PyTorch Workshop 1_From Gradient Flow to Optimisation Intuition_v1.0

February 6, 2026

1 Workshop 1: From Gradient Flow to Optimisation Intuition

This workshop builds on Tutorials 1–4 and closes **Part 1** of the series, consolidating core ideas about PyTorch `autograd` and gradient flow before moving on to explicit optimisation in Workshop 2 and Part 2.

Rather than treating gradients as black-box training signals, the workshop emphasises **gradients as sensitivity measures**, especially in settings where model outputs are **tensor-valued** and gradients are computed using **explicit upstream directions**.

The problems are designed to shift perspective: - from *"what gradient does PyTorch give me?"* - to *"what does this gradient represent, and why?"*

The emphasis is on developing intuition for: - how gradients flow through structured linear and nonlinear computations, - how upstream gradients select directions in output space, - how `.grad` reflects sensitivity rather than optimisation, - and how gradient structure anticipates optimisation behaviour.

Key ideas explored include: - vector–Jacobian products as the fundamental object computed by `backward(v)`, - the role of upstream gradients in shaping gradient flow, - sparsity and structure in gradients induced by linear maps and nonlinearities (e.g., ReLU), - interpreting gradients statistically and geometrically rather than procedurally, - and using controlled experiments to reason about gradient behaviour.

This workshop serves as a conceptual bridge between: - `autograd` mechanics and gradient flow (Tutorials 3–4), - and **objective design and optimisation dynamics** (Workshop 2 and Part 2).

The focus is intentionally *not* on training pipelines, datasets, or optimisers, but on understanding how gradients behave in controlled settings—laying the groundwork for effective optimisation.

Recommended prerequisites: - Familiarity with PyTorch tensors and `.backward()` - Understanding of scalar vs tensor-valued gradients - Basic comfort with linear algebra and nonlinear activations

---

Author: Angze Li

Last updated: 2026-02-06

Version: v1.0

1.1 Problem 1: Sensitivity of a Feature Transformation (Warm-up)

In many models, inputs are transformed into feature representations before being used by a downstream objective. Understanding which inputs influence which features is critical for interpretability and optimisation.

Consider the following setup:

```
x = torch.randn(6, requires_grad=True)

W = torch.tensor([
    [1.0,  0.0,  0.0, 0.0, 0.0, 0.0],
    [0.0,  1.0,  0.0, 0.0, 0.0, 0.0],
    [0.0,  0.0,  2.0, 0.0, 0.0, 0.0],
    [0.0,  0.0,  0.0, 3.0, 0.0, 0.0],
], requires_grad=False)

out = torch.relu(W @ x) # what does this line do?
```

This produces a 4-dimensional tensor output, not a scalar.

1.1.1 Task

1. Construct an upstream gradient vector `v` such that:
   - the last feature of out is weighted most heavily,
   - the first two features are ignored.
2. Call:

```
out.backward(v)
```

3. Inspect `x.grad`.

1.1.2 Questions to think about

- Which components of `x` receive non-zero gradients?
- How does the structure of `W` affect gradient flow?
- How does ReLU change which inputs are "active"?
- Why is `x.grad` sparse in some cases?

1.1.3 Hint (optional)

Think in terms of which inputs influence which outputs, and which outputs are being emphasised by `v`.

1.1.4 Learning outcomes

After this problem, you should be comfortable with: - interpreting `.grad` as a sensitivity map, - reasoning about gradient flow through linear + nonlinear layers, - understanding how upstream weighting reshapes gradient influence.

This problem bridges Tutorials 3 and 4 cleanly and warms up the optimisation mindset.

