Lecture 3: PyTorch networks and training

CSC 352 / CSC 552 · 2026-09-03

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

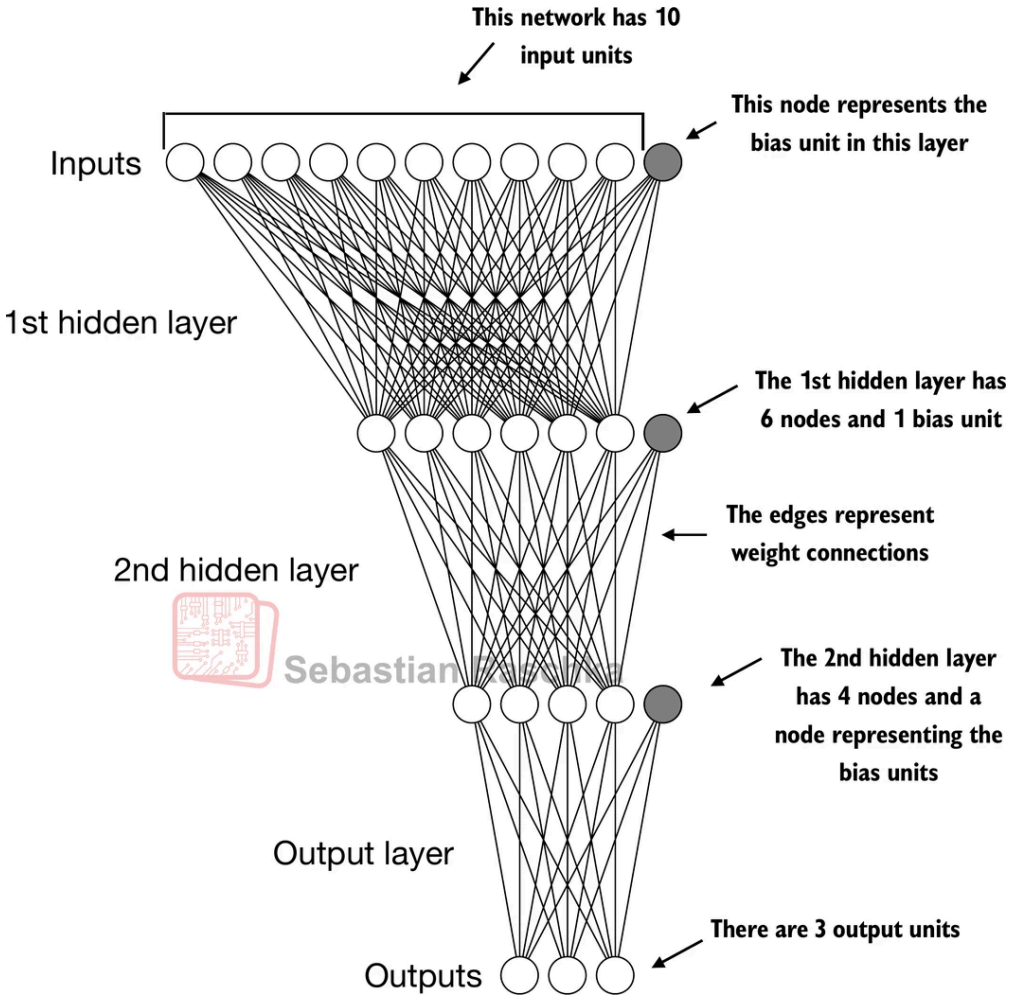
Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. On tensors, what is the difference between `*` and `@`?
2. What does calling `.backward()` write into `.grad`?
3. In one sentence, what is autograd for?

A.5 Implementing multilayer neural networks

A **neural network** here is a stack of mix-and-squash steps.

- `nn.Module` is PyTorch's box: knobs plus a `forward` recipe.
- `Linear` mixes inputs with knobs (weights and a bias), like `z = xw + b` last time, for many numbers at once.
- **ReLU**: if a number is negative, make it 0; if positive, leave it. Another squash, different from sigmoid. Later lectures use more of these.
- A **hidden layer** is a pile of numbers in the middle — not the final label yet.
- `forward` returns **logits** (raw scores), same word as Lecture 1. Softmax and loss show up in the training loop.



