

Neural Networks

Last modified: 2026-10-27

i Source

Adapted from Brian Hutchinson’s Fall 2025 lecture notes, reorganized as exercises. The threshold neuron, its two diagrams, and the deep-network and autoencoder sketches come from Kameron Decker Harris’ 2022 Week 7 notes. The layer-counting convention, depth and width, and the region-counting argument for depth come from Logan Sizemore’s Spring 2025 lectures on representational power and on deep networks.

This chapter covers what a network can *represent*. We converted the source’s second half — training, minibatching, and back-propagation — in the back-propagation chapter, where it belongs and where the source itself defers it (“will come back to this”). We kept the unsplit original as an outtake in this page’s source. We kept the polished derivation this chapter carried before the rebuild, and the “Training deep networks” material (vanishing, exploding and shattered gradients; residual connections; choosing the hidden non-linearity), as an outtake too. The back-propagation chapter has since converted the substance of the training material: vanishing, exploding and shattered gradients, the rectified linear unit (ReLU) remedy, residual connections and choosing the hidden non-linearity.

We do not repeat the source’s cross-entropy and maximum-likelihood derivations here, since logistic regression already works them; Exercise 8.1 cross-references them instead.

We qualify one claim rather than reproduce it. The source says an arbitrary logic function “may require $O(2^p)$ hidden nodes”, the cost of the disjunctive normal form (DNF) construction. Exercise 5.10 builds that construction, and then shows that parity — the standard example of a hard Boolean function — needs only p units.

1 When a line is not enough

Linear regression fits a hyperplane and logistic regression separates with one. Both are useful far beyond what their simplicity suggests, and both stop at the same limit: the boundary they draw is flat.

Plenty of problems are not flat. Which object a grid of pixels shows depends on colour, rotation and lighting together; which speech sound a spectrum encodes depends on the speaker’s age, their vocal tract and the microphone at once. In each case the quantity you want is a complicated function of the measurements you have, and no weighting of those measurements recovers it.

There are three ways to go beyond a flat boundary, and this chapter is about the third.

- **Map the features into a new space with a transform you choose.** This is a *basis expansion*.
- **Map them into a new space with a transform you never write down.** This is the kernel trick, and it belongs to support vector machines.
- **Learn the transform from the data.** This is a *neural network*.

The first is the cheapest to understand and the most expensive to run, so it makes the right starting point.

Exercise 1.1 (Three ways out of a straight line). Every model in these notes so far draws a straight boundary, or a boundary made of straight pieces. Logistic regression is the clearest case: its boundary is the hyperplane $\vec{\beta} \cdot \vec{x} = 0$.

The exclusive-or (XOR) data of the decision boundary exercise has a point of each class in each of two opposite quadrants. Say in one sentence why no choice of $\vec{\beta}$ classifies all four points, and why this is a statement about the *model* rather than about the fitting algorithm^a.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

Solution

Solution 1.1. A linear model assigns $\hat{y} = 1$ exactly on one side of a hyperplane. The XOR circles sit at $(-1, 1)$ and $(1, -1)$ and the crosses at $(1, 1)$ and $(-1, -1)$, so each class occupies two *opposite* quadrants. Any line through the plane leaves the two opposite quadrants on the same side or splits both of them, so no line has all circles on one side.

This is a statement about the model because the failure holds for every $\vec{\beta}$ at once, not for the particular $\vec{\beta}$ some algorithm happened to return. A better optimizer, more iterations or a different starting point change nothing: the function class does not contain a separator. The distinction matters because the two failures look identical from the outside — a fit that does not classify the training data — and they have opposite remedies.

Exercise 1.2 (Three ways out of a straight line (part 2)). Use the setup of Exercise 1.1.

Suppose you are allowed to invent one new feature, computed from x_1 and x_2 , and then fit a linear model in the enlarged feature vector. Give a feature that makes XOR linearly separable, and give the $\vec{\beta}$ that separates it.

Solution

Solution 1.2. Take $x_3 = x_1 x_2$. The class is the sign of $x_1 x_2$: the crosses have $x_3 = 1$ and the circles have $x_3 = -1$. So $\vec{\beta} = (0, 0, 0, 1)^\top$, reading the coefficients as $(\beta_0, \beta_1, \beta_2, \beta_3)$, gives $\vec{\beta} \cdot \vec{x} = x_1 x_2$, which is positive on exactly the crosses.

The boundary in the *enlarged* space is the hyperplane $x_3 = 0$. Pulled back to the original plane it is the set $x_1 x_2 = 0$, which is the pair of axes — not a line. A linear boundary in the enlarged space is a curved one in the original space, and that is the whole idea.

Exercise 1.3 (Three ways out of a straight line (part 3)). Use the setup of Exercise 1.1.

Exercise 1.2 is the whole idea of this chapter in miniature. Three strategies follow from it, and they differ in *who chooses* the new features:

- a fixed transform you write down yourself;
- a fixed transform you never write down, used only through inner products;
- a transform whose parameters are fitted alongside everything else.

Name the method this course attaches to each, and say which one requires you to know in advance which features matter.

Solution

Solution 1.3.

who chooses the features	method	needs advance knowledge?
you write the transform down	basis expansion	yes
the transform is implicit, used only through inner products	kernels and SVMs	partly: you choose the kernel
the transform is fitted	neural networks	no

Basis expansion is the one that needs you to know which features matter. You must commit to $\tilde{D}$ before seeing how well it works, and Exercise 2.5 shows that guarding against a wrong choice by including everything is expensive.

Theorem 1.1 (XOR is not linearly separable). *The four points $(-1, 1)$ and $(1, -1)$ of one class and $(1, 1)$ and $(-1, -1)$ of the other cannot be separated by any hyperplane $\vec{\beta} \cdot \tilde{x} = 0$ in the plane, for any choice of $\vec{\beta}$ [1].*

i Proof

Proof. By Solution 1.1. □

Example 1.1 (Searching for a separating line). Try 10,000 directions $\vec{\beta}$ and count how many of the four XOR points each classifies correctly by the sign of $\vec{\beta} \cdot \tilde{x}$:

1.1 Python

```
import numpy as np

pts_x = np.array([[1, 1], [-1, -1], [-1, 1], [1, -1]], float)
lab_x = np.array([1, 1, -1, -1])
ang = np.linspace(0, 2 * np.pi, 10000)
W_x = np.column_stack([np.cos(ang), np.sin(ang)])
correct = (np.sign(pts_x @ W_x.T) == lab_x[:, None]).sum(0)
print(correct.max())
```

2

1.2 R

```
pts_x <- rbind(c(1, 1), c(-1, -1), c(-1, 1), c(1, -1))
lab_x <- c(1, 1, -1, -1)
ang <- seq(0, 2 * pi, length.out = 10000)
W_x <- rbind(cos(ang), sin(ang))
correct <- colSums(sign(pts_x %*% W_x) == lab_x)
max(correct)
```

[1] 2

No direction classifies all four points, as Theorem 1.1 says.

Theorem 1.2 (The product feature separates XOR). *Add the feature $x_3 = x_1 x_2$ to the four XOR points of Theorem 1.1. In the enlarged feature space, $\vec{\beta} = (0, 0, 0, 1)^\top$ classifies all four points correctly, with boundary the hyperplane $x_3 = 0$. In the original plane that boundary is the set $x_1 x_2 = 0$, the pair of axes.*

i Proof

Proof. By Solution 1.2. □

Example 1.2 (The score with the product feature). Add $x_3 = x_1 x_2$ and score the four points with $\vec{\beta} = (0, 0, 0, 1)^\top$:

1.3 Python

```
import numpy as np

pts_p = np.array([[1, 1], [-1, -1], [-1, 1], [1, -1]], float)
feat = np.column_stack([np.ones(4), pts_p, pts_p[:, 0] * pts_p[:, 1]])
print(feat @ np.array([0, 0, 0, 1.0]))

[ 1.  1. -1. -1.]
```

1.4 R

```
pts_p <- rbind(c(1, 1), c(-1, -1), c(-1, 1), c(1, -1))
feat <- cbind(1, pts_p, pts_p[, 1] * pts_p[, 2])
drop(feat %*% c(0, 0, 0, 1))

[1] 1 1 -1 -1
```

The scores are positive for the first two points and negative for the last two, which separates the classes.

2 Basis expansion

Adding x_1x_2 by hand solved the problem in Exercise 1.1. The obvious generalization is to stop choosing and add everything: every product of the input coordinates up to some degree.

Exercise 2.1 (Write the expansion out). A **basis expansion** replaces $\tilde{x}$ by a vector of functions of $\tilde{x}$:

$$\tilde{D}(\tilde{x}) = \begin{bmatrix} 1 \\ f_1(\tilde{x}) \\ \vdots \\ f_{p^*}(\tilde{x}) \end{bmatrix}$$

and then fits an ordinary linear model in $\tilde{D}(\tilde{x})$ instead of $\tilde{x}$. The L th-order **polynomial** expansion takes the f_j to be every monomial in $x_1, \dots, x_p$ of total degree^a at most L . Write $\tilde{D}(\tilde{x})$ in full for $p = 1$, $L = 3$.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Solution

Solution 2.1. $p = 1$, $L = 3$:

$$\tilde{D}(x) = \begin{bmatrix} 1 \\ x_1 \\ x_1^2 \\ x_1^3 \end{bmatrix}$$

Four entries: one of each degree from 0 to 3.

Exercise 2.2 (Write the expansion out (part 2)). Use the setup of Exercise 2.1.

Write it in full for $p = 2$, $L = 2$. Be careful to list the first-order terms separately; a source that writes a single x in this list means^a both x_1 and x_2 .

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 2.2. $p = 2, L = 2$:

$$D(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}$$

Six entries: one of degree^a 0, two of degree 1, three of degree 2. We easily drop the cross term $x_1 x_2$, and it carries the interaction, so we should check for it.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Exercise 2.3 (Write the expansion out (part 3)). Use the setup of Exercise 2.1.

Write it in full for $p = 2, L = 3$, and check your list against a count you can make without writing anything down: the number of monomials of total degree^a exactly k in 2 variables is $k + 1$.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Solution

Solution 2.3. $p = 2, L = 3$ adds the four monomials of degree^a exactly 3:

$$D(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \\ x_1^3 \\ x_2^3 \\ x_1^2 x_2 \\ x_1 x_2^2 \end{bmatrix}$$

Ten entries, and the count confirms it: degrees^b 0, 1, 2, 3 contribute $1 + 2 + 3 + 4 = 10$. Getting the degree-3 block right means^c remembering $x_1^2 x_2$ and $x_1 x_2^2$ as well as the two pure cubes — the same omission as in Solution 2.2, one degree up.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^c<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 2.4 (Write the expansion out (part 4)). Use the setup of Exercise 2.1.

Nothing forces the f_j to be polynomials^a. Give an expansion suited to a feature that is strictly positive and spans several orders of magnitude, and one suited to a feature you expect to act periodically, such as hour of the day.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

Solution

Solution 2.4. For a strictly positive feature spanning orders of magnitude, use $\log x$, alone or alongside x . A linear model in $\log x$ says the *response* is linear in the number of doublings rather than in the raw amount, which is usually the claim you actually believe about income, population^a or concentration. A

cubic in the raw feature cannot express that and will use its flexibility mostly to fit the largest values. For an hour-of-day feature, use $\sin(2\pi x/24)$ and $\cos(2\pi x/24)$. The point is continuity where the day ends and starts again: hour 23 and hour 0 are one hour apart, and every polynomial^b in x makes them 23 apart. Adding the pair also builds in the period, which the data then does not have to use degrees^c of freedom to rediscover.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-population>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Adding everything works, and the drawback appears immediately.

Exercise 2.5 (Count the cost of hedging). Exercise 2.1 asks you to commit to a transform before you know which features matter. The obvious precaution is to include everything up to some order. This exercise computes the cost of that precaution.

How many monomials of total degree^a *exactly* k are there in p variables? Derive the count rather than quoting it: a monomial $x_1^{a_1} \cdots x_p^{a_p}$ of degree k is a choice of non-negative integers $a_1, \dots, a_p$ summing to k , so count those.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Solution

Solution 2.5. Count the solutions of $a_1 + \cdots + a_p = k$ in non-negative integers. Lay out k identical tokens in a row and insert $p - 1$ dividers among them; the tokens before the first divider give a_1 , those between the first and second give a_2 , and so on. Every arrangement of k tokens and $p - 1$ dividers gives exactly one solution and every solution gives exactly one arrangement, so the count is the number of ways to choose which k of the $k + p - 1$ positions hold tokens:

$$\#\{\text{degree exactly } k\} = \binom{k + p - 1}{k}$$

Check it at $p = 2$: $\binom{k+1}{k} = k + 1$, which is the count used in Exercise 2.3.

Exercise 2.6 (Count the cost of hedging (part 2)). Use the setup of Exercise 2.5.

Sum over $k = 0, \dots, L$ to get p^* , the length of $\tilde{D}(\tilde{x})$, and simplify to a single binomial coefficient.

Solution

Solution 2.6. Summing over k and using the hockey-stick identity $\sum_{k=0}^L \binom{k+p-1}{k} = \binom{L+p}{L}$:

$$p^* = \binom{p + L}{L}$$

We can derive the identity directly instead of looking it up. Choosing L items from $p + L$ is the same as deciding how many of the first few you skip, and grouping the choices by that number reproduces the sum term by term.

Check against Exercise 2.1: $\binom{4}{3} = 4$, $\binom{4}{2} = 6$, $\binom{5}{3} = 10$, matching the three lists written out there.

Exercise 2.7 (Count the cost of hedging (part 3)). Use the setup of Exercise 2.5.

Evaluate your formula at $p = 100$ for $L = 2$ and $L = 3$.

Solution

Solution 2.7.

$$\begin{aligned}\binom{102}{2} &= \frac{102 \cdot 101}{2} & \binom{103}{3} &= \frac{103 \cdot 102 \cdot 101}{6} \\ &= 5151, & &= 176,851\end{aligned}$$

A hundred features become five thousand, or a hundred and seventy-seven thousand.

Exercise 2.8 (Count the cost of hedging (part 4)). Use the setup of Exercise 2.5.

Show that for fixed L your answer grows like $p^L/L!$ as $p \rightarrow \infty$, which is the precise form of the claim that a second-order expansion costs $O(p^2)$ and a third-order one $O(p^3)$.

Solution

Solution 2.8. Write the binomial out:

$$\binom{p+L}{L} = \frac{(p+L)(p+L-1)\cdots(p+1)}{L!}$$

The numerator is a product of L factors, each of which is p plus a constant^a, so it equals $p^L(1 + O(1/p))$. Hence $p^* \sim p^L/L!$ with L held fixed.

At $p = 100$, $L = 3$ the approximation gives $10^6/6 \approx 166,667$ against the exact 176,851 — close, and an underestimate, as it must be since every discarded factor exceeds p .

Note what the statement does *not* say. The growth is polynomial^b in p for fixed L , and *exponential in L* for fixed p . “ $O(p^3)$ ” is a claim about widening the feature vector, not about raising the order.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

Exercise 2.9 (Count the cost of hedging (part 5)). Use the setup of Exercise 2.5.

A common guideline asks for several times more training points than parameters. Using Exercise 2.7, say what a third-order expansion of 100 features would demand, and connect this to the bias–variance decomposition: which term does an expansion like this reduce, and which does it inflate?

Solution

Solution 2.9. At $p = 100$, $L = 3$ there are 176,851 parameters, so the guideline asks for something on the order of a million labelled examples to fit a cubic in a hundred features. Very few problems supply that.

In bias–variance terms the expansion enlarges the function class, so it can only reduce bias^a: the best attainable function is now at least as good as before, since the old class sits inside the new one. What it inflates is variance^b. Each additional parameter is one more quantity estimated^c from the same fixed sample, so the fitted function moves more from one training set to the next, which is overfitting^d.

