Lecture 2: PyTorch tensors and autograd

CSC 352 / CSC 552 · 2026-09-01

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.

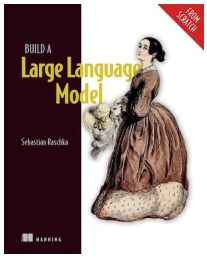
1. Last class we split penguins **before** fitting. Why is a 99% number from fitting on every row not the number you quote?
2. What is the **test** set for: reporting how well you did, or choosing the model?
3. Sundar wants a species name. Mark wants weight in grams. Which task is classification, and which is regression?

Appendix A.1–A.4 (Raschka)

Official companion: code-part1. Today we stop after **automatic differentiation**. Next class starts at **A.5 Implementing multilayer neural networks**.

Supplementary code for the Build a Large Language Model From Scratch book by Sebastian Raschka

Code repository: https://github.com/rasbt/LLMs-from-scratch



Appendix A: Introduction to PyTorch (Part 1)

A.1 What is PyTorch

PyTorch is a Python library for boxes of numbers (tensors) and for turning knobs using gradients. The `import torch` cell checks the version. `cuda` is the GPU; `False` means this machine is using the CPU, which is fine for today.

In [8]:
```
import torch
```

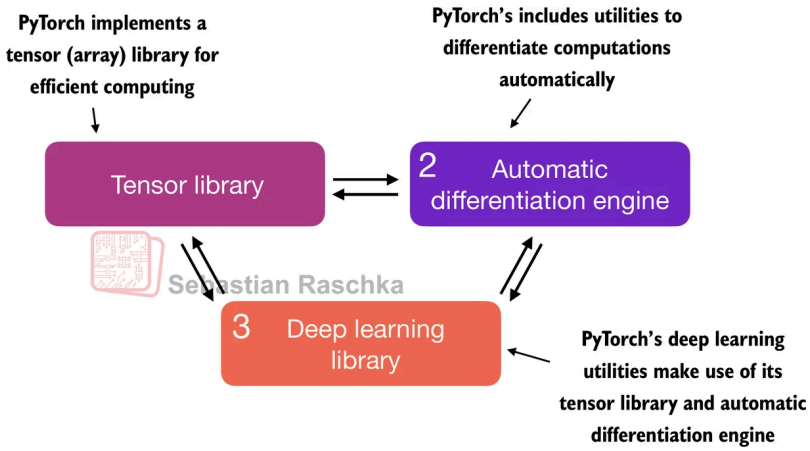
```
print(torch.__version__)
```

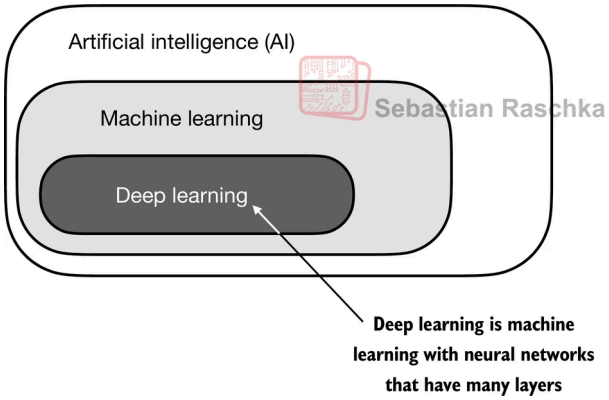
```
2.8.0
```

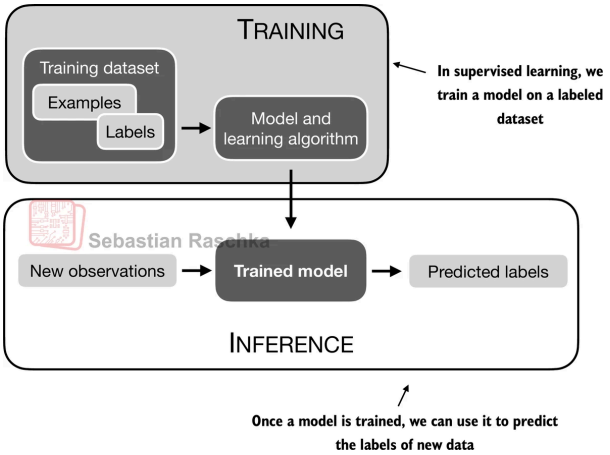
The next cell continues the same idea — read the new names before you run it.