In [5]:
```
import torch
```

The next cell is the net: mix (`Linear`) then squash (`ReLU`), twice, then scores out.

In [7]:
```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(

            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),

            # 2nd hidden layer
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),

            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
```

```
        logits = self.layers(x)
        return logits
```

This class takes `num_inputs` and `num_outputs`. Hidden sizes **30** and **20** are written in the cell — we chose them; the data did not. Next we build a **50-in, 3-out** example just to look at the layers. The training loop later uses **2-in, 2-out** (the toy table).

In [9]:
```
model = NeuralNetwork(50, 3)
```

`print(model)` lists the layers: 50 → 30 → 20 → 3.

In [11]:
```
print(model)
```

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

`numel()` counts knobs (weights and biases) the trainer may change.

In [13]:
```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

```
Total number of trainable model parameters: 2213
```

These numbers are the first layer's weights. They start random; training will change them.

In [15]:
```
print(model.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.0939,  0.0778,  0.0266,  ..., -0.1266, -0.0908, -0.0487],
        [ 0.0185, -0.0729, -0.0017,  ..., -0.0703, -0.1057, -0.1132],
        [ 0.0582,  0.0991, -0.0928,  ..., -0.1204,  0.1111,  0.1160],
        ...,
        [ 0.1023,  0.0386,  0.0728,  ...,  0.1362,  0.0801, -0.0960],
        [-0.0566,  0.0301,  0.0552,  ...,  0.0940, -0.0761,  0.1075],
        [-0.0261,  0.0220,  0.0668,  ..., -0.0832,  0.1229,  0.0672]],
       requires_grad=True)
```

Fix the random seed so we all see the same knobs.

In [17]:
```
torch.manual_seed(123)

model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.0865],
        [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.0297],
        [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.1063],
        ...,
        [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.1351],
        [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.1058],
        [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.0509]],
       requires_grad=True)
```

The first `Linear` is 30 rows × 50 columns: one row of weights per hidden unit.

In [19]:
```
print(model.layers[0].weight.shape)
```

```
torch.Size([30, 50])
```

One fake row with 50 numbers goes in. Three logits come out.

In [21]:
```
torch.manual_seed(123)

X = torch.rand((1, 50))
out = model(X)
print(out)
```

```
tensor([[-0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

Still a **forward** pass (the net still computes logits). `torch.no_grad()` means: do **not** record anything for `backward`. Use this when we are only looking or scoring. The training loop — where we update weights and bias — must **not** be wrapped in `no_grad`.

In [23]:
```
with torch.no_grad():
    out = model(X)
print(out)
```

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

Softmax turns those three logits into three probabilities that add to 1.

In [25]:
```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

```
tensor([[0.3113, 0.3934, 0.2952]])
```

`no_grad` does not skip the forward math. It only skips building the gradient tape. Scoring and accuracy need forward, not `.grad`.

Exercise (A.5)

Build `NeuralNetwork(2, 2)`. Count how many parameters training is allowed to change. Print that count.

In [ ]:
```
# Exercise A.5 — your attempt
model_ex = NeuralNetwork(2, 2)
n = None
print('trainable parameters', n)
```

Solution (A.5). Count parameters the model can change during training.

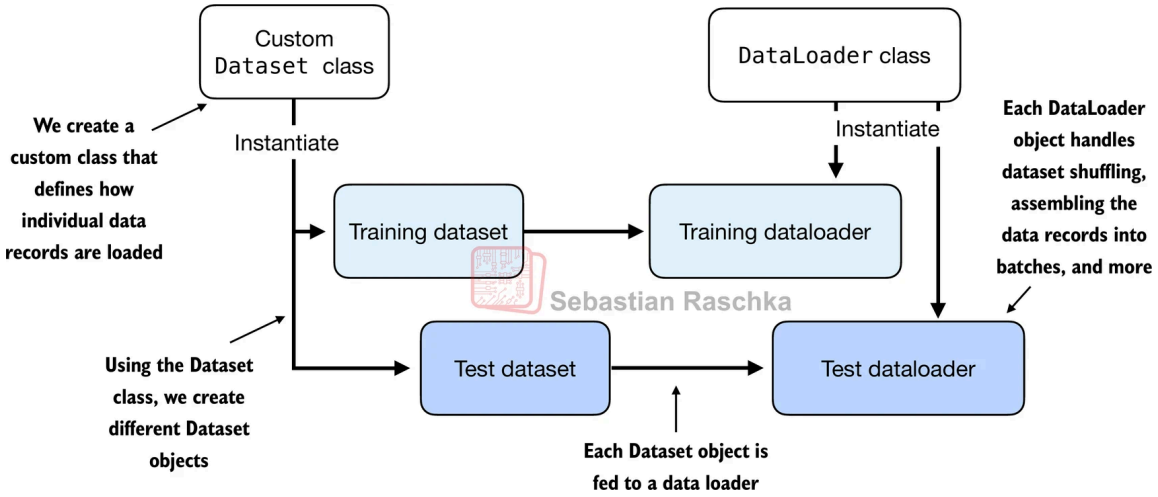
In [30]:
```
# Solution A.5
model_ex = NeuralNetwork(2, 2)
n = sum(p.numel() for p in model_ex.parameters() if p.requires_grad)
print('trainable parameters', n)
```