This chapter exists to avoid that difficulty. Basis expansion gains expressiveness by adding parameters you chose in advance and cannot revise. A neural network fixes the number of parameters first — through the width L — and then *fits* the transform, so expressiveness and parameter count are no longer the same thing.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

^b<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^d<https://morrison-lab.github.io/rme/chapters/predictor-selection.html#def-overfitting>

Theorem 2.1 (Number of polynomial features). For a polynomial^a in p variables, the number of monomials of degree^b at most L , including the constant^c 1, is

$$p^* = \binom{p+L}{L}.$$

For fixed L , $p^* \sim p^L/L!$ as $p \rightarrow \infty$.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

i Proof

Proof. By Solution 2.5, 2 and 4. □

Example 2.1 (Counting monomials). Count the monomials of degree^a at most L in p variables by listing exponent vectors, and compare with $\binom{p+L}{L}$ of Theorem 2.1:

2.1 Python

```
from itertools import product
from math import comb

def count_monomials(D, L):
    return sum(1 for e in product(range(L + 1), repeat=D) if sum(e) <= L)

print([(D, L, count_monomials(D, L), comb(D + L, L)) for D, L in [(2, 3), (3, 2), (4, 3)]]
      [(2, 3, 10, 10), (3, 2, 10, 10), (4, 3, 35, 35)])

print(comb(103, 3))

176851
```

2.2 R

```
count_monomials <- function(D, L) {
  sum(rowSums(do.call(expand.grid, rep(list(0:L), D))) <= L)
}

rbind(c(count_monomials(2, 3), choose(5, 3)),
      c(count_monomials(3, 2), choose(5, 2)),
      c(count_monomials(4, 3), choose(7, 3)))

[,1] [,2]
[1,] 10 10
[2,] 10 10
[3,] 35 35

choose(103, 3)

[1] 176851
```

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Before moving on, we state what a basis expansion does and does not change.

Exercise 2.10 (Why the old machinery still works). A basis expansion looks like it should break least squares: the fitted function $h(\tilde{x}) = \tilde{w}^\top \tilde{D}(\tilde{x})$ is a cubic in $\tilde{x}$, and nothing about a cubic is linear. Say precisely which variables the model is linear in, and why that is the only linearity least squares ever required.

Solution

Solution 2.10. The model is linear in the **parameters** $\tilde{w}$, not in the inputs. Least squares never asked for linearity in $\tilde{x}$: the normal equations come from differentiating $\hat{R}$ with respect to $\tilde{w}$ and setting the result to zero, and that calculation sees $\tilde{D}(\tilde{x}_i)$ only as a fixed vector of numbers. Whether those numbers arose as x_i , x_i^3 or $\sin x_i$ is invisible to the optimization. The phrase “linear model” is about the parameters throughout, and reading it as a claim about the shape of the fitted curve is a common and costly misreading.

Exercise 2.11 (Why the old machinery still works (part 2)). Use the setup of Exercise 2.10. Let D be the $n \times p^*$ matrix whose i th row is $\tilde{D}(\tilde{x}_i)^\top$. Express the empirical risk $\hat{R}(\tilde{w})$ in matrix form and give the normal equations, quoting the normal-equations exercise rather than re-deriving them.

Solution

Solution 2.11. With $D \in \mathbb{R}^{n \times p^*}$ and $\tilde{y} \in \mathbb{R}^n$,

$$\hat{R}(\tilde{w}) = \frac{1}{n} \|\tilde{y} - D\tilde{w}\|^2$$

and the normal equations are

$$D^\top D \tilde{w} = D^\top \tilde{y}$$

identical in form to the ordinary case with D in place of X .

Exercise 2.12 (Why the old machinery still works (part 3)). Use the setup of Exercise 2.10. State the condition under which those equations have a unique solution, in terms of D rather than $D(\mathbf{x})$. Then give a concrete expansion, on data you choose, that violates it — so the failure is a property of the transform and not of the data being unlucky.

Solution

Solution 2.12. The solution is unique if and only if $D^\top D$ is invertible^a, which holds if and only if D has full column rank^b p^* ($D^\top D$ is invertible exactly when D has full column rank^c). Two things can break it, and the expansion itself creates the second.

Too few points. D has n rows, so its rank is at most n . Since p^* grows as Theorem 2.1 shows, $p^* > n$ is easy to reach: a cubic in two features already needs $n \geq 10$.

A dependency among the columns, at any n . Take $p = 2$ and a dataset in which $x_2 = 2x_1$ for every point. Then in the second-order expansion the column for x_1x_2 equals twice the column for x_1^2 , and the column for x_2^2 equals four times it. D has rank at most 4 however many points you collect, because the dependency is between the columns rather than a shortage of rows.

The contrast is the point of the part. Collecting more data fixes the first and never fixes the second.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-matrix-inverse>

^b<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

^c<https://morrison-lab.github.io/mds/linear-algebra-diagonalization.html#cor-gram-invertible-iff>

Exercise 2.13 (Why the old machinery still works (part 4)). Use the setup of Exercise 2.10. Parts 1–3 hold for logistic regression too, with $\tilde{D}(\tilde{x}_i)$ in place of $\tilde{x}_i$. Say what the decision boundary of that model looks like in the original feature space, and check your answer against Solution 1.2.

Solution

Solution 2.13. The boundary is the set where $\tilde{w}^\top \tilde{D}(\tilde{x}) = 0$. In the enlarged space that is a hyperplane; in the original feature space it is the zero set of a degree- L polynomial^a — a conic for $L = 2$, a cubic curve for $L = 3$, and in general a curved surface rather than a flat one.

Theorem 1.2 is the smallest instance: the single added feature $x_3 = x_1x_2$ with $\tilde{w} = (0, 0, 0, 1)^\top$ gives the hyperplane $x_3 = 0$ in the enlarged space, which pulls back to $x_1x_2 = 0$, the two axes. Flat in the enlarged space, curved in the original space, and fitted by exactly the machinery of the previous two chapters.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

i Watch

Hutchinson covers nonlinear predictions by feature transformation^a (10 min) [2]. The video site requires a login, which enrolled students find in the syllabus.

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#nonlinear_prediction

3 Learning the features instead

A basis expansion fixes the features before it sees the data, and keeps every one it did not need. The alternative is to fix only *how many* features there will be, and let the fit decide what they are.

Write L for that number. The network computes

$$\tilde{h} = f\left(W^\top \begin{bmatrix} 1 \\ \tilde{x} \end{bmatrix}\right), \quad \tilde{y} = g\left(U^\top \begin{bmatrix} 1 \\ \tilde{h} \end{bmatrix}\right), \quad (1)$$

where f is applied coordinate by coordinate and g depends on the task. $\tilde{h}$ is the **hidden layer**: L features, computed from the input, with W fit alongside everything else.

Each coordinate of $\tilde{h}$ is a unit modelled on a neuron. A single artificial neuron computes a weighted sum of its inputs and fires when that sum passes a threshold:

$$y = f\left(\sum_{i=1}^p w_i x_i - \Theta\right) \quad (2)$$

where p is the number of inputs¹ and f is a non-linear activation, which fires above the threshold Θ (Figure 1).

¹<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

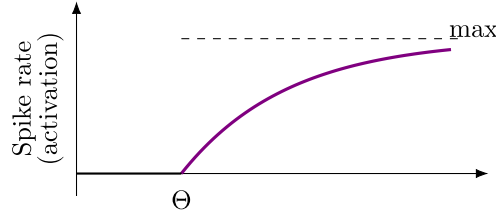


Figure 1: A threshold activation. A neuron's spike rate is zero until its input passes the threshold Θ , then rises and saturates at a maximum.

Read

GeeksforGeeks explains the perceptron^a [3], a single threshold unit.

^a<https://www.geeksforgeeks.org/deep-learning/what-is-perceptron-the-simplest-artificial-neural-network/>

Stacking a layer of L such units gives a vector of features (Figure 2). Each unit carries its own weights *and* its own threshold, so both are indexed by i :

$$D_i = f(w_i \cdot \tilde{x} - \Theta_i), \quad i = 1, \dots, L \quad (3)$$

and combining them is linear or non-linear depending on whether a second activation is applied:

$$\left. \begin{aligned} y &= \sum_{i=1}^L D_i(\tilde{x}) \beta_i + \beta_0 \\ y &= \tilde{D} \cdot \tilde{\beta} + \beta_0 \end{aligned} \right\} \text{Linear} \quad (4)$$

$$y = g(\tilde{D} \cdot \tilde{\beta} + \beta_0) \left\} \text{Non-linear}$$

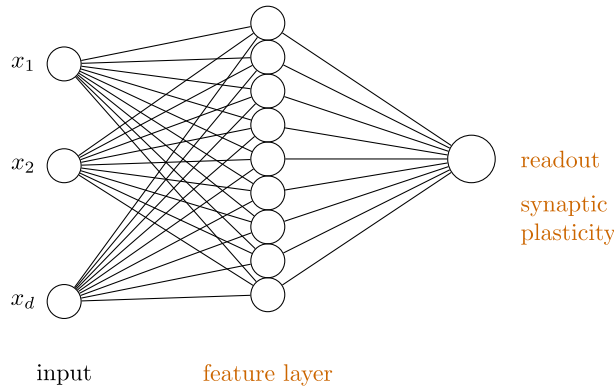


Figure 2: A layer of threshold units used as features. Every input feeds every unit of the feature layer, and one readout unit combines the features. The annotation *synaptic plasticity* marks the readout's connections as the ones that learn.

Exercise 3.1 (From a threshold to a bias). Equation 2 writes a unit with an explicit threshold Θ , and Equation 3 gives each of the L units its own weights w_i and its own threshold Θ_i . The network of Equation 1 has no Θ anywhere. Show that $f(\sum_{i=1}^p w_i x_i - \Theta)$ is a unit of the form $f(w \cdot \tilde{x})$, with $\tilde{x} \in \mathbb{R}^{p+1}$ carrying a constant^a 1 in its

first position.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 3.1. Put the threshold in front of the weights, with its sign flipped:

$$\sum_{i=1}^p w_i x_i - \Theta = \begin{bmatrix} -\Theta \\ \underset{\sim}{w} \end{bmatrix}^\top \begin{bmatrix} 1 \\ \underset{\sim}{x} \end{bmatrix}.$$

Exercise 3.2 (From a threshold to a bias (part 2)). Use the setup of Exercise 3.1. What is the weight on that constant^a?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 3.2. The weight on the constant^a is $-\Theta$. A threshold and a bias^b are one parameter under two names: the unit fires when the weighted sum exceeds Θ , which is when the weighted sum with the bias included, called the pre-activation, is positive. These notes use the bias, written w_0 for a single unit and stored in the row of W that multiplies the constant. Sources that start from the neuron write the threshold, as Equation 2 does.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/sds/estimation.html#def-bias>

Exercise 3.3 (From a threshold to a bias (part 3)). Use the setup of Exercise 3.1. The linear readout of Equation 4 reads a vector of features. Which model of this chapter is it when the D_i are fixed in advance?

Solution

Solution 3.3. With the D_i fixed in advance, the readout is a linear model in a chosen set of features, which is a basis expansion (Exercise 2.1).

Exercise 3.4 (From a threshold to a bias (part 4)). Use the setup of Exercise 3.1. Which model of this chapter is it when their weights and thresholds are fitted?

Solution

Solution 3.4. With their weights and thresholds fitted, it is the network of Equation 1 with g the identity and $C = 1$.

Exercise 3.5 (From a threshold to a bias (part 5)). Use the setup of Exercise 3.1. Say which entries of W and U the Θ_i , the β_i and β_0 become.

Solution

Solution 3.5. w_i and $-\Theta_i$ form column i of W , the β_i are the entries of U 's single column that multiply $h_1, \dots, h_L$, and β_0 is the entry that multiplies the constant^a. The two models differ only in whether the

features are written down or fitted.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

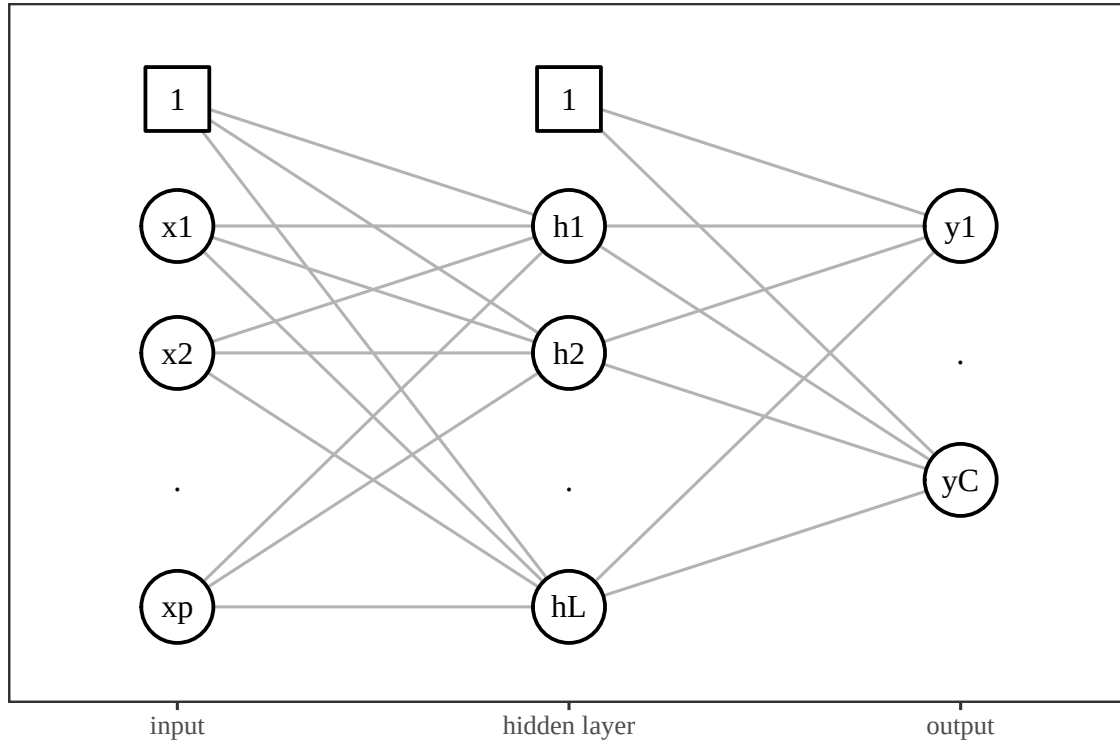


Figure 3: One hidden layer, drawn as columns of nodes from left to right. The squares are the extra entry 1 at the top of the first two columns; the circles are the inputs, the hidden units and the outputs; the vertical dots stand for the units that are not drawn.

Exercise 3.6 (Read the shapes off the diagram). With one hidden layer (Figure 3), the computation is two matrix multiplications with a function applied between them:

$$\tilde{h} = f(W^\top \tilde{x}), \quad \tilde{y} = g\left(U^\top \begin{bmatrix} 1 \\ \tilde{h} \end{bmatrix}\right)$$

Here $\tilde{x} \in \mathbb{R}^{p+1}$ carries a constant^a 1 in its first position, $\tilde{h} \in \mathbb{R}^L$ is the **hidden layer** whose entries are called units or neurons, and $\tilde{y} \in \mathbb{R}^C$ is the output. f and g are the **activation functions**, applied elementwise in the case of f .

Read the diagram to give the shapes of W and U , and say where each +1 comes from.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 3.6. $W \in \mathbb{R}^{(p+1) \times L}$ and $U \in \mathbb{R}^{(L+1) \times C}$.

$W^\top \tilde{x}$ must be a vector of length L , and $\tilde{x}$ has length $p+1$, so $W^\top$ is $L \times (p+1)$ and W is its transpose^a. The +1 in $p+1$ is the constant^b placed first in the input^c; the +1 in $L+1$ is the constant placed in front of $\tilde{h}$. Each supplies a bias, so that a layer computes an affine function rather than one forced through the origin.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-transpose>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 3.7 (Read the shapes off the diagram (part 2)). Use the setup of Exercise 3.6. Give a formula for the total number of parameters.

Solution

Solution 3.7.

$$\# \text{parameters} = (p + 1)L + (L + 1)C$$

Exercise 3.8 (Read the shapes off the diagram (part 3)). Use the setup of Exercise 3.6. Evaluate it for $p = 784$, $L = 100$, $C = 10$.

Solution

Solution 3.8. At $p = 784$, $L = 100$, $C = 10$:

$$\begin{aligned} 785 \times 100 + 101 \times 10 &= 78,500 + 1,010 \\ &= 79,510 \end{aligned}$$

Exercise 3.9 (Read the shapes off the diagram (part 4)). Use the setup of Exercise 3.6. Which of the two matrices holds almost all of them?

Solution

Solution 3.9. W holds 98.7% of them. That is the usual situation, and it says where the modelling capacity sits: almost the whole network is the map from raw inputs^a to learned features, and the classifier that uses those features has few parameters. Comparing 78,500 with the 176,851 of Solution 2.5's cubic in a hundred features is the argument for this chapter in one line.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 3.10 (Read the shapes off the diagram (part 5)). Use the setup of Exercise 3.6. Write y_c , a single output coordinate, as an explicit double sum over the entries of W and U . Written this way you can see each individual parameter, which the matrix form hides.

