ETH zürich

CAMLab



Tutorial 2: Cross Validation and intro to CNNs

AI in Science and Engineering 2026HS

Shizheng Wen | Sep 28, 2026

Seminar for Applied Mathematics (D-MATH) & ETH AI Center

ETH AI CENTER

SAM Seminar for Applied Mathematics

Outline

01 **Motivation**

Why do we need CNNs?

02 **Loss Function – cross entropy**

Loss function for classification problem

03 **K-fold Cross validation**

How we can assess the performance and generalization of a machine learning model

04 **Practice**

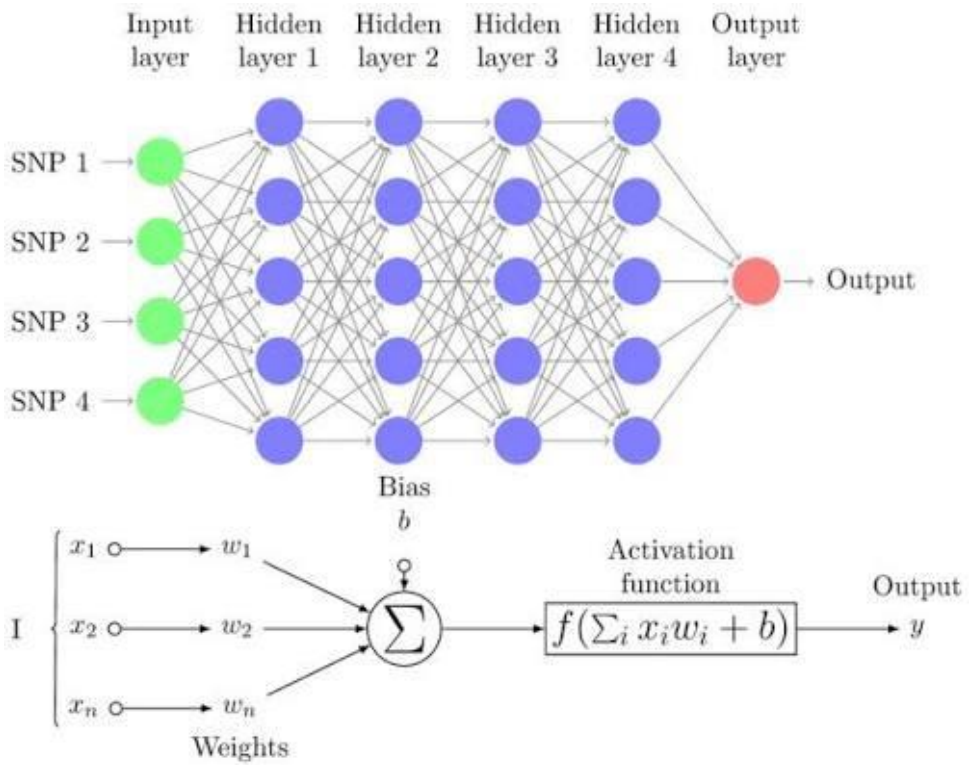
CNN for handwritten digital recognition

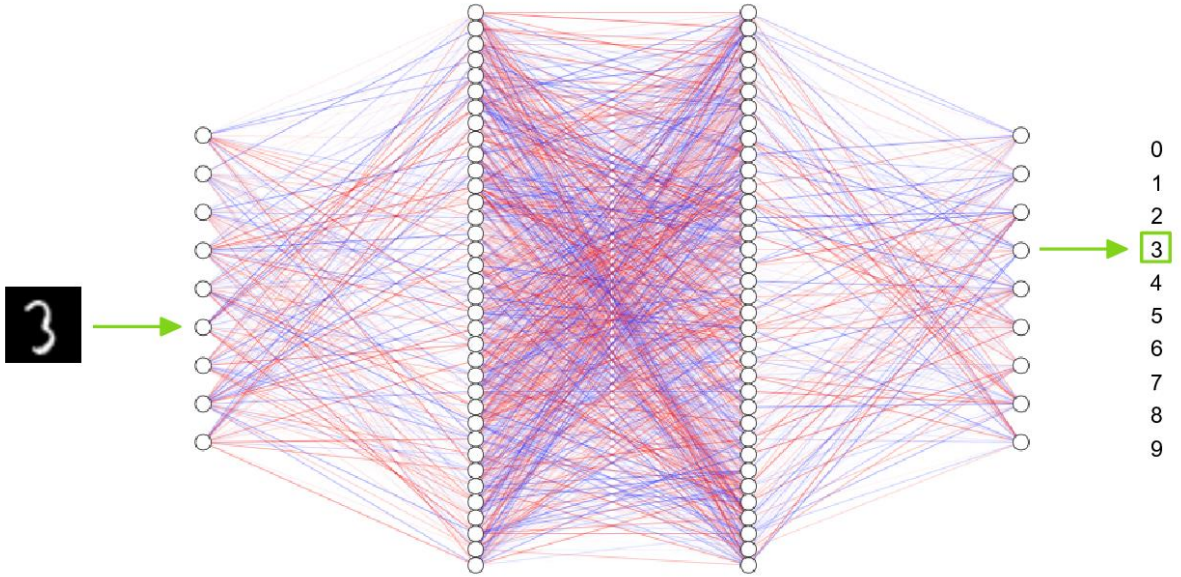
Outline

01 **Motivation**

Why do we need CNNs?

Motivation



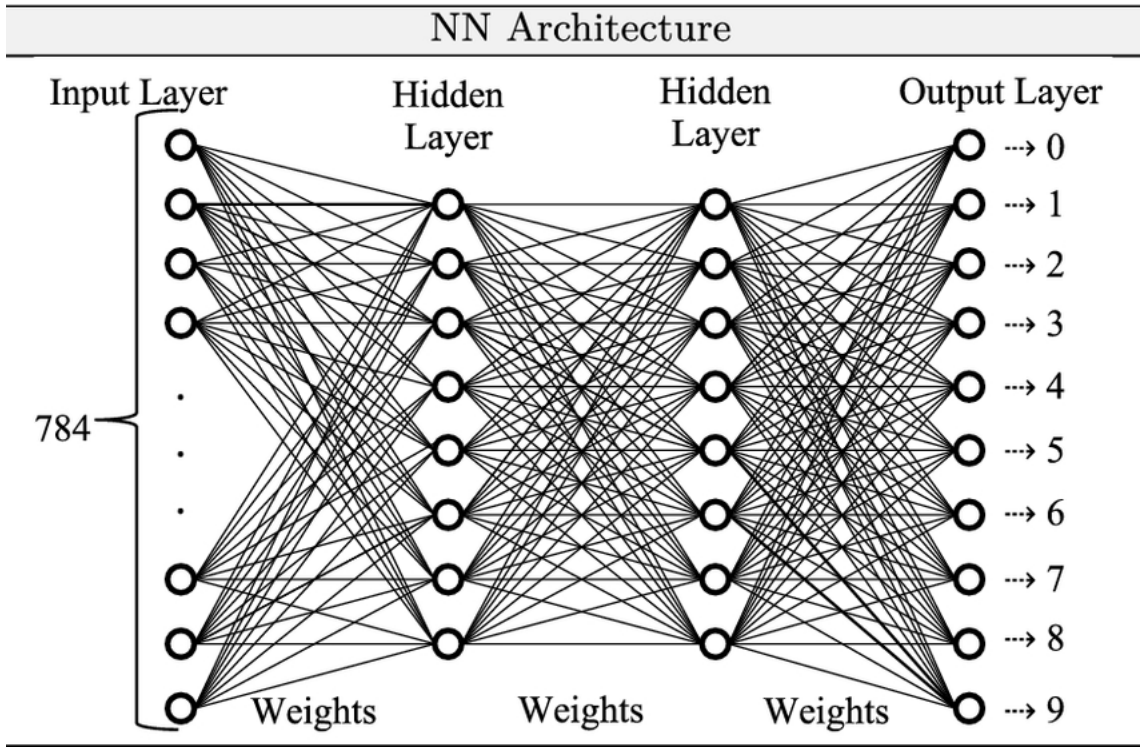


Basic bone of ANN (MLPs)