```
trainable parameters 752
```

A.6 Setting up efficient data loaders

The penguin table was small enough to feed at once. Real text is not.

- A **Dataset** answers "give me row *i*."
- A **DataLoader** is the waiter: it brings a **batch** (a few rows) at a time, and can shuffle.
- One **epoch** = one pass through every training row.
- Keep reading `.shape` : a table is `(rows, features)` ; one Dataset row is `(features,)` ; one batch is `(batch, features)` .



Five training rows. Two numbers on each row are the **features**. `y_train` is one label per row. The print shows the numbers and `.shape` : `(5, 2)` is 5 rows × 2 features.

In [34]:
```
X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
    [2.7, -1.5]
])

y_train = torch.tensor([0, 0, 0, 1, 1])
```

```
print("X_train =")
print(X_train)
print("X_train.shape =", tuple(X_train.shape), "  # (rows, features)")
print("y_train =", y_train)
print("y_train.shape =", tuple(y_train.shape), "  # one label per row")
```

```
X_train =
tensor([[-1.2000,  3.1000],
        [-0.9000,  2.9000],
        [-0.5000,  2.6000],
        [ 2.3000, -1.1000],
        [ 2.7000, -1.5000]])
X_train.shape = (5, 2)   # (rows, features)
y_train = tensor([0, 0, 0, 1, 1])
y_train.shape = (5,)   # one label per row
```

Hold out two rows as **test**. We will not study these while training. Print them the same way: 2 rows × 2 features, and 2 labels.

In [36]:
```
X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])

y_test = torch.tensor([0, 1])

print("X_test =")
print(X_test)
print("X_test.shape =", tuple(X_test.shape), "  # (rows, features)")
print("y_test =", y_test)
print("y_test.shape =", tuple(y_test.shape))
```

```
X_test =
tensor([[-0.8000,  2.8000],
        [ 2.6000, -1.6000]])
X_test.shape = (2, 2)   # (rows, features)
y_test = tensor([0, 1])
y_test.shape = (2,)
```

Wrap each table so we can ask for one row at a time (`Dataset`).

In [38]:
```
from torch.utils.data import Dataset


class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y
```

```
    def __len__(self):
        return self.labels.shape[0]
```

`__getitem__` is "give me one row." `__len__` is "how many rows." That is the whole Dataset contract. The next print is **one row**, not a batch: a vector of length 2 plus one label.

In [40]:
```
train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
print("train rows", len(train_ds), "test rows", len(test_ds))

one_x, one_y = train_ds[0]
print("train_ds[0] x =", one_x)
print("train_ds[0] y =", one_y)
print("one row x.shape =", tuple(one_x.shape), "  # (features,) — not a batc
print("one row y =", one_y, "  # a single label")
```

```
train rows 5 test rows 2
train_ds[0] x = tensor([-1.2000,  3.1000])
train_ds[0] y = tensor(0)
one row x.shape = (2,)   # (features,) — not a batch
one row y = tensor(0)   # a single label
```

A **DataLoader** chops the train rows into batches of 2 and shuffles them. `len(loader)` is how many batches, not how many rows.

In [42]:
```
from torch.utils.data import DataLoader

