zebras frequently co-occur with grass backgrounds. It uses grass as a shortcut feature instead of learning actual zebra patterns.

11 Quiz 11: Transformers

Topics: Encoder/Decoder, Attention (Q/K/V), Causal Masking, Cross-Attention

Q1

Which transformer architecture would be most appropriate for building a text classification model?

A. Decoder-only, because it can generate the classification label as text

B. Encoder-decoder, because it can transform input into classification output

C. **Encoder-only, because encoders are designed for understanding/encoding tasks** ✓

D. Any architecture works equally well for classification

Reasoning

Encoder-only (BERT-style) is optimal for understanding tasks like classification—it processes the full input bidirectionally. Decoder-only (GPT) is for generation; encoder-decoder for seq2seq tasks.

Q2

In the attention mechanism, what do Query (Q), Key (K), and Value (V) represent conceptually?

A. Q = how this token appears, K = what this token searches for, V = information provided

B. **Q = what this token is searching for, K = how tokens appear/advertise themselves, V =** ✓

C. Q = information provided, K = how tokens appear, V = what this token searches for

D. Q = search query, K = information provided, V = how tokens appear

Reasoning

Query = "what am I looking for?", Key = "what do I contain / how should I be found?", Value = "what information do I provide when matched?" Attention weights come from Q·K similarity.

Q3

What is the correct sequence of operations for calculating attention weights from Q, K, V?

A. Apply softmax to queries, scale the result, then compute dot product with keys

B. **Compute dot product of Q and $\mathbf{K}^T$, scale by $\sqrt{d_k}$, apply softmax, multiply by V** ✓

C. Scale the queries first, compute dot product with keys, then apply softmax

D. Compute dot product of Q and V, scale the result, then apply softmax

Reasoning

Attention formula: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$. Order: dot product → scale → softmax → multiply by V.

Q4

When using nn.MultiheadAttention for self-attention in an encoder, how should you pass the input tensor?

A. Pass the input once and let PyTorch automatically create separate Q, K, V projections

B. Pass three completely different pre-transformed versions of the input

C. **Pass the same input tensor three times—once each for query, key, and value arguments** ✓

D. Pass input twice (once for keys+values combined) and separately for queries

Reasoning

Self-attention: pass same input x for all three: attn(x, x, x). The layer's internal learned projection matrices (W_Q, W_K, W_V) create the distinct Q, K, V representations.

Q5

When creating nn.TransformerEncoderLayer, what does the d_model parameter specify?

A. The number of attention heads to use

B. The number of encoder layers to stack

C. **The dimensionality of the token embeddings (hidden size)** ✓

D. The size of the feed-forward network's hidden layer

Reasoning

d_model is the embedding/hidden dimension (e.g., 512, 768)—the core dimensionality of token representations throughout the transformer. nhead is for attention heads, dim_feedforward for FFN.

Q6

When building a decoder in PyTorch, how can you apply causal masking to prevent attending to future tokens?

A. Set bidirectional=False in the attention layer configuration

B. **Set is_causal=True or pass an explicit upper-triangular causal mask** ✓

C. Use nn.CausalAttention instead of nn.MultiheadAttention

D. Set decoder_mode=True in the layer configuration

Reasoning

Causal masking in PyTorch: use is_causal=True parameter (newer API) or pass an explicit upper-triangular boolean mask that blocks attention to future positions.

Q7

How can you build a decoder-only model (like GPT) using PyTorch's existing transformer components?

A. You must use nn.TransformerDecoderLayer which has built-in causal masking

B. **Reuse nn.TransformerEncoderLayer or nn.TransformerEncoder and apply a causal mas** ✓

C. Create a completely custom decoder class from scratch

D. Use nn.TransformerEncoder with a special reverse=True flag

Reasoning

Decoder-only models (GPT-style) can be built using TransformerEncoderLayer + causal mask. No cross-attention is needed, just masked self-attention—which encoder layers can do with proper masking.