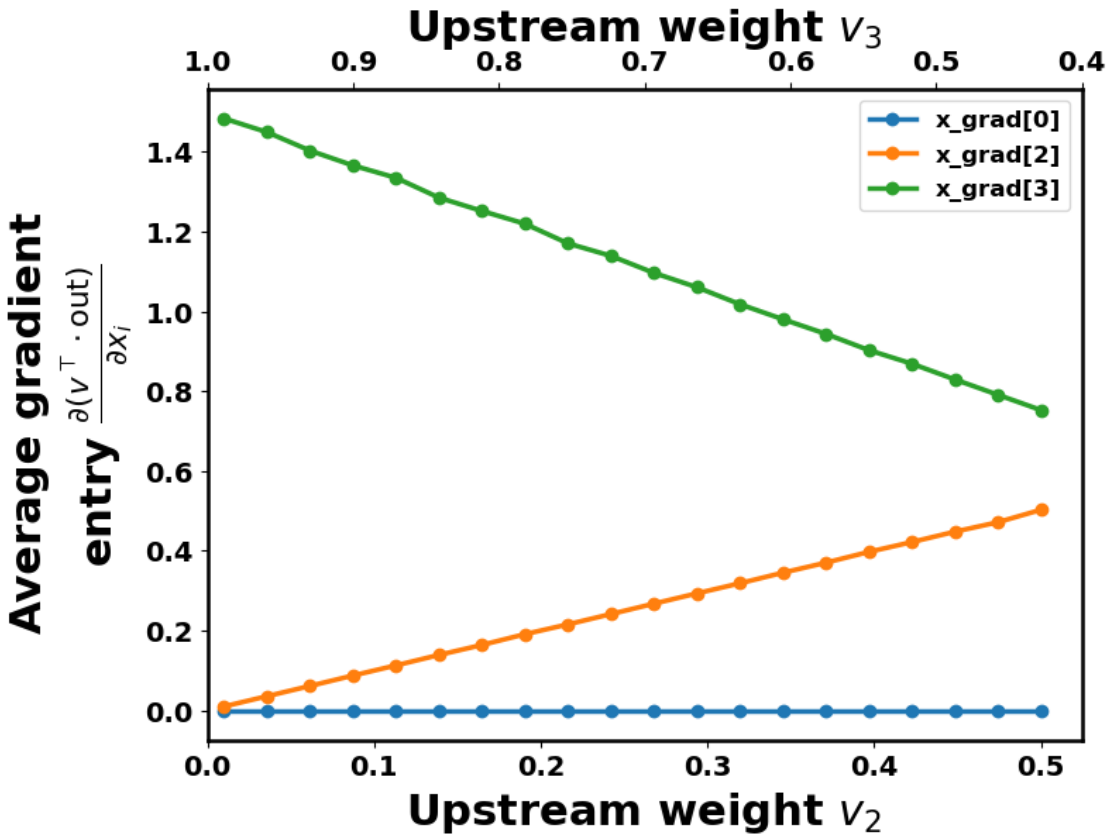
1.2 Solution 1

[152]:
```
%%time
import torch
import numpy as np

W = torch.tensor([
        [1.0,  0.0,  0.0, 0.0, 0.0, 0.0],
        [0.0,  1.0,  0.0, 0.0, 0.0, 0.0],
        [0.0,  0.0,  2.0, 0.0, 0.0, 0.0],
        [0.0,  0.0,  0.0, 3.0, 0.0, 0.0],
    ], requires_grad=False)

weight = np.linspace(0.01,0.50,20)
repeats = 100000
x_grad0, x_grad2, x_grad3, x_grad_ratio = [], [], [], []

for i in range(len(weight)):
    v = torch.zeros(4)
    v[2] = weight[i]
    v[3] = 1-weight[i]

    x_grad_sum = torch.zeros(6)

    for j in range(repeats):
        x = torch.randn(6, requires_grad=True)
        out = torch.relu(W @ x)
        out.backward(v)

        with torch.no_grad():
            x_grad_sum += x.grad

    print(f"Iteration {i}, Weight {round(weight[i],3)}, Monte Carlo average of␣
 ↪x_grad: {x_grad_sum/repeats}")
    x_grad_ratio.append(float(x_grad_sum[3]/x_grad_sum[2]))
    x_grad3.append(x_grad_sum[3]/repeats)
    x_grad2.append(x_grad_sum[2]/repeats)
    x_grad0.append(x_grad_sum[0]/repeats)

print(f"\nAveraged x_grad[3]/x_grad[2] at each weight:\n {x_grad_ratio}")
```

```
Iteration 0, Weight 0.01, Monte Carlo average of x_grad: tensor([0.0000, 0.0000,
0.0100, 1.4830, 0.0000, 0.0000])
Iteration 1, Weight 0.036, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.0357, 1.4488, 0.0000, 0.0000])
Iteration 2, Weight 0.062, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.0614, 1.4025, 0.0000, 0.0000])
Iteration 3, Weight 0.087, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.0875, 1.3653, 0.0000, 0.0000])
Iteration 4, Weight 0.113, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.1129, 1.3344, 0.0000, 0.0000])
Iteration 5, Weight 0.139, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.1391, 1.2846, 0.0000, 0.0000])
Iteration 6, Weight 0.165, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.1641, 1.2513, 0.0000, 0.0000])
