# PyTorch Tutorial 2_A minimal learning problem

February 1, 2026

## 1 Scope of this notebook

This notebook introduces a **minimal learning problem** in PyTorch, focusing on how gradient-based learning works at a mechanical level.
It demonstrates how model parameters are updated through repeated application of forward computation, loss evaluation, backpropagation, and parameter updates.

The emphasis is on developing **clear intuition** for: - how losses are defined and computed, - how gradients are obtained using PyTorch's automatic differentiation engine, - how parameter updates lead to learning over multiple iterations, - and how the learning rate affects convergence behaviour.

All concepts are illustrated using explicit tensor operations and a fully manual training loop, without relying on higher-level abstractions.

This notebook is intended as a **conceptual bridge** between basic tensor operations and more advanced PyTorch workflows.
Readers should finish this notebook with a solid understanding of *what learning means computationally*, before introducing optimisers or neural network modules.

It does **not** introduce `nn.Module`, built-in optimisers, datasets, or data loaders.
These abstractions will be introduced in subsequent notebooks once the underlying mechanics are well understood.

**Recommended prerequisites:** Familiarity with PyTorch tensor fundamentals, basic Python programming, and elementary linear algebra.

---

**Author:** Angze Li

**Last updated:** 2026-02-01

**Version:** v1.0

```
[ ]: import torch
```

### 1.0.1 What problem are we setting up?

We consider a very small model with: - an input vector `x` - trainable parameters `w` and `b` - a loss function that measures how wrong the model output is

The model produces an output by combining the input with its parameters:

- `w` (the **weights**) determine how strongly each component of the input `x` influences the output.

- b (the **bias**) provides an offset that allows the model to shift its output independently of the input.
- Together, w and b define the behaviour of the model and are the quantities we want to learn.

The **loss function** compares the model output to the expected output and returns a single scalar value that quantifies the error.

This scalar loss is what drives learning: by analysing how the loss changes with respect to w and b, we can adjust the parameters to improve the model.

```python
# Input and target
x = torch.ones(5)        # input features
y = torch.zeros(3)       # expected output (target)

# Trainable parameters
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
```

### 1.0.2 What these tensors represent

- x is a fixed input vector.

- w is a weight matrix mapping 5 inputs to 3 outputs.

- b is a bias vector added to the output. (Note the dimensionalities of x, w and b.)

- `requires_grad=True` tells PyTorch:

    "Track how this tensor influences the final loss."

Only tensors with `requires_grad=True` will receive gradients.

```python
z = torch.matmul(x, w) + b
print("Model output z:", z)
```

### 1.0.3 What just happened mathematically

This line computes the **forward pass** of the model:

$$z = xW + b$$

- x is shape (5,)
- w is shape (5, 3)
- result z is shape (3,)

Each element of z depends on **all entries of w and b**.

PyTorch remembers this dependency internally.

```python
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
print("Loss:", loss)
```

### 1.0.4 Why we need a loss

The loss is a **single scalar number** that measures how far the model output `z` is from the target `y`.

In this example, we use **binary cross-entropy with logits** (hence the mame of the function `torch.nn.functional.binary_cross_entropy_with_logits()` , which is computed *element-wise* as:

$$\ell(z_i, y_i) = - \left[ y_i \ln(\sigma(z_i)) + (1 - y_i) \ln(1 - \sigma(z_i)) \right]$$

where: - $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function, - $z_i$ is the model output (logit) for the $i$-th entry, - $y_i \in \{0, 1\}$ is the corresponding target.

The total loss is obtained by averaging over all entries:

$$\text{loss} = \frac{1}{N} \sum_i \ell(z_i, y_i)$$

Key points: - The loss is a **scalar**, which is required for backpropagation. - Using logits (raw values `z`) instead of probabilities is **numerically more stable**. - PyTorch combines the sigmoid operation and cross-entropy computation internally.

At this point, PyTorch has built a **computation graph** linking
`w, b → z → loss`.

One can manually verify that the loss computed above follows this formula exactly by applying the sigmoid function to $z$ and evaluating the expression entry by entry.

```
[ ]: print("Gradient of w before backward:", w.grad)
     print("Gradient of b before backward:", b.grad)
```

### 1.0.5 Why gradients are `None`

Gradients are not computed automatically.

Up to now, we have only done a **forward pass**. No differentiation has happened yet.

To compute gradients, we must explicitly ask PyTorch to do so.

```
[ ]: loss.backward()
```

### 1.0.6 What `backward()` actually means

Calling `loss.backward()` tells PyTorch to compute **how the loss changes with respect to each trainable parameter**.

More concretely, PyTorch performs the following steps:

1. **Start from the scalar `loss`**
   The loss is the final output of the computation graph. Gradients are always computed with respect to a scalar, because only then is the gradient well-defined.

3

2. **Traverse the computation graph backwards**
   PyTorch follows the recorded operations in reverse order, moving from the loss back toward the parameters (`w` and `b`).