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)
print("train batches:", len(train_loader), "  # 5 rows, batch size 2 → 3 bat
```

```
train batches: 3   # 5 rows, batch size 2 → 3 batches (last one leftover)
```

Test loader: no shuffle. We do not want a random order when we score.

In [44]:
```
test_ds = ToyDataset(X_test, y_test)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
print("test batches:", len(test_loader))
```

```
test batches: 1
```

Print each train batch. Look at `.shape` first: `(batch_size, features)`. Five rows and batch size 2 → the last batch is smaller, so its first number is 1, not 2.

In [46]:
```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}  x.shape={tuple(x.shape)}  y.shape={tuple(y.shape)
    print(x)
    print("y =", y)
```

```
Batch 1  x.shape=(2, 2)  y.shape=(2,)
tensor([[ 2.3000, -1.1000],
        [-0.9000,  2.9000]])
y = tensor([1, 0])
Batch 2  x.shape=(2, 2)  y.shape=(2,)
tensor([[-1.2000,  3.1000],
        [-0.5000,  2.6000]])
y = tensor([0, 0])
Batch 3  x.shape=(1, 2)  y.shape=(1,)
tensor([[ 2.7000, -1.5000]])
y = tensor([1])
```

`drop_last=True` throws away a leftover batch that is smaller than `batch_size`, so every batch has the same size.

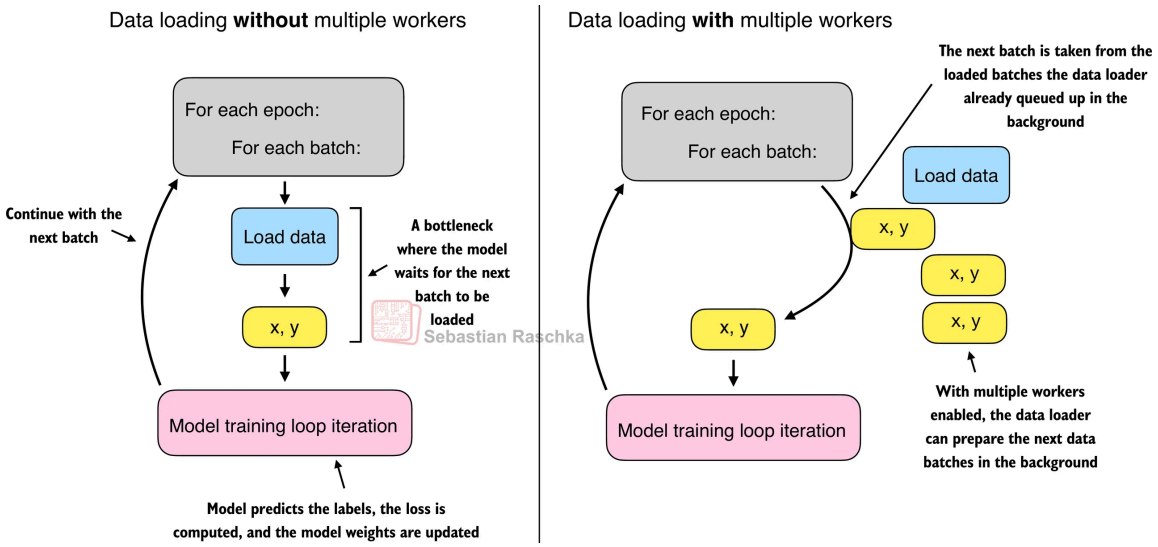
In [48]:
```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
print("train batches after drop_last:", len(train_loader), "  # leftover row
```

```
train batches after drop_last: 2   # leftover row dropped
```

Print batches again. The leftover row is gone, so every `x.shape` is `(2, 2)`: 2 rows, 2 features.

In [50]:
```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}  x.shape={tuple(x.shape)}  y.shape={tuple(y.shape)
    print(x)
    print("y =", y)
```