Iteration 7, Weight 0.191, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.1911, 1.2192, 0.0000, 0.0000])
Iteration 8, Weight 0.216, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.2157, 1.1701, 0.0000, 0.0000])
Iteration 9, Weight 0.242, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.2415, 1.1388, 0.0000, 0.0000])
Iteration 10, Weight 0.268, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.2675, 1.0961, 0.0000, 0.0000])
Iteration 11, Weight 0.294, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.2933, 1.0602, 0.0000, 0.0000])
Iteration 12, Weight 0.319, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.3186, 1.0176, 0.0000, 0.0000])
Iteration 13, Weight 0.345, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.3451, 0.9800, 0.0000, 0.0000])
Iteration 14, Weight 0.371, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.3702, 0.9434, 0.0000, 0.0000])
Iteration 15, Weight 0.397, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.3973, 0.9026, 0.0000, 0.0000])
Iteration 16, Weight 0.423, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.4219, 0.8686, 0.0000, 0.0000])
Iteration 17, Weight 0.448, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.4479, 0.8292, 0.0000, 0.0000])
Iteration 18, Weight 0.474, Monte Carlo average of x_grad: tensor([0.0000,
0.0000, 0.4716, 0.7908, 0.0000, 0.0000])
Iteration 19, Weight 0.5, Monte Carlo average of x_grad: tensor([0.0000, 0.0000,
0.5026, 0.7526, 0.0000, 0.0000])

Averaged x_grad[3]/x_grad[2] at each weight:
 [147.82485961914062, 40.62276077270508, 22.836936950683594, 15.60584831237793,
11.819583892822266, 9.23569107055664, 7.6236252784729, 6.379469871520996,
5.424524784088135, 4.715577602386475, 4.097631454467773, 3.6148970127105713,
3.1935925483703613, 2.83980393409729, 2.547971487045288, 2.271894693374634,
2.05904221534729, 1.8515218496322632, 1.6770366430282593, 1.4972246885299683]
CPU times: user 29.9 s, sys: 20.5 ms, total: 29.9 s
```