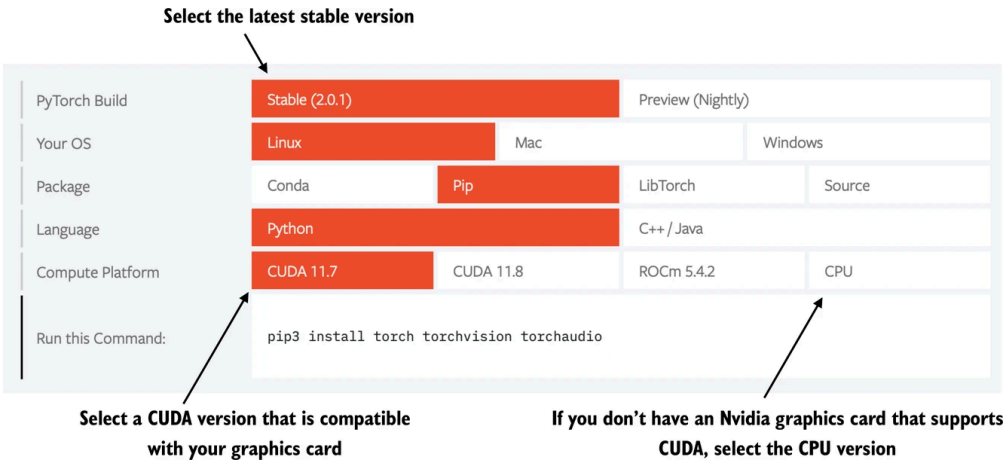
In [10]:
```
print(torch.cuda.is_available())
```

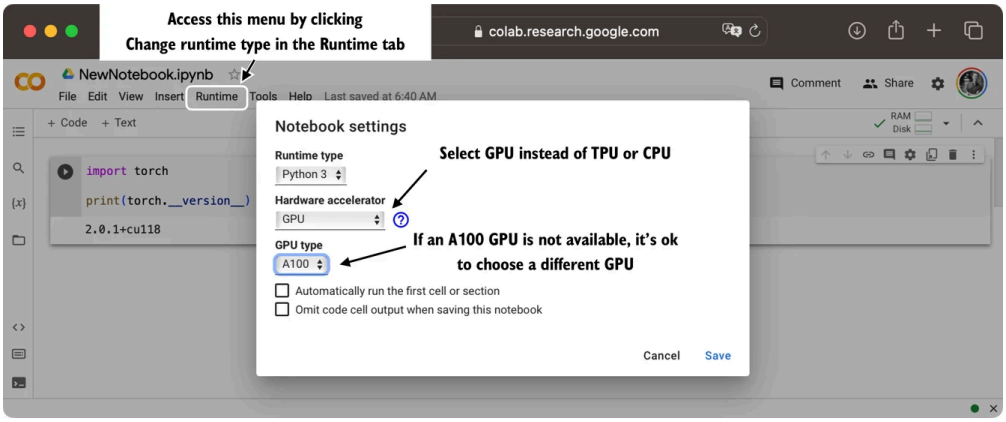
```
False
```





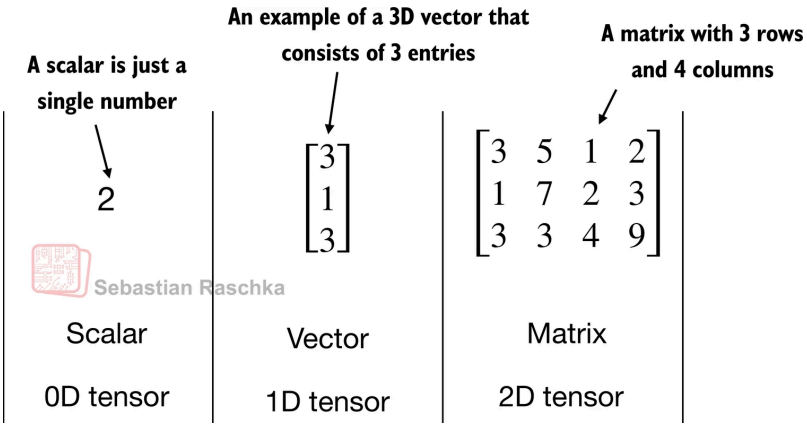






A.2 Understanding tensors

A **tensor** is PyTorch's box of numbers: one number, a list, a table, or a stack of tables. Same idea as a NumPy array. `shape` is how the box is laid out (rows × columns, ...). `dtype` is the kind of number (integer vs float). We will use these operations when we build the net.