```
Batch 1  x.shape=(2, 2)  y.shape=(2,)
tensor([[-1.2000,  3.1000],
        [-0.5000,  2.6000]])
y = tensor([0, 0])
Batch 2  x.shape=(2, 2)  y.shape=(2,)
tensor([[ 2.3000, -1.1000],
        [-0.9000,  2.9000]])
y = tensor([1, 0])
```



Exercise (A.6)

Make a DataLoader over `train_ds` that serves 3 rows at a time, does not shuffle, and does not drop a leftover batch. Print each batch index together with the feature shape and the label shape. How many batches are there, and why is the last feature batch not 3 rows?

In [ ]:
```
# A.6 — your attempt
from torch.utils.data import DataLoader

loader_ex = None
for i, (x, y) in enumerate(loader_ex):
    print()
```

Solution (A.6). 5 rows and batch size 3 → two batches: 3 rows, then 2 leftover. Last `x.shape` is `(2, 2)`, not `(3, 2)`.

In [55]:
```
# Solution A.6
from torch.utils.data import DataLoader

loader_ex = DataLoader(
    dataset=train_ds,
    batch_size=3,
    shuffle=False,
    drop_last=False,
)
print("batches:", len(loader_ex))
for i, (x, y) in enumerate(loader_ex):
    print(f"Batch {i+1}  x.shape={tuple(x.shape)}  y.shape={tuple(y.shape)}"
    print(x)
    print("y =", y)
```

```
batches: 2
Batch 1  x.shape=(3, 2)  y.shape=(3,)
tensor([[-1.2000,  3.1000],
        [-0.9000,  2.9000],
        [-0.5000,  2.6000]])
y = tensor([0, 0, 0])
Batch 2  x.shape=(2, 2)  y.shape=(2,)
tensor([[ 2.3000, -1.1000],
        [ 2.7000, -1.5000]])
y = tensor([1, 1])
```

A.7 A typical training loop

Each batch, in order:

1. **Forward:** `logits = model(features)` — guesses. This pass **does** record a tape, because we will call `backward`.
2. **Loss:** `cross_entropy` — how wrong the probabilities are (same family as last class).
3. **`zero_grad`:** forget the last nudge, or new slopes pile onto old ones.
4. **`backward`:** fill `.grad` on every **weight** and **bias**. A **gradient** is the slope of the loss for that knob (if I nudge it, does loss go up or down?), same as last class.
5. **`optimizer.step`:** actually **update** those weights and bias. **SGD** is one optimizer — one recipe that uses the slopes: `new = old − lr × gradient`. **`lr`** (learning rate) is how big a step. SGD is not a second slope.

In the loop, `features` is one of those batches. After `drop_last`, `features.shape` is `(2, 2)`.

`model.train()` when learning; `model.eval()` when only scoring.

In [57]:
```python
import torch.nn.functional as F


torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):

        logits = model(features)

        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx+1:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Optional model evaluation
```

```
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 002/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 002/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 002/002 | Train/Val Loss: 0.00
```

The printed loss should trend down across batches. Those updates changed the weights and bias. Next we **score**: still forward, but we will **not** call `backward`, so we do not need gradients.

In [59]:
```
model.eval()

with torch.no_grad():
    outputs = model(X_train)

print(outputs)
```

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

Softmax → probabilities; `argmax` → the guessed class name.

In [61]:
```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)

predictions = torch.argmax(probas, dim=1)
print(predictions)
```

```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
tensor([0, 0, 0, 1, 1])
```

`argmax` on the raw logits is the same guess — softmax does not change the winner.

In [63]:
```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

```
tensor([0, 0, 0, 1, 1])
```

True where the guess matches the label.

In [65]:
```
predictions == y_train
```

Out[65]:
```
tensor([True, True, True, True, True])
```

How many matches.

In [67]:
```
torch.sum(predictions == y_train)
```

Out[67]:
```
tensor(5)
```

Same check, wrapped as a function. Inside you will see `torch.no_grad()`: accuracy is a forward pass only. No gradient, no update of weights or bias.

In [69]:
```
def compute_accuracy(model, dataloader):

    model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):

        with torch.no_grad():
            logits = model(features)

        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions
        correct += torch.sum(compare)
        total_examples += len(compare)

    return (correct / total_examples).item()
```

Accuracy on the **train** rows (easy to overfit).

