

Perceptrons (Augmented with Gradient Descent)

Aayush Bajaj

July 10, 2026

Contents

1	rosenblatt's unit	1
2	the perceptron learning rule	2
3	novikoff's convergence theorem	2
4	the xor limitation	3
5	the smooth perceptron: sigmoid + gradient descent	4
5.1	log-loss and its gradient	4
5.2	the boundary is still linear	5
6	code: both perceptrons from scratch	5
6.1	the classic rule, with the mistake bound checked	5
6.2	the sigmoid version, trained by gradient descent	6
7	see also	7

the perceptron is the hydrogen atom of neural networks: one neuron, one weight vector, one threshold — and yet it already exhibits the two behaviours that define the whole field. it *learns* from mistakes with a provable convergence guarantee, and it *fails* on problems its geometry cannot express. this page covers both, then swaps the hard threshold for a sigmoid so that gradient descent can take over — a companion to the [sign-loss perceptron](#) page, which treats the classical algorithm on its own.

rosenblatt's 1958 demo ran in software on an ibm 704; the mark i hardware — motorised potentiometers for weights — was assembled in 1959 and demonstrated in 1960

1 rosenblatt's unit

a perceptron takes an input $x \in \mathbb{R}^d$, forms a weighted sum, and thresholds it:

$$\hat{y} = \text{sign}(w^\top x + b) = \begin{cases} +1 & \text{if } w^\top x + b > 0, \\ -1 & \text{otherwise.} \end{cases} \quad (1)$$

- the weights $w \in \mathbb{R}^d$ measure how much each input coordinate matters, and in which direction.
- the bias b shifts the threshold away from the origin. absorbing it as $w \leftarrow (w, b)$, $x \leftarrow (x, 1)$ tidies every formula below, and we do so silently from here on.
- the hard nonlinearity is the point: without it the unit is just a linear functional,

and stacking linear maps yields another linear map.

the lineage runs from mcculloch and pitts' logical calculus of nervous activity (McCulloch, Warren S. and Pitts, Walter, 1943) — fixed weights, pure logic — to rosenblatt's 1958 innovation: make the weights *learnable* from data.¹

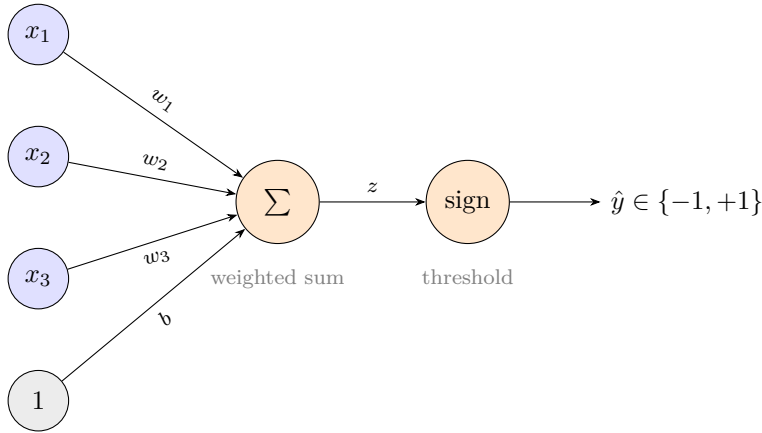


Figure 1: a single perceptron unit: inputs are scaled by weights, summed with the bias, and thresholded.

2 the perceptron learning rule

given labelled data (x_i, y_i) with $y_i \in \{-1, +1\}$, the algorithm is almost embarrassingly simple:

- **input** a training set, weights initialised to $w = 0$.
- **loop** over examples. if the current example is correctly classified — $y_i w^\top x_i > 0$ — do nothing.
- **update** on a mistake:

$$w \leftarrow w + y_i x_i. \quad (2)$$

- **stop** after a full pass with no mistakes.

why this update? after it, the score on the offending example changes by

$$y_i (w + y_i x_i)^\top x_i = y_i w^\top x_i + \|x_i\|^2, \quad (3)$$

which is strictly larger — every update nudges the score of the mistaken point in the correct direction. in modern language: sgd on $\ell(w) = \max(0, -y w^\top x)$, the “perceptron criterion” (Goodfellow, Ian, 2016).

the rule is exactly stochastic gradient descent with step size 1 on the perceptron criterion, the hinge-at-zero loss

3 novikoff's convergence theorem

the astonishing part is that this local, greedy rule terminates — and the bound does not depend on the number of data points.

