

Tiny Network, Visible Computation

Gavin · AI-assisted course design

Contents

1 Goal	3
2 The fixed network	3
3 Part 1 — Make a hand trace	3
4 Part 2 — Predict interventions	4
5 Part 3 — Implement or independently recalculate	4
5.1 Coding route	4
5.2 Paper route	5
5.3 Interpret the comparison	5
6 Part 4 — Optional one-parameter training step	6
7 Checkpoints and solutions	6
8 Submit an explanation	7
9 More Learning	7

1 Goal

Build a forward-pass trace you can check line by line. Predict how an intervention changes the output, then explain the result using the computation rather than a story about what a neuron “wants.”

Prerequisite: Lesson 1.3, through the microscopic network example. Read its training section before the optional derivative check.

Time: about 45–75 minutes, including the written interpretation. The optional derivative check adds about 15 minutes.

Requirements: paper or a text editor. The coding route uses an existing Python 3 installation, ordinary CPU execution, and the standard library only. It requires no API, account, internet access, GPU, model download, or paid compute. A complete hand-worked submission is a valid alternative if Python is unavailable.

Expected artifact: a short `submission.md` containing predictions, arithmetic, observations, and interpretation. For the coding route, also keep `tiny_network.py` and a copy of its actual output. Do not substitute the answer key for observations from a run.

! Calculate before coding

Finish Part 1 and record the predictions in Part 2 before executing a program or asking an LLM to solve them. If you use an assistant, ask it to check your reasoning after your first attempt. The prediction is part of the evidence of learning.

2 The fixed network

Use exactly these values:

$$W = \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \\ -1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 1 \\ 0.5 \end{bmatrix}, \quad v = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad c = 0.5.$$

For every input column x , calculate

$$z = Wx + b, \quad h = \text{ReLU}(z), \quad \hat{y} = v^T h + c.$$

These parameters were chosen for transparent arithmetic. The network has not been fitted to a dataset. A successful Lab verifies computation and controlled changes, not predictive performance on an application.

3 Part 1 — Make a hand trace

Use the fresh input $x = [1, 2]^T$.

1. Write the shape of each object: x , W , b , z , h , v^T , and $\hat{y}$.
2. Expand each of the three weighted sums, including its bias.
3. Apply ReLU to obtain h .

4. Write each hidden unit's separate contribution $v_j h_j$ to the output. Add c once.
5. Count the weights and biases. Explain why the three computed values in h do not add three learned parameters.
6. Describe how the arithmetic would differ if ReLU were removed. Predict whether that changes the output for this input.

Keep your original attempt if you correct it later. A clear correction is useful evidence: name the particular sign, bias, or shape mistake rather than merely replacing the number.

4 Part 2 — Predict interventions

Return to the lesson's baseline input $x = [2, -1]^T$. Use the original parameters at the start of every case. Each case changes only what is named; do not accidentally accumulate edits.

Record a numerical prediction and one sentence of reasoning for each case:

1. **Ablation:** force $h_2 = 0$ after ReLU and before the output calculation.
2. **Weight change:** set $v_3 = 101$ while leaving every other parameter unchanged.
3. **Different input:** keep that same $v_3 = 101$ change, but now use $x = [0, 1]^T$. Compare with the original network on that input.
4. **Remove ReLU:** use $h = z$ for the baseline input, with all original parameters restored.

For the ablation, keep both the unmodified activation vector and the vector actually sent to the readout. This distinguishes observing an activation from changing it.

Which case do you expect to disprove the statement “if changing this weight has no effect once, this weight is useless”? Explain your prediction before checking the answer.

5 Part 3 — Implement or independently recalculate

5.1 Coding route

Implement the equations directly with lists, loops, multiplication, addition, and `max`. Avoid a machine-learning library for the first pass. The small scale makes it practical to see every operation.

The following is a **learner scaffold**, not a complete reference implementation. Its unfinished functions deliberately raise errors. Complete them before running the checks; the code shown here is not presented as an executed or verified solution.

```
from copy import deepcopy

PARAMS = {
    "W": [[1.0, -1.0], [0.5, 1.0], [-1.0, 2.0]],
    "b": [0.0, 1.0, 0.5],
    "v": [2.0, -1.0, 1.0],
    "c": 0.5,
}

def dot(a, b):
```

```

# Reject mismatched lengths before computing the sum.
raise NotImplementedError("Implement a checked dot product")

def forward(x, params, ablate_index=None, use_relu=True):
    # Check: len(x) == 2; W has 3 rows of length 2;
    # b and v each have length 3.
    # 1. Compute and retain z.
    # 2. Compute and retain the natural activation vector h.
    # 3. Copy h to readout_h before any intervention.
    # 4. If requested, set readout_h[ablate_index] to zero.
    # 5. Compute the scalar output from readout_h.
    # Return a dict containing z, h, readout_h, and output.
    raise NotImplementedError("Implement the forward trace")

# Python uses zero-based indices: hidden unit 2 has index 1.
# Use deepcopy(PARAMS) for each independent parameter change.
# Do not mutate PARAMS or an earlier run's activation vectors.

```

For dot, do not rely on an unchecked zip: it can silently truncate mismatched inputs. A deliberate shape error should raise a clear `ValueError`. Reject an invalid ablation index too.