Solution

Solution 3.10. With f applied elementwise and U_{0c} the bias^a row,

$$y_c = g \left(U_{0c} + \sum_{j=1}^L U_{jc} f \left(\sum_{d=0}^p W_{dj} x_d \right) \right)$$

writing $x_0 = 1$ for the input^b's constant^c. Every parameter appears exactly once, W_{dj} inside the activation and U_{jc} outside it. Because of that asymmetry, fitting the two matrices is not the same problem twice: a change to U acts on the output directly, while a change to W has to pass through f first.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Exercise 3.11 (Read the shapes off the diagram (part 6)). Use the setup of Exercise 3.6. The task dictates the choice of g . Say which g you would use for a regression problem, for a two-class problem, and for a C -class problem, and in each case say what constraint on y that choice enforces.

Solution

Solution 3.11.

task	g	constraint on y
regression	identity, $g(z) = z$	none: any real vector
two classes	logistic expit	a single number in $(0, 1)$
C classes	softmax	non-negative, summing to 1

The constraint is the reason for the choice in each case. A regression target is unbounded, so any squashing function would cap the prediction at a value the data can exceed. A two-class output must be a probability. A C -class output must be a probability *distribution*^a, and softmax enforces the sum rather than merely bounding each coordinate — C separate sigmoids^b bound each one and do not make them sum to 1.

^a<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

^b<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 3.12 (Read the shapes off the diagram (part 7)). Use the setup of Exercise 3.6. Suppose you drop the constant^a 1 from the bracket in the second equation, so that $U \in \mathbb{R}^{L \times C}$. Describe a function the network can no longer represent, and connect this to the role of β_0 in logistic regression.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 3.12. Without the constant^a 1, the output is $g(U^\top h)$, which is $g(0)$ whenever $h = \vec{0}$. So the network can no longer set its output freely when the hidden layer is silent: with g the softmax, a zero hidden layer forces the uniform distribution^b, and the network cannot learn an unequal base rate for the classes.

This is the role of β_0 exactly. In logistic regression the intercept^c shifts the surface without rotating or scaling it, and removing it forces the boundary through the origin. Here U_{0c} shifts each output, and removing it forces the network to route every departure from $g(\vec{0})$ through the hidden units — using learned features for something a single parameter would supply.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Theorem 3.1 (Parameters of a one-hidden-layer network). A *network* with p inputs^a, L hidden units and C outputs, with a constant^b input^c to the hidden layer and a constant^d feeding the output layer, has $W \in \mathbb{R}^{(p+1) \times L}$, $U \in \mathbb{R}^{(L+1) \times C}$ and

$$(p+1)L + (L+1)C$$

parameters.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^d<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

i Proof

Proof. By Solution 3.6. □

Example 3.1 (Counting the entries of W and U). For $p = 4$, $L = 3$ and $C = 2$, build the two matrices and compare their total size with the formula of Theorem 3.1:

3.1 Python

```
import numpy as np

D_n, L_n, C_n = 4, 3, 2
W_n = np.zeros((D_n + 1, L_n))
U_n = np.zeros((L_n + 1, C_n))
print(W_n.size + U_n.size, (D_n + 1) * L_n + (L_n + 1) * C_n)

23 23
```

3.2 R

```
D_n <- 4; L_n <- 3; C_n <- 2
W_n <- matrix(0, D_n + 1, L_n)
U_n <- matrix(0, L_n + 1, C_n)
c(length(W_n) + length(U_n), (D_n + 1) * L_n + (L_n + 1) * C_n)

[1] 23 23
```

i Counting layers

The network of Equation 1 has an input layer, a hidden layer and an output layer, so some sources call it a three-layer network. These notes call it a **two-layer** network instead, counting the two weight matrices — input-to-hidden and hidden-to-output — rather than the three rows of units, or say that it has one hidden layer. The notation choices page^a gives the reason for counting this way. Both names are in use, so check which one a source means before comparing depths.

^a<https://morrison-lab.github.io/macros/notation-choices.html#sec-counting-layers>

This network — $h = f(W^\top[1; \tilde{x}])$ and $y = g(U^\top[1; h])$ — contains logistic regression as a special case, which we check explicitly before going further.

Exercise 3.13 (Find the models you already know). Take $f = \text{expit}$ and $g = \text{softmax}$ in Exercise 3.6. Set $L = p$ and $W = I$, the identity, so the hidden layer copies the input^a unchanged. Show that the network reduces to multinomial logistic regression, and say which of W and U plays the part of the coefficient matrix B .

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 3.13. With $W = I$ the pre-activation is $\tilde{x}$ itself, so $h = \text{expit}(\tilde{x})$ — elementwise, and note this is not quite the raw input^a, a point Solution 3.16 returns to. The output is then

$$\tilde{y} = \text{softmax}\left(U^\top \begin{bmatrix} 1 \\ h \end{bmatrix}\right)$$

which is exactly the multinomial logistic regression of the softmax exercise with $B = U$: one column of

U per class, each giving that class's score as a linear function of the features, passed through softmax. So U plays the part of B . W has no counterpart in logistic regression at all.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 3.14 (Find the models you already know (part 2)). Use the setup of Exercise 3.13. Now take $C = 1$ with $g = \text{expit}$ instead. Under the same choice of W , which model is this, and what is $\tilde{\beta}$?

Solution

Solution 3.14. $C = 1$ with $g = \text{expit}$ is ordinary two-class logistic regression, and $\tilde{\beta}$ is the single column of U — its bias^a entry U_{01} is β_0 and the remaining L entries are the slopes^b. The one thing to check is the label convention: a single output through expit models $P(Y = 1 \mid \tilde{x})$ directly, matching the 0/1 convention rather than the ± 1 margin form.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Exercise 3.15 (Find the models you already know (part 3)). Use the setup of Exercise 3.13. Parts 1 and 2 put the familiar model *on top of* the hidden layer. Say in one sentence what the network is therefore doing that logistic regression cannot, and why the phrase “learned features” describes it.

Solution

Solution 3.15. The network applies logistic regression to a representation it has learned, instead of to the features it was given. Logistic regression's boundary is linear in whatever it is given, and that is not a limitation the network shares, because W is fitted: the network can move the data into a space where a linear boundary works, and choose that space using the labels.

“Learned features” is the right phrase because $\tilde{h}$ occupies the same position in the model that $\tilde{D}(\tilde{x})$ occupies in Exercise 2.1 — a vector of derived quantities the classifier reads — with the difference that $\tilde{D}$ was written down in advance and $\tilde{h}$ is estimated^a.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 3.16 (Find the models you already know (part 4)). Use the setup of Exercise 3.13. $W = I$ needs $L = p$. Explain why a network with $L \neq p$ still contains a logistic regression as a special case whenever $L \geq p$, and what goes wrong when $L < p$.

Solution

Solution 3.16. For $L \geq p$, choose W so that its first p columns are the identity and the rest are zero. The first p hidden units then reproduce the argument of Solution 3.13, and the remaining $L - p$ units are constant^a, so the classifier can ignore them by setting the corresponding rows of U to zero. The network contains the logistic regression as a special case, so its best achievable fit is at least as good. For $L < p$ no choice of W recovers it. $W^\top$ maps $\mathbb{R}^{p+1}$ to $\mathbb{R}^L$ with $L < p$, so it has a non-trivial null space: some direction in feature space is annihilated before the classifier ever sees it, and two inputs^b differing only along that direction produce identical predictions. This is the bottleneck case of Exercise 3.24, and it is a restriction rather than a defect — it is the whole point when you want that representation. One caveat on Solution 3.13 and Solution 3.16, both of which set $W = I$. $W = I$ gives $\tilde{h} = \text{expit}(\tilde{x})$, not $\tilde{x}$, so the embedded model is a logistic regression on the *squashed* features. To recover it on the raw features, use $W = \varepsilon I$ for small ε and rescale U by $1/\varepsilon$: since $\text{expit}(z) \approx \frac{1}{2} + \frac{z}{4}$ near the origin, the hidden layer becomes affine in $\tilde{x}$ to any accuracy wanted, and the bias^c absorbs the $\frac{1}{2}$. The conclusion

— that the network’s class contains logistic regression’s — survives, but it needs this limiting argument rather than a single choice of W .

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/sds/estimation.html#def-bias>

The activation f looks like a detail, but it changes what the network can represent.

Exercise 3.17 (What happens if you leave the non-linearity out). Suppose f is the identity, so $\tilde{h} = W^\top \tilde{x}$, and take g to be the identity as well. Write y in terms of $\tilde{x}$, W and U , and show it is an affine function of $\tilde{x}$.

Solution

Solution 3.17. Writing $\tilde{h} = W^\top \tilde{x}$ and splitting U into its bias row $\tilde{u}_0^\top$ and its remaining block $U_{1:L,:}$,

$$\begin{aligned} \tilde{y} &= U^\top \begin{bmatrix} 1 \\ \tilde{h} \end{bmatrix} \\ &= U_{1:L,:}^\top W^\top \tilde{x} + \tilde{u}_0 \\ &= (WU_{1:L,:})^\top \tilde{x} + \tilde{u}_0 \end{aligned}$$

That is a matrix times $\tilde{x}$ plus a vector, so y is an affine function of $\tilde{x}$.

Exercise 3.18 (What happens if you leave the non-linearity out (part 2)). Use the setup of Exercise 3.17. Give the single matrix and the single vector it is built from.

Solution

Solution 3.18. So $y = M^\top \tilde{x} + \tilde{u}_0$ with the single matrix $M = WU_{1:L,:} \in \mathbb{R}^{(p+1) \times C}$, an affine function of $\tilde{x}$. (The constant^a coordinate of $\tilde{x}$ puts part of the intercept^b inside M already; $\tilde{u}_0$ is the part the second layer adds on top.)

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Exercise 3.19 (What happens if you leave the non-linearity out (part 3)). Use the setup of Exercise 3.17. Conclude what the network with K hidden layers and identity activations can represent, and say how that set compares with what one layer of the same kind represents.

Solution

Solution 3.19. By induction the same collapse happens at any depth: each layer contributes one more matrix factor, and a product of matrices is a matrix. A network of K identity-activation layers computes $y = M^\top \tilde{x} + b$ for some M and b , so it represents exactly the affine maps $\mathbb{R}^{p+1} \rightarrow \mathbb{R}^C$ — neither more nor fewer than a single layer of the same kind, and neither more nor fewer than linear regression. Depth changes nothing about what is representable.

Exercise 3.20 (What happens if you leave the non-linearity out (part 4)). Use the setup of Exercise 3.17. So depth adds nothing here. Say what it costs: compare the number of parameters against the number of parameters actually identified by the function being computed, taking

$$\begin{aligned} p &= C \\ &= 10 \end{aligned}$$

and $L = 100$ as a concrete case.

Solution

Solution 3.20. With $L = 100$ and

$$\begin{aligned} p &= C \\ &= 10, \end{aligned}$$

the parameter count of Theorem 3.1 is

$$\begin{aligned} 11 \times 100 + 101 \times 10 &= 1100 + 1010 \\ &= 2110. \end{aligned}$$

The function computed is an affine map $\mathbb{R}^{11} \rightarrow \mathbb{R}^{10}$, which is determined by $11 \times 10 = 110$ numbers. So 2110 parameters describe a 110-dimensional family: the remaining 2000 directions move the parameters without moving the function at all.

That is not merely wasteful. It is the non-identifiability of the softmax shift in a more severe form, and it makes the optimization worse behaved — the empirical risk is constant^a along a 2000-dimensional set through every point, so the Hessian^b is singular everywhere and nothing pins down where along that set the fit lands.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/mds/vector-calculus.html#def-hessian>

Exercise 3.21 (What happens if you leave the non-linearity out (part 5)). Use the setup of Exercise 3.17. Now let g be softmax but keep f the identity. Which model is this?

Solution

Solution 3.21. This is multinomial logistic regression, exactly. Solution 3.18 shows the two linear layers collapse to M , and softmax of an affine function of $\tilde{x}$ is the definition of the multinomial model, with $B = M$.

Exercise 3.22 (What happens if you leave the non-linearity out (part 6)). Use the setup of Exercise 3.17. Is anything gained over plain multinomial logistic regression?

Solution

Solution 3.22. Nothing is gained. The representable set is identical, and the reparameterization $M = WU_{1:L}$, loses the identifiability of Solution 3.19. The lesson is that a non-linear *output* does not rescue a linear interior: the squashing happens once, at the end, and cannot manufacture the interaction between features that f was there to supply.

Exercise 3.23 (What happens if you leave the non-linearity out (part 7)). Use the setup of Exercise 3.17. State the general principle in one sentence, in the form “a composition of ... is ...”.

Solution

Solution 3.23. A composition of affine maps is an affine map.

i Read

GeeksforGeeks explains activation functions^a [4].

^a<https://www.geeksforgeeks.org/machine-learning/activation-functions-neural-networks/>

That leaves L , the one number the architecture asks you for.

Exercise 3.24 (What the width of the hidden layer decides). Take a hidden layer with L units on the input^a that has p features. Three cases are worth naming.

$L > p$. Say what this does to the function class and what it does to the risk of overfitting^b, quoting the bias–variance reasoning of Solution 2.9.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/rme/chapters/predictor-selection.html#def-overfitting>

Solution

Solution 3.24. $L > p$ is an **expansion**: the hidden layer has more coordinates than the input, so it can hold each original feature plus derived combinations, and the function class strictly grows with L (the argument of Solution 3.16 embeds any narrower network in a wider one).

The drawback is the one Solution 2.5 quantifies. More units means more parameters estimated from the same sample, so the fitted function varies more from one training set to the next: bias falls, variance rises. A basis expansion fixes the enlarged space in advance; here the fit chooses it, so a network spends a given parameter budget on directions the labels say are useful.

Exercise 3.25 (What the width of the hidden layer decides (part 2)). Use the setup of Exercise 3.24. $L = p$. The notes call this “nothing special”. Say why — and be careful: Solution 3.16 shows that a width- p layer with a fitted W is *not* the same as no layer at all. Distinguish what is special about the number p from what the layer is doing.

Solution

Solution 3.25. Only the *number* is unremarkable when $L = p$: nothing about the input^a having p coordinates makes p hidden units a natural choice, and the count alone tells you nothing, since the function class is neither maximally rich nor deliberately restricted.

The layer itself still does real work. With W fitted, a width- p layer applies a learned non-linear transformation of the feature space, which is a strictly larger class than logistic regression on the raw features. The case is uncommon because nothing recommends it, not because it is inert — and reading “nothing special” as “equivalent to no hidden layer” is the mistake Solution 3.21 rules out.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 3.26 (What the width of the hidden layer decides (part 3)). Use the setup of Exercise 3.24. $L < p$, a bottleneck. Say what the layer must do, why this can be desirable rather than merely lossy, and name one thing you could do with $\tilde{h}$ that you could not do with $\tilde{x}$.

Solution

Solution 3.26. A bottleneck must represent p -dimensional data in $L < p$ numbers, so it must discard something; the question is what. Fitted against a label, it discards the directions that do not help predict it, which is exactly right when the raw features are redundant — neighbouring pixels, correlated sensors, one-hot columns that move together.

It is desirable rather than lossy whenever the signal genuinely occupies fewer dimensions than the

measurement. Compression is then a form of regularization: the network cannot overfit along a direction it has discarded.

With $\tilde{h}$ you can plot the data. Take $L = 2$ and every example becomes a point in the plane, so the classes can be seen to separate or not. $\tilde{x}$ with $p = 784$ admits no such picture, and this is the ordinary use of a bottleneck outside prediction altogether.

Exercise 3.27 (What the width of the hidden layer decides (part 4)). Use the setup of Exercise 3.24. Give the bottleneck’s failure mode: describe a $p = 2$, $L = 1$ dataset on which any bottleneck network must misclassify, however W is chosen.

Solution

Solution 3.27. Take $p = 2$, $L = 1$, and put the classes in concentric rings: label 1 for points with $\|\tilde{x}\| < 1$ and label 0 for $\|\tilde{x}\| > 2$, sampled all the way around both circles.

The first layer computes $a = \tilde{w} \cdot \tilde{x} + b$, a single number, and everything downstream sees only a . The level sets^a of a are parallel lines, and every such line crosses both the inner disc and the outer ring, so some inner point and some outer point share a value of a and receive the same prediction. One of them is wrong, whatever W is and whatever the rest of the network does.

