

Convolutional and Recurrent Networks

Last modified: 2026-10-25

i Source

Written for these notes, following the treatment of convolutional and recurrent networks in Goodfellow et al. [1], Chapters 9 and 10, and James et al. [2], Chapter 10. The output-size formula follows Dumoulin and Visin [3]. The handwritten digits are the test set of the `optdigits` data of Alpaydin and Kaynak [4], and the series is the `Smarket` data of James et al. [5]. The chapter builds on Neural Networks and Backpropagation, and the sequence half ends where Attention and Transformers begins.

Neural Networks built layers that take a vector and connect every input to every unit. That choice treats an image as a list of unrelated numbers, and a time series as a vector of fixed length. Two layers drop that assumption. A **convolutional layer** reads a small patch of an image and reuses its weights at every position. A **recurrent layer** reads a sequence one step at a time and carries a state from step to step. Both reuse the same weights at many places, and the chapter's questions follow from that: how large each output is, how many parameters it saves, and how gradients flow through the reuse.

1 Images as grids and as vectors

The images in this chapter are handwritten digits. A model from the earlier chapters would flatten each into a vector, and a convolutional network would keep it as a grid. We start by moving between the two.

Exercise 1.1 (A digit as a grid and as a vector). The `optdigits` data hold handwritten digits prepared by Alpaydin and Kaynak [4]. Each digit was a 32×32 bitmap, reduced to an 8×8 grid by counting the inked pixels in each 4×4 block, so every grid entry is an integer from 0 to 16. The UCI Machine Learning Repository distributes them under a CC BY 4.0 licence. The file `data/optdigits-test.csv`^a is a copy of the repository's test file `optdigits.tes`. It has no header row: each line holds the 64 grid entries of one digit, row by row, followed by the digit it shows. Load the file and report how many digits it holds.

^a<https://morrisson-lab.github.io/lds/chapters/data/optdigits-test.csv>

Solution

Solution 1.1.

1.1 Python

```
import numpy as np
import pandas as pd

digits = pd.read_csv("data/optdigits-test.csv", header=None).to_numpy()
X_dig = digits[:, :64]      # one row per digit, 64 entries
y_dig = digits[:, 64]      # the digit each row shows
print(X_dig.shape)

(1797, 64)
```

1.2 R

```
digits_r <- as.matrix(read.csv("data/optdigits-test.csv", header = FALSE))
X_dig_r <- digits_r[, 1:64]  # one row per digit, 64 entries
y_dig_r <- digits_r[, 65]   # the digit each row shows
dim(X_dig_r)

[1] 1797 64
The file holds 1797 digits.
```

Exercise 1.2 (A digit as a grid and as a vector (part 2)). Use the setup of Exercise 1.1. Print the second digit twice: as an 8×8 grid, and as the vector of 64 numbers the file stores.

Solution

Solution 1.2.

1.3 Python

```
print(X_dig[1].reshape(8, 8)) # the same 64 numbers as an 8 x 8 grid

[[ 0  0  0 12 13  5  0  0]
 [ 0  0  0 11 16  9  0  0]
 [ 0  0  3 15 16  6  0  0]
 [ 0  7 15 16 16  2  0  0]
 [ 0  0  1 16 16  3  0  0]
 [ 0  0  1 16 16  6  0  0]
 [ 0  0  1 16 16  6  0  0]
 [ 0  0  0 11 16 10  0  0]]

print(X_dig[1])

[ 0  0  0 12 13  5  0  0  0  0  0 11 16  9  0  0  0  0  3 15 16  6  0  0
  0  7 15 16 16  2  0  0  0  0  1 16 16  3  0  0  0  0  1 16 16  6  0  0
  0  0  1 16 16  6  0  0  0  0  0 11 16 10  0  0]
```