¹f. rosenblatt (1958), *the perceptron: a probabilistic model for information storage and organization in the brain*, psychological review 65(6).

theorem (novikoff, 1962).² suppose $\|x_i\| \leq R$ for all i , and there exist a unit vector u and a margin $\gamma > 0$ such that

$$y_i u^\top x_i \geq \gamma \quad \text{for all } i. \quad (4)$$

then the perceptron algorithm, started from $w = 0$, makes at most $(R/\gamma)^2$ updates, on any presentation order of the data.

proof. let w_k denote the weight vector after the k -th update, $w_0 = 0$. track two quantities.

alignment grows linearly. if the k -th update is on example (x, y) , then

$$\begin{aligned} u^\top w_k &= u^\top w_{k-1} + y u^\top x \\ &\geq u^\top w_{k-1} + \gamma, \end{aligned}$$

so by induction $u^\top w_k \geq k\gamma$.

norm grows at most like the square root. the update was triggered by a mistake, so $y w_{k-1}^\top x \leq 0$, and

$$\begin{aligned} \|w_k\|^2 &= \|w_{k-1}\|^2 + 2y w_{k-1}^\top x + \|x\|^2 \\ &\leq \|w_{k-1}\|^2 + R^2, \end{aligned}$$

so by induction $\|w_k\|^2 \leq kR^2$.

squeeze. cauchy–schwarz with $\|u\| = 1$ gives

$$k\gamma \leq u^\top w_k \leq \|w_k\| \leq \sqrt{k} R \quad \implies \quad k \leq \left(\frac{R}{\gamma}\right)^2. \quad \blacksquare \quad (5)$$

three remarks:

- the bound is *dimension-free* and *sample-size-free*: only the geometry (how big the data, how wide the corridor between the classes) matters. this observation is the seed of margin theory and, eventually, of support vector machines.
- the mistake cap holds for any ordering, even an adversarial one — it is an online mistake bound, not a statistics statement.
- the theorem says nothing when no separating u exists. in that case the algorithm cycles forever, which brings us to the famous failure.

4 the xor limitation

a perceptron’s decision boundary is the hyperplane $w^\top x + b = 0$: it can only ever carve the input space into two half-spaces. xor asks for more.

²a. novikoff (1962), *on convergence proofs on perceptrons*, proceedings of the symposium on the mathematical theory of automata, vol. 12. the proof given here is the standard modern presentation.

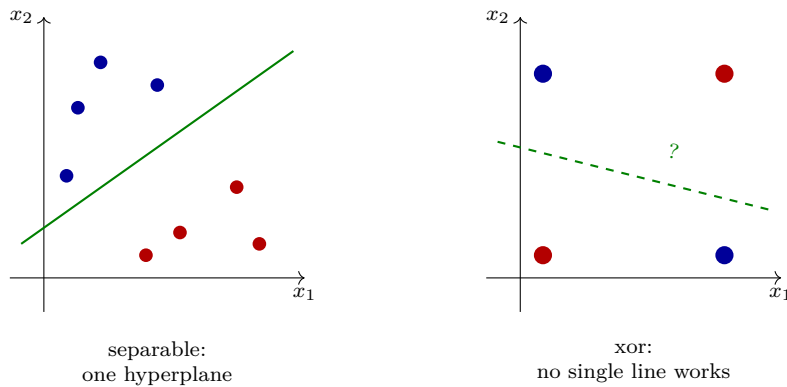


Figure 2: left: a linearly separable problem — one line suffices. right: xor — any line that separates one diagonal pair misclassifies the other.

the two-line argument: for xor we need $f(0,0) = f(1,1) = -1$ and $f(1,0) = f(0,1) = +1$. a linear score would require

$$\begin{aligned} b &\leq 0, & w_1 + w_2 + b &\leq 0, \\ w_1 + b &> 0, & w_2 + b &> 0. \end{aligned}$$

adding the last two: $w_1 + w_2 + 2b > 0$, so $w_1 + w_2 + b > -b \geq 0$ — contradicting the second inequality. no weights exist.