- Fully connected
- dense

Think about you job is to do the handwritten recognition - MINIST dataset

Motivation



784

inputs for one MNIST digit, 28 × 28 pixels

400k

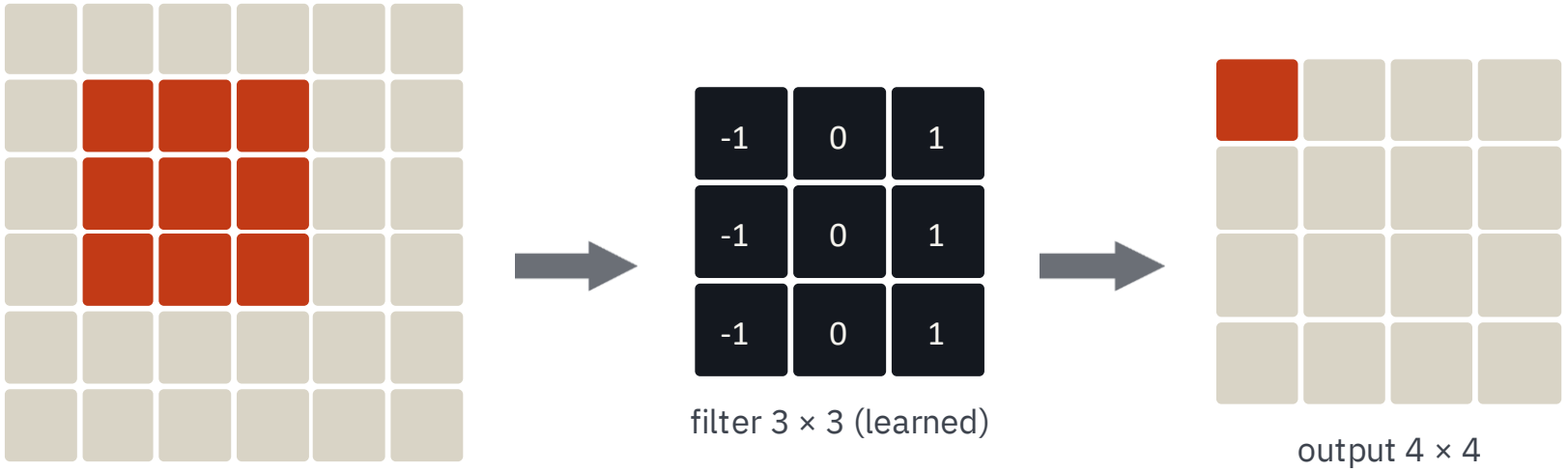
weights in the first layer alone, with 512 hidden units

150k

inputs for a small colour photo, 224 × 224 × 3

- **Too many parameters:** every pixel is connected to every hidden unit. The count grows with the image size.
- **No notion of ”nearby”:** The image is flattened to a vector. Neighbouring pixels are treated like any other pair.
- **Not shift invariant:** Move the digit by one pixel and the input vector looks completely different.