1.4 R

```
# matrix() fills column by column unless told otherwise, so ask for row by row.
matrix(X_dig_r[2, ], nrow = 8, byrow = TRUE)

[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]  0   0   0  12  13   5   0   0
[2,]  0   0   0  11  16   9   0   0
[3,]  0   0   3  15  16   6   0   0
[4,]  0   7  15  16  16   2   0   0
[5,]  0   0   1  16  16   3   0   0
```

```
[6,] 0 0 1 16 16 6 0 0
[7,] 0 0 1 16 16 6 0 0
[8,] 0 0 0 11 16 10 0 0
```

```
X_dig_r[2, ]
```

```
V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
0 0 0 12 13 5 0 0 0 0 0 11 16 9 0 0 0 0 3 15
V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40
16 6 0 0 0 7 15 16 16 2 0 0 0 0 1 16 16 3 0 0
V41 V42 V43 V44 V45 V46 V47 V48 V49 V50 V51 V52 V53 V54 V55 V56 V57 V58 V59 V60
0 0 1 16 16 6 0 0 0 0 1 16 16 6 0 0 0 0 0 11
V61 V62 V63 V64
16 10 0 0
```

The largest grid entry is 16, and the ink forms a vertical stroke.

Exercise 1.3 (A digit as a grid and as a vector (part 3)). Use the setup of Exercise 1.1. Which digit is the second one?

Solution

Solution 1.3.

1.5 Python

```
print(y_dig[1])
```

```
1
```

1.6 R

```
y_dig_r[2]
```

```
[1] 1
```

The second digit is a 1.

Exercise 1.4 (Where does a pixel go?). Count rows and columns of the grid from 1, and entries of the vector from 1. Which entry of the length-64 vector holds the pixel in row 4, column 3 of the grid?

Solution

Solution 1.4. The file stores the grid row by row. The first three rows come first and fill $8 \times 3 = 24$ entries, and the pixel is the third entry of row 4, so it sits at entry $24 + 3 = 27$.

Exercise 1.5 (Where does a pixel go? (part 2)). Use the setup of Exercise 1.4. Give a formula for the entry that holds row i , column j of an $H \times W$ grid.

Solution

Solution 1.5. Rows $1, \dots, i - 1$ come first and fill $W(i - 1)$ entries, and the pixel is the j th entry of row i , so it sits at entry $W(i - 1) + j$.

Exercise 1.6 (Where does a pixel go? (part 3)). Use the setup of Exercise 1.4. Check your formula on the second digit.

Solution

Solution 1.6.

1.7 Python

```
grid_py = X_dig[1].reshape(8, 8)
# Python counts from 0, so subtract 1 from the row, column and result.
print(grid_py[4 - 1, 3 - 1], X_dig[1][8 * (4 - 1) + 3 - 1])
```

15 15

1.8 R

```
grid_r <- matrix(X_dig_r[2, ], nrow = 8, byrow = TRUE)
c(grid_r[4, 3], X_dig_r[2, 8 * (4 - 1) + 3])
```

V27

15 15

Both routes return the same pixel, 15.

Definition 1.1 (Flattening (vectorizing, unrolling)). **Flattening** an $H \times W$ grid X gives the vector $x \in \mathbb{R}^p$ with $p = HW$ whose entry $W(i - 1) + j$ is X_{ij} . The grid and the vector hold the same numbers. Flattening is also called **vectorizing** or **unrolling** the grid. This row-by-row order is the one NumPy uses by default; R fills a matrix column by column, so `byrow = TRUE` is needed to match it. Models in the chapters on linear regression and logistic regression take $\tilde{x}$ as their input. Convolutional networks keep the grid X , and the entries of X near each other in the grid are the pixels a filter reads together.

Example 1.1 (Neighbours in the grid are not neighbours in the vector). In the second digit, the pixel at row 3, column 4 and the pixel directly below it, at row 4, column 4, are vertical neighbours in the grid. Find their entries in the flattened vector.