The general statement: the bottleneck’s level sets are fixed by W alone, so any two points that W maps together are indistinguishable to the whole network — a constraint on the architecture, not on the fit. Widening to $L = 2$ removes it; $a_1 = x_1$, $a_2 = x_2$ already suffices.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

Exercise 3.28 (Watch the hidden width change a fit to real cars). The `Auto` data of James et al. [5] give the fuel efficiency (`mpg`) and horsepower of 392 cars. Download them from the ISLP labs repository^a or the book’s R resources page^b, or use the copy `data/auto.csv`^c on this site. Any computer with R or Python can run the code in the solution.

Standardize `horsepower` using the training rows, and call the result x . Use every fourth row, starting with the first, as the test set, and the other rows for training. A network with one hidden layer of L units predicts

$$\hat{y}(x) = b + \sum_{j=1}^L w_j \tanh(3(x - c_j)),$$

where we space the L centres c_j evenly from -2 to 2 . To keep the exercise small, we fix the hidden units by this rule and fit only b and the weights w_j , by least squares on the training rows.

Before computing anything, predict the shape of the test sample mean squared error (sMSE) as L grows from 2 to 16. Compare it to the test error of a straight line in x .

^ahttps://github.com/Morrison-Lab/ISLP_labs

^b<https://www.statlearning.com/resources-second-edition>

^c<https://morrison-lab.github.io/lds/chapters/data/auto.csv>

Solution

Solution 3.28. A straight line cannot follow the curve of fuel economy against horsepower, so a network with enough units should beat it. With only two units, the centres sit at the two edges of the range, far from most cars, so the network has little room to bend and can do worse than the line. After that the test error should fall as units are added, and then flatten once the curve is captured.

Exercise 3.29 (Watch the hidden width change a fit to real cars (part 2)). Use the setup of Exercise 3.28. Fit the network for $L \in \{2, 4, 8, 16\}$, and fit the straight line. Report the test sMSE of each.

Solution

Solution 3.29.

3.2.1 Python

```
import numpy as np
import pandas as pd

auto_py = pd.read_csv("data/auto.csv").dropna(subset=["mpg", "horsepower"])
hp = auto_py["horsepower"].to_numpy(dtype=float)
mpg = auto_py["mpg"].to_numpy(dtype=float)
is_test = np.arange(len(hp)) % 4 == 0

x = (hp - hp[~is_test].mean()) / hp[~is_test].std(ddof=1)

def design(x, L):
    centres = np.linspace(-2, 2, L)
    return np.column_stack([np.ones_like(x)] +
                           [np.tanh(3 * (x - c)) for c in centres])

def test_mse_py(L):
    X = np.column_stack([np.ones_like(x), x]) if L == 0 else design(x, L)
    beta = np.linalg.lstsq(X[~is_test], mpg[~is_test], rcond=None)[0]
    return np.mean((mpg[is_test] - X[is_test] @ beta) ** 2)

width_py = pd.Series({L: test_mse_py(L) for L in [0, 2, 4, 8, 16]})
width_py
```

0	27.354129
2	50.475812
4	22.680171
8	22.433200
16	22.799542

dtype: float64

3.2.2 R

```

auto <- read.csv("data/auto.csv") |>
  tidyr::drop_na(mpg, horsepower)
is_test <- (seq_len(nrow(auto)) - 1) %% 4 == 0

x <- (auto$horsepower - mean(auto$horsepower[!is_test])) /
  sd(auto$horsepower[!is_test])

design <- function(x, L) {
  if (L == 0) return(cbind(1, x))
  centres <- seq(-2, 2, length.out = L)
  cbind(1, sapply(centres, function(cj) tanh(3 * (x - cj))))
}

test_mse <- function(L) {
  X <- design(x, L)
  fit <- lm.fit(X[!is_test, , drop = FALSE], auto$mpg[!is_test])
  pred <- X[is_test, , drop = FALSE] %*% fit$coefficients
  mean((auto$mpg[is_test] - pred)^2)
}

widths <- c(0, 2, 4, 8, 16)
width_mse <- setNames(sapply(widths, test_mse), widths)
width_mse

      0      2      4      8     16
27.35413 50.47581 22.68017 22.43320 22.79954

# The claims the prose makes: both languages agree, and the best network
# beats the straight line.
stopifnot(
  max(abs(width_mse - unlist(reticulate::py$width_py)[as.character(widths)])) < 1e-6,
  min(width_mse[-1]) < width_mse[["0"]],
  width_mse[["2"]] > width_mse[["0"]],
  width_mse[["16"]] < width_mse[["2"]]
)
best_L <- widths[-1][which.min(width_mse[-1])]

```

Here $L = 0$ is the straight line. Its test sMSE is 27.4. The best network in the table uses $L = 8$ units and has test error 22.4. The full table is $L = 0$: 27.4; $L = 2$: 50.5; $L = 4$: 22.7; $L = 8$: 22.4; $L = 16$: 22.8.

Exercise 3.30 (Watch the hidden width change a fit to real cars (part 3)). Use the setup of Exercise 3.28. Plot the data with the straight line and the $L = 8$ fit.

Solution

Solution 3.30. Figure 4 shows the line and the $L = 8$ fit.

```

grid <- seq(min(x), max(x), length.out = 200)
fit_curve <- function(L) {
  X <- design(x, L)
  lm.fit(X[!is_test, , drop = FALSE], auto$mpg[!is_test])$coefficients
}
library(ggplot2)
curve_df <- tibble::tibble(
  x = rep(grid, 2),
  mpg = c(design(grid, 0) %*% fit_curve(0), design(grid, 8) %*% fit_curve(8)),
  model = rep(c("straight line", "8 hidden units"), each = length(grid))
)
point_df <- tibble::tibble(x = x, mpg = auto$mpg, is_test = is_test)
# The fit uses standardized horsepower; the ticks show horsepower.
hp_train_mean <- mean(auto$horsepower[!is_test])
hp_train_sd <- sd(auto$horsepower[!is_test])
hp_ticks <- pretty(x * hp_train_sd + hp_train_mean)
ggplot(point_df, aes(x, mpg)) +
  geom_point(aes(shape = is_test), colour = "grey50") +
  geom_line(aes(linetype = model), data = curve_df) +
  scale_shape_manual(values = c("FALSE" = 16, "TRUE" = 1), guide = "none") +
  scale_linetype_manual(values = c("straight line" = "dashed",
                                   "8 hidden units" = "solid")) +

  scale_x_continuous(
    breaks = (hp_ticks - hp_train_mean) / hp_train_sd, labels = hp_ticks
  ) +
  labs(x = "horsepower", y = "mpg", linetype = NULL) +

  theme(legend.position = "bottom")

```

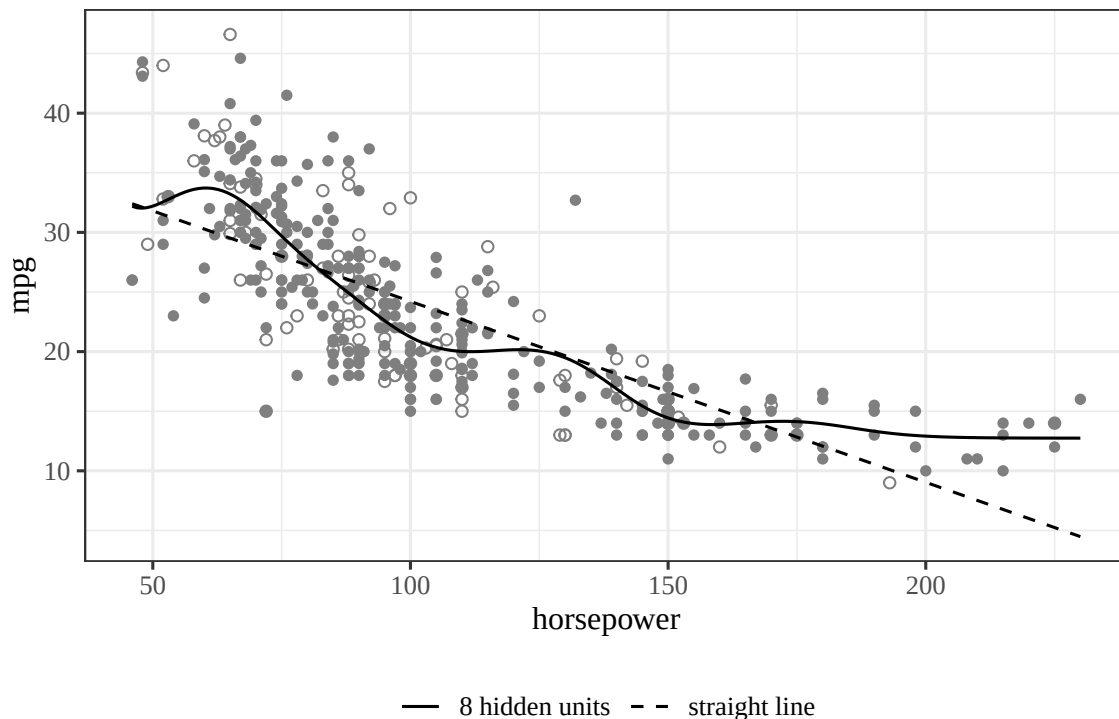


Figure 4: Fuel efficiency against horsepower for the Auto data, with the straight-line fit (dashed) and the fit of a hidden layer of 8 tanh units (solid), both fitted to the training rows on standardized horsepower.

Exercise 3.31 (Watch the hidden width change a fit to real cars (part 4)). Use the setup of Exercise 3.28. Say what the hidden units let the fit do that the line cannot.

Solution

Solution 3.31. Each hidden unit is a smooth step at its own centre, so a weighted sum of them can bend the curve. Fuel economy falls steeply at low horsepower and levels off at high horsepower, a bend that a line cannot follow.

Exercise 3.32 (Watch the hidden width change a fit to real cars (part 5)). Use the setup of Exercise 3.28. The notes ask what L decides. Use your table to say what you would choose for L here, and why the training error alone could not tell you.

Solution

Solution 3.32. Choose the smallest L whose test error is close to the best in the table, because beyond that point extra units add parameters without improving the fit to new cars. The training error cannot make this choice: it measures how well the fit matches the rows it was fitted to, and a more flexible network can match those rows ever more closely whether or not it predicts new cars better. Only error on held-out rows can show where adding units stops helping.

Definition 3.1 (Autoencoder). An **autoencoder** is a network trained to reproduce its own input [6]: its target is $\tilde{x}$ itself, and a bottleneck layer in the middle (Exercise 3.26) holds a compressed representation of $\tilde{x}$ from which the layers after the bottleneck layer rebuild $\tilde{x}$.

Example 3.2 (Compressing six car measurements into two numbers). Take the 6 numeric features of the Auto data [5] that describe a car's engine and body: **cylinders**, **displacement**, **horsepower**, **weight**, **acceleration** and **year**, standardized to mean 0 and variance 1. Train an autoencoder with a bottleneck of 2 units, so the network is $6 \rightarrow 2 \rightarrow 6$ and its target is its own input. A network that ignored its input and always predicted the mean would have sMSE 1 on each standardized feature:

3.3 Python

```
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPRegressor

cols_ae = ["cylinders", "displacement", "horsepower", "weight", "acceleration", "year"]
auto_ae = pd.read_csv("data/auto.csv").dropna(subset=cols_ae)
X_ae = auto_ae[cols_ae].to_numpy(float)
X_ae = (X_ae - X_ae.mean(axis=0)) / X_ae.std(axis=0)

# The target is the input itself; the single hidden layer of 2 units is the bottleneck.
ae_fit = MLPRegressor(hidden_layer_sizes=(2,), activation="tanh", max_iter=5000,
                      random_state=0).fit(X_ae, X_ae)
mse_ae = np.mean((ae_fit.predict(X_ae) - X_ae) ** 2)
print(round(mse_ae, 3))

0.198
```

3.4 R

```
cols_ae <- c("cylinders", "displacement", "horsepower", "weight", "acceleration", "year")
auto_ae <- na.omit(read.csv("data/auto.csv")[, cols_ae])
X_ae <- scale(auto_ae)

# The target is the input itself; the single hidden layer of 2 units is the bottleneck.
set.seed(0)
ae_fit_r <- nnet::nnet(X_ae, X_ae, size = 2, linout = TRUE, maxit = 2000, trace = FALSE)
mse_ae_r <- mean((predict(ae_fit_r, X_ae) - X_ae)^2)
round(mse_ae_r, 3)
```

[1] 0.15

The autoencoder rebuilds the six features from two numbers with an sMSE of 0.15 in R (0.20 in Python), against 1 for the constant prediction. Both errors are measured on the cars the autoencoder was trained on, and from one random start, so they have no confidence interval: the fit chose its weights to make them small, and they say nothing about how well it would rebuild new cars. It can do this because the features are strongly related: heavy cars have more cylinders, bigger engines and more horsepower, so two numbers carry most of what the six do. The two hidden-unit values for a car are its compressed representation.

i Read

GeeksforGeeks explains autoencoders^a [7].

^a<https://www.geeksforgeeks.org/machine-learning/auto-encoders/>

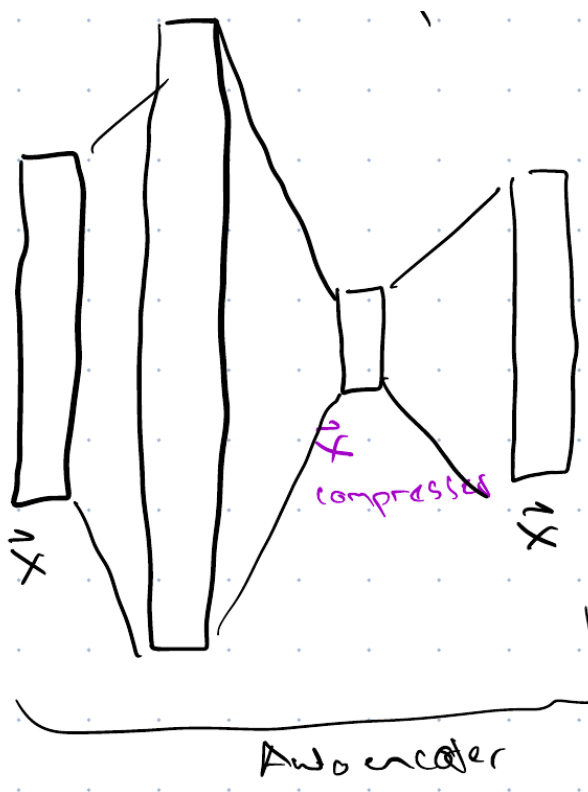


Figure 5: An [autoencoder](#) widens the feature vector x , squeezes it through a small compressed layer, and produces the reconstruction $\hat{x}$.

i Watch

- Hastie and Tibshirani introduce neural networks^a [8], video 10.1.
- Starmer gives the essential main ideas of neural networks^b [9].

^a<https://www.youtube.com/watch?v=jJb2qytbcNg>

^b<https://www.youtube.com/watch?v=CqOfi41LfDw>

i Watch

- The CS229 lecture introduces neural networks^a [10], Lecture 10.
- Abu-Mostafa also gives a lecture on neural networks^b [11], Lecture 10.

^a<https://www.youtube.com/watch?v=MfIjxPh6Pys>

^b<https://www.youtube.com/watch?v=Ih5Mr93E-2c>

i Watch

Hutchinson covers:

- layers^a (4 min) [2]
- feed-forward networks and hidden layers^b (25 min) [2]

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#layers

^bhttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#neural_networks

i Read

GeeksforGeeks explains:

- neural networks, for beginners^a [12]
- feedforward networks^b [13]

^a<https://www.geeksforgeeks.org/deep-learning/neural-networks-a-beginners-guide/>

^b<https://www.geeksforgeeks.org/deep-learning/feedforward-neural-network/>

i Read

The University of Washington's CSE 446 slides cover neural nets^a [14], Lecture 19.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/19-neural%20nets.pdf>

4 What a hidden unit can do

A hidden unit is $\text{expit}(w_0 + \tilde{w} \cdot \tilde{x})$ — the same object logistic regression fits, now used as a feature rather than as a prediction. Asking what one of them can represent turns out to have a simple answer.

Exercise 4.1 (What a single unit computes). Consider one hidden unit, $h = \text{expit}(\tilde{w} \cdot \tilde{x})$, with $\tilde{x} \in \mathbb{R}^{p+1}$ carrying a constant^a 1.