Motivation – Convolution: one small filter, slid over the image



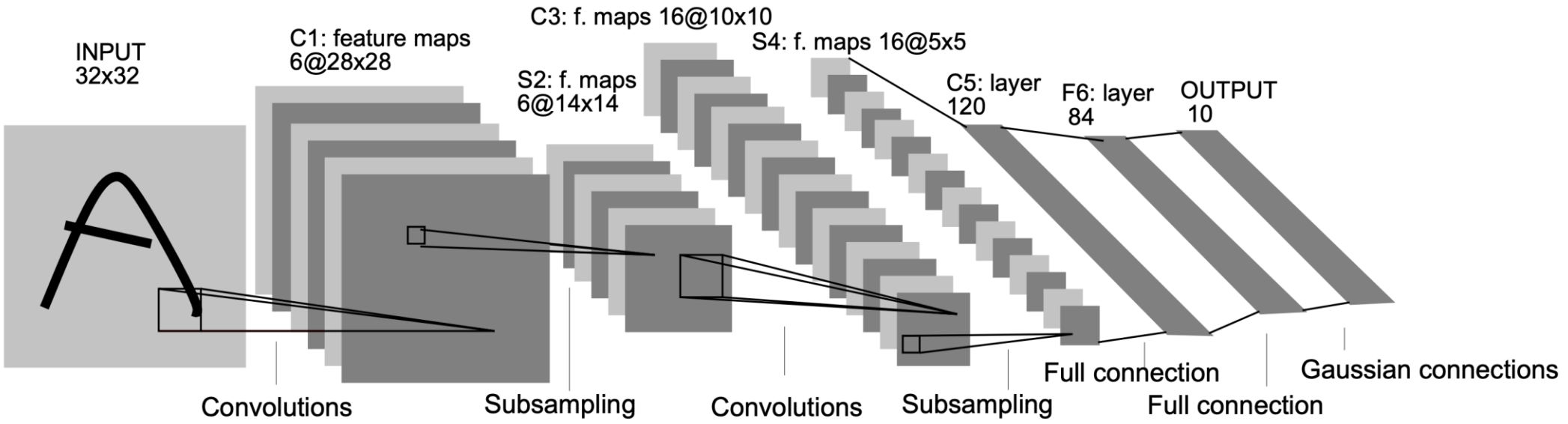
$$out[i, j] = \Sigma_{ab} K[a, b] \cdot I[i+a, j+b]$$

Multiply the window by the filter, add up, write one number. Then move one pixel and repeat.

Core Idea:

- **Local:** Each output looks at a 3 x 3 window, not at the whole image.
- **Shared weights:** The same 9 numbers are used at every position. 9 parameters, whatever the image size.
- **Shift equivariant:** Move the digit, and the feature map moves with it.

Motivation – LeNet (1998): the first CNN that worked



LeNet:

- **Conv -> pool -> conv -> pool -> FC:** Two rounds of feature extraction, then three fully connected layers. About 60,000 parameters
- **Built for handwritten digits:** Yann LeCun at Bell Labs, reading cheques and postal codes. The same task as your exercise today.
- **What changed since:** tanh become ReLU, average pooling become max pooling, and the networks became much deeper.

Implementation - A CNN: four building blocks, repeated

nn.Conv2d

Convolution

Many learned filters. Each one makes a feature map.

→

nn.ReLU

Activation

max(0, x). Without it, two convolutions are still one linear map.

→

nn.MaxPool2d

Pooling

Keep the largest value in each 2 × 2 block. The image gets half as big.

→

nn.Linear

Fully connected

At the end. Flatten the features and make the decision.

Early layers see edges and strokes, later layers see whole shapes

After each pooling step, the same 3 × 3 filter covers a larger part of the original image. Conv → ReLU → Pool is repeated a few times, then one or two linear layers finish the job.

Implementation - Follow the shape through the network

Layer	Output shape	Parameters
input: one grey image	(B, 1, 28, 28)	0
Conv2d(1, 16, 3, padding = 1) + ReLU	(B, 16, 28, 28)	16·9 + 16 = 160
MaxPool2d(2)	(B, 16, 14, 14)	0
Conv2d(16, 32, 3, padding = 1) + ReLU	(B, 32, 14, 14)	32·16·9 + 32 = 4 640
MaxPool2d(2)	(B, 32, 7, 7)	0
flatten	(B, 1568)	0
Linear(1568, 10)	(B, 10)	1568·10 + 10 = 15 690

About 20 000 parameters in total. An MLP with one hidden layer of 512 units has about 400 000, and most of them sit in the first layer.