Q8

In an encoder-decoder Transformer for machine translation, what does cross-attention allow the decoder to do?

A. Process its own outputs using Q, K, V all derived from decoder states

B. **Use queries from decoder states to attend to keys and values from encoder outputs** ✓

C. Share attention weight matrices with the encoder layers

D. Apply bidirectional attention to future target tokens

Reasoning

Cross-attention: decoder queries attend to encoder keys/values. This lets the decoder "look at" the encoded source sequence while generating the target, aligning translation with source content.

Q9

When calling nn.TransformerDecoderLayer's forward method, what tensor is passed as the "memory" parameter?

A. The target sequence that the decoder is currently generating

B. The previous decoder layer's output representation

C. **The encoder's final output representation of the source sequence** ✓

D. The causal mask tensor for preventing attention to future tokens

Reasoning

"memory" = encoder output. This is passed to all decoder layers for cross-attention, allowing each decoder layer to attend to the full encoded source representation.

Q10

How does nn.TransformerDecoderLayer differ structurally from nn.TransformerEncoderLayer?

A. Decoder layers use more feed-forward sublayers

B. **Decoder layers have two attention blocks (self + cross) while encoder has only one (self** ✓

C. Decoder layers can only process one token at a time sequentially

D. Decoder layers don't use residual connections around sublayers

Reasoning

Encoder layer: 1 self-attention + 1 FFN. Decoder layer: 1 masked self-attention + 1 cross-attention + 1 FFN = 2 attention mechanisms. The extra cross-attention connects to encoder output.

12 Quiz 12: Model Deployment

Topics: Checkpoints, ONNX, Pruning, Quantization

Q1

What should you include in a checkpoint to seamlessly resume training from where you left off?

A. {model_state_dict, optimizer_state_dict, learning_rate, batch_size}

B. **{model_state_dict, optimizer_state_dict, epoch, loss}** ✓

C. {model_state_dict, epoch, loss, best_accuracy}

D. {optimizer_state_dict, epoch, loss, random_seed}

Reasoning

Resume training requires: model weights (state_dict), optimizer state (momentum buffers, Adam statistics), epoch number (where to continue), and loss (for logging). Learning rate is inside optimizer state.

Q2

Why do you need to explicitly save trained models in PyTorch?

A. PyTorch automatically saves models to disk after training completes

B. **Trained model weights exist only in RAM and are lost when the process terminates** ✓

C. To compress the model files for smaller storage

D. Jupyter notebooks automatically persist all Python variables

Reasoning

PyTorch models exist only in RAM during execution. Without explicit torch.save(), all trained weights are lost when the Python process terminates. Always save your models!

Q3

Why does torch.onnx.export() require an example input tensor to be provided?

A. To validate that the model produces numerically correct outputs

B. **To trace the model and determine input shapes for the static ONNX graph** ✓

C. To include sample data embedded inside the ONNX file

D. To permanently lock the batch size to a fixed value

Reasoning

ONNX export traces the model by running a forward pass with example input. This determines tensor shapes and constructs the static computation graph. Dynamic axes can be specified separately.

Q4

In torch.onnx.export(), what does setting do_constant_folding=True accomplish?

A. Converts all model weights to int8 precision

B. Enables variable-length inputs at runtime

C. **Precomputes operations with constant inputs and stores the results directly** ✓

D. Forces the export to use the latest ONNX opset version

Reasoning

do_constant_folding=True precomputes operations that only involve constants at export time (e.g., shape calculations), optimizing the ONNX graph by replacing ops with their computed values.

Q5

Which statement about structured vs. unstructured pruning is correct?