Describe the set of $\tilde{x}$ on which $h = \frac{1}{2}$, and the sets on which $h > \frac{1}{2}$ and $h < \frac{1}{2}$.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 4.1. $\text{expit}(z) = \frac{1}{2}$ exactly at $z = 0$, and expit is strictly increasing, so

$$h = \frac{1}{2} \iff \tilde{w} \cdot \tilde{x} = 0, \quad h > \frac{1}{2} \iff \tilde{w} \cdot \tilde{x} > 0, \quad h < \frac{1}{2} \iff \tilde{w} \cdot \tilde{x} < 0$$

The first is a hyperplane in $\mathbb{R}^{p+1}$; in the original p coordinates it is an affine hyperplane, positioned by the bias entry of $\tilde{w}$. The other two are the open halfspaces on either side.

Exercise 4.2 (What a single unit computes (part 2)). Use the setup of Exercise 4.1.

Replace $\tilde{w}$ by $t\tilde{w}$ and let $t \rightarrow \infty$. What does h converge to, at a point not on the set from Exercise 4.1? Conclude what a single unit computes in the limit, and name the object.

Solution

Solution 4.2. At a point with $\tilde{w} \cdot \tilde{x} = z_0 \neq 0$, scaling gives $\text{expit}(tz_0)$, and

$$\lim_{t \rightarrow \infty} \text{expit}(tz_0) = \begin{cases} 1 & z_0 > 0 \\ 0 & z_0 < 0 \end{cases}$$

since $tz_0 \rightarrow \pm\infty$ with the sign of z_0 . So in the limit the unit computes the **indicator of a halfspace**: it returns 1 on one side of a hyperplane and 0 on the other. The direction of $\tilde{w}$ fixes the orientation, the bias^a fixes the offset, and the magnitude fixes only how abruptly the unit switches — the same separation of direction from scale as in logistic regression.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

Exercise 4.3 (What a single unit computes (part 3)). Use the setup of Exercise 4.1.

The argument in Exercise 4.2 needs the qualification “not on the set from Exercise 4.1”. Say what happens there, and why no choice of t removes the qualification.

Solution

Solution 4.3. On the hyperplane itself $z_0 = 0$, so

$$\begin{aligned} \text{expit}(t \cdot 0) &= \text{expit}(0) \\ &= \frac{1}{2} \end{aligned}$$

for every t . The limit is $\frac{1}{2}$ there, not 0 or 1.

No t removes this because the value at $z_0 = 0$ does not depend on t at all. The convergence is pointwise off the hyperplane and never uniform near it: for any fixed t there are points either side within distance $1/t$ of the boundary where the unit is still near $\frac{1}{2}$. That transition band is not a technicality to be dismissed: it makes the unit differentiable, and so makes gradient descent^a applicable at all. A true step function has zero derivative everywhere it is defined and would supply no gradient to train on.

^a<https://morrison-lab.github.io/mds/optimization.html#def-gradient-descent>

Exercise 4.4 (What a single unit computes (part 4)). Use the setup of Exercise 4.1.

Now restrict the inputs^a to $\tilde{x} \in \{0, 1\}^p$. Exercise 4.2 says a unit computes a halfspace indicator; say what that becomes when only the corners of the cube are available, and how many distinct functions $\{0, 1\}^p \rightarrow \{0, 1\}$ there are in total. Exercise 4.5 asks which ones.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 4.4. On the corners of the cube a halfspace indicator becomes a **linearly separable** Boolean function: those $\{0,1\}^p \rightarrow \{0,1\}$ functions whose 1-corners can be cut off from the 0-corners by a single hyperplane.

There are 2^{2^p} Boolean functions on p inputs^a in total, since each of the 2^p corners may independently be sent to 0 or 1. At $p = 2$ that is 16, of which 14 are linearly separable — the two exceptions being XOR and its negation, which is Theorem 1.1 again in Boolean form. The count 2^{2^p} grows far faster than the number of halfspaces, so for larger p almost no Boolean function is computable by one unit, so we need a hidden layer, as in Exercise 5.1.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

When the inputs are binary, that halfspace becomes something familiar. Throughout what follows we use $\text{expit}(10) \approx 1$ and $\text{expit}(-10) \approx 0$.

Exercise 4.5 (Which gates can one unit compute?). Take binary inputs $x_1, x_2 \in \{0,1\}$ and one unit $h = \text{expit}(w_0 + w_1x_1 + w_2x_2)$. Read the output as the bit it is nearer to.

Write the pre-activation $w_0 + w_1x_1 + w_2x_2$ on each of the four corners, in terms of the weights.

Solution

Solution 4.5. On the corners $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ the pre-activation is

$$w_0, \quad w_0 + w_2, \quad w_0 + w_1, \quad w_0 + w_1 + w_2.$$

Exercise 4.6 (Which gates can one unit compute? (part 2)). Use the setup of Exercise 4.5.

What does the unit have to arrange, corner by corner, to output a value near 1 on some corners and a value near 0 on the rest?

Solution

Solution 4.6. The unit outputs a value near 1 where the pre-activation is large and positive, and a value near 0 where it is large and negative. So the design problem is entirely about signs and margins: the pre-activation has to be positive, by a wide margin, on the corners that should give 1, and negative, by a wide margin, on the rest.

Exercise 4.7 (Which gates can one unit compute? (part 3)). Use the setup of Exercise 4.5.

Which of NOT x_1 , AND, OR and XOR can one unit compute?

Solution

Solution 4.7. The unit is on exactly where $w_0 + w_1x_1 + w_2x_2 > 0$, which is one side of a line in the plane (Solution 4.2). So it can compute a Boolean function precisely when a line cuts that function's 1-corners off from its 0-corners: the linearly separable functions, 14 of the 16 (Solution 4.4).

NOT x_1 , AND and OR are all among them. NOT x_1 is on the edge $x_1 = 0$, AND on the single corner $(1,1)$, and OR on every corner except $(0,0)$; in each case a line cuts those corners off from the others. Once a line does, the margins of Solution 4.6 follow directly: every corner sits a positive distance from the line, so multiplying all three weights by a large enough positive number pushes every pre-activation as far from zero as we like without moving the line.

Exercise 4.8 (Which gates can one unit compute? (part 4)). Use the setup of Exercise 4.5. Give a geometric reason for any of them that one unit cannot compute.

Solution

Solution 4.8. XOR is not. Its 1-corners are $(0, 1)$ and $(1, 0)$. If both were strictly on the positive side of a line, so would be their midpoint $(\frac{1}{2}, \frac{1}{2})$, since a half-plane is convex^a. But that is also the midpoint of $(0, 0)$ and $(1, 1)$, so at least one of those would be on the positive side too. This is Theorem 1.1 again, in Boolean form.

^a<https://morrison-lab.github.io/mds/algebra-sums.html#def-convex-function>

Exercise 4.9 (Which gates can one unit compute? (part 5)). Use the setup of Exercise 4.5. Evaluate $\text{expit}(10)$ and $\text{expit}(-10)$ to six decimal places. The output is never exactly 0 or 1: say why that is unavoidable for any finite weights, and why it is a feature rather than a defect.

Solution

Solution 4.9. $\text{expit}(10) = 0.999955$ and $\text{expit}(-10) = 0.000045$.

Exactness is impossible because $\text{expit}(z) = 1/(1 + e^{-z})$ and $e^{-z} > 0$ for every finite z , so $\text{expit}(z) \in (0, 1)$ strictly: the values 0 and 1 are approached and never attained. Larger weights move closer without reaching it, which is the separation problem seen from the other side.

It is a feature because it makes the unit differentiable. A true step function has derivative 0 wherever it is defined and no derivative at the jump, so gradient descent^a would receive no signal anywhere. The sigmoid^b trades exactness for a usable gradient, and Solution 4.3 shows what that trade gives up.

^a<https://morrison-lab.github.io/mds/optimization.html#def-gradient-descent>

^b<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 4.10 (Which gates can one unit compute? (part 6)). Use the setup of Exercise 4.5. Take any corner c of the cube $\{0, 1\}^p$. Show that some single unit is on at c and off at every other corner. An argument that such weights exist is enough; you do not need to write them down. You will use this in Exercise 5.10.

Solution

Solution 4.10. Every corner of the cube is a vertex of it: no corner is an average of the others. So c lies outside the convex^a hull of the remaining $2^p - 1$ corners, and a point outside a closed convex set can be strictly separated from it by a hyperplane. Put c on the positive side, and scale the weights up as in Solution 4.6 until every pre-activation is decisive.

One unit per corner, firing on that corner alone: this is the building block Exercise 5.10 counts.

^a<https://morrison-lab.github.io/mds/algebra-sums.html#def-convex-function>

Watch

- Sanderson introduces neurons and layers, from 2:42^a [15, Ch. 1].
- He explains why networks have layers, from 5:31^b.
- He then compares ReLU with the sigmoid, from 17:03^c.
- Starmer shows ReLU in action^d [16].

^a<https://www.youtube.com/watch?v=aircAruvnKk&t=162s>

^b<https://www.youtube.com/watch?v=aircAruvnKk&t=331s>

^c<https://www.youtube.com/watch?v=aircAruvnKk&t=1023s>

^d<https://www.youtube.com/watch?v=68BZ5f7P94E>

5 What a layer of them can do

Exercise 1.1 said a linear boundary cannot represent XOR. Exercise 4.5 showed which pieces one unit can supply.

Exercise 5.1 (Two layers are enough for XOR). Exercise 1.1 showed that no single linear boundary separates XOR. Exercise 4.5 showed that one unit can compute OR, and AND. Put the two facts together.

Write XOR as a Boolean expression^a in OR and AND. There is more than one right answer; pick one that uses each of OR and AND exactly once.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-expression>

Solution

Solution 5.1. $x_1 \text{ XOR } x_2 = (x_1 \text{ OR } x_2) \text{ AND NOT } (x_1 \text{ AND } x_2)$: at least one of them is on, and not both.

Exercise 5.2 (Two layers are enough for XOR (part 2)). Use the setup of Exercise 5.1. Design a network with two hidden units h_1, h_2 and one output y that computes your expression^a.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-expression>

Solution

Solution 5.2. Take h_1 to be an OR unit and h_2 an AND unit, both reading x_1, x_2 . The output has to compute h_1 AND NOT h_2 .

Exercise 5.3 (Two layers are enough for XOR (part 3)). Use the setup of Exercise 5.1. Say which Boolean function each of the three units computes.

Solution

Solution 5.3. One hidden unit computes OR, the other computes AND, and the output computes h_1 AND NOT h_2 .

Exercise 5.4 (Two layers are enough for XOR (part 4)). Use the setup of Exercise 5.1. Use Exercise 4.7 to check that one unit can compute it.

Solution

Solution 5.4. Read as a function of (h_1, h_2) , h_1 AND NOT h_2 is on at the single corner $(1, 0)$. A line cuts one corner of the square off from the other three, so by Exercise 4.7 one more unit of the same kind computes it.

Exercise 5.5 (Two layers are enough for XOR (part 5)). Use the setup of Exercise 5.1. Tabulate h_1, h_2 and y on all four corners, reading each unit as the bit it is nearer to, and check the output is right on each.

Solution

Solution 5.5. Reading each unit as its nearest bit:

x_1	x_2	h_1	h_2	y	XOR
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

Right on all four corners. The hidden values are only approximately 0 and 1, but we set the output unit's margin in the same way as the hidden units', so an error of order $\exp(-10)$ in its inputs^a moves its pre-activation by far less than that margin.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

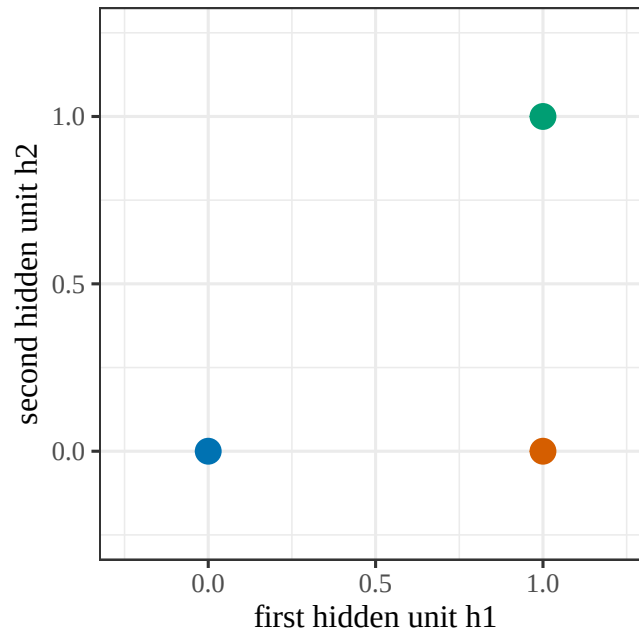
Exercise 5.6 (Two layers are enough for XOR (part 6)). Use the setup of Exercise 5.1. The hidden layer takes the four corners of the input^a square to four points in the (h_1, h_2) plane. Plot those four points, approximately.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 5.6. The four corners land at approximately

$$(0,0) \mapsto (0,0), \quad (0,1) \mapsto (1,0), \quad (1,0) \mapsto (1,0), \quad (1,1) \mapsto (1,1).$$



input corners ● (0, 0) ● (0, 1) and (1, 0) ● (1, 1)

Figure 6: The four corners of the unit square in the plane of the two hidden units. The corners $(0,1)$ and $(1,0)$ land on the same point.

Exercise 5.7 (Two layers are enough for XOR (part 7)). Use the setup of Exercise 5.1. Why is the output unit's job now easy, when Exercise 1.1 said it was impossible?

Solution

Solution 5.7. The two corners that should give 1 have been mapped to the *same* point, $(1, 0)$, and the two that should give 0 sit at $(0, 0)$ and $(1, 1)$. In the (h_1, h_2) plane the problem is no longer XOR at all: the positive class is one point, and a line through it separates it from the other two. The line $h_1 - h_2 = \frac{1}{2}$ will do.

So the output unit is not doing anything clever. It is solving a linearly separable problem, because the hidden layer handed it one. That is the whole idea: a network does not evade Exercise 1.1, it *moves the data* until the last layer's linear boundary is enough.

Exercise 5.8 (Two layers are enough for XOR (part 8)). Use the setup of Exercise 5.1. In Exercise 1.1 you separated XOR by adding the feature $x_3 = x_1x_2$ by hand. Which of h_1, h_2 is playing that role here?

Solution

Solution 5.8. h_2 is the AND unit, and $x_1 \text{ AND } x_2 = x_1x_2$ on $\{0, 1\}$, so h_2 is the hand-built feature $x_3 = x_1x_2$ of Exercise 1.1, rediscovered.

Exercise 5.9 (Two layers are enough for XOR (part 9)). Use the setup of Exercise 5.1. State the difference between the two solutions in one sentence.

Solution

Solution 5.9. The difference is who chose it. In Exercise 1.1 you looked at the data, saw that a product would do it, and wrote the product down; here the same feature is a unit whose weights are numbers the fitting procedure^a could have found on its own.

That is the trade Exercise 2.5 set up. A basis expansion writes down every product up to some degree^b, and pays $\binom{p+L}{L}$ for the ones it does not need. A hidden layer writes down L features and lets the fit decide what they are, paying $L(p+1)$ parameters regardless of what it discovers.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-average-fit>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

i Read

GeeksforGeeks explains how neural networks solve the XOR problem^a [17].

^a<https://www.geeksforgeeks.org/artificial-intelligence/how-neural-networks-solve-the-xor-problem/>

XOR took two hidden units. The natural next question is how many units an arbitrary function needs.

Exercise 5.10 (How many hidden units does a logic function need?). XOR needed two hidden units. Ask what an arbitrary Boolean function $f : \{0, 1\}^p \rightarrow \{0, 1\}$ needs.

Exercise 4.10 showed that one unit can fire on **one** named corner of $\{0, 1\}^p$ and nowhere else. Use that to write any f as a one-hidden-layer network. How many hidden units does your construction use, in terms of the number M of corners where $f = 1$?

Solution

Solution 5.10. Every corner where $f = 1$ gets its own unit. Write $M = |\{x : f(x) = 1\}|$ and list those corners $c^{(1)}, \dots, c^{(M)}$. For each corner c , Solution 4.10 gives a unit h_c that outputs about 1 at $x = c$ and about 0 everywhere else. So the construction uses M hidden units.

This is the disjunctive normal form (DNF) of f — f is true exactly when the input x is one of the corners where it is true — written as a network.

Exercise 5.11 (How many hidden units does a logic function need? (part 2)). Use the setup of Exercise 5.10.

Say what the output unit has to compute, and why the construction needs no more than an OR at the top.