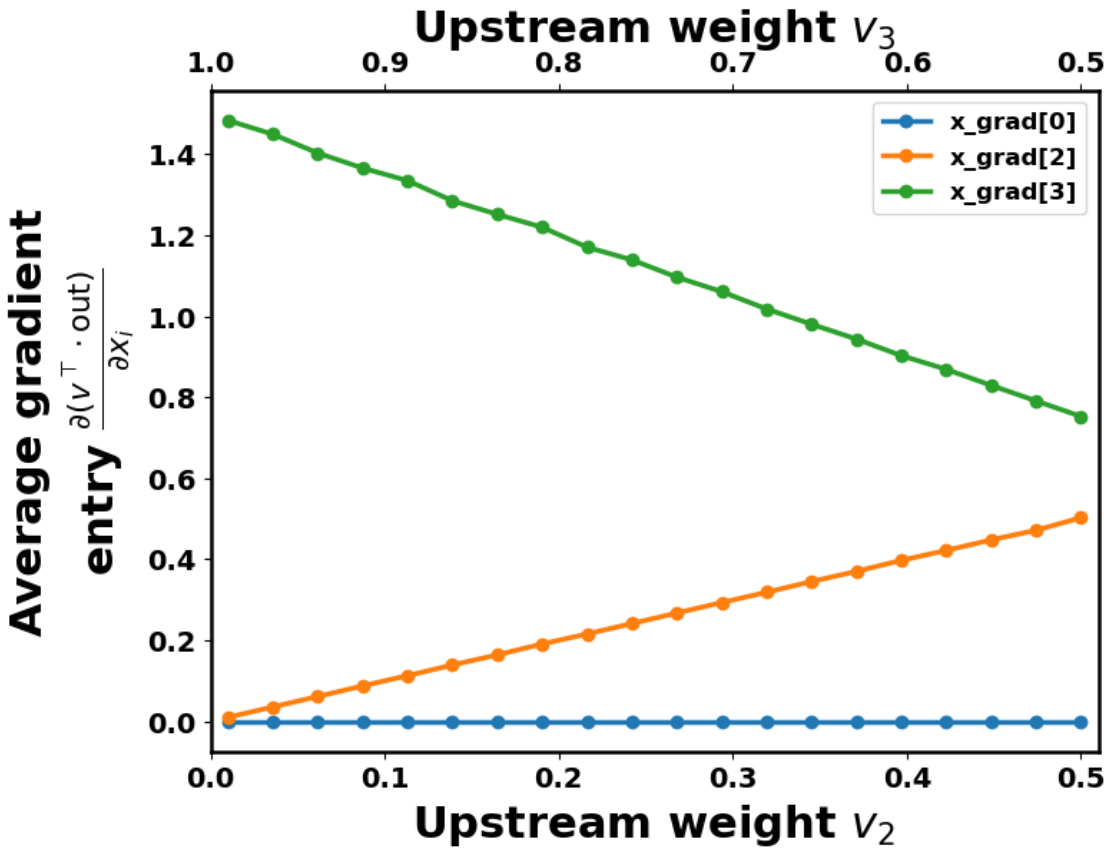
```
Wall time: 29.9 s
```

[172]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,6))
ax = plt.gca()
for spine in ax.spines.values():
        spine.set_linewidth(1.8)


plt.plot(weight, np.array(x_grad0), '-o', linewidth=2.5, label="x_grad[0]")
plt.plot(weight, np.array(x_grad2), '-o', linewidth=2.5, label="x_grad[2]")
plt.plot(weight, np.array(x_grad3), '-o', linewidth=2.5, label="x_grad[3]")
plt.xlim(0)
plt.xticks(fontsize=14, fontweight='bold')
plt.yticks(fontsize=14, fontweight='bold')
plt.legend(loc='upper right',prop={"size": 12, "weight": "bold"})
plt.xlabel(r"Upstream weight $v_2$", fontsize=22, fontweight='bold')
plt.ylabel("Average gradient \n"r"entry $\frac{\partial (v^\top \cdot␣
 ↪\mathrm{out})}{\partial x_i}$", fontsize=22, fontweight='bold')

ax_top = ax.twiny()
ax_top.set_xlim(ax.get_xlim())
ticks = ax.get_xticks()
ax_top.set_xticks(ticks)
ax_top.set_xticklabels([f"{1 - t:.1f}" for t in ticks],fontsize=14,␣
 ↪fontweight='bold')
ax_top.set_xlabel(r"Upstream weight $v_3$", fontsize=22, fontweight='bold')