minsky and papert's *perceptrons* (1969) turned this observation into a systematic theory of what single-layer machines cannot compute (parity, connectedness), and is popularly blamed for the first neural-network winter. the fix was known even then: stack units, so that hidden neurons re-represent the input in a space where the classes *are* linearly separable. that is the **multilayer perceptron**, and with it comes the need for a trainable, differentiable unit — the subject of the rest of this page.

the book's actual thesis was subtler — about how coefficients must scale with input size — but funding agencies read the cover, not the lemmas

5 the smooth perceptron: sigmoid + gradient descent

the step function has derivative zero everywhere it is differentiable, so gradient-based training has nothing to hold onto. replace it with the logistic sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z) (1 - \sigma(z)), \quad (6)$$

and reinterpret the output as a probability: $p(y = 1 \mid x) = \sigma(w^\top x)$, with labels now in $\{0, 1\}$.

5.1 log-loss and its gradient

maximum likelihood gives the cross-entropy (log-loss) objective over n examples:

$$L(w) = -\frac{1}{n} \sum_{i=1}^n \left[y_i \log p_i + (1 - y_i) \log(1 - p_i) \right], \quad p_i = \sigma(w^\top x_i). \quad (7)$$

the gradient collapses beautifully. for one example, chain rule through $p = \sigma(z)$, $z = w^\top x$:

$$\frac{\partial L}{\partial z} = - \left[\frac{y}{p} - \frac{1-y}{1-p} \right] \sigma'(z) = - \frac{y-p}{p(1-p)} \cdot p(1-p) = p-y,$$

$$\nabla_w L = (p-y) x.$$

the sigmoid’s derivative exactly cancels the loss’s denominators — this pairing is why log-loss (and not squared error) is the canonical partner of the sigmoid (Goodfellow, Ian, 2016). gradient descent is then

with squared error the gradient keeps a sigma-prime factor that vanishes when the unit saturates confidently wrong — learning stalls precisely when it is most needed

$$w \leftarrow w - \eta \cdot \frac{1}{n} \sum_{i=1}^n (p_i - y_i) x_i, \quad (8)$$

structurally the same “error times input” rule as rosenblatt’s — but with a *graded* error $p - y \in (-1, 1)$ instead of an all-or-nothing one. this smooth unit trained on log-loss is exactly logistic regression; nielsen’s book develops the same unit as the building block of deep networks ([nielsen, neural networks and deep learning](#)).

5.2 the boundary is still linear

smoothing buys trainability, not expressiveness. the decision rule $p > \frac{1}{2}$ is

$$\sigma(w^\top x) > \frac{1}{2} \iff w^\top x > 0, \quad (9)$$

the same hyperplane as before. a single sigmoid unit still cannot solve xor; only *depth* changes the representational class. what changes is everything else: the loss is differentiable and convex in w , and the same machinery scales to stacked layers via backpropagation.

6 code: both perceptrons from scratch

6.1 the classic rule, with the mistake bound checked

```

1  import numpy as np
2
3  rng = np.random.default_rng(42)
4
5  # --- linearly separable data with a known margin ---
6  n = 100
7  X = rng.uniform(-1, 1, size=(n, 2))
8  w_true = np.array([2.0, -1.5])
9  b_true = 0.3
10 y = np.sign(X @ w_true + b_true)
11 # enforce a margin: discard points too close to the boundary
12 keep = np.abs(X @ w_true + b_true) / np.linalg.norm(w_true) > 0.1
13 X, y = X[keep], y[keep]
14
15 # --- classic perceptron learning rule ---
16 def perceptron(X, y, max_epochs=100):
17     Xa = np.hstack([X, np.ones((len(X), 1))]) # absorb bias
18     w = np.zeros(Xa.shape[1])
19     mistakes = 0
20     for epoch in range(max_epochs):
21         errors = 0
22         for xi, yi in zip(Xa, y):