A.2.1 Scalars, vectors, matrices, and tensors

In [17]:
```
import torch
import numpy as np
```

```
# create a 0D tensor (scalar) from a Python integer
tensor0d = torch.tensor(1)

# create a 1D tensor (vector) from a Python list
tensor1d = torch.tensor([1, 2, 3])

# create a 2D tensor from a nested Python list
tensor2d = torch.tensor([[1, 2],
                         [3, 4]])

# create a 3D tensor from a nested Python list
tensor3d_1 = torch.tensor([[[1, 2], [3, 4]],
                           [[5, 6], [7, 8]]])

# create a 3D tensor from NumPy array
ary3d = np.array([[[1, 2], [3, 4]],
                  [[5, 6], [7, 8]]])
tensor3d_2 = torch.tensor(ary3d)  # Copies NumPy array
tensor3d_3 = torch.from_numpy(ary3d)  # Shares memory with NumPy array
```

The next cell continues the same idea — read the new names before you run it.

In [19]:
```
ary3d[0, 0, 0] = 999
print(tensor3d_2) # remains unchanged
```

```
tensor([[[1, 2],
         [3, 4]],

        [[5, 6],
         [7, 8]]])
```

The next cell continues the same idea — read the new names before you run it.

In [21]:
```
print(tensor3d_3) # changes because of memory sharing
```

```
tensor([[[999,   2],
         [  3,   4]],

        [[  5,   6],
         [  7,   8]]])
```

A.2.2 Tensor data types

In [23]:
```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

```
torch.int64
```

The next cell continues the same idea — read the new names before you run it.

In [25]:
```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

```
torch.float32
```

The next cell continues the same idea — read the new names before you run it.

In [27]:
```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

```
torch.float32
```

A.2.3 Common PyTorch tensor operations

This 2 × 3 table is the running example: 2 rows, 3 columns.

In [30]:
```
tensor2d = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])  # 2 rows, 3 columns
tensor2d
```

Out[30]:
```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

`shape` is (rows, columns). Here that is `(2, 3)`.

In [32]:
```
tensor2d.shape  # torch.Size([2, 3]) means 2 rows × 3 columns
```

Out[32]:
```
torch.Size([2, 3])
```

`reshape` and `view` put the **same numbers** into a new layout. They do not change the values. `view` needs the numbers to sit in one contiguous block of memory; `reshape` will copy if it has to. For this table they match.

In [34]:
```
tensor2d.reshape(3, 2)  # same six numbers, now 3 rows × 2 columns
```

Out[34]:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

`view` does the same rearrangement for this table:

In [36]:
```
tensor2d.view(3, 2)  # same result as reshape here
```

Out[36]:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

`.T` **transposes** the table: rows become columns. `(2, 3)` → `(3, 2)`. We need this before matrix multiply, because the inner sizes have to match.

In [38]:
```
tensor2d.T  # columns of tensor2d become rows
```

Out[38]:
```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

Matrix multiply is not “multiply matching entries.”

- Python `*` on tensors is **element-wise**: each pair of numbers in the **same position** is multiplied. Both sides need the **same shape**.
- `.matmul(...)` and `@` (the operator is read **"at"**) are **matrix multiplication**. Each output number is a **dot product**: take one **row of the left** table, one **column of the right** table, multiply those pairs, then add.

Shape rule: `(m × n) @ (n × p)` → `(m × p)`. The inner `n` must match.

`tensor2d` is 2 × 3, so `tensor2d @ tensor2d` is illegal (3 ≠ 2). `tensor2d.T` is 3 × 2, so `tensor2d @ tensor2d.T` is 2 × 3 times 3 × 2 → **2 × 2**.

Top-left entry of that 2 × 2: first row of `tensor2d` is `[1, 2, 3]`, first column of `tensor2d.T` is also `[1, 2, 3]`:

`1·1 + 2·2 + 3·3 = 14`.

In [40]:
```
# matrix multiply: (2×3) @ (3×2) → (2×2)
# top-left 14 = 1*1 + 2*2 + 3*3   (row 0 · row 0, because .T turns row 0 int
# top-right 32 = 1*4 + 2*5 + 3*6
# bottom-left 32 = 4*1 + 5*2 + 6*3
# bottom-right 77 = 4*4 + 5*5 + 6*6
tensor2d.matmul(tensor2d.T)
```

Out[40]:
```
tensor([[14, 32],
        [32, 77]])
```

`@` is the same operation as `.matmul`. People write `@` because it is shorter. Both are matrix multiply. Neither is `*`.

In [42]:
```
tensor2d @ tensor2d.T  # identical to tensor2d.matmul(tensor2d.T)
```

Out[42]:
```
tensor([[14, 32],
        [32, 77]])
```

`*` needs the same shape, so here both sides stay 2 × 3. Each matching pair is multiplied. That is **not** the 2 × 2 matrix product above.

In [44]:
```
tensor2d * tensor2d  # element-wise: [[1, 4, 9], [16, 25, 36]]
```

Out[44]:
```
tensor([[ 1,  4,  9],
        [16, 25, 36]])
```

On two **vectors** of the same length, `@` is exactly the **dot product** (one number). That is the same arithmetic as **one entry** of a matrix multiply. `torch.dot` is the same thing for 1-D tensors.

In [46]:
```
u = torch.tensor([1, 2, 3])
v = torch.tensor([1, 2, 3])
print('u @ v (dot product)', u @ v)          # 1*1 + 2*2 + 3*3 = 14
```