3. **Apply the chain rule automatically**
   At each operation, PyTorch computes local derivatives and combines them using the chain rule, propagating gradients backward through the graph.

4. **Accumulate gradients in leaf tensors**
   For every tensor marked with `requires_grad=True` that is a *leaf* of the graph (such as `w` and `b`), PyTorch stores the resulting gradients in the `.grad` attribute.

After this process: - `w.grad` contains $\partial \text{ loss}/\partial w$ - `b.grad` contains $\partial \text{ loss}/\partial b$

This entire backward pass from the loss back to the parameters is known as **backpropagation**.

```
[ ]: print("Gradient of w:\n", w.grad)
     print("Gradient of b:\n", b.grad)
```

### 1.0.7 How to interpret these gradients

- `w.grad` tells us how the loss changes if each entry of `w` is *increased* slightly.
- `b.grad` tells us how the loss changes if each entry of `b` is *increased* slightly.

The gradients have the **same shape** as the parameters they belong to. This is not an accident — it allows **parameter-wise** updates.

```
[ ]: learning_rate = 0.1

     with torch.no_grad():
         w -= learning_rate * w.grad
         b -= learning_rate * b.grad
```

### 1.0.8 Learning rate and `torch.no_grad(`

The variable `learning_rate` controls **how large each parameter update is**.
It determines how far we move the parameters in the direction that reduces the loss.

- A larger learning rate leads to **bigger steps**, which may speed up learning but risk overshooting.
- A smaller learning rate leads to **smaller, more cautious steps**, which are more stable but slower.

In this example, `learning_rate = 0.1` is chosen as a simple, reasonable value to clearly illustrate the update step.

---

We use `torch.no_grad()` because: - Updating parameters is **not** part of the model itself - We do not want PyTorch to track these operations

Otherwise, the computation graph would grow incorrectly.

Expand the two cells below to learn more about learning rate and the necessity of `torch.no_grad()`.

Learning rate: scale, intuition, and practical behaviour

**What is the usual range of the learning rate?**  There is no single "correct" learning rate, but in practice:

- Common values range from **1e—4 to 1e—1**
- Typical starting points are **0.1**, **0.01**, or **0.001**
- The appropriate value depends on:
    - the model,
    - the loss function,
    - and the scale of the gradients

In this notebook, we use `learning_rate = 0.1` purely for illustration, not because it is optimal.

---

**Intuition: the hill analogy**  Imagine the loss as a landscape with hills and valleys:

- The **height** represents the loss value
- The **slope** represents the gradient
- The parameters (`w`, `b`) represent your current position on the hill

The learning rate controls **how large a step you take downhill**:

- Too small → you move very slowly and make little progress
- Too large → you may overshoot the valley and oscillate
- Just right → you move steadily toward a minimum

---

**Why the learning rate matters**  The gradient tells you *which direction* reduces the loss. The learning rate determines *how far* you move in that direction.

Effective learning requires **both** a meaningful gradient and an appropriate learning rate.

Why torch.no_grad() is needed during parameter updates

We use `torch.no_grad()` because updating model parameters is **not part of the forward computation** that defines the model itself.

During the forward pass, PyTorch records operations to build a computation graph.
However, parameter updates are a **bookkeeping step** performed after gradients have been computed.

If parameter updates were performed without `torch.no_grad()`: - PyTorch would attempt to track these updates as part of the computation graph - The graph would incorrectly include the update operations themselves - Subsequent calls to `backward()` could fail or produce incorrect gradients

Using `torch.no_grad()` temporarily disables gradient tracking, ensuring that: - parameter updates do not become part of the computation graph - each training step starts from a clean graph - gradients are computed only for the forward computation

This separation between **model computation** and **parameter updates** is essential for correct and stable training.

```
[ ]: w.grad.zero_()
     b.grad.zero_()

     print("w:\n", w)
     print("b:\n", b)
```

### 1.0.9 Why gradients must be cleared

By default, PyTorch **accumulates gradients**.

This means: - Calling `backward()` multiple times without resetting gradients will **add new gradients** to the existing values stored in `.grad` - This behaviour is intentional and allows gradients to be accumulated across multiple backward passes (e.g. when summing losses)

However, for most training loops—especially when learning—this accumulation can be confusing and lead to subtle bugs.

Clearing gradients explicitly ensures that each backward pass computes gradients **only for the current forward computation**.

---

### 1.0.10 Make it a loop

So far, we have seen how to compute gradients and perform a **single parameter update**.

Learning, however, does not happen in one step.
Instead, the process of: 1. computing the loss, 2. computing gradients via backpropagation, 3. updating parameters,

is repeated **many times**.