Once the functions are complete:

1. Run both the fresh hand-worked input and the lesson baseline. Print every returned field.
2. Compare intermediate values before comparing the final output. An incorrect implementation can occasionally produce the right output through cancelling errors.
3. Run the four intervention cases with independent parameter copies.
4. Test $x = [1, 2, 3]$ and confirm that the shape check rejects it.
5. Run the baseline again and check that earlier interventions did not change it.

You can run your completed file with:

```
python3 tiny_network.py
```

Record the Python version and the actual command. Save the observed trace, including any failing checks you used to diagnose a problem. No benchmark or training claim is expected.

5.2 Paper route

Recalculate the same cases using a second arrangement of the arithmetic: compute Wx first, add b , apply ReLU, and only then form the readout. Compare this with your earlier unit-by-unit expansion. State that your results are hand-checked rather than program outputs.

5.3 Interpret the comparison

For each case, write the prediction, checked result, and explanation. If they differ, identify the earliest intermediate value that diverged. Answer these questions:

- Why can removing a positive activation increase the final output?
- Why does a weight change matter on one input and disappear on another?
- Did the ablation retrain the model?
- What would you need to know before assigning a real-world meaning to these hidden units?

6 Part 4 — Optional one-parameter training step

This part connects inference to training without implementing a training framework. Start again from the original parameters and input $[2, -1]^T$. Let the target be $t = 4$ and define $L = \frac{1}{2}(\hat{y} - t)^2$.

1. Calculate the original loss and the derivative $\partial L / \partial v_1$ by hand.
2. Predict the effect of changing only v_1 with learning rate 0.1. Freeze all other parameters.
3. Recalculate the output and loss with the new v_1 .
4. Explain why this improvement on one example does not establish generalization.

For an optional numerical derivative check, choose $\epsilon = 10^{-4}$. Evaluate the loss twice with $v_1 + \epsilon$ and $v_1 - \epsilon$, resetting the other parameters each time, and compute

$$g_{\text{finite difference}} = \frac{L(v_1 + \epsilon) - L(v_1 - \epsilon)}{2\epsilon}.$$

Compare that estimate with the hand derivative. Ordinary floating-point arithmetic can introduce small differences; report what you observe rather than promising bit-for-bit equality. This finite-difference check concerns one scalar parameter. It is not an efficient substitute for backpropagation in a large network.

Do not describe merely computing this derivative as training. The parameter update is a separate operation. PyTorch's optimization tutorial illustrates this separation in a framework.

7 Checkpoints and solutions

The values below are **analytically derived expectations**, not a record of a software run. Compare them only after making your own attempt.

i Hand trace and interventions

For the fresh input $[1, 2]^T$:

- $z = [-1, 3.5, 3.5]^T$.
- $h = [0, 3.5, 3.5]^T$.
- The three contributions are 0, -3.5 , and 3.5 .
- The output is 0.5 after adding the output bias.
- Without ReLU, the first contribution becomes -2 , so the output is -1.5 .

Shapes are $x : 2 \times 1$, $W : 3 \times 2$, $b, z, h : 3 \times 1$, $v^T : 1 \times 3$, and $\hat{y}$: scalar. The parameter count is $6 + 3 + 3 + 1 = 13$. Activations are calculated from the input and parameters, not independently stored coefficients to be learned in this model.

For the baseline input $[2, -1]^T$, $z = [3, 1, -3.5]^T$, $h = [3, 1, 0]^T$, and the output is 5.5.

1. Ablating hidden unit 2 yields readout activations $[3, 0, 0]^T$ and output 6.5. The removed contribution was negative.
2. Changing v_3 to 101 leaves the baseline output at 5.5, because the third activation is zero.
3. On $[0, 1]^T$, $z = [-1, 2, 2.5]^T$ and $h = [0, 2, 2.5]^T$. The original output is 1; the changed output is 251. The same weight now multiplies a nonzero activation.
4. Removing ReLU at the baseline yields output 2. The negative third preactivation now contributes -3.5 .

Ablation changes an intermediate value for a controlled forward computation. It does not, by itself, fit parameters to data. The parameters and axes have no assigned application meaning here, so these effects do not justify labels such as a “truth neuron” or “planning neuron.”

i Optional derivative check

The original error is 1.5 and loss is 1.125. Since $h_1 = 3$, the derivative is $(\hat{y} - t)h_1 = 4.5$.

Updating only v_1 gives $2 - 0.1(4.5) = 1.55$. The new output is 4.15 and loss is 0.01125.

In exact arithmetic, the centered finite difference equals 4.5 here because the loss is quadratic in v_1 . An implemented floating-point estimate should be close; a tolerance such as 10^{-6} is a reasonable check for this specific calculation using ordinary Python floats. Its observed behavior still needs to be checked in the completed program.

8 Submit an explanation

Your `submission.md` should include:

- **Prediction:** your pre-run answers and reasoning;
- **What I did:** hand calculations or implementation details, with version and command if applicable;
- **Results:** checked intermediate values and the four intervention outcomes;
- **What surprised me:** or an explanation of why the results matched expectations;
- **Interpretation:** the causal role of the changed quantity in these examples;
- **Limits:** what these checks do not establish about training or real-world performance.

A strong submission makes it possible for someone else to reconstruct the arithmetic. It also distinguishes a parameter change from an activation intervention and avoids generalizing from a single zero effect.

9 More Learning

- Lesson 1.3 supplies the full mathematical and conceptual explanation.
- PyTorch Linear documents the corresponding affine layer if you later reproduce your checked implementation with a library.

- PyTorch Automatic Differentiation connects the optional hand derivative to computational graphs. Framework use is optional and outside the core Lab.