```
print('torch.dot(u, v)     ', torch.dot(u, v))  # same 14
print('u * v (element-wise)', u * v)          # [1, 4, 9], still a vector
```

```
u @ v (dot product) tensor(14)
torch.dot(u, v)      tensor(14)
u * v (element-wise) tensor([1, 4, 9])
```

Exercise (matrix multiply)

`t` is the 2×3 tensor in the next cell. Print its shape, its transpose, and the matrix product of the tensor with its transpose (not element-wise). Check that the top-left entry equals the first row dotted with itself.

In [ ]:
```
# Exercise 3 — your attempt
import torch

t = torch.tensor([[1, 2, 3], [4, 5, 6]])
shape = None
transposed = None
product = None
top_left = None
print(shape)
print(transposed)
print(product)
print('top-left', top_left)
```

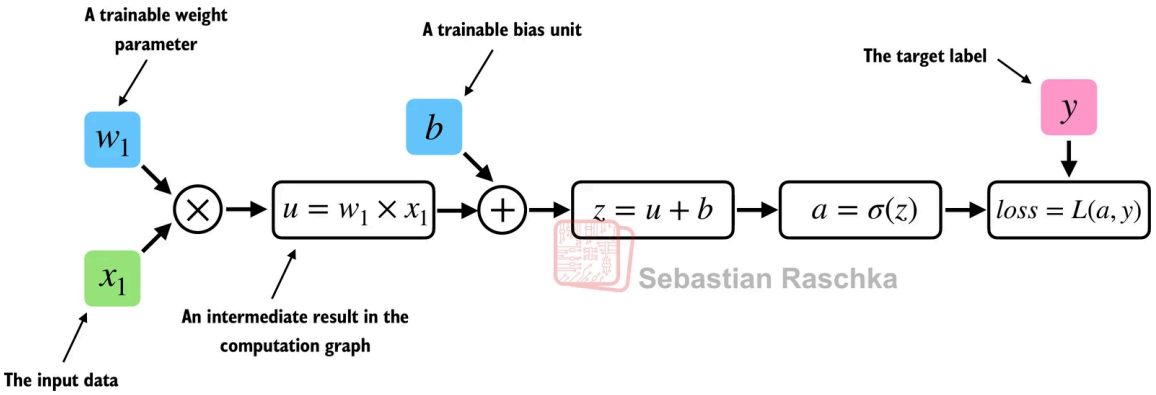
Solution (matrix multiply)

In [50]:
```
# Solution 3 — shape is rows × columns. .T flips them.
# @ (and .matmul) is matrix multiply: each entry is a row·column dot product
import torch

t = torch.tensor([[1, 2, 3], [4, 5, 6]])
print('shape', tuple(t.shape))
print(t.T)
print(t @ t.T)
print('top-left by hand', 1*1 + 2*2 + 3*3)
```