Each repetition is called a **training step** (or iteration).
Over successive steps, the parameters are gradually adjusted to reduce the loss.

```
[ ]: # Simple training loop to demonstrate learning

     learning_rate = 0.1
     num_steps = 20

     for step in range(num_steps):
         # Forward pass
         z = torch.matmul(x, w) + b
         loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)

         # Backward pass
         loss.backward()

         # Parameter update
         with torch.no_grad():
             w -= learning_rate * w.grad
```

```
        b -= learning_rate * b.grad

    # Clear gradients
    w.grad.zero_()
    b.grad.zero_()

    # Monitor progress
    print(f"Step {step:2d} | Loss: {loss.item():.6f}")
```

### 1.0.11 The training loop: how learning emerges

- Each loop iteration performs one **forward**–**backward**–**update** cycle.
- The loss is recomputed using the updated parameters at every step.
- If the learning rate is reasonable, the loss should gradually decrease.

This loop demonstrates the core mechanism behind training neural networks: repeated application of backpropagation and parameter updates.

### 1.0.12 What backpropagation really did here

- We computed a forward pass to get a scalar loss
- PyTorch recorded how that loss depends on parameters
- `loss.backward()` computed all required gradients automatically
- Gradients told us how to update parameters to reduce the loss

No neural network magic — just calculus applied systematically.

---

### 1.0.13 Convergence in gradient-based learning

At each iteration: 1. A forward pass computes the model output and the loss. 2. Backpropagation computes gradients of the loss with respect to the parameters. 3. The parameters are updated in the direction that reduces the loss. 4. Gradients are cleared in preparation for the next step.

As the loop progresses, the loss stored in `loss_history` decreases and eventually falls below the chosen threshold.

This indicates that successive parameter updates are making **smaller and smaller improvements** (see the figure below for the visualisation of decrease in `loss`), and the model is approaching a minimum of the loss function.

The stopping condition is: "'python loss_value > 0.01

```
[ ]: %%time

    x = torch.ones(5)
    y = torch.zeros(3)
    w = torch.randn(5, 3, requires_grad=True)
    b = torch.randn(3, requires_grad=True)

    learning_rate = 0.1
```

```python
loss_value = float("inf")
learning_step = 0
max_steps = 1000

loss_history = []

while loss_value > 0.01 and learning_step < max_steps:
    # Forward pass
    z = torch.matmul(x, w) + b
    loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)

    # Backward pass
    loss.backward()

    # Parameter update
    with torch.no_grad():
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad

    # Clear gradients
    w.grad.zero_()
    b.grad.zero_()

    # Extract scalar loss
    loss_value = loss.item()
    loss_history.append(loss_value)

    # Monitor progress
    if learning_step % 100 == 0:
        print(f"Step {learning_step:4d} | Loss: {loss_value:.6f}")

    learning_step += 1

print("\nTotal learning steps:", learning_step)
print("Final loss:", round(loss_value, 6))
```

```python
import matplotlib.pyplot as plt

plt.plot(loss_history)
plt.xlabel("Training step")
plt.ylabel("Loss")
plt.title("Loss convergence over training")
plt.show()
```

### 1.0.14   The effect of learning rate

The experiment below examines how the **learning rate** affects convergence in gradient-based learning.

For each value in `learning_rates`, the model is re-initialised and trained under identical conditions. This ensures that any differences in behaviour arise solely from the choice of learning rate, rather than from initialisation or model structure.

The reported results show how many steps are required for the loss to fall below the threshold "'python loss_value > 0.01

```python
learning_rates = [0.001, 0.01, 0.1, 1.0, 10, 100]
max_steps = 1000
threshold = 0.01

for lr in learning_rates:
    # Re-initialise model for fair comparison
    x = torch.ones(5)
    y = torch.zeros(3)
    w = torch.randn(5, 3, requires_grad=True)
    b = torch.randn(3, requires_grad=True)

    loss_value = float("inf")
    learning_step = 0
    loss_history = []

    while loss_value > threshold and learning_step < max_steps:
        # Forward pass
        z = torch.matmul(x, w) + b
        loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)

        # Backward pass
        loss.backward()

        # Parameter update
        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad

        # Clear gradients
        w.grad.zero_()
        b.grad.zero_()

        # Extract scalar loss
        loss_value = loss.item()
        loss_history.append(loss_value)

        learning_step += 1

    print(f"learning_rate = {lr:<5} | steps = {learning_step:<4} | final loss =␣
    ↪{loss_value:.6f}")
```

### 1.0.15 Interpreting learning rate effects

In this simple and well-conditioned example, larger learning rates reduce the number of steps required for convergence, allowing the loss to fall below the chosen threshold more quickly.

However, this behaviour does not generalise to all optimisation problems. In more complex loss landscapes, overly large learning rates can lead to: - unstable updates, - divergence of the loss, - or convergence to poor solutions that generalise poorly.

For this reason, smaller learning rates are often preferred in practice, as they provide greater **robustness and stability** across a wide range of models and training conditions.

### 1.0.16 Summary

In this section, we examined a minimal learning problem and followed the entire training process step by step.

Starting from a simple linear model, we: - defined a loss function, - used backpropagation to compute gradients, - updated parameters using gradient descent, - and observed convergence over multiple iterations.

This example captures the essential mechanics underlying gradient-based learning. In the next sections, these same ideas will be reused and extended using higher-level PyTorch abstractions.