In [71]:
```
compute_accuracy(model, train_loader)
```

Out[71]:
```
1.0
```

Accuracy on the **test** rows — the number we believe. Train accuracy can look great from memorizing.

In [73]:
```
compute_accuracy(model, test_loader)
```

Out[73]:
```
1.0
```

`argmax` picks the larger logit — the guessed class. `compute_accuracy` is percent correct, same idea as Lecture 1. It runs forward under `no_grad` because scoring does not update knobs.

Exercise (A.7)

Score the trained model on the train loader and on the test loader. Print both accuracies. Which number would you report, and why? (Same rule as Lecture 1.)

In [ ]:
```
# Exercise A.7 — your attempt
train_acc = None
test_acc = None
print('train', train_acc)
print('test', test_acc)
```

Solution (A.7). Quote **test** accuracy. Train accuracy can be high just from memorizing.

In [78]:
```
# Solution A.7
print('train', compute_accuracy(model, train_loader))
print('test', compute_accuracy(model, test_loader))
```

```
train 1.0
test 1.0
```

A.8 Saving and loading models

`state_dict` is only the knob values, not the Python class. Build the same `NeuralNetwork(...)` first, then pour the knobs in.

In [80]:
```
torch.save(model.state_dict(), "model.pth")
```

Build an empty net with the **same** shape, then pour the saved knobs in.

In [82]:
```
model = NeuralNetwork(2, 2) # needs to match the original model exactly
model.load_state_dict(torch.load("model.pth", weights_only=True))
```

Out[82]:
```
<All keys matched successfully>
```

Exercise (A.8)

Make a new `NeuralNetwork(2, 2)` that has never been trained. Load the knobs saved in `model.pth`. Print the shape of the first layer's weight matrix.

In [ ]:
```
# Exercise A.8 — your attempt
loaded = NeuralNetwork(2, 2)
# Load the saved knobs into `loaded`.
print()
```

Solution (A.8). Same class shape as the saved net; `state_dict` is only the knob values.

In [86]:
```
# Solution A.8
loaded = NeuralNetwork(2, 2)
loaded.load_state_dict(torch.load("model.pth", weights_only=True))
print(loaded.layers[0].weight.shape)
```

```
torch.Size([30, 2])
```

Comprehensive exercises

Comprehensive 1. Train a net whose first hidden size is **10** instead of **30**. Print test accuracy. More parameters are not automatically better.

In [ ]:
```
# Comprehensive 1 — your attempt
class SmallerNet(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = torch.nn.Sequential(
            # Linear → ReLU → Linear → ReLU → Linear
        )
    def forward(self, x):
        return self.layers(x)

small = SmallerNet()
# Train like the demo loop, then print test accuracy.
```

Solution. Fewer knobs. Test accuracy can go up or down.

In [112…
```
# Solution Comprehensive 1
class SmallerNet(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = torch.nn.Sequential(
            torch.nn.Linear(2, 10),
            torch.nn.ReLU(),
            torch.nn.Linear(10, 20),
            torch.nn.ReLU(),
            torch.nn.Linear(20, 2),
        )
    def forward(self, x):
        return self.layers(x)

import torch.nn.functional as F
torch.manual_seed(123)
small = SmallerNet()
opt = torch.optim.SGD(small.parameters(), lr=0.5)
for epoch in range(3):
    small.train()
    for features, labels in train_loader:
        logits = small(features)
        loss = F.cross_entropy(logits, labels)
        opt.zero_grad()
        loss.backward()
```

```
        opt.step()
print('small-net test', compute_accuracy(small, test_loader))
```

```
small-net test 1.0
```

Takeaways

1. `nn.Module` + `forward` = knobs plus a recipe. Hidden layers are middle piles of numbers. Knobs are **weights** and **bias**.
2. `Dataset` / `DataLoader` feed **batches**. One **epoch** is one pass through train.
3. Train: logits → loss → `zero_grad` → `backward` → `optimizer.step`.
4. Accuracy: forward only, wrapped with `no_grad`. Report **test**.
5. `state_dict` saves knobs. Today CPU is enough. GPU / multi-GPU (Appendix A.9) is at the start of the training-pipeline lecture.