```
shape (2, 3)
tensor([[1, 4],
        [2, 5],
        [3, 6]])
tensor([[14, 32],
        [32, 77]])
top-left by hand 14
```

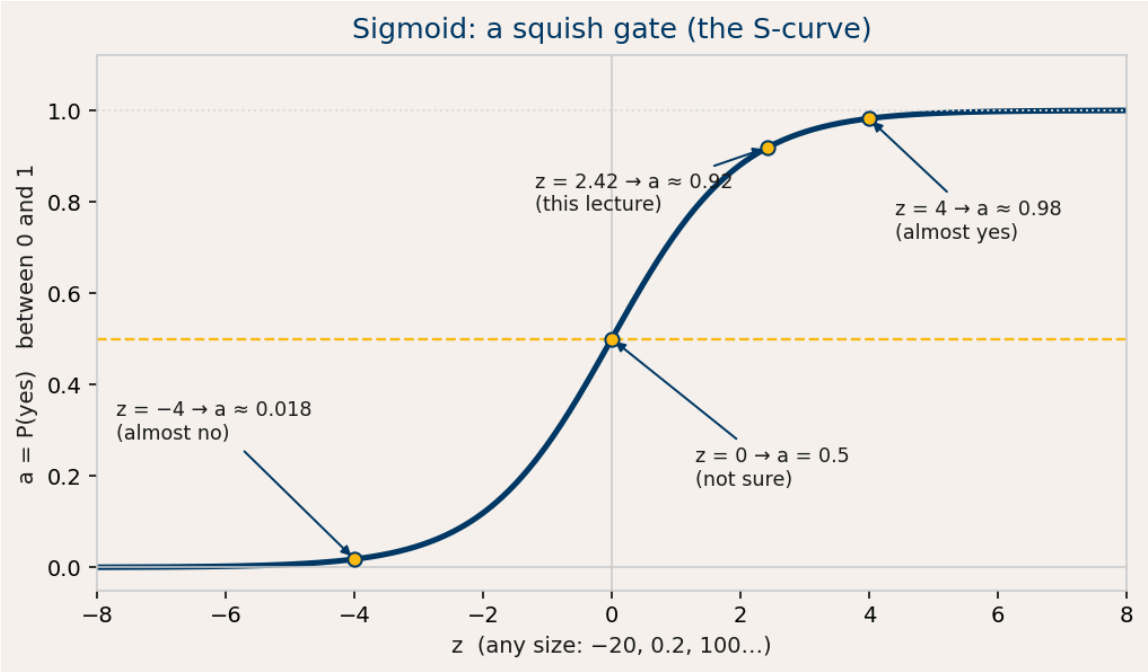
A.3 Seeing models as computation graphs



The picture is **one neuron**: a tiny mixing desk with two knobs.

- The **ingredient** is the input `x1`.
- `w1` is a **volume knob** (the **weight**): how loud that ingredient is.
- `b` is a **nudge** (the **bias**): a constant push, even if `x1` is zero.
- **Mix:** `z = x1 * w1 + b`. `z` can be 100, 0.2, or −20.
- **Squish: sigmoid** forces *any* mix into a number between **0 and 1**. Call it `a`. Read it as **P(yes)**. The curve is an **S**.

Look at that S first, then a few numbers through it.



Sigmoid is the S-shaped squish gate: any z becomes a number between 0 and 1.

Each `z` in, one `a` out. Big negative mix → almost no. Mix near 0 → coin flip. Big positive mix → almost yes. `z = 2.42` is the mix we get from `x1 = 1.1` and `w1 = 2.2` in the next cells.

In [56]:
```
# a few mixes through the squish gate
z = torch.tensor([-20.0, -4.0, 0.0, 2.42, 4.0, 100.0])
```

```python
a = torch.sigmoid(z)
print('   z (mix)     a = P(yes)')
for zi, ai in zip(z, a):
    print(f'{float(zi):10.2f}   {float(ai):.6f}')
```

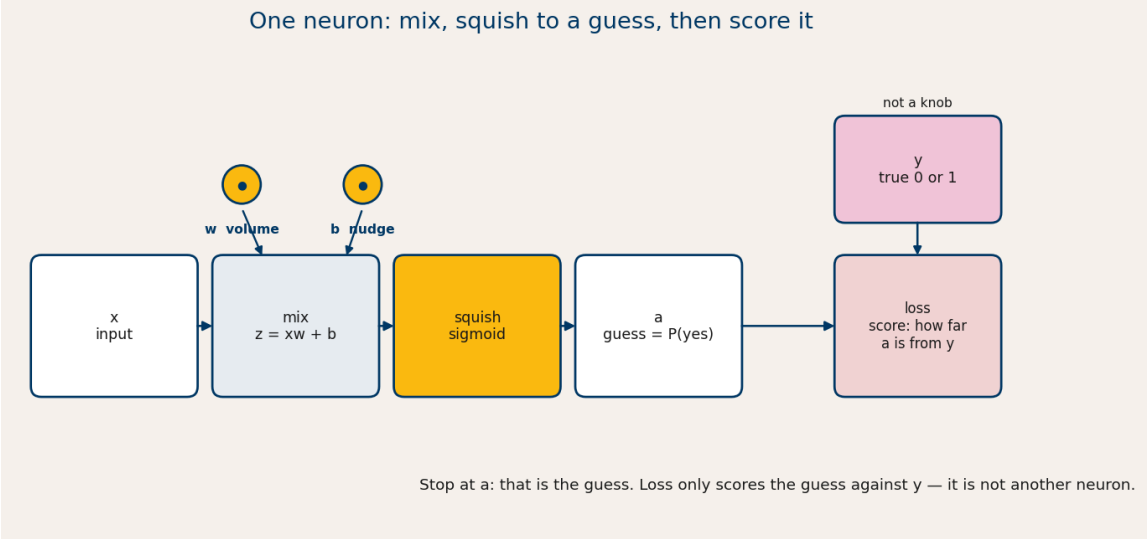
```
   z (mix)     a = P(yes)
    -20.00   0.000000
     -4.00   0.017986
      0.00   0.500000
      2.42   0.918340
      4.00   0.982014
    100.00   1.000000
```

The printed `a` **is the model's guess** — not the true answer.

In Raschka's picture (and the mixing-desk picture next):

- `a` = the guess, `P(yes)`, after sigmoid.
- `y` = the true 0 or 1 (the pink box). It is **not** a knob.
- `loss` = a **score** for that guess: how far `a` is from `y`. Loss is not another neuron and not another mix.

The desk **stops at** `a`. Loss only scores the guess.



The desk outputs a guess a. y is the true 0 or 1. loss scores how far a is from y — it is not another neuron.

Binary cross-entropy is the yes/no version of the next-word **cross-entropy** above. Same question: *how much probability did we put on the true answer?* Then take `-log` of that number (natural log — the same `log` PyTorch uses). Small `-log` means we were not surprised; large `-log` means we almost missed.

- True answer is **yes** (`y = 1`): loss = `-log(a)`. We wanted `a` near 1.
  - `a = 0.9` → `-log(0.9) ≈ 0.105` (small)

    - `a = 0.1` → `-log(0.1) ≈ 2.30` (large — we said "probably no" when it was yes)
- True answer is **no** (`y = 0`): loss = `-log(1 - a)`. We wanted `a` near 0.

One line that covers both cases:

`loss = -(y * log(a) + (1 - y) * log(1 - a))`

`F.binary_cross_entropy(a, y)` computes that. Next-word uses the many-class cousin: `-log` of the **true word's** probability. Smaller is better.

In [60]:
```
import torch.nn.functional as F