plt.show()
```



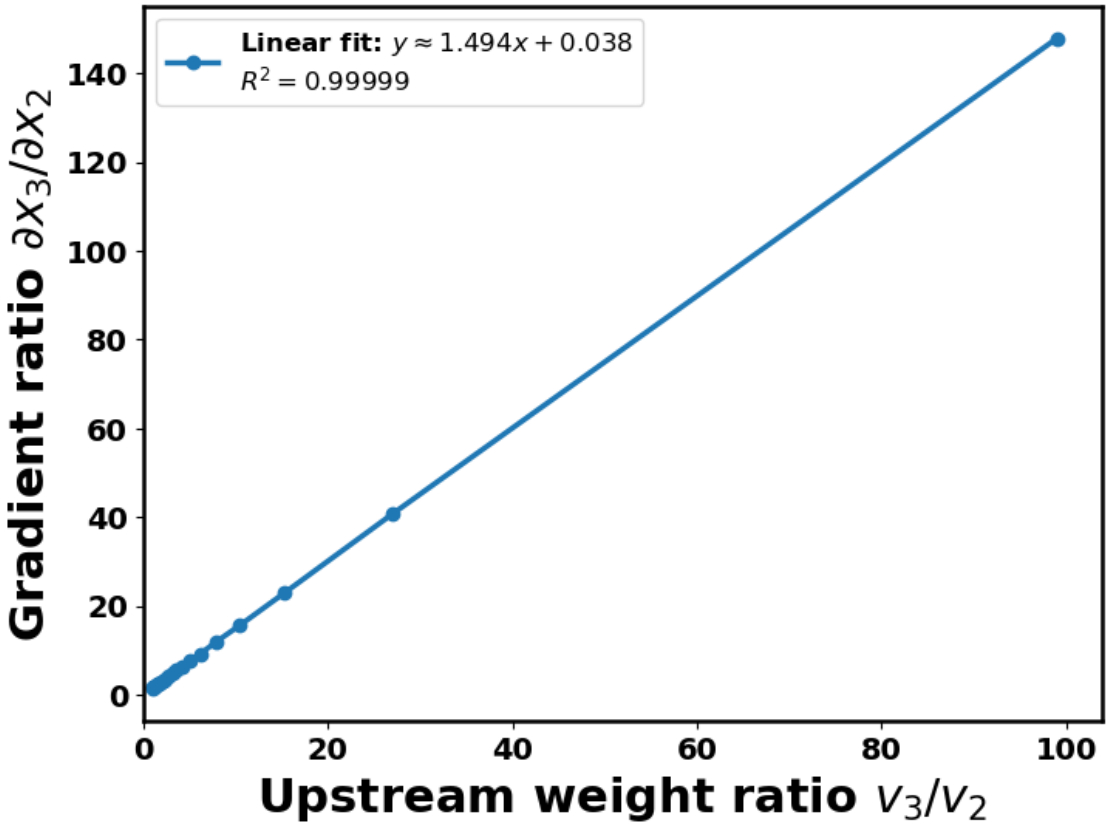
[183]:
```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(8, 6))

for spine in ax.spines.values():
    spine.set_linewidth(1.8)

ax.plot(weight, np.array(x_grad0), "-o", lw=2.5, label="x_grad[0]")
ax.plot(weight, np.array(x_grad2), "-o", lw=2.5, label="x_grad[2]")
ax.plot(weight, np.array(x_grad3), "-o", lw=2.5, label="x_grad[3]")


ax.set(xlim=(0, 0.51),
       xlabel=r"Upstream weight $v_2$",
       ylabel="Average gradient\n"
              r"entry $\frac{\partial (v^\top\cdot \mathrm{out})}{\partial␣
 ↪x_i}$")
ax.xaxis.label.set_fontsize(22)
ax.yaxis.label.set_fontsize(22)
```

```
ax.xaxis.label.set_fontweight("bold")
ax.yaxis.label.set_fontweight("bold")

ax.tick_params(axis="both", labelsize=14)
ax.legend(loc="upper right", prop={"size": 12, "weight": "bold"})


secax = ax.secondary_xaxis("top", functions=(lambda v2: 1 - v2, lambda v3: 1 -␣
 ↪v3))
secax.set(xlabel=r"Upstream weight $v_3$")
secax.xaxis.label.set_fontsize(22)
secax.xaxis.label.set_fontweight("bold")

bottom = ax.get_xticks()
secax.set_xticks(1 - bottom)
secax.set_xticklabels([f"{t:.1f}" for t in 1 - bottom])
secax.tick_params(axis="x", labelsize=14)

for t in ax.get_xticklabels() + ax.get_yticklabels() + secax.get_xticklabels():
    t.set_fontweight("bold")
ax.xaxis.label.set_fontweight("bold")
ax.yaxis.label.set_fontweight("bold")
secax.xaxis.label.set_fontweight("bold")

plt.show()
```



[185]:
```
weight_ratio = (1 - weight) / weight
x_grad_ratio = np.array(x_grad_ratio)

m, c = np.polyfit(weight_ratio, x_grad_ratio, 1)
x_grad_fit = m * weight_ratio + c

ss_res = np.sum((x_grad_ratio - x_grad_fit) ** 2)
ss_tot = np.sum((x_grad_ratio - np.mean(x_grad_ratio)) ** 2)
R2 = 1 - ss_res / ss_tot if ss_tot != 0 else np.nan

fig, ax = plt.subplots(figsize=(8, 6))
for spine in ax.spines.values():
    spine.set_linewidth(1.8)

ax.plot(weight_ratio, x_grad_ratio, '-o', linewidth=2.5, label=rf"Linear fit:␣
 ↪$y \approx {m:.3f}x + {c:.3f}$" f"\n$R^2 = {R2:.5f}$")

ax.set_xlim(0)
ax.set_xlabel(r"Upstream weight ratio $v_3 / v_2$")
ax.set_ylabel(r"Gradient ratio $\partial x_3 / \partial x_2$")
```

