AI Research Labs

Section 11 of 11 · 3 lessons

Network Design and PyTorch

Depth versus width, designing a network for a real task, and implementing and training a network in PyTorch.

11.1 Depth versus Width

11.2 Designing a Network for Tabular Data

11.3 Neural Networks in PyTorch



Free course with quizzes, progress tracking and Scoopy, an AI tutor:
airesearchlabs.si/learn/dl

Deep Learning Foundations · Edition November 2026 · © 2026 AI Research Labs. Free for personal study.

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

M3

BUILDING AND TRAINING A NETWORK · DENSE LANE · STEP M3

Size the network

How deep, how wide, and for which task?

Lessons here: 11.1, 11.2 · feeds into Regularise

7

BUILDING AND TRAINING A NETWORK · STEP 7

Update the weights

How do I turn gradients into a better network?

Lessons here: 11.3 · feeds into Evaluate honestly

Depth versus Width

More layers or more units per layer? How depth and width differ in expressive power, parameter count and trainability, and why most successful networks are deep rather than very wide.

Two ways to add capacity

	MORE DEPTH (LAYERS)	MORE WIDTH (UNITS PER LAYER)
Expressive power	Builds **hierarchical, compositional** features; some functions need far fewer units when built deep	More features at the **same level** of abstraction; a shallow net may need exponentially many units for the same function
Parameters	Grow roughly linearly with the number of layers	Grow with the **product** of adjacent widths: doubling two adjacent layers quadruples their weights
Training	Harder: vanishing or exploding gradients, more sequential computation	Easier to optimise, highly parallel
Main risk	Optimisation difficulty	Overfitting from sheer parameter count

Why depth usually wins

The universal approximation theorem says one wide hidden layer *can* represent any continuous function, but possibly with an enormous number of units. Composing layers lets a network reuse intermediate features (lesson 3.3), so it represents structured functions such as images, speech and language much more efficiently. In practice, depth often contributes more to useful expressive power than width.

Making depth trainable

- **ReLU** activations, to avoid vanishing gradients (Section 4).
- **Careful initialisation** and normalised inputs.
- **Batch normalisation** and **residual (skip) connections**, which let very deep networks train well (beyond this course).

A BALANCED ANSWER

"Depth gives hierarchical features and can be exponentially more efficient than width, but is harder to optimise. Width is easier to train but adds parameters fast and risks overfitting. Modern networks are deep, with ReLU, normalisation and residual connections to keep them trainable."

Common doubts

FIRST PASS **So should I always make my network as deep as possible?**

No. Extra depth only helps if the problem has structure to exploit and you can train it. For small tabular datasets, two or three hidden layers are usually plenty; deeper networks mostly add training trouble and overfitting risk. Let validation performance decide.

FIRST PASS **How do I answer a "trade-offs between depth and width" question?**

Use the layout **two options × two criteria (power, cost) + a verdict**.

	DEEPER (MORE LAYERS)	WIDER (MORE NEURONS PER LAYER)
Power	Features of features: edges → shapes → objects	More features at the same level
Efficiency	The same function with far fewer neurons	May need exponentially more neurons
Training	Harder: vanishing gradients (eased by ReLU, batch norm, residual connections)	Easier and more stable
Speed	Slower: layers run one after another	Faster on GPUs: a layer's neurons run in parallel
Risk	Both add parameters, so **both** can overfit	

Link it to the universal approximation theorem: one wide layer is enough in theory, but the theorem says nothing about how many neurons that takes. **Verdict:** depth gives more power per parameter, so modern networks go deep and use ReLU, batch norm and residual connections to keep them trainable.

Memory line: "Deep is efficient but hard to train. Wide is easy to train but wasteful."

Cheatsheet

Depth vs width

Depth: hierarchical features, efficient, harder to train (vanishing gradients). Width: easy to train, parallel, but may need far more neurons. Both add capacity, so both can overfit.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/design-and-pytorch/depth-versus-width

Designing a Network for Tabular Data

A worked design for a regression problem: predicting house prices from features such as size, bedrooms and location. Architecture, activations, loss, regularisation, categorical features and the challenges to expect.

The task

Predict a house's price (a number) from features: floor area, number of bedrooms, location, property type, and so on. This is **regression** on **tabular** data.

A sensible starting design

CHOICE	PROPOSAL	WHY
Layers	2–3 fully connected hidden layers, e.g. 64 → 32 → 16 units	Tabular features have no spatial structure, so dense layers suit; shrinking widths compress the features
Hidden activation	ReLU	Non-linear, cheap, avoids vanishing gradients
Output	1 unit, **linear**	A price is any positive real number; a sigmoid would cap it
Loss	MSE (MAE if outliers matter)	Standard for regression; MAE resists extreme prices
Regularisation	L2 weight decay; optionally dropout; early stopping	Tabular datasets are often small, so overfitting is the main risk

It's a starting point, not a recipe: the best architecture depends on the data and is chosen on validation performance.