y = torch.tensor([1.0])   # true label: 1 = yes
x1 = torch.tensor([1.1])  # input feature
w1 = torch.tensor([2.2])  # weight (a knob)
b = torch.tensor([0.0])   # bias (the other knob)

z = x1 * w1 + b           # net input; can be any real number
a = torch.sigmoid(z)      # squash z into (0, 1) = P(yes)

# y is 1, so this is -log(a). Same idea as next-word: -log(prob on the true
loss = F.binary_cross_entropy(a, y)
print('a = P(yes)', a)
print('loss     ', loss)
print('-log(a)  ', -torch.log(a))  # matches loss because y = 1
```

```
a = P(yes) tensor([0.9183])
loss       tensor(0.0852)
-log(a)    tensor([0.0852])
```

The printed `loss` is already small here because this made-up `w1` pushes sigmoid close to 1 (the true label). A worse guess makes the same formula large:

In [62]:
```
# same formula, two made-up probabilities, true answer still yes (y = 1)
print('good guess a=0.9, y=1  loss', round(float(-torch.log(torch.tensor(0.9
print('bad guess  a=0.1, y=1  loss', round(float(-torch.log(torch.tensor(0.1
# if the true answer had been no (y = 0), we would use -log(1 - a) instead
```

```
good guess a=0.9, y=1  loss 0.105
bad guess  a=0.1, y=1  loss 2.303
```

Exercise (binary cross-entropy)

The true answer is yes. Two guesses for P(yes) are 0.8 and 0.2. Print the binary cross-entropy loss of each guess. Which guess is worse, and by roughly how much?

In [ ]:
```
# Exercise (binary cross-entropy) — your attempt
import torch

a_good = torch.tensor(0.8)
a_bad = torch.tensor(0.2)
loss_good = None
```

```
loss_bad = None
print('a=0.8 loss', loss_good)
print('a=0.2 loss', loss_bad)
```

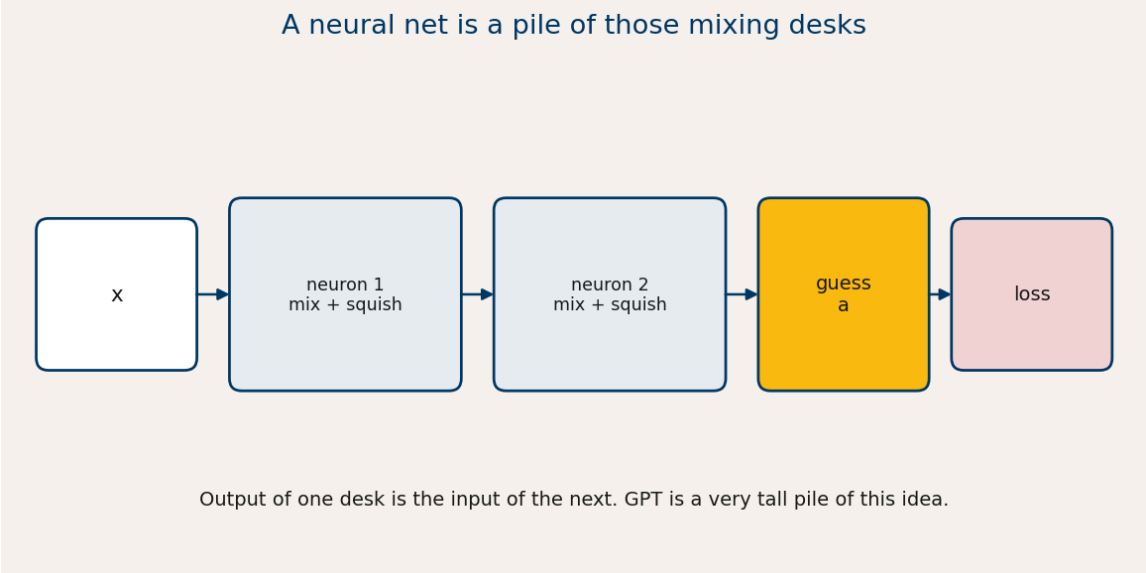
Solution (binary cross-entropy)

In [66]:
```
# Solution — y = 1, so loss is -log(a). Smaller a on the true answer → large
import torch