1.9 Python

```
W_dig = 8
idx_py = [W_dig * (i - 1) + j for i, j in [(3, 4), (4, 4)]]
print(idx_py, idx_py[1] - idx_py[0])
```

[20, 28] 8

1.10 R

```
idx_r <- sapply(list(c(3, 4), c(4, 4)), function(ij) 8 * (ij[1] - 1) + ij[2])
c(idx_r, diff(idx_r))
```

[1] 20 28 8

The two pixels sit 8 entries apart, a whole row of the grid. A pixel's horizontal neighbour is one entry away, but its vertical neighbour is W entries away, and the vector does not record which pairs of entries were neighbours.

Exercise 1.7 (Shuffling the pixels). Draw one random permutation of the 64 pixel positions, and apply it to every digit in the same way. A fully connected layer with weight matrix W reads the shuffled vector instead of the original.

Can some other weight matrix W' reproduce the original layer's outputs from the shuffled vectors?

Solution

Solution 1.7. Yes: apply the same permutation to the columns of W . The product $W\tilde{x}$ adds up $W_{kj}x_j$ over j , and the sum does not depend on the order of its terms, so $W'\tilde{x}' = W\tilde{x}$ whenever the columns of W' and the entries of $\tilde{x}'$ are permuted together.

Exercise 1.8 (Shuffling the pixels (part 2)). Use the setup of Exercise 1.7.

What does this say about how a fully connected layer uses the neighbourhood structure of the grid?

Solution

Solution 1.8. A fully connected layer gives each pixel its own weight, and so it fits shuffled images as well as it fits the originals. It cannot tell that entries 27 and 35 are vertical neighbours in the grid while entries 27 and 60 are far apart, so any use of the grid has to be learned from data.

Exercise 1.9 (Shuffling the pixels (part 3)). Use the setup of Exercise 1.7.

Check your answer to Exercise 1.7 on the first digit.

Solution

Solution 1.9.

1.11 Python

```
rng_sh = np.random.default_rng(0)
perm_py = rng_sh.permutation(64)
W_sh = rng_sh.normal(size=(5, 64))          # 5 units, one weight per pixel
x_sh = X_dig[0].astype(float)
same_py = np.allclose(W_sh @ x_sh, W_sh[:, perm_py] @ x_sh[perm_py])
print(same_py)
```

True

1.12 R

```
set.seed(0)
perm_r <- sample(64)
W_sh_r <- matrix(rnorm(5 * 64), nrow = 5) # 5 units, one weight per pixel
x_sh_r <- X_dig_r[1, ]
# Column k of the new matrix goes with entry k of the shuffled vector.
same_r <- all.equal(drop(W_sh_r %*% x_sh_r),
                     drop(W_sh_r[, perm_r] %*% x_sh_r[perm_r]))
same_r
```

[1] TRUE

Both checks return TRUE.

2 Convolution

A fully connected layer ignores the grid. A layer that respects it looks at a patch of neighbouring pixels, and asks the same question of every patch.

Exercise 2.1 (One filter on one patch). Take the second digit as an 8×8 grid X (Solution 1.4), and the 3×3 array of weights

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}.$$

Multiply K entry by entry with the 3×3 patch of X in rows 1 to 3 and columns 3 to 5, and add the nine products. Check your sum with code.

Solution

Solution 2.1.

2.1 Python

```
K_v = np.array([[ -1, 0, 1], [ -1, 0, 1], [ -1, 0, 1]])
patch_py = X_dig[1].reshape(8, 8)[0:3, 2:5] # rows 1-3, columns 3-5
print(patch_py)
```

```
[[ 0 12 13]
 [ 0 11 16]
 [ 3 15 16]]
```

```
print((K_v * patch_py).sum())
```

```
42
```

2.2 R

```
K_v_r <- matrix(c(-1, 0, 1), nrow = 3, ncol = 3, byrow = TRUE)
patch_r <- matrix(X_dig_r[2, ], nrow = 8, byrow = TRUE)[1:3, 3:5]
patch_r
```

```
      [,1] [,2] [,3]
[1,]    0   12   13
[2,]    0   11   16
[3,]    3   15   16
```

```
sum(K_v_r * patch_r)
```

```
[1] 42
```

The sum is 42.