```
ax.xaxis.label.set_fontsize(22)
ax.yaxis.label.set_fontsize(22)
ax.xaxis.label.set_fontweight("bold")
ax.yaxis.label.set_fontweight("bold")

ax.tick_params(axis='both', labelsize=14)
for tick in ax.get_xticklabels() + ax.get_yticklabels():
    tick.set_fontweight("bold")

ax.legend(loc='upper left', prop={"size": 12, "weight": "bold"})

plt.show()
```



Explanation 1

1.2.1 Understanding the solution

This exercise explores how gradients flow through a simple linear–nonlinear system when the output is tensor-valued and gradients are computed via a vector–Jacobian product.

The model we studied is:

$$\text{out} = \text{ReLU}(Wx),$$

with a user-chosen upstream gradient `v` supplied to `backward()`.

1.2.2 Which components of `x` receive non-zero gradients?

Only a **subset of the components of `x`** ever receive non-zero gradients.

This happens for two reasons: 1. **Linear structure**: Each row of `W` depends only on one component of ‘x.

`As a result, each output entry is influenced by exactly one input coordinate.`

2. **Upstream weighting**: Only the output components with non-zero entries in `v` contribute to the gradient.

If a component of `x` does not influence any weighted output entry, its gradient is exactly zero.

1.2.3 How does the structure of `W` affect gradient flow?

The matrix `W` acts as a **routing mechanism** for gradients.

Because `W` is sparse and diagonal-like: - each output dimension maps directly to a single input dimension, - gradient flow is strictly localised, - there is no mixing between unrelated components of `x`.

This makes the gradient behaviour easy to interpret: changing the weight of one output only affects the gradient of its corresponding input.

1.2.4 How does ReLU change which inputs are “active”?

ReLU introduces **input-dependent gating**.

For each output component:

$$\text{ReLU}(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

This means: - gradients flow **only when the pre-activation is positive**, - inputs corresponding to negative pre-activations are *temporarily invisible* to backpropagation.

In this exercise, averaging gradients over many random samples reveals that: - each active ReLU contributes gradients roughly **half** the time, - inactive paths contribute nothing.

This explains the appearance of **fractional average gradients** in the Monte Carlo results.

1.2.5 Why is `x.grad` sparse in some cases?

`x.grad` is sparse because gradient flow is conditional: - conditional on which outputs are weighted by `v`, - conditional on which ReLU units are active, - conditional on the sparsity pattern of `W`.

If any link in this chain is broken (zero weight, inactive ReLU, or missing linear connection), the gradient for that input vanishes.

This sparsity is not a numerical artifact — it is a structural property of the computation graph.

1.2.6 Why are the figures necessary?

The figures serve two essential purposes.

1 Revealing how upstream weights redistribute gradients

The first figure shows how changing the upstream weights `v` **redistributes gradient magnitude** across different input components.

While the total sensitivity remains bounded, its allocation shifts smoothly as `v` changes. This visually reinforces the idea that:

> vector–Jacobian products do not create new gradients — they reweight existing ones.

2 Confirming a linear relationship between gradient ratios and weight ratios

The second figure plots:

$$\frac{\partial x_3}{\partial x_2} \quad \text{vs.} \quad \frac{v_3}{v_2}$$

The near-perfect linear relationship confirms that: - gradient ratios scale linearly with upstream weight ratios, - the computation behaves exactly as predicted by the chain rule, - ReLU's gating only affects *when* gradients flow, not *how* they scale once active.

This provides strong empirical validation of the theory behind vector–Jacobian products.

Key takeaway

This exercise demonstrates that in PyTorch: - gradients of tensor-valued outputs are **not single objects**, but **directional sensitivities**, - `backward(v)` computes how a *chosen direction in output space* propagates back to inputs, - sparsity and structure in gradients arise naturally from the computation graph itself.

Together, the code and figures make vector–Jacobian products concrete, intuitive, and experimentally verifiable — exactly the conceptual bridge needed before moving on to optimisation in Part 2.