a_good = torch.tensor(0.8)
a_bad = torch.tensor(0.2)
loss_good = float(-torch.log(a_good))
loss_bad = float(-torch.log(a_bad))
print('a=0.8 loss', round(loss_good, 3))
print('a=0.2 loss', round(loss_bad, 3))
print('worse is a=0.2; about', round(loss_bad / loss_good, 1), 'times as lar
```

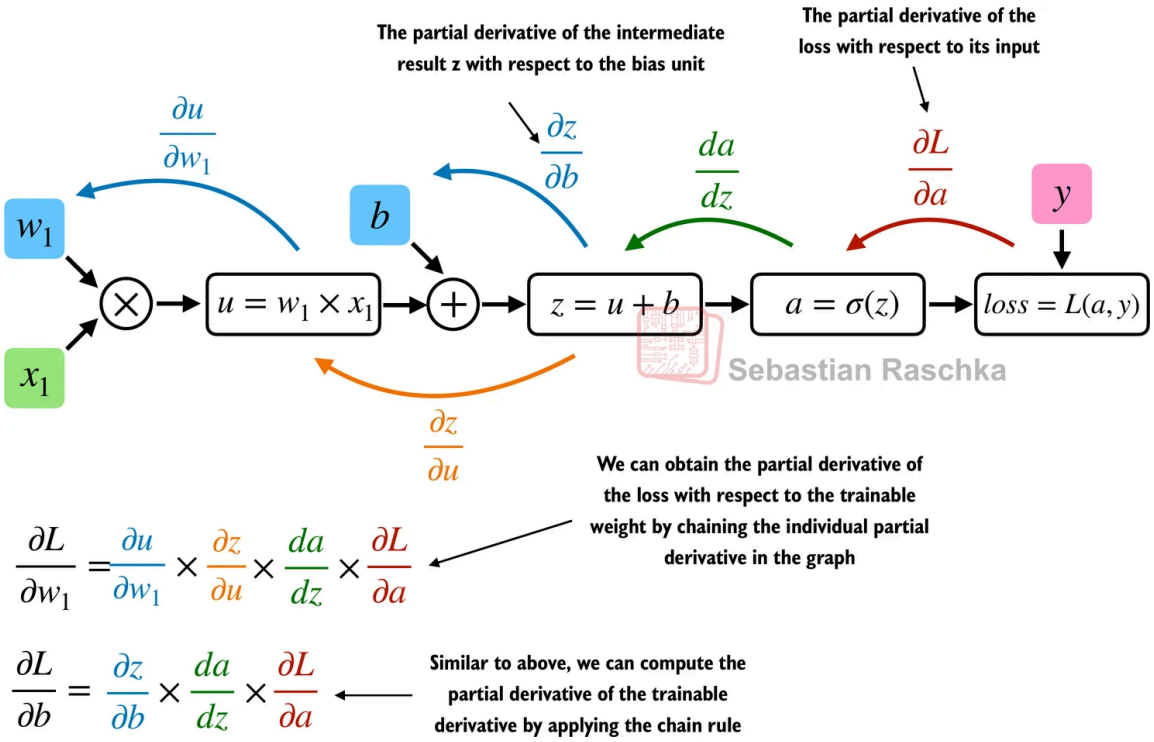
```
a=0.8 loss 0.223
a=0.2 loss 1.609
worse is a=0.2; about 7.2 times as large
```

A **neural net** is a **pile** of these mixing desks. The output of one is the input of the next. GPT is a very tall pile of the same idea. We just ran **one** desk. Stacking them as `nn.Module` is Appendix A.5.



A neural net is a pile of mixing desks. Output of one is the input of the next.

A.4 Automatic differentiation made easy



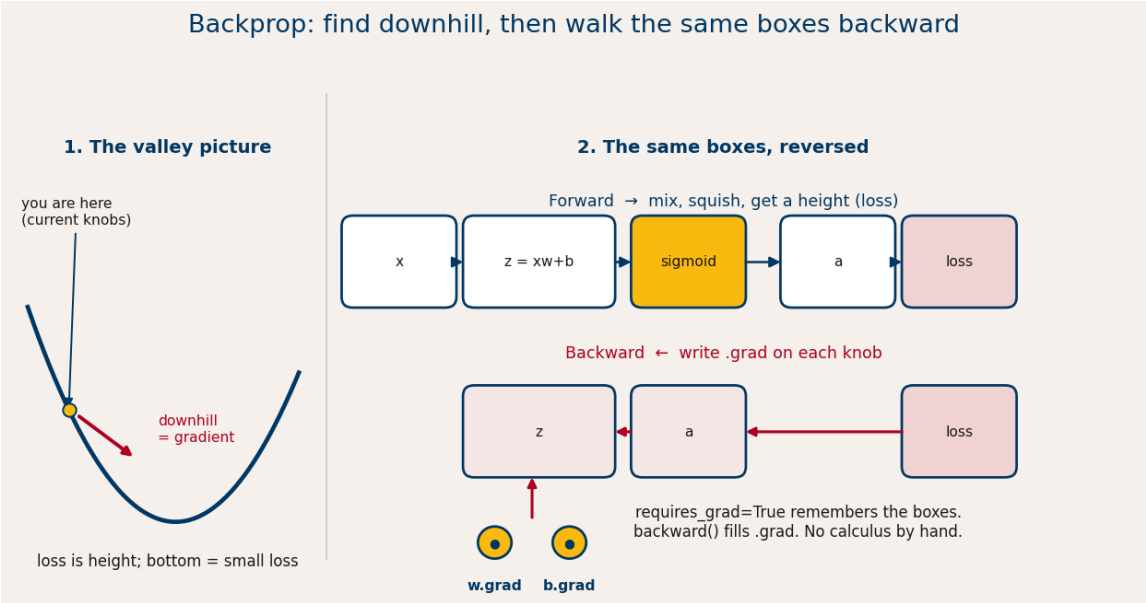
Two pictures for the same move.

The valley. Loss is **height**. We stand on a slope. The **gradient** is the arrow "downhill is that way." Later we take a **small step** along that arrow. We do not jump across the valley.

The chain of boxes. Forward is left to right: mix, squish, then a height (`loss`). `requires_grad=True` means "remember this box." **Backpropagation** — `loss.backward()` — walks **the same boxes in reverse** (loss → a → z → knobs) and writes `.grad` on each knob.

- Negative `.grad` : turning the knob **up** would make the height **smaller** (locally).
- Positive `.grad` : turning the knob **up** would make the height **larger**.

PyTorch walks the chain. We do not differentiate by hand.



Left: loss is height; the red arrow is downhill. Right: forward left to right, backward writes .grad on each knob.

In [73]:
```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)  # height of the valley (still -log(a) b