Preparing the features

- **Numeric** features: standardise using training statistics (also the target, then undo it on the predictions).
- **Categorical** features (location, property type): **one-hot encode**. With many categories (hundreds of postcodes), use an **embedding layer** that learns a short dense vector per category.
- **Missing values:** impute with training medians and add "was missing" indicators.
- **Outliers:** cap or remove clear errors; consider a log of the price as the target.

Challenges to expect

- **Overfitting** on a small dataset: regularise, use cross-validation, collect more data.
- **Mixed feature types and scales**: encode and scale consistently.
- **Irrelevant features**: L1 or simply checking which features matter.
- **Leakage:** a feature known only after the sale (such as the final agent fee) must be excluded.

Common doubts

FIRST PASS **Why not encode location as a number, like 1 = north, 2 = south?**

Because the network would treat it as a quantity: "south" would be "twice north", and locations 1 and 3 would look further apart than 1 and 2. One-hot encoding gives each location its own input with no false ordering; embeddings go further and learn which locations behave similarly.

Cheatsheet

Tabular regression design

Dense ReLU layers (64→32→16), 1 linear output, MSE, L2 + early stopping (+ dropout). Standardise numerics, one-hot or embed categories.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/design-and-pytorch/designing-a-network-for-tabular-data

Neural Networks in PyTorch

Define a network, choose a loss and optimiser, and write the training loop. A 100 → 50 → 10 classifier line by line, the logits rule, train and eval modes, and a custom activation function.

Defining the network

A classifier with 100 input features, 50 ReLU hidden units and 10 classes:

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(100, 50)   # 100*50 + 50 = 5,050 parameters
        self.fc2 = nn.Linear(50, 10)    # 50*10 + 10  =   510 parameters

    def forward(self, x):
        return self.fc2(torch.relu(self.fc1(x)))   # logits: no softmax here
```

THE LOGITS RULE

`nn.CrossEntropyLoss` applies log-softmax itself, so the model must return **raw logits**. A question may ask for "an output layer with softmax": for training with `CrossEntropyLoss`, leave it out and say why, then apply `torch.softmax(logits, dim=1)` only when you need probabilities.

Training

```
model = MLP()
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)   # or torch.optim.Adam

for epoch in range(num_epochs):
    model.train()                       # training mode (dropout on)
    for xb, yb in train_loader:         # mini-batches
        optimizer.zero_grad()           # 1. clear old gradients
        loss = loss_fn(model(xb), yb)   # 2. forward pass and loss
        loss.backward()                 # 3. backpropagation
        optimizer.step()                # 4. gradient descent update

    model.eval()                        # evaluation mode (dropout off)
    with torch.no_grad():               # no gradients needed
        val_loss = sum(loss_fn(model(xb), yb).item() for xb, yb in val_loader)
```

- `zero_grad()` : PyTorch **adds** gradients by default, so clear them each step.
- `backward()` is backpropagation; `step()` is the update. L2 weight decay: `torch.optim.SGD(..., weight_decay=1e-4)` .
- Record training and validation loss every epoch to watch for overfitting.

A custom activation function

Activations are just modules. For example "LeakySigmoid": $\sigma(x)$ for $x \geq 0$ and $0.01\,\sigma(x)$ for $x < 0$:

```
class LeakySigmoid(nn.Module):
    def forward(self, x):
        s = torch.sigmoid(x)
        return torch.where(x ⩾ 0, s, 0.01 * s)
```

Use it like any layer, e.g. `self.act = LeakySigmoid()` and `self.fc2(self.act(self.fc1(x)))` . Autograd differentiates `torch.where` and `torch.sigmoid` automatically, so no hand-written backward is needed.

Common doubts

FIRST PASS **What happens if I forget optimizer.zero_grad()?**

Gradients from every previous batch keep adding up, so each step uses a growing sum of old gradients instead of the current one. Training behaves erratically or diverges. Clear them at the start of every iteration.

GOING DEEPER **Does the LeakySigmoid fix the vanishing-gradient problem?**

Not really. For negative inputs it's $0.01\,\sigma(x)$, so its gradient there is even **smaller** than the sigmoid's, and for large positive inputs it saturates just like a sigmoid. It's a coding exercise more than a recommended activation; ReLU or Leaky ReLU are the practical choices.

Cheatsheet

Training loop

`zero_grad()` → forward + loss → `loss.backward()` → `optimizer.step()`. `model.train()` / `model.eval()` + `torch.no_grad()` for validation.

Classifier output

Return logits from the last `nn.Linear`; `nn.CrossEntropyLoss` applies log-softmax. Softmax only for reporting probabilities.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/design-and-pytorch/neural-networks-in-pytorch