Solution

Solution 5.11. At most one hidden unit is on at a time, because the corners are distinct and each unit fires on only its own. So the output has to answer “is any of them on”, which is an OR over M inputs^a. That OR, read as a function of the hidden layer, is off at the single corner where every h is 0 and on everywhere else. A hyperplane cuts that one corner off, so one unit computes the OR (Exercise 4.7, in M dimensions). Nothing more elaborate is needed at the top, because the hidden layer has already done the work of recognizing each corner.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 5.12 (How many hidden units does a logic function need? (part 3)). Use the setup of Exercise 5.10.

Bound M above. Then improve the bound by noticing that a network computing f is easily turned into one computing $1 - f$.

Solution

Solution 5.12. $M \leq 2^p$, since there are only that many corners, so the construction never needs more than 2^p hidden units.

The bound halves. If f is 1 on more than half the corners, build a network for $1 - f$ instead, which is 1 on fewer than half, and then negate the output unit’s weights and bias^a, which turns “is any of them on” into “are all of them off”. So $\min(M, 2^p - M) \leq 2^{p-1}$ units always suffice.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

Exercise 5.13 (How many hidden units does a logic function need? (part 4)). Use the setup of Exercise 5.10.

The “off” hidden units do not output exactly 0, and there are up to 2^{p-1} of them, so we might expect their leakage into the output sum to grow with p and eventually exceed the output’s margin. Suppose each corner unit’s pre-activation is $m > 0$ at its own corner and falls by at least $a > m$ for every coordinate in which the input^a disagrees with that corner. Bound the total leakage, by asking *which* off units contribute much, and show it grows only linearly in p .

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 5.13. Because the off units are not all equally off. Take an input^a x and a corner unit whose corner differs from x in k coordinates. Its pre-activation is at most $m - ak$, so its output is at most $\expit(m - ak) < e^{m-ak}$, which falls exponentially in k . There are $\binom{p}{k}$ corners at distance k from x , so

the total leakage from all the off units is at most

$$\sum_{k=1}^p \binom{p}{k} e^{m-ak} = e^m [(1 + e^{-a})^p - 1] \approx p e^{m-a} \quad \text{when } p e^{-a} \ll 1.$$

The leakage is dominated by the at most p units at distance 1, and grows only linearly in p , regardless of how large M is. The far-away units contribute nothing that matters, because expit falls exponentially in the distance faster than the number of corners at that distance grows.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 5.14 (How many hidden units does a logic function need? (part 5)). Use the setup of Exercise 5.10.

Now take the specific function $f(x) = x_1 \oplus x_2 \oplus \dots \oplus x_p$, the parity of the input^a. Exercise 5.12 gives a bound of order 2^p for it. Show that p hidden units are in fact enough, by building the network from the quantities $\mathbb{1}\{s \geq k\}$ where $s = \sum_d x_d$.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 5.14. Let $s = \sum_d x_d$, the number of ones in the input^a, which runs $0, 1, \dots, p$. Parity is the parity of s . Write $\mathbb{1}\{s \geq k\}$ for the indicator that at least k inputs^b are on, and check the alternating sum

$$\begin{aligned} \sum_{k=1}^p (-1)^{k+1} \mathbb{1}\{s \geq k\} &= \underbrace{1 - 1 + 1 - \dots}_{s \text{ terms}} \\ &= \begin{cases} 1 & s \text{ odd} \\ 0 & s \text{ even.} \end{cases} \end{aligned}$$

Exactly s of the indicators are on, so the sum telescopes to $s \bmod 2$.

Each indicator is one hidden unit. The count s is a linear function of the input, so the corners with $s \geq k$ and those with $s \leq k - 1$ lie on opposite sides of the hyperplane $s = k - \frac{1}{2}$, and a unit whose boundary is that hyperplane, with its weights scaled up, computes the indicator (Exercise 4.7, in p dimensions). The output unit then needs to be on when the alternating sum of the h_k is 1 and off when it is 0. That sum is a linear function of the hidden layer, so a threshold on it at $\frac{1}{2}$ is again one unit. So parity takes p hidden units, not 2^{p-1} .

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 5.15 (How many hidden units does a logic function need? (part 6)). Use the setup of Exercise 5.10.

Parity has half its corners set to 1, so Exercise 5.12's bound is as bad for it as for any function, and yet Exercise 5.14 does it in p units. What, then, does the $O(2^p)$ figure actually describe?

Solution

Solution 5.15. The $O(2^p)$ is the cost of the *construction* in parts 1–3, not a floor for any particular function. It says: here is a recipe that works for every f , and here is its cost. It does not say that some named f forces you to pay it, and Solution 5.14 shows that the obvious candidate does not.

Parity looks like it should be the hard case. Parity is the standard example of a function that circuits of AND and OR gates cannot compute at constant^a depth without exponentially many gates. But a hidden unit is not an AND or an OR gate. It thresholds a *weighted sum*, so it can count, and counting is

precisely what parity needs: the units h_k in Solution 5.14 are doing arithmetic on s , which no fixed-depth collection of AND and OR gates can do. The DNF construction of Solution 5.10 discards that ability, using each unit as a single-corner detector when it could have been a counter.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Exercise 5.16 (How many hidden units does a logic function need? (part 7)). Use the setup of Exercise 5.10.

Does anything survive as a statement about *most* functions?

Solution

Solution 5.16. Something does survive, but only on average. There are 2^{2^p} Boolean functions on p bits (Exercise 4.1), a doubly exponential count, while a network with L hidden units is fixed by $(p+1)L + (L+1)$ numbers. Suppose we store each number in b bits. Then there are at most $2^{b((p+1)L + L + 1)}$ networks, hence at most that many functions realisable with L units, and covering all 2^{2^p} functions needs

$$b((p+1)L + L + 1) \geq 2^p, \quad \text{so} \quad L \gtrsim \frac{2^p}{bp}.$$

All but a vanishing fraction of Boolean functions therefore need on the order of $2^p/p$ hidden units. Replacing the b -bit fiction with a bound on how many distinct functions a real-parameter family can realise — a Vapnik–Chervonenkis (VC) dimension argument — changes the polynomial^a factors in p and not the shape.

That argument counts, so it proves that almost every f is expensive without naming a single one, and it cannot name one: any f you can write down briefly is, by that very fact, drawn from a tiny and unrepresentative part of the 2^{2^p} . Parity is an example. This is the representational question only. We take up next whether we can *find* such a network, given data rather than the function itself.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

i Watch

Starmer builds a network with several inputs and outputs^a [18].

^a<https://www.youtube.com/watch?v=83LYR-1IcJA>

6 Universal approximation

Introductions to networks usually state a strong guarantee. The guarantee is true, and we need to read it carefully.

Exercise 6.1 (What “universal approximator” promises, and what it does not). Introductions to neural networks often claim that they are *universal approximators*: any function can be approximated arbitrarily well by a network with enough hidden units. The claim is true. This exercise is about reading it carefully.

Take $p = 1$ and one hidden layer. Using two units, build a function that is about 1 on an interval (a, b) and about 0 outside it.

Solution

Solution 6.1. Take the difference of two sigmoids with the same steepness and different offsets:

$$\text{bump}_{a,b}(x) = \text{expit}(t(x-a)) - \text{expit}(t(x-b)), \quad a < b.$$

Each sigmoid is one hidden unit and the subtraction is two output weights, +1 and -1, so the bump costs two units. Larger t makes the two edges sharper.

Exercise 6.2 (What “universal approximator” promises, and what it does not (part 2)). Use the setup of Exercise 6.1.

Sketch the two sigmoids^a and their difference.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Solution

Solution 6.2. The first sigmoid^a switches on near a and the second near b . For $x \ll a$ both terms are near 0, so the difference is near 0. For $x \gg b$ both are near 1, so the difference is near 0 again. Between a and b the first has switched on and the second has not, so the difference is near 1.

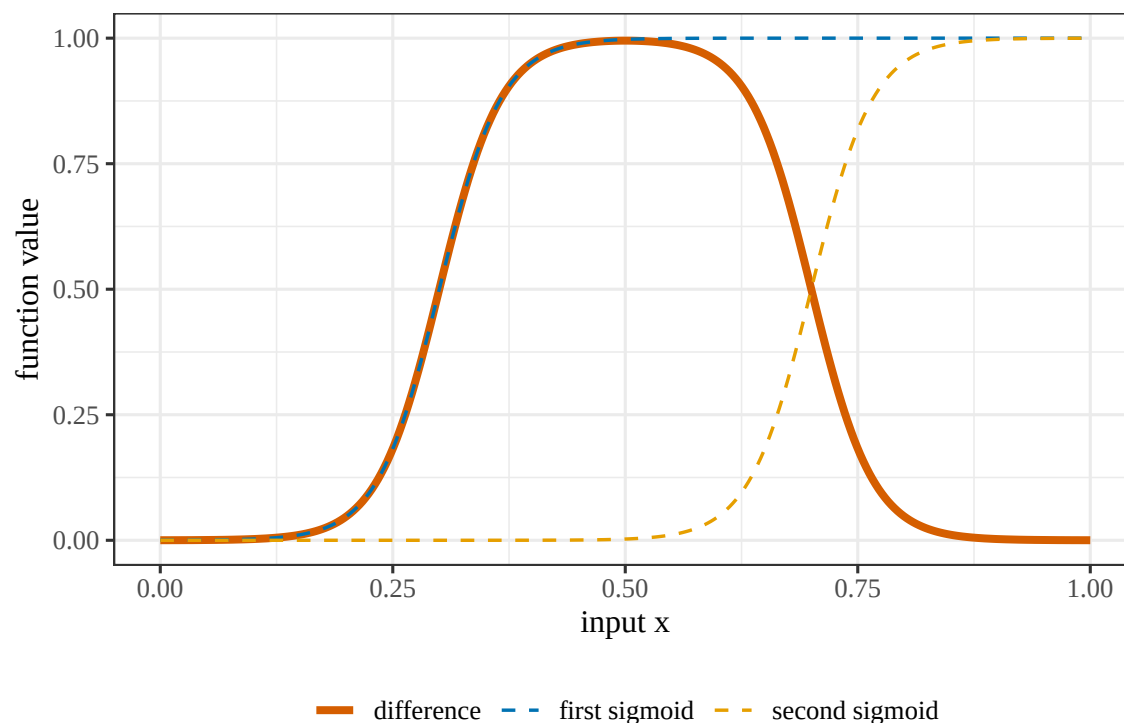


Figure 7: Two sigmoids^b with edges at $a = 0.3$ and $b = 0.7$, and their difference. The difference is near 1 between the edges and near 0 outside them.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^b<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 6.3 (What “universal approximator” promises, and what it does not (part 3)). Use the setup of Exercise 6.1.

Use N of these bumps, side by side, to approximate a continuous $f : [0, 1] \rightarrow \mathbb{R}$. How many hidden units does that cost?

Solution

Solution 6.3. Cut $[0, 1]$ into N intervals of length $1/N$ and put a bump on each. Each bump takes two hidden units, so this costs $2N$ hidden units.

Exercise 6.4 (What “universal approximator” promises, and what it does not (part 4)). Use the setup of Exercise 6.1.

How should you choose each bump’s height?

Solution

Solution 6.4. Give bump j the output weight $f((j + \frac{1}{2})/N)$, the value of f at the middle of that interval:

$$g(x) = \sum_{j=0}^{N-1} f\left(\frac{j + \frac{1}{2}}{N}\right) \text{bump}_{j/N, (j+1)/N}(x).$$

Exercise 6.5 (What “universal approximator” promises, and what it does not (part 5)). Use the setup of Exercise 6.1.

Roughly how does the error behave as N grows, and why?

Solution

Solution 6.5. Since at most one bump is on at a time, g is a piecewise-constant approximation of f — a staircase — so the error on an interval is how far f moves across a length of $1/N$. For differentiable f that is about $\max |f'|/(2N)$: halving the interval width halves the error, so the error falls like $1/N$, which is first order and not fast.

Measured on $f(x) = \sin(2\pi x) + 0.3x^2$ over $[0.05, 0.95]$, with $t = 25N$:

N	hidden units	$\max g - f $
4	8	0.436
8	16	0.252
16	32	0.139
32	64	0.068
64	128	0.034
128	256	0.017

Each doubling of N halves the error, as expected^a. Getting two decimal places took 256 hidden units for one input variable.

^a<https://morrison-lab.github.io/pds/expectation.html#def-expectation>

Exercise 6.6 (What “universal approximator” promises, and what it does not (part 6)). Use the setup of Exercise 6.1.

Now take $p = 2$ and try to build an indicator of the box $(a_1, b_1) \times (a_2, b_2)$. Show that you can do it with $2p$ units in a first hidden layer and one unit in a second. Then say what goes wrong if you try to assemble the same box out of one hidden layer in the obvious way.

Solution

Solution 6.6. Build the p one-dimensional bumps in the first layer, at $2p$ units, then ask the second layer whether all p of them are on:

$$\text{box}(x) = \text{expit} \left(T \left[\sum_{d=1}^p \text{bump}_{a_d, b_d}(x_d) - (p - \tfrac{1}{2}) \right] \right).$$

Inside the box every bump is on and the sum is about p ; outside, at least one is off and the sum is at most about $p - 1$. Thresholding at $p - \frac{1}{2}$ separates the two. This is an AND over p coordinates, and the second layer computes it.

Checked at $p = 2$ with $(a, b) = (0.3, 0.6) \times (0.4, 0.7)$, $t = 300$, $T = 40$: the box returns 1.000000 at interior points including the corners $(0.31, 0.41)$ and $(0.59, 0.69)$, and 0.000000 at $(0.45, 0.2)$, $(0.1, 0.55)$, $(0.1, 0.1)$ and $(0.61, 0.55)$.

In one hidden layer the obvious attempt fails. A single layer can only form $\sum_m u_m \text{expit}(w_m^T x + w_{0m})$, a weighted sum of halfspace-like ridges, and the AND you need is a *threshold of a sum*, not a sum. Adding the p bumps without thresholding gives a function that is p inside the box but also $p - 1$ on the whole slab where all but one coordinate is in range — a plus-sign shape, not a box. Applying a threshold to the sum removes the slab, and the threshold is a second layer.

Exercise 6.7 (What “universal approximator” promises, and what it does not (part 7)). Use the setup of Exercise 6.1.

Continuing Exercise 6.2’s strategy into p dimensions requires tiling the cube $[0, 1]^p$ with boxes. How many boxes, if each side is cut into N pieces?

Solution

Solution 6.7. Cutting each of p sides into N pieces gives N^p boxes, so the tiling costs N^p second-layer units, exponential in p .

Exercise 6.8 (What “universal approximator” promises, and what it does not (part 8)). Use the setup of Exercise 6.1.

Compare that count with Exercise 2.5, and say which of the two is worse.

Solution

Solution 6.8. That is worse than Exercise 2.5. A basis expansion of degree^a L costs $\binom{p+L}{L} \approx p^L/L!$, which is exponential in the *degree* L and only polynomial^b in the dimension p . The tiling is exponential in p itself, and p is the thing you do not get to choose. At $p = 784$, a modest degree-2 expansion is 308,505 terms while a tiling at $N = 2$ is 2^{784} boxes.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

Exercise 6.9 (What “universal approximator” promises, and what it does not (part 9)). Use the setup of Exercise 6.1.

The theorem holds for one hidden layer in any p , even though Exercise 6.3 says the tiling argument does not give it to you. What does that tell you about the *kind* of statement the theorem is?

Solution

Solution 6.9. That it is an existence theorem, proved by an argument that does not build the network. The tiling is one route to universality; it happens to need two layers and exponentially many units, and

the theorem says neither of those is necessary for the conclusion. What it does not say is how few units *are* necessary — an existence proof that does not construct the object does not come with a count. So “enough hidden units” in the statement carries real content, and the theorem leaves it unquantified. Exercise 5.10 is the same gap in the Boolean case: a construction gives 2^{p-1} , a counting argument gives $2^p/p$ for almost every function, and neither tells you what any particular function costs.

Exercise 6.10 (What “universal approximator” promises, and what it does not (part 10)). Use the setup of Exercise 6.1.

The Weierstrass approximation theorem says that polynomials^a are also universal approximators on a closed bounded interval. Given that, what does universality actually tell you about the choice between a network and a basis expansion?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

Solution

Solution 6.10. That universality is not the distinguishing property. Both families are universal, so a claim that a network can approximate anything is not an argument for a network over a basis expansion; it rules out neither and recommends neither.