grad_L_w1 = grad(loss, w1, retain_graph=True)  # which way is downhill for w
grad_L_b = grad(loss, b, retain_graph=True)    # which way is downhill for b

print(grad_L_w1)
print(grad_L_b)
```

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

`loss.backward()` fills `.grad` on every tensor that had `requires_grad=True`. Same slopes as `grad(...)` above, stored on the knobs themselves.

In [75]:
```
loss.backward()  # walk the same boxes backward; fills .grad on every knob

print(w1.grad)  # same number as grad(...) above; negative → turning w1 up l
print(b.grad)
```

```
tensor([-0.0898])
tensor([-0.0817])
```

Exercise (backward)

Look at `w1.grad` from the cell above. If we increase `w1` a little, does the loss go up or down? Print one word.

In [ ]:
```
# Exercise (backward) — your attempt
# w1.grad is already filled from backward.
slope = None
direction = None
print(direction)
```

Solution (backward)

In [79]:
```
# Solution — a negative slope means: locally, bigger w1 → smaller loss.
# Downhill on the valley is toward increasing w1.
print('down')
print('negative .grad: increasing the knob lowers loss (here)')
```

```
down
negative .grad: increasing the knob lowers loss (here)
```

Comprehensive exercises

Work these if we finish early. They use more than one idea from today.

Takeaways

1. A **tensor** is PyTorch's box of numbers. `*` is element-wise. `.matmul` and `@` are **matrix multiply**.
2. **Cross-entropy** is `-log` of the probability on the true answer. **Backprop** walks the same boxes backward; `.grad` says which way is downhill for that knob.
3. Next class: Appendix **A.5–A.9** (`nn.Module`, DataLoader, the training loop). Save/load and GPU if we have time.