AI Research Labs

Deep Learning Foundations · Practice test 11 · 15 questions

Network Design and PyTorch

How to use this test. Answer all 15 questions without notes, then mark yourself with the answer key at the back. Each question has exactly one best answer. Score 75% or more and you've mastered the material; below that, reread the lessons for the questions you missed.

11.1 Depth versus Width

Q1 concept What is the main expressive advantage of adding depth rather than width?

A. Fewer parameters in every possible architecture
B. Easier optimisation with no vanishing gradients
C. Guaranteed better generalisation on any data
D. Hierarchical features that compose simpler ones

Q2 calculate Two adjacent hidden layers each have 100 units. You double both to 200. Roughly how many times more weights connect them?

A. 8 times
B. 2 times
C. 4 times
D. The same

Q3 concept What is the main practical difficulty of very deep networks?

A. They can only be used for image classification tasks
B. They are harder to optimise, e.g. vanishing gradients
C. They need far more training data than any wide network
D. They cannot represent non-linear functions at all

Q4 apply For a small tabular dataset of a few thousand rows, what depth is a sensible starting point?

A. One hidden layer with a million units in it
B. No hidden layer at all, just linear regression
C. Fifty hidden layers with residual connections
D. Two or three hidden layers

Q5 interpret Which statement balances the depth–width trade-off correctly?

A. Depth and width are interchangeable in every respect
B. Depth is efficient but harder to train; width is easy but costly
C. Width always beats depth because the theorem needs one layer
D. Depth is always better, so width should be minimal

11.2 Designing a Network for Tabular Data

Q6 `apply` For predicting house prices, which output layer and loss should you start with?

- A One sigmoid unit with binary cross-entropy
- B One ReLU unit with the accuracy as the loss
- C One linear unit with MSE
- D Ten softmax units, one per price band, with cross-entropy

Q7 `apply` A "postcode" feature has 2,000 distinct values. What's the best way to feed it in?

- A Standardise the postcode strings to have a mean of zero
- B An embedding layer with a learned vector per postcode
- C Drop the feature, since it has far too many values
- D A single numeric input coded from 1 up to 2,000

Q8 `concept` Why might you train on the logarithm of the price instead of the price itself?

- A It tames the skew and stops expensive houses dominating
- B Because the logarithm makes the loss function convex
- C Because networks cannot output numbers above 1,000
- D Because prices must be categorical before training

Q9 `apply` You have only 800 houses. Which combination best reduces the risk of overfitting?

- A A much deeper network trained for many more epochs
- B Using the test set to choose the number of epochs
- C L2 weight decay, early stopping and cross-validation
- D Removing the validation set to train on more data

Q10 `interpret` One feature is "final agent fee", known only after the sale completes. Should it be an input?

- A No, but only because it would need to be one-hot encoded
- B No: it isn't available when predicting, so it's leakage
- C Yes, as long as it is standardised with training statistics
- D Yes: it is strongly correlated with the price, so it helps

11.3 Neural Networks in PyTorch

Q11 `concept` In the standard PyTorch training loop, what is the correct order?

- A zero_grad → forward and loss → backward → step
- B step → zero_grad → forward and loss → backward
- C forward and loss → step → backward → zero_grad
- D backward → forward and loss → zero_grad → step

Q12 apply The model ends with `nn.Softmax` and you train with `nn.CrossEntropyLoss`. What should you change?

A. Replace the softmax with a sigmoid
B. Add a second softmax before the loss
C. Remove the softmax and return raw logits
D. Switch the loss function to nn.MSELoss

Q13 concept Why must `optimizer.zero_grad()` be called every iteration?

A. PyTorch adds new gradients to the old ones by default
B. It reduces the learning rate after every single step
C. It resets the model's weights back to their initial values
D. It switches the model between training and evaluation

Q14 calculate `nn.Linear(100, 50)` followed by `nn.Linear(50, 10)`. How many trainable parameters?

A. 160
B. 5,560
C. 5,000
D. 5,500

Q15 apply How do you add L2 weight decay when training with SGD in PyTorch?

A. Divide the learning rate by the batch size
B. Pass `weight_decay` to the optimiser
C. Add `nn.Dropout` after every layer
D. Call `model.eval()` before training

Answer key

Each answer comes with the reasoning, including why the most tempting wrong option is wrong.

Q1 **D** · lesson 11.1

Depth lets later layers build on earlier features, which can represent structured functions with far fewer units. It doesn't always reduce parameters, it makes optimisation harder (not easier), and it guarantees nothing about generalisation.

Q2 **C** · lesson 11.1

The connecting weights number $100 \times 100 = 10{,}000$, becoming $200 \times 200 = 40{,}000$: four times as many. Width grows parameters with the product of adjacent widths, which is why wide networks get large fast.

Q3 **B** · lesson 11.1

Signals must pass back through many layers, where gradients can vanish or explode, so training is harder; ReLU, normalisation and residual connections help. Depth adds expressive power and is used well beyond images.

Q4 **D** · lesson 11.1

Tabular data rarely have deep compositional structure, and small datasets overfit easily, so 2–3 dense layers are a good start, tuned on validation data. Very deep or extremely wide networks mostly add trouble.

Q5 **B** · lesson 11.1

Depth can be exponentially more efficient yet harder to optimise; width trains easily but adds parameters and overfitting risk. The universal approximation theorem allows one wide layer but doesn't make it the better design.

Q6 **C** · lesson 11.2

Price is a continuous number: a linear output and a regression loss (MSE, or MAE if outliers matter). Sigmoid and softmax are for classification, binning prices throws away information, and accuracy has no gradient.

Q7 **B** · lesson 11.2

With many categories, one-hot encoding gives 2,000 sparse inputs; an embedding learns a short vector per postcode, so similar areas end up close. Integer codes impose a false order, and location is too informative to drop.

Q8 **A** · lesson 11.2

Prices are right-skewed; on a log scale errors become relative, and a few mansions don't dominate an MSE loss. Networks can output large numbers, the log doesn't make a network's loss convex, and price is numeric.

Q9 **C** · lesson 11.2

Small datasets overfit easily: penalise large weights, stop at the best validation epoch, and use cross-validation to make reliable choices. More depth and epochs increase overfitting, and using the test set for tuning is leakage.

Q10 **B** · lesson 11.2

A model used to price houses before sale can't use information that exists only afterwards; including it would give great test scores and useless predictions. Scaling or encoding doesn't fix leakage.

Q11 **A** · lesson 11.3

Clear old gradients, compute the predictions and loss, backpropagate to fill the gradients, then update the weights. Stepping before backward would use stale or empty gradients.

Q12 **C** · lesson 11.3

CrossEntropyLoss applies log-softmax itself; an extra softmax distorts the gradients. Return logits and apply softmax only for reporting probabilities. MSE is the wrong loss for classification.

Q13 **A** · lesson 11.3

Gradients accumulate across `backward()` calls, so without clearing them each step uses the sum of all previous gradients. It doesn't touch the weights, the mode (that's `train()`/`eval()`), or the learning rate.

Q14 **B** · lesson 11.3

$100 \times 50 + 50 = 5{,}050$ plus $50 \times 10 + 10 = 510$, total 5,560. 5,500 leaves out the biases, 160 counts units instead of weights, and 5,000 counts only the first layer's weights.

Q15 **B** · lesson 11.3

`torch.optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)` adds λW to each gradient, which is L2 weight decay. Dropout is a different regulariser, and `eval()` switches dropout off for evaluation.