Exercise 2.2 (One filter on one patch (part 2)). Use the setup of Exercise 2.1. Read K for what the sum is large for.

Solution

Solution 2.2. The first column of K subtracts the patch's left column, and the last column adds its right column. The middle column is ignored. The sum is large and positive where the right side of the patch is brighter than the left, and large and negative where it is darker. So K responds to a vertical edge that goes from dark to bright, left to right. In this patch the left column is 0, 0, 3 and the right

column is brighter, so the [filter](#) reports a left-to-right dark-to-bright edge.

Exercise 2.3 (Sliding the filter over the image). Repeat [Exercise 2.1](#) at every position where the 3×3 filter K fits entirely inside the 8×8 grid. Write a function that takes a grid and a filter and returns the array of sums.

Solution

Solution 2.3.

2.3 Python

```
def corr2d(X, K, stride=1, pad=0):
    """Slide K over X: at each position, multiply entry by entry and add."""
    X = np.pad(X, pad)                # pad zeros on all four sides
    kh, kw = K.shape
    n_i = (X.shape[0] - kh) // stride + 1
    n_j = (X.shape[1] - kw) // stride + 1
    Z = np.zeros((n_i, n_j))
    for i in range(n_i):
        for j in range(n_j):
            patch = X[i * stride:i * stride + kh, j * stride:j * stride + kw]
            Z[i, j] = (K * patch).sum()
    return Z
```

2.4 R

```
corr2d <- function(X, K, stride = 1, pad = 0) {
  # Slide K over X: at each position, multiply entry by entry and add.
  if (pad > 0) {
    X_pad <- matrix(0, nrow(X) + 2 * pad, ncol(X) + 2 * pad)
    X_pad[pad + seq_len(nrow(X)), pad + seq_len(ncol(X))] <- X
    X <- X_pad                # zeros on all four sides
  }
  kh <- nrow(K)
  kw <- ncol(K)
  n_i <- (nrow(X) - kh) %/% stride + 1
  n_j <- (ncol(X) - kw) %/% stride + 1
  Z <- matrix(0, n_i, n_j)
  for (i in seq_len(n_i)) {
    for (j in seq_len(n_j)) {
      rows <- (i - 1) * stride + seq_len(kh)
      cols <- (j - 1) * stride + seq_len(kw)
      Z[i, j] <- sum(K * X[rows, cols])
    }
  }
  Z
}
```

Exercise 2.4 (Sliding the filter over the image (part 2)). Use the setup of [Exercise 2.3](#). How many positions are there along each side?

Solution

Solution 2.4. Along one side, the **filter**'s left edge can start at columns 1 through $8 - 3 + 1 = 6$, so there are $6 \times 6 = 36$ positions.

Exercise 2.5 (Sliding the filter over the image (part 3)). Use the setup of Exercise 2.3. Which positions give the largest and smallest sums on the second digit?

Solution

Solution 2.5.

2.5 Python

```
img_py = X_dig[1].reshape(8, 8)
Z_v = corr2d(img_py, K_v)
print(Z_v.astype(int))

[[ 3  38  42 -18 -45 -20]
 [ 18  35  30 -25 -48 -17]
 [ 19  40  29 -36 -48 -11]
 [ 17  41  31 -37 -48 -11]
 [ 3  48  45 -33 -48 -15]
 [ 2  43  46 -21 -48 -22]]

print(np.unique(np.argwhere(Z_v == Z_v.max())[:, 1]) + 1) # columns of the largest sum
[2]
print(np.unique(np.argwhere(Z_v == Z_v.min())[:, 1]) + 1) # columns of the smallest
[5]
```