A. Unstructured pruning always dramatically shrinks file size and speeds up inference

B. Structured pruning is typically less harmful to model accuracy than unstructured

C. **Structured pruning removes entire neurons, channels, or filters as complete units** ✓

D. Both methods automatically remove weights from the model's state dictionary

Reasoning

Structured pruning removes whole structures (neurons, channels, filters)—actually reducing dimensions. Unstructured removes individual weights (creates sparse matrices with zeros), which doesn't shrink dense tensors.

Q6

When using prune.ln_structured() with dim=0 and amount=0.5, what gets pruned from a Linear layer?

A. 50% of the input features, selected randomly

B. **50% of the output neurons, selected by L-n norm importance** ✓

C. 50% of the input features, selected by importance

D. 50% of the output neurons, selected randomly

Reasoning

dim=0 targets the output dimension (weight matrix rows = output neurons). ln_structured uses L-n norm to rank neurons by importance (not random), then prunes the 50% with smallest norms.

Q7

Why is 32-bit floating point precision important during training but often unnecessary for inference?

A. Training requires GPU hardware which only supports 32-bit operations

B. **Training accumulates small gradient updates where precision loss compounds; inference** ✓

C. 32-bit precision specifically prevents overfitting during training

D. Training data inherently contains more numerical variation

Reasoning

Training: small gradient updates accumulate over millions of steps—precision loss compounds catastrophically. Inference: weights are fixed, single forward pass, minor rounding from quantization is tolerable.

Q8

When should you use dynamic quantization versus static quantization?

A. Dynamic for CNNs, static for RNNs and transformers

B. **Dynamic for models with large linear layers (RNN/Transformer); static for CNNs requ** ✓

C. Dynamic when you have calibration data available, static when you don't

D. Always use static quantization for best results

Reasoning

Dynamic quantization works well for nn.Linear-heavy models (RNN, Transformer). Static quantization supports nn.Conv2d and requires calibration data to determine activation ranges—better for CNNs.

Q9

During Quantization-Aware Training (QAT), what does a "fake quantization" module do?

A. Permanently converts values to int8 storage format

B. Stores both float32 and int8 versions of weights side-by-side

C. **Simulates quantization by rounding to int8 range then immediately converting back to** ✓

D. Completely skips quantization during the training phase

Reasoning

Fake quantization: round values to int8 range, then immediately dequantize back to float32. This simulates quantization effects during training while keeping values differentiable for backprop.

Q10

Why should you call fuse_modules() before prepare_qat() when quantizing Conv-BatchNorm-ReLU sequences?

A. Fusing reduces memory usage by combining layer weight tensors

B. PyTorch strictly requires layers to be fused before any quantization

C. **Fusing combines operations so quantization can observe the true output range** ✓

D. Fusing prevents the model from learning incorrect quantization parameters

Reasoning

Fusing Conv+BN+ReLU into one operation lets quantization see the combined transformation's actual output distribution. Without fusing, quantization might set suboptimal ranges based on intermediate values.

A Quick Reference: Key Formulas

Loss Functions

MSE Loss (Regression):

$$\mathcal{L}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Cross-Entropy Loss (Classification):

$$\mathcal{L}_{CE} = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

Attention Mechanism

Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where d_k is the dimension of the key vectors.

ResNet Skip Connection

Forward Pass:

$$y = F(x) + x$$

Gradient (enables deep training):

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \left(\frac{\partial F}{\partial x} + 1\right)$$

The $+1$ term provides a direct gradient highway.

ImageNet Normalization

Standard values for pretrained models:

```
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
```

B Quick Reference: Essential Code Patterns

Listing 1: Standard Training Loop

```
for epoch in range(num_epochs):
    model.train()
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
```

```
        loss.backward()
        optimizer.step()
```

Listing 2: Evaluation Loop

```
model.eval()
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        outputs = model(batch_x.to(device))
        # compute metrics...
```

Listing 3: Save/Load Checkpoint

```
# Save
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'epoch': epoch,
    'loss': loss,
}, 'checkpoint.pth')

# Load
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

Listing 4: Device Setup

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```