Implementation – The same nn.Module, new layers

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.classifier = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):          # (B, 1, 28, 28)
        x = self.features(x)       # (B, 32, 7, 7)
        x = x.flatten(1)           # (B, 1568)
        return self.classifier(x)  # (B, 10)
```

Conv2d(in, out, kernel)

Input channels, output channels, filter size. padding=1 keeps the image size.

flatten(1)

Keep the batch axis, merge the rest into one vector.

10 outputs, not 1

One score per digit. What these scores mean is the next slide.

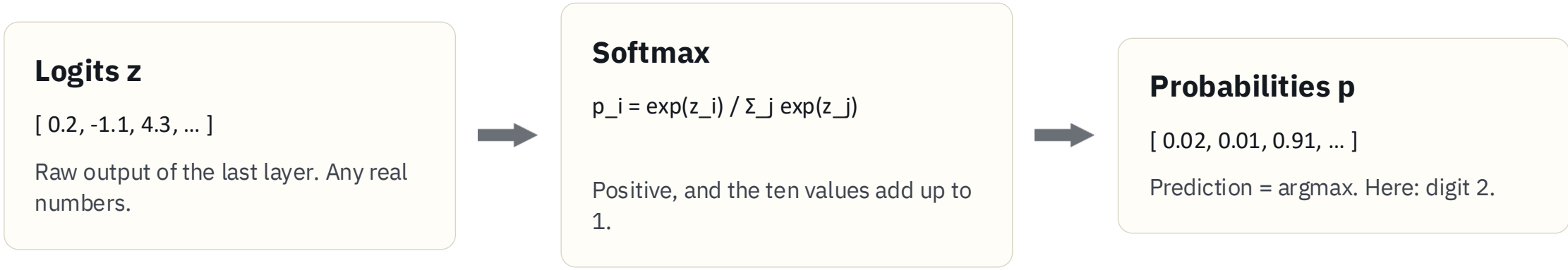
Outline

02

Loss function - cross entropy

Loss function for classification problem

Classification: from 10 scores to a probability



Regression, last week

Output: one real number. Loss: squared distance to the true value.

Classification, today

Output: a probability for each class. Loss: how much probability we gave to the correct class.

Cross-entropy: minus the log of the correct probability

$$H(y, p) = -\sum_i y_i \log p_i = -\log p_{\text{correct}}$$

y is one-hot, so only the correct class survives the sum.

p for the correct class	loss = − log p
0.9 (confident, right)	0.11
0.5	0.69
0.1 (uniform over 10 classes)	2.30
0.01 (confident, wrong)	4.61

```
loss_fn = nn.CrossEntropyLoss()

logits = model(xb)   # (64, 10)
loss = loss_fn(logits, yb)
# yb: (64,) integers 0..9
```

Give it logits, not probabilities

The softmax is inside the loss. Adding your own softmax before it is a common bug.

Labels are integers

Not one-hot vectors. PyTorch does the one-hot for you.

Training loop: the same five lines, in batches

```
for epoch in range(5):
    model.train()
    for xb, yb in train_loader:            # mini-batches of 64 images
        xb, yb = xb.to(device), yb.to(device)
        optimizer.zero_grad()
        loss = loss_fn(model(xb), yb)      # logits (64, 10), labels (64,)
        loss.backward()
        optimizer.step()