Cost and adaptivity differ. Exercise 2.5 fixes the features in advance and pays $\binom{p+L}{L}$ of them, most of which the data never needed. A hidden layer fixes the *number* of features at L and pays $L(p+1)$ parameters to let the fit choose what they are (Exercise 5.8). The question worth asking about an architecture is how many parameters it needs for the accuracy you want on the problem you have, and universality does not address it.

Exercise 6.11 (What “universal approximator” promises, and what it does not (part 11)). Use the setup of Exercise 6.1.

List what the theorem does **not** say. Consider at least: the number of units required; whether fitting finds the approximating network; and what happens when you have n data points rather than the function itself.

Solution

Solution 6.11. The theorem is silent on all three.

- **How many units.** No bound, as Solution 6.5 says. The honest reading of “any function can be well approximated by some network” is that no function is ruled out, not that any particular one is cheap.
- **Whether fitting finds it.** The theorem is about *representation*: some setting of the weights approximates f . Gradient descent^a searches a non-convex surface and is not promised to reach it — and the constructions here are a warning, since they use weights like $t = 300$ that a fit starting from small random values and descending a gradient has no particular reason to visit. Representable and findable are different properties.
- **What happens with data.** The theorem approximates a *known* f on a compact set. In practice you hold n pairs^b (x_i, y_i) and not f , and a network with enough units to approximate any f has enough units to interpolate any n points, noise included. Universality is therefore a statement about the prediction function family^c being large, which is exactly the condition under which fitting the training data says least about new data.

A fourth omission worth naming: the approximation holds on a compact set, fixed before the units are chosen. It promises nothing about x outside the region the argument was run on, which is the formal version of the usual warning not to extrapolate.

^a<https://morrison-lab.github.io/mds/optimization.html#def-gradient-descent>

^b<https://morrison-lab.github.io/mds/sets.html#def-ordered-pair>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-prediction-function-family>

i Watch

Weinberger discusses universal approximation, from 27:05^a [19], Lecture 35.

^a<https://www.youtube.com/watch?v=kPXxbmBsFxs&t=1625s>

i Watch

Hutchinson covers the representational power of networks^a (23 min) [2].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#representational_power

i Read

GeeksforGeeks explains the universal approximation theorem^a [20].

^a<https://www.geeksforgeeks.org/deep-learning/universal-approximation-theorem-for-neural-networks/>

7 Depth and width

Exercise 6.1 part 3 already needed a second hidden layer to build a box. Nothing forces a network to stop at two. A deep network stacks several hidden layers, each taking the previous layer's output as its input, and the back-propagation chapter writes out the forward rule for K of them. Two words describe the result, and they mean different quantities.

Definition 7.1 (Depth). The **depth** of a network is its number of layers of weights: $K + 1$ for a network with K hidden layers, counted as in the counting-layers note of Section 3. A network is **deep** when it has more than one hidden layer.

Definition 7.2 (Width). The **width** of a layer is its number of units. The hidden layer of Equation 1 has width L .

Example 7.1 (Two networks for fuel economy with the same number of parameters). Predict mpg in the Auto data [5] from the 6 other numeric columns: `cylinders`, `displacement`, `horsepower`, `weight`, `acceleration` and `year`. Compare a network with two hidden layers, of widths 8 and 4, with a network with one hidden layer. List each network's layer sizes, read off its depth and widths, and count its weights and biases:

7.1 Python

```
# Layer sizes from the input to the output: 6 inputs, hidden layers, 1 output.
deep_dw = [6, 8, 4, 1]
wide_dw = [6, 12, 1]

def n_params_dw(sizes):
    """Each layer has a weight matrix (units x inputs) and one bias per unit."""
    return sum(n_out * n_in + n_out for n_in, n_out in zip(sizes[:-1], sizes[1:]))

for name, sizes in [("deep", deep_dw), ("wide", wide_dw)]:
    # K hidden layers means K + 1 layers of weights, which is the depth.
    print(name, "depth", len(sizes) - 1, "widths", sizes[1:-1], "parameters", n_params_dw(sizes))

deep depth 3 widths [8, 4] parameters 97
```

wide depth 2 widths [12] parameters 97

7.2 R

```
# Layer sizes from the input to the output: 6 inputs, hidden layers, 1 output.
deep_dw <- c(6, 8, 4, 1)
wide_dw <- c(6, 12, 1)

# Each layer has a weight matrix (units x inputs) and one bias per unit.
n_params_dw <- function(sizes) {
  n_in <- head(sizes, -1)
  n_out <- tail(sizes, -1)
  sum(n_out * n_in + n_out)
}

# K hidden layers means K + 1 layers of weights, which is the depth.
data.frame(
  network = c("deep", "wide"),
  depth = c(length(deep_dw) - 1, length(wide_dw) - 1),
  parameters = c(n_params_dw(deep_dw), n_params_dw(wide_dw))
)
```

```
# A tibble: 2 x 3
  network depth parameters
  <chr>   <dbl>     <dbl>
1 deep       3         97
2 wide       2         97
```

The first network has $K = 2$ hidden layers, so its depth is 3, and its hidden layers have widths 8 and 4. The second has $K = 1$, depth 2, and one hidden layer of width 12. Both have 97 parameters, so the choice between them is not about size: we can spend the same budget on depth or on width. Only the first is deep in the sense of the definition, because it has more than one hidden layer.

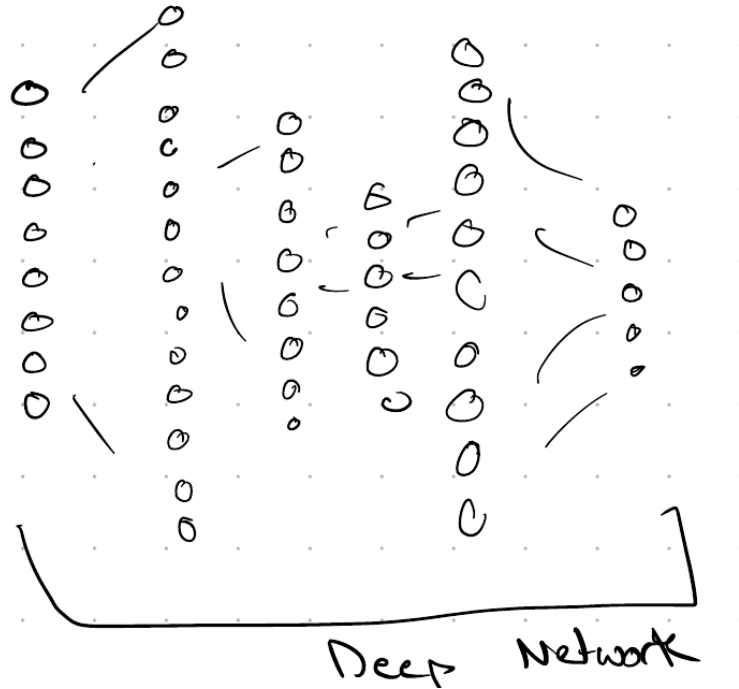


Figure 8: A deep network (Definition 7.1): several hidden layers between the input and the output, each with its own width.

Depth and width are independent settings, and the rest of this section is about why they are not interchangeable.

Nothing requires the hidden layers to match each other, and in principle every weight matrix could have its own shape and every layer its own non-linearity. In practice:

- We usually give the hidden layers all the same width, so we describe the architecture by two numbers, the depth and the common width, rather than by one width per hidden layer.
- We usually use the same f in every hidden layer.
- The task fixes the output layer's width: C units, one per class, or one unit for scalar regression.
- The task fixes the output activation g as well, exactly as it was for one hidden layer in Exercise 3.6 part 4.

So the free choices in an ordinary feedforward architecture are the depth, the common width, and the hidden non-linearity. All three are hyperparameters: nothing in the data determines them, and we choose them by the procedure of model selection like any other.

If depth only had to compete with width on what is representable, it would not be worth using: Exercise 6.1 says one hidden layer already approximates any continuous function. The argument for depth is about *efficiency* of representation rather than about what is representable at all.

The simplest version of it counts regions. A network whose non-linearity is piecewise linear — the rectified linear unit $f(a) = \max(0, a)$, say — computes a piecewise linear function, and the number of linear pieces it can divide the input space into is a reasonable measure of how complex a decision boundary it can draw. The largest number of pieces a network can produce grows **exponentially in the depth and only polynomially in the width** [21]. A deep network can therefore match a shallow one's expressiveness with far fewer parameters, and the gap widens as the functions get more complex.

Networks also differ in how their units are connected, not only in how many there are.

Definition 7.3 (Fully connected layer). A layer is **fully connected** when every one of its units reads every unit of the layer below it [22]. Every layer in this chapter is fully connected.

Example 7.2 (Every unit reads every input). A **fully connected layer** of $L = 4$ units on the $p = 6$ numeric features of the Auto data [5] has a weight matrix W with L rows and p columns. Row k holds the weights of unit k , one for every input. Build such a layer with random weights, apply it to the first car, and check that changing any one input changes every unit's output:

7.3 Python

```
import numpy as np
import pandas as pd

cols_fc = ["cylinders", "displacement", "horsepower", "weight", "acceleration", "year"]
auto_fc = pd.read_csv("data/auto.csv").dropna(subset=cols_fc)
x_fc = auto_fc[cols_fc].iloc[0].to_numpy(float)
x_fc = (x_fc - auto_fc[cols_fc].mean().to_numpy()) / auto_fc[cols_fc].std().to_numpy()

rng_fc = np.random.default_rng(0)
W_fc = rng_fc.normal(size=(4, 6))      # one row per unit, one column per input
b_fc = np.zeros(4)
out_fc = np.tanh(W_fc @ x_fc + b_fc)

# Nudge each input in turn: how many of the 4 outputs move?
moved_fc = []
for j in range(6):
    x_nudged = x_fc.copy()
    x_nudged[j] += 0.1
    moved_fc.append(int(np.sum(np.tanh(W_fc @ x_nudged + b_fc) != out_fc)))
print(W_fc.shape, W_fc.size, moved_fc)

(4, 6) 24 [4, 4, 4, 4, 4, 4]
```

7.4 R

```
cols_fc <- c("cylinders", "displacement", "horsepower", "weight", "acceleration", "year")
auto_fc <- na.omit(read.csv("data/auto.csv")[, cols_fc])
x_fc <- as.numeric(scale(auto_fc)[1, ])

set.seed(0)
W_fc <- matrix(rnorm(4 * 6), nrow = 4, ncol = 6)      # one row per unit, one column per input
b_fc <- rep(0, 4)
out_fc <- tanh(W_fc %*% x_fc + b_fc)

# Nudge each input in turn: how many of the 4 outputs move?
moved_fc <- sapply(1:6, function(j) {
  x_nudged <- x_fc
  x_nudged[j] <- x_nudged[j] + 0.1
  sum(tanh(W_fc %*% x_nudged + b_fc) != out_fc)
})
list(shape = dim(W_fc), n_weights = length(W_fc), outputs_moved = moved_fc)

$shape
[1] 4 6
```

```
$n_weights
```

```
[1] 24
```

```
$outputs_moved
```

```
[1] 4 4 4 4 4 4
```

The layer has $4 \times 6 = 24$ weights, one for every pair of an input and a unit. Each nudge moves all 4 of the layer's outputs, because every unit reads every input. A layer that connected each unit to only a few neighbouring inputs, as convolutional layers do, would have zeros in most of W , and a nudge would reach only the units that read that input.

Full connection is one choice among several, and each choice encodes an assumption about the data. A convolutional layer has each unit read only a small neighbourhood and share its weights across positions, which assumes that what counts as a useful local pattern does not depend on where it occurs. A recurrent layer feeds a unit's own output back as an input at the next step, which assumes the inputs form a sequence whose earlier terms matter. And a deeper stack of narrow layers and a shallower stack of wide ones can hold the same number of weights and still learn quite different functions. Convolutional and Recurrent Networks builds the first two kinds of layer.

Watch

Weinberger compares wide and deep networks, from 38:20^a [19], Lecture 35.

^a<https://www.youtube.com/watch?v=kPXxbmBsFxs&t=2300s>

Watch

Hutchinson covers deep networks^a (21 min) [2].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#dnns

Watch

Starmer explains:

- convolutional networks^a [23]
- recurrent networks^b [24]

^a<https://www.youtube.com/watch?v=HGwBXDKFk9I>

^b<https://www.youtube.com/watch?v=AsNTP8Kwu80>

8 Choosing an objective

Everything so far has been about what a network can represent. Fitting one needs a loss, and the loss is not a free choice: it is a statement about the distribution of y given x , whether or not you state it.

Exercise 8.1 (Choosing the loss, and why it is not a free choice). Exercise 3.6 left the output activation g to be chosen by the task. We have to choose the loss too, and the two choices are not independent. For regression the usual loss is squared error, and for classification it is cross-entropy:

$$L(y, \hat{y}) = (y - \hat{y})^2, \quad L(y, h(x)) = - \sum_{c=1}^C y_c \log h_c(x),$$

where $\hat{y} = h(x)$ is the fitted value.

Neither is an arbitrary choice. For squared error, name a distribution^a for y given x whose log-likelihood^b is the negative of that loss, up to a constant^c and a positive factor, and carry out the derivation. Linear regression states this one without deriving it; the point here is to say what it *assumes*.

^a<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

^b<https://morrison-lab.github.io/sds/intro-MLEs.html#def-loglik>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 8.1. Squared error is the Gaussian assumption. Take $y \mid x \sim \mathcal{N}(h(x), \sigma^2)$ with σ^2 fixed. Over independent training points the log-likelihood is

$$\ell = \sum_{i=1}^n \left[-\frac{1}{2} \log(2\pi\sigma^2) - \frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right],$$

where $\hat{y}_i = h(x_i)$ is the fitted value. The only parameter-dependent part is $-\frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - \hat{y}_i)^2$. Maximizing ℓ is minimizing the sum of squared errors.

Exercise 8.2 (Choosing the loss, and why it is not a free choice (part 2)). Use the setup of Exercise 8.1. For cross-entropy^a, name a distribution^b for y given x whose log-likelihood^c is the negative of that loss, up to a constant^d and a positive factor, and carry out the derivation. Logistic regression derives this one; the point here is to say what it *assumes*.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

^b<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

^c<https://morrison-lab.github.io/sds/intro-MLEs.html#def-loglik>

^d<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 8.2. Cross-entropy^a is the categorical^b assumption. Take y one-hot and model $P(y_c = 1 \mid x) = h_c(x)$, so that $P(y \mid x) = \prod_c h_c(x)^{y_c}$. Then

$$\ell = \sum_{i=1}^n \sum_{c=1}^C y_{ic} \log h_c(x_i),$$

the negative of the summed cross-entropy.

So the two losses are the same procedure — maximum likelihood^c — under two different assumptions about what kind of thing y is. Writing down a loss is writing down a noise model, whether or not you state it. The Gaussian assumption also makes squared error a poor fit for classification before any gradient argument: a label in $\{0, 1\}$ is not a continuous quantity with symmetric^d additive noise, and the model is wrong about the response in an obvious way.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

^b<https://morrison-lab.github.io/sds/data.html#def-categorical-var>

^c<https://morrison-lab.github.io/sds/intro-MLEs.html#def-mle>

^d<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-symmetric-matrix>

Exercise 8.3 (Choosing the loss, and why it is not a free choice (part 3)). Use the setup of Exercise 8.1. Nothing stops you writing squared error for a classification problem. Take one binary output $h = \text{expit}(z)$ and a label $y \in \{0, 1\}$. Differentiate both losses with respect to the pre-activation z .

Solution

Solution 8.3. With $h = \text{expit}(z)$:

$$\frac{\partial}{\partial z} (y - \text{expit}(z))^2 = -2(y - \text{expit}(z)) \text{expit}(z)(1 - \text{expit}(z)),$$

using $\text{expit}'(z) = \text{expit}(z)(1 - \text{expit}(z))$, while for binary cross-entropy^a $L = -y \log \text{expit}(z) - (1 - y) \log(1 - \text{expit}(z))$ the expit' cancels against the $1/\text{expit}$ from the logarithm and leaves

$$\frac{\partial L}{\partial z} = \text{expit}(z) - y.$$

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Exercise 8.4 (Choosing the loss, and why it is not a free choice (part 4)). Use the setup of Exercise 8.1. Evaluate both derivatives at $y = 1$ and $z = -10$ — the model is confidently wrong.

Solution

Solution 8.4. At $y = 1$ and $z = -10$, the sigmoid^a gives $\text{expit}(z) = 0.000045$:

loss

^a<https://morrison-lab.github.io/mds/functions.html#def-log-sigmoid-function>

squared error	−0.000091
cross-entropy ^b	−0.999955

Exercise 8.5 (Choosing the loss, and why it is not a free choice (part 5)). Use the setup of Exercise 8.1. Which loss still supplies a gradient?