```

```

23         if yi * (w @ xi) <= 0:                # misclassified (or on
           ↳ boundary)
24             w += yi * xi                    # the update
25             mistakes += 1
26             errors += 1
27         if errors == 0:
28             return w, mistakes, epoch + 1
29     return w, mistakes, max_epochs
30
31 w, mistakes, epochs = perceptron(X, y)
32 print(f"n = {len(X)} points, converged after {epochs} epochs, {mistakes} total
   ↳ updates")
33 print(f"learned w = {np.round(w, 3)}")
34
35 # novikoff bound check:  $R^2 / \gamma^2$  with the true separator as witness u
36 Xa = np.hstack([X, np.ones((len(X), 1))])
37 u = np.append(w_true, b_true)
38 u /= np.linalg.norm(u)
39 R = np.linalg.norm(Xa, axis=1).max()
40 gamma = (y * (Xa @ u)).min()
41 print(f"R = {R:.3f}, gamma = {gamma:.3f}, bound (R/gamma)^2 =
   ↳ {(R/gamma)**2:.1f} >= {mistakes} mistakes")

```

n = 92 points, converged after 7 epochs, 23 total updates

learned w = [4.11 -2.659 1.]

R = 1.607, gamma = 0.109, bound (R/gamma)² = 217.4 >= 23 mistakes

23 actual updates against a certified cap of 217 — the bound is loose (it always is; it must hold for adversarial orderings) but finite, and that is the theorem's whole content. note the margin γ here is measured in the *augmented* space where the bias is a coordinate, matching the theorem's assumptions.

6.2 the sigmoid version, trained by gradient descent

```

1  import numpy as np
2
3  rng = np.random.default_rng(42)
4
5  # same separable data as before
6  n = 100
7  X = rng.uniform(-1, 1, size=(n, 2))
8  w_true = np.array([2.0, -1.5])
9  b_true = 0.3
10 y01 = (X @ w_true + b_true > 0).astype(float) # labels in {0, 1} now
11
12 sigmoid = lambda z: 1.0 / (1.0 + np.exp(-z))
13
14 def train_sigmoid_gd(X, y, lr=0.5, steps=2000):
15     Xa = np.hstack([X, np.ones((len(X), 1))])
16     w = np.zeros(Xa.shape[1])
17     for t in range(steps + 1):
18         p = sigmoid(Xa @ w)
19         if t % 400 == 0:
20             eps = 1e-12
21             loss = -np.mean(y * np.log(p + eps) + (1 - y) * np.log(1 - p +
   ↳ eps))
22             acc = np.mean((p > 0.5) == y)
23             print(f"step {t:>4} log-loss {loss:.4f} accuracy {acc:.3f}")
24             grad = Xa.T @ (p - y) / len(y) # the whole derivation, one
   ↳ line
25             w -= lr * grad
26     return w
27

```

```

28 w = train_sigmoid_gd(X, y01)
29 print(f"learned w = {np.round(w, 3)} (true direction ~ [2, -1.5, 0.3])")
30 print(f"normalised: {np.round(w / np.linalg.norm(w[:2]) *
    ↪ np.linalg.norm(w_true), 3)}")

```

```

step    0  log-loss 0.6931  accuracy 0.440
step  400  log-loss 0.1085  accuracy 0.990
step  800  log-loss 0.0805  accuracy 0.990
step 1200  log-loss 0.0679  accuracy 0.990
step 1600  log-loss 0.0602  accuracy 1.000
step 2000  log-loss 0.0549  accuracy 1.000
learned w = [11.117 -8.616  1.382] (true direction ~ [2, -1.5, 0.3])
normalised: [ 1.976 -1.531  0.246]

```

two things worth noticing in the trace:

- the loss starts at $\log 2 \approx 0.6931$ — the entropy of a coin flip, as it must with $w = 0$.
- the raw weights keep growing (on separable data the log-loss optimum is at infinity — the sigmoid wants to become a step function again), but the *direction* converges: rescaled, the learned vector recovers the true boundary to two decimal places.

7 see also

- [perceptron \(sign loss\)](#) — the classical algorithm in its supervised-learning habitat
- [multilayer perceptron](#) — stacking units to conquer xor
- [feedforward deep neural networks](#) — where the sigmoid unit becomes a layer
- [convolutional neural networks](#) — weight sharing as structured perceptrons

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.

McCulloch, Warren S. and Pitts, Walter (1943). *A Logical Calculus of the Ideas Immanent in Nervous Activity*.