

PRACTICE QUESTIONS

Linux Foundation / CNCF

PTCA

PyTorch Certified Associate

SHARED BY [u/Cozy-Pixel-4078](#) on aiexamcerts

EDITION Free Set 2

QUESTIONS 10

SHARED October 18, 2026

EXAM Linux Foundation / CNCF PTCA, PyTorch Certified Associate

Shared by u/Cozy-Pixel-4078 on aiexamcerts

10 practice questions for the Linux Foundation / CNCF PTCA exam, PyTorch Certified Associate, each followed by its answer and a short explanation. More free sets and the full practice exam:

aiexamcerts.com/linux-foundation-cncf-ptca. Spotted a mistake? Email support@aiexamcerts.com with the exam code PTCA and the question number.

1. Which tensor operation is the standard way to compute the matrix product of two 2D tensors?

- A `torch.matmul(a, b)` or `a @ b`
- B `a * b`, which multiplies the two tensors element by element with broadcasting
- C `torch.cat((a, b), dim=0)`, which joins the tensors along their first dimension
- D `a.T.sum()`, which transposes `a` and then sums all of its elements together

Answer: A

`torch.matmul` and the `@` operator perform matrix multiplication for 2D tensors and generalize to batched inputs. Element-wise multiplication with `*` is the common mistake, because it looks like a product but computes something different. Concatenation and summing join or reduce values without multiplying matrices.

2. `nn.CrossEntropyLoss` expects its input to be probabilities after softmax, so the model should apply softmax before calling it.

- A True
- B False

Answer: B (False)

False. `nn.CrossEntropyLoss` takes raw logits and applies log-softmax internally, so adding softmax first produces a weaker and incorrect loss. Logits in and class indices as targets is the expected pattern.

3. Which PyTorch 2.x feature compiles a model's Python code into optimized kernels to speed up execution?

- A `torch.nn.DataParallel`, which replicates the model across GPUs within one process
- B `torch.utils.checkpoint`, which recomputes activations to reduce peak memory use
- C `torch.compile`, which captures the model and generates optimized code for it
- D `torch.no_grad`, which disables gradient tracking for the enclosed operations

Answer: C

`torch.compile` captures the model's computation graph and generates optimized code, which can reduce overhead and fuse operations. `DataParallel` distributes work across GPUs rather than optimizing the code itself. Checkpointing and `no_grad` trade memory or gradient tracking, and neither compiles the model.

4. The DataLoader argument _____ set to True copies batches into page-locked host memory, which speeds up transfers to a CUDA device.

Answer: pin_memory (also accepted: pin memory)

`pin_memory=True` places batches in page-locked host memory, which allows faster and asynchronous copies to a CUDA device when combined with `non_blocking` transfers. It has no effect on training results. It is a loading and transfer setting, so it does not change how many batches are formed.

5. Which statement about the `.grad` attribute is correct after two `backward()` calls without zeroing the gradients?

- A It is set to None automatically after the second backward call finishes
- B It accumulates, so it holds the sum of the gradients from both backward calls
- C It is recomputed from the loss alone when the optimizer step is called later
- D It is replaced by the first gradient and the second one is discarded entirely

Answer: B

`backward()` adds new gradients into `.grad`, so after two calls without zeroing, the attribute holds their sum. This is intentional for gradient accumulation and a bug in ordinary training loops. The option that claims the value is replaced is the most tempting, but PyTorch does not overwrite it by default.

6. Which two practices reduce host-to-device transfer overhead in a training pipeline? (Choose two.)

- A Pinning host memory and using `non_blocking=True` copies for CUDA tensors
- B Moving each sample to the GPU separately inside the Python loop that builds the batch
- C Using float64 tensors for all inputs so that precision stays as high as possible
- D Pre-batching data on the host so that fewer and larger transfers occur
- E Calling `.cpu()` on the output of every layer of the network during the forward pass

Answer: A, D

Pinned memory with `non_blocking` copies lets transfers overlap computation, and batching on the host sends larger chunks in fewer transfers. Moving samples one at a time multiplies the transfer overhead, which makes that option the most tempting wrong choice. Float64 doubles the bytes moved, and calling `.cpu()` on every layer forces repeated device round trips.

7. Which tool records the time spent by each operator, including CPU and CUDA activity, during a training run?
- A A stopwatch placed around the optimizer call, reporting only the total elapsed time
 - B Print statements placed inside the forward method of the model's layers
 - C The Python garbage collector statistics, which list objects freed during each step
 - D `torch.profiler.profile`, which records operator timings across CPU and CUDA activity

Answer: D

`torch.profiler.profile` records per-operator timings for both CPU and CUDA activity, and its traces can be inspected to find bottlenecks. A single stopwatch around the optimizer gives only one total and cannot attribute time to operators. Print statements and garbage collector statistics do not measure operator time.

8. What does `tensor.numel()` return?
- A The number of dimensions of the tensor, such as two for a matrix
 - B The size of the first dimension only, regardless of the other dimensions
 - C The total count of elements across all dimensions of the tensor
 - D The number of bytes that the tensor's storage occupies in device memory

Answer: C

`numel()` returns the total number of elements in the tensor, which is the product of all its dimension sizes. The number of dimensions is returned by `dim()` or `ndim`, which is the first option. Bytes used depend on the dtype and are a different quantity from the element count.

9. `torch.manual_seed` controls PyTorch's random number generators, which helps make training runs reproducible on the same setup.
- A True
 - B False

Answer: A (True)

True. Setting the seed fixes the sequence of random draws used for weight initialization, shuffling and augmentation on the same software and hardware. Full determinism can still need extra settings, but the seed is the first step.

10. Which optimizer uses decoupled weight decay as its standard variant of Adam-style updates?

- A torch.optim.AdamW
- B torch.optim.SGD with momentum set to zero and no weight decay of any kind
- C torch.optim.Adagrad with a fixed learning rate and no accumulation of squared gradients
- D torch.optim.Adam with a learning rate of zero, so that no parameter is changed at all

Answer: A

AdamW applies weight decay separately from the adaptive gradient step, which is the decoupled form that Adam's plain `weight_decay` option does not provide. SGD and Adagrad do not implement this Adam variant. Adam with a zero learning rate leaves every weight unchanged, so it cannot be a training optimizer in any useful sense.