Solution

Solution 8.5. Cross-entropy^a still supplies a gradient of essentially full size, and squared error supplies almost none. The reason is the $\text{expit}'(z)$ factor. Being confidently wrong makes $\text{expit}(z)$ near 0, so $\text{expit}'(z)$ is near 0 too, and squared error multiplies the error by it. Cross-entropy's logarithm cancels that factor exactly, and this cancellation makes the pairing standard.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Exercise 8.6 (Choosing the loss, and why it is not a free choice (part 6)). Use the setup of Exercise 8.1. By what factor do they differ?

Solution

Solution 8.6. Squared error supplies a gradient about 11,000 times smaller than cross-entropy^as.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Exercise 8.7 (Choosing the loss, and why it is not a free choice (part 7)). Use the setup of Exercise 8.1. Plot, or tabulate, the squared-error derivative as z runs from -10 to $+5$.

Solution

Solution 8.7. Sweeping z at $y = 1$:

z	$\text{expit}(z)$	squared error	cross-entropy ^a
-10	0.000045	-0.000091	-0.999955
-5	0.006693	-0.013207	-0.993307
-2	0.119203	-0.184956	-0.880797
0	0.500000	-0.250000	-0.500000
2	0.880797	-0.025031	-0.119203
5	0.993307	-0.000089	-0.006693

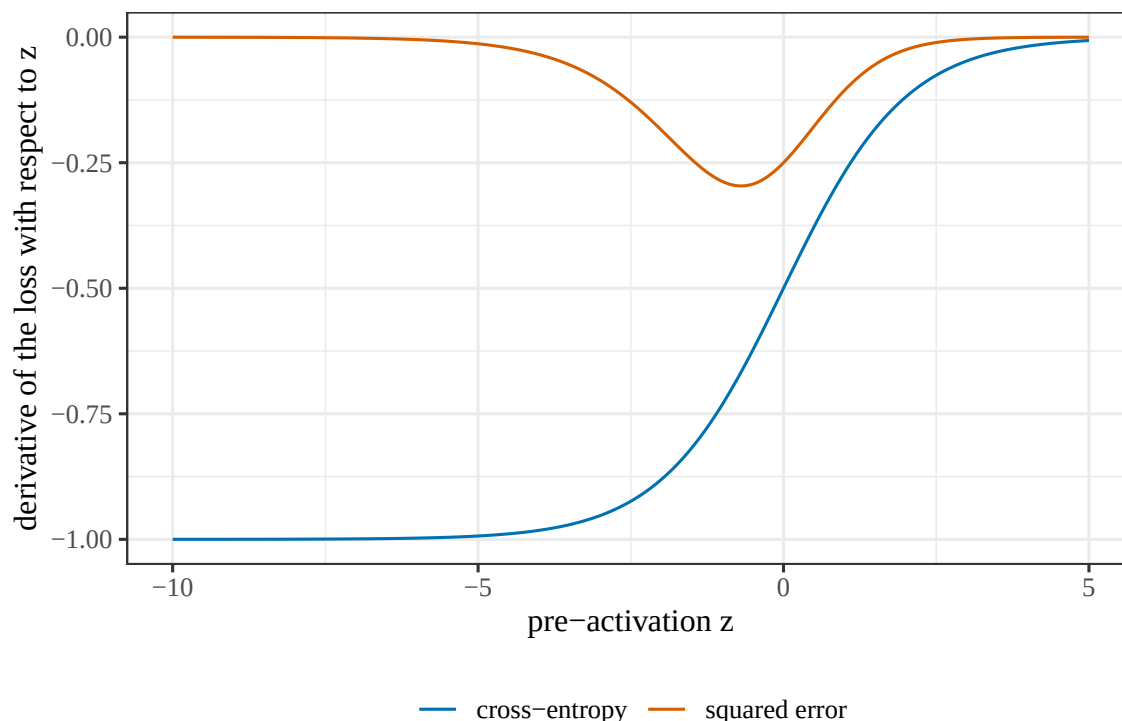


Figure 9: The derivative of the squared error and of the cross-entropy^b with respect to z , for a label of 1. The squared-error derivative is nearly 0 where the fitted value $\text{expit}(z)$ is most wrong, so a gradient step barely moves a unit that is confidently wrong.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

^b<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Exercise 8.8 (Choosing the loss, and why it is not a free choice (part 8)). Use the setup of Exercise 8.1. Where is it largest?

Solution

Solution 8.8. The squared-error derivative is largest in magnitude at $z = 0$, where the model is maximally *uncertain*, and approaches zero in both directions.

Exercise 8.9 (Choosing the loss, and why it is not a free choice (part 9)). Use the setup of Exercise 8.1. Say why that shape is the wrong one for learning.

Solution

Solution 8.9. So it pushes hardest on the points the model is least sure about, and barely at all on the points it has got badly wrong — the wrong way round. Cross-entropy^a's derivative is monotone in z : the more wrong the prediction, the larger the gradient, which is the shape you want.

Note that squared error's derivative is also small at $z = +5$, where the model is confidently *right*. That part is fine and both losses do it. The defect is specifically the left tail.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Exercise 8.10 (Choosing the loss, and why it is not a free choice (part 10)). Use the setup of Exercise 8.1. Now do Exercise 8.2 for the matched combinations: identity output with squared error, sigmoid^a output with binary cross-entropy^b, and softmax output with cross-entropy. Write $\partial L/\partial z_c$ in each case.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^b<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Solution

Solution 8.10.

output g	loss	$\partial L/\partial z_c$
identity	$\frac{1}{2}(y - h)^2$	$h - y$
sigmoid ^a	binary cross-entropy ^b	$\text{expit}(z) - y$
softmax	cross-entropy	$h_c - y_c$

Exercise 8.11 (Choosing the loss, and why it is not a free choice (part 11)). Use the setup of Exercise 8.1. What do the three answers have in common?

Solution

Solution 8.11. All three matched combinations give the same thing.

In every case the derivative with respect to the output pre-activation is prediction minus target. We checked the softmax case numerically at $z = (2, -1, 0.5)$ with the target in class 2: central differences give $(+0.785597, -0.960887, +0.175290)$, and $h - y$ gives the same three numbers.

Exercise 8.12 (Choosing the loss, and why it is not a free choice (part 12)). Use the setup of Exercise 8.1. What advantage does that give you when you implement backpropagation?

Solution

Solution 8.12. That is the benefit of the pairing. Backpropagation starts from $\partial L/\partial z$ at the output and pushes it backward through the layers, so this is the quantity it needs first, and the matched pair makes it a subtraction rather than a product of an error with a derivative that may be near zero. The same cancellation explains Solution 8.3: the two facts are one fact.

Mismatching the pair — softmax with squared error, say — is not an error the code will catch. It runs, and it trains slowly for the reason Solution 8.4 gives.

Exercise 8.13 (Choosing the loss, and why it is not a free choice (part 13)). Use the setup of Exercise 8.1. A network with $g = \text{softmax}$ and cross-entropy loss, trained on data it fits perfectly, drives the loss toward 0. What happens to $\|U\|$ meanwhile?

Solution

Solution 8.13. $\|U\|$ grows without bound. Cross-entropy^a reaches 0 only in the limit $h_c \rightarrow 1$ on every training point, which needs the correct class's pre-activation to exceed the others by an unbounded margin. The direction of U settles while its scale grows without bound.

^a<https://morrison-lab.github.io/pds/expectation.html#def-cross-entropy>

Exercise 8.14 (Choosing the loss, and why it is not a free choice (part 14)). Use the setup of Exercise 8.1. Which earlier exercise is this?

Solution

Solution 8.14. This is the separation problem, since Exercise 3.13 showed the output layer *is* multinomial logistic regression with $B = U$. The remedy carries over unchanged: penalize $\|U\|^2$, or stop early. A network adds a second place for it to happen. W is fit by the same descent on the same objective, so nothing stops its scale from growing without bound too, and Exercise 3.24's observation that a hidden layer is never inert has a downside here as well.

i Watch

Starmer explains cross-entropy^a [25] as a network's loss.

^a<https://www.youtube.com/watch?v=6ArSys5qHAU>

i Read

GeeksforGeeks explains loss functions in deep learning^a [26].

^a<https://www.geeksforgeeks.org/deep-learning/loss-functions-in-deep-learning/>

9 Regularizing a network

Exercise 9.1 (A network that fits the training set too well). A fully connected network reaches near-zero training loss, but its validation loss is much higher and rises as training continues. List the ways covered in these notes to reduce the gap without changing the data collected, and say what each one does to the fit.

Solution

Solution 9.1. A network has so many parameters that it can fit noise, so each remedy limits how closely it can. The notes develop five, in four different places:

- **Weight decay** shrinks the weights at every step, which is an ℓ_2 penalty under plain gradient descent. See Weight decay.
- **Early stopping** halts training when the validation loss stops improving, so the parameters have less time to move away from their initial values. See Regularizing without a penalty.
- **Data augmentation** trains on label-preserving transformations of the examples, which tells the network what it should ignore. See Regularizing without a penalty.
- **Dropout** zeroes random units during training, so no unit can rely on a particular other unit, and approximates averaging an ensemble of subnetworks. See Ensembling neural networks.
- **Smaller networks**: fewer units or layers limit what the network can represent. See [Depth and width](#).

The strength of each is a hyperparameter, so we tune each on the validation set, as in Model Selection.

10 Where this goes next

Two topics remain.

Fitting. We find W and U by gradient descent, as in the gradient descent chapter, with one complication: a network is a composition, so the derivative with respect to W has to pass back through f . Doing that efficiently is **back-propagation**, the subject of the next chapter.

i Watch

- Hastie and Tibshirani explain how neural networks are fitted^a [8], video 10.6.
- Their Python lab^b fits a multilayer network to handwritten digits.

^a<https://www.youtube.com/watch?v=07zslA8BXZY>

^b<https://www.youtube.com/watch?v=csmHZGGeONU>

Depth. Section 7 gave the case for depth in representation. Depth is cheap to *write* — the next chapter shows the gradient recursion is the same at every layer — but costly to *train*. Gradient Descent covers much of what makes deep networks train well.

11 Further reading

- Goodfellow et al. [27], Chapter 6
- James et al. [5], Chapter 10
- Hastie et al. [28], Chapter 11
- Starmer [29], on neural networks, and Starmer [30], which builds from a single network up to transformers, with PyTorch tutorials

i Watch

The Data Science Learning Community’s book clubs discuss *An Introduction to Statistical Learning*, Chapter 10, in recorded meetings:

- the R book club, in part 1^a and part 2^b [31];
- the Python book club, in part 1^c and part 2^d [32].

^a<https://www.youtube.com/watch?v=FRcGhmyzxY8&list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

^b<https://www.youtube.com/watch?v=iqJdAlANjio&list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

^c<https://www.youtube.com/watch?v=ZA99FHf08DQ&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

^d<https://www.youtube.com/watch?v=f19nIgi0Cgg&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

i Watch

Andrew Ng’s lectures for DeepLearning.AI introduce neural networks: what a neural network is [33]; how its layers are written down [34]; activation functions [35]; why they must be non-linear [36]; and why depth helps [37].

References

- [1] Wikipedia contributors, “Perceptron — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/Perceptron>
- [2] B. Hutchinson, “DATA 471/571 (machine learning) and CSCI 481/581 (deep learning) video lectures.” Western Washington University. Accessed: Sep. 28, 2026. [Online]. Available: https://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/
- [3] GeeksforGeeks, “What is perceptron.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/what-is-perceptron-the-simplest-artificial-neural-network/>

- [4] GeeksforGeeks, “Activation functions in neural networks.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/activation-functions-neural-networks/>
- [5] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning with applications in R*, 2nd ed. Springer, 2021. Available: <https://www.statlearning.com/>
- [6] Wikipedia contributors, “Autoencoder — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/Autoencoder>
- [7] GeeksforGeeks, “Autoencoders in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/auto-encoders/>
- [8] T. Hastie, R. Tibshirani, and J. Taylor, “Statistical learning with python.” Stanford Online, lecture videos on YouTube, 2023. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ>
- [9] J. Starmer, “The essential main ideas of neural networks.” StatQuest, video on YouTube, 2020. Available: <https://www.youtube.com/watch?v=CqOfi41LfDw>
- [10] A. Ng *et al.*, “CS229: Machine learning, autumn 2018.” Stanford Online, lecture videos on YouTube, 2018. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU>
- [11] Y. S. Abu-Mostafa, “Learning from data: CS 156 lectures.” Caltech, lecture videos on YouTube, 2012. Available: <https://www.youtube.com/playlist?list=PLD63A284B7615313A>
- [12] GeeksforGeeks, “Introduction to neural networks.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/neural-networks-a-beginners-guide/>
- [13] GeeksforGeeks, “Feedforward neural network.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/feedforward-neural-network/>
- [14] N. Jaques, “CSE 446/546: Machine learning, lecture slides.” University of Washington, 2026. Accessed: Oct. 12, 2026. [Online]. Available: <https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/>
- [15] G. Sanderson, “Neural networks.” 3Blue1Brown, video series on YouTube; chapters 5 and 6 added in 2024, 2017. Available: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- [16] J. Starmer, “Neural networks pt. 3: ReLU in action.” StatQuest, video on YouTube, 2020. Available: <https://www.youtube.com/watch?v=68BZ5f7P94E>
- [17] GeeksforGeeks, “How neural networks solve the XOR problem.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/artificial-intelligence/how-neural-networks-solve-the-xor-problem/>
- [18] J. Starmer, “Neural networks pt. 4: Multiple inputs and outputs.” StatQuest, video on YouTube, 2021. Available: <https://www.youtube.com/watch?v=83LYR-IIcjA>
- [19] K. Weinberger, “CS4780: Machine learning for intelligent systems, spring 2017.” Cornell University, lecture videos on YouTube, 2017. Available: <https://www.youtube.com/playlist?list=PLl8OIHZGYOQ7bkVbuRthEsaLr7bONzbXS>
- [20] GeeksforGeeks, “Universal approximation theorem for neural networks.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/deep-learning/universal-approximation-theorem-for-neural-networks/>
- [21] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, “On the number of linear regions of deep neural networks,” in *Advances in neural information processing systems*, 2014. Available: <https://arxiv.org/abs/1402.1869>
- [22] Wikipedia contributors, “Multilayer perceptron — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Multilayer_perceptron
- [23] J. Starmer, “Neural networks part 8: Image classification with convolutional neural networks (CNNs).” StatQuest, video on YouTube, 2021. Available: <https://www.youtube.com/watch?v=HGwBXDKFk9I>
- [24] J. Starmer, “Recurrent neural networks (RNNs), clearly explained.” StatQuest, video on YouTube, 2022. Available: <https://www.youtube.com/watch?v=AsNTP8Kwu80>
- [25] J. Starmer, “Neural networks part 6: Cross entropy.” StatQuest, video on YouTube, 2021. Available: <https://www.youtube.com/watch?v=6ArSys5qHAU>
- [26] GeeksforGeeks, “Loss functions in deep learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/loss-functions-in-deep-learning/>

- [27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. Available: <https://www.deeplearningbook.org/>
- [28] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, 2nd ed. Springer, 2009. Available: <https://hastie.su.domains/ElemStatLearn/>
- [29] J. Starmer, *The StatQuest illustrated guide to machine learning*. StatQuest Publications, 2022. Available: <https://statquest.org/statquest-store/>
- [30] J. Starmer, *The StatQuest illustrated guide to neural networks and AI: With hands-on examples in PyTorch*. Independently published, 2025. Available: <https://statquest.org/statquest-store/>
- [31] Data Science Learning Community, “Introduction to statistical learning using R (islr06).” Book club meeting recordings on YouTube, 2023. Available: <https://www.youtube.com/playlist?list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>
- [32] Data Science Learning Community, “Introduction to statistical learning using Python (islp03).” Book club meeting recordings on YouTube, 2025. Available: <https://www.youtube.com/playlist?list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>
- [33] A. Ng, “What is a neural network?” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=n1l-9IIMW7E>
- [34] A. Ng, “Neural network representations.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=CcRkHl75Z-Y>
- [35] A. Ng, “Activation functions.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=Xvg00QnyaIY>
- [36] A. Ng, “Why non-linear activation functions.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=NkOv_k7r6no
- [37] A. Ng, “Why deep representations?” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=5dWp1mw_XNk