    model.eval(); correct = 0
    with torch.no_grad():                  # no graph needed for evaluation
        for xb, yb in val_loader:
            pred = model(xb.to(device)).argmax(dim=1)
            correct += (pred == yb.to(device)).sum().item()
    print(epoch, loss.item(), correct / len(val_loader.dataset))
```

Two new things: a DataLoader that gives mini-batches, and an evaluation pass on data the model never trains on.

Outline

03

K-fold Cross validation

How we can assess the performance and generalization of a machine learning model

Training loss is not the number you care about

train

Fit the parameters

The loss here always goes down if you train long enough. A big network can memorize every image.

validation

Choose the design

Learning rate, width, depth, number of epochs. Pick whatever works best here.

test

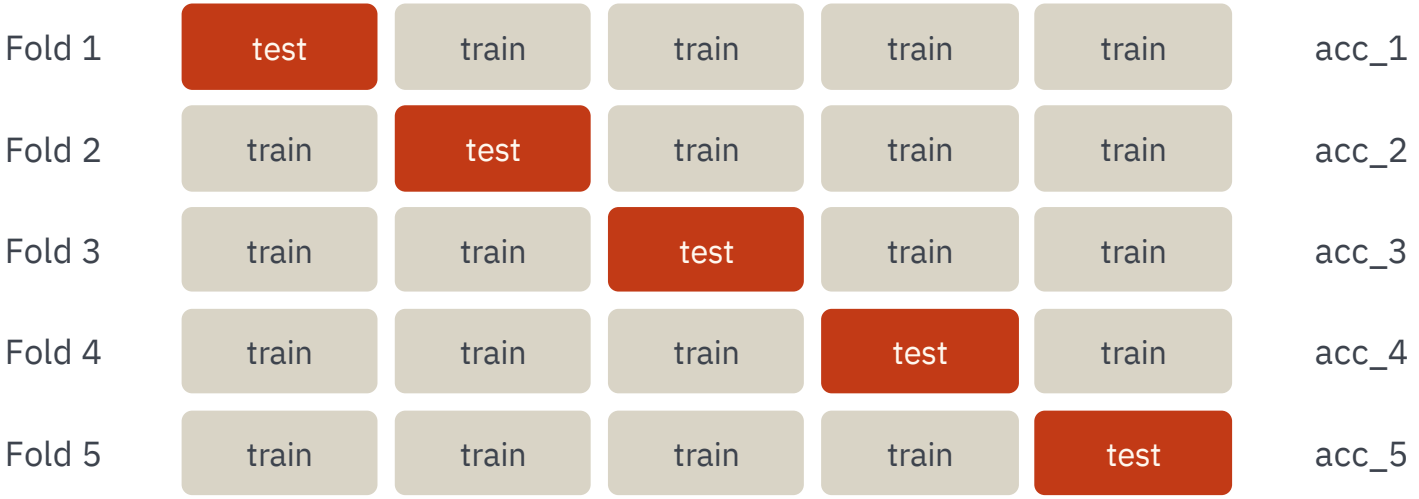
Report the result

Touch it once, at the very end. If you tune on it, the number means nothing.

Overfitting: train loss keeps falling, validation accuracy stops improving or gets worse

The model learns the noise in the training images instead of the digits. The validation set is how you notice. Typical split: 80 / 10 / 10, or use the fixed test set that comes with the dataset.

K-fold cross validation: every point is tested once



Split into k folds

Train on k – 1, test on the one left out.
Repeat k times, a different fold each time.

Report mean ± std

The mean is a better estimate than one split.
The std tells you how much to trust it.

Cost: k trainings

Five folds means five full training runs for every model you want to compare.

When is k-fold worth it?

Use it when

- The dataset is small, so one validation split is noisy
- One training run is cheap, seconds or minutes
- You need error bars to compare two models honestly
- Classic ML: linear models, trees, small networks

Skip it when

- One training run takes hours or days
- The dataset is large, so one validation split is already stable
- Most deep learning today: one fixed train / val / test split
- Instead: repeat a few random seeds and report mean ± std

In scientific ML the data is often expensive, so k-fold comes back

A few hundred simulations or experiments is a small dataset. There, five folds of a small network are often worth the cost.

Outline

04 **Practice**

CNN for handwritten digital recognition

Classify handwritten digits

Build the CNN, train it on MNIST with cross-entropy, and check the accuracy on data it has not seen. Then compare it with an MLP, and try k-fold on a small subset.