2.6 R

```
img_r <- matrix(X_dig_r[2, ], nrow = 8, byrow = TRUE)
Z_v_r <- corr2d(img_r, K_v_r)
Z_v_r

[,1] [,2] [,3] [,4] [,5] [,6]
[1,] 3  38  42 -18 -45 -20
[2,] 18  35  30 -25 -48 -17
[3,] 19  40  29 -36 -48 -11
[4,] 17  41  31 -37 -48 -11
[5,] 3  48  45 -33 -48 -15
[6,] 2  43  46 -21 -48 -22

max_col <- unique(which(Z_v_r == max(Z_v_r), arr.ind = TRUE)[, "col"]) # columns of the largest sum
min_col <- unique(which(Z_v_r == min(Z_v_r), arr.ind = TRUE)[, "col"]) # columns of the smallest
c(max_col, min_col)

[1] 2 5
```

The result is a 6×6 array. Its largest sum is 48, in column 2, and its smallest is -48, in column 5 (several rows tie for the smallest). The left edge of the stroke gives the large positive sums, and its right edge the large negative ones.

Definition 2.1 (Filter (kernel)). A **filter**, also called a **kernel**, is a $k \times k$ array K of weights [6].

Definition 2.2 (Convolution). Let X be an $H \times W$ grid and K a filter (Definition 2.1). The **convolution** of X with the filter K and bias b is the array Z with entries

$$Z_{ij} = b + \sum_{u=1}^k \sum_{v=1}^k K_{uv} X_{i+u-1, j+v-1}, \quad i = 1, \dots, H - k + 1, \quad j = 1, \dots, W - k + 1. \quad (1)$$

Definition 2.3 (Feature map (activation map, response map)). The array Z of the convolution in Equation 1 is a **feature map**, also called an **activation map** or **response map**: entry (i, j) measures how well the patch of X with top-left corner (i, j) matches K . The notes say filter and feature map.

Definition 2.4 (Convolutional layer). The weights K and the bias b of a convolution (Definition 2.2) are the parameters, learned like the weights of any other layer. A **convolutional layer** applies several filters to the same input, and usually passes each feature map (Definition 2.3) through an activation function [6].

Example 2.1 (Vertical and horizontal edges in a handwritten 1). Apply K_v from Exercise 2.1 and its transpose $K_h = K_v^\top$ to the second digit, which is a handwritten 1. K_h subtracts the row above and adds the row below, so it responds to horizontal edges. Figure 1 shows the grid and both feature maps.

2.7 Python

```
Z_h = corr2d(img_py, K_v.T)
print(Z_h.astype(int))

[[ 3  6  9  7  4  1]
 [22 27 20 -2 -7 -7]
 [-2 -1 -1 -2 -3 -3]
 [-21 -21 -14  4  4  4]
 [ 0  0  0  3  3  3]
 [-1 -6 -6 -1  4  4]]

print(Z_v.shape, Z_h.shape)

(6, 6) (6, 6)
```

2.8 R

```
Z_h_r <- corr2d(img_r, t(K_v_r))
Z_h_r

      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    3    6    9    7    4    1
[2,]   22   27   20   -2   -7   -7
[3,]   -2   -1   -1   -2   -3   -3
[4,]  -21  -21  -14    4    4    4
[5,]    0    0    0    3    3    3
[6,]   -1   -6   -6   -1    4    4

dim(Z_h_r)

[1] 6 6
```

The vertical-edge map has the largest magnitudes, up to 48, along the two long sides of the stroke. The horizontal-edge map is smaller, at most 27 in magnitude, since a straight vertical stroke has few horizontal edges. Each map is 6×6 .

```

show_grid <- function(M, main) {
  # image() draws row 1 at the bottom, so flip the rows to show the grid as printed.
  image(t(M[rev(seq_len(nrow(M)))]), col = gray.colors(64, start = 0, end = 1),
        axes = FALSE, main = main, asp = 1)
}
op <- par(mfrow = c(1, 3), mar = c(0.5, 0.5, 2, 0.5))
show_grid(img_r, "digit (8 x 8)")
show_grid(Z_v_r, "vertical-edge map (6 x 6)")
show_grid(Z_h_r, "horizontal-edge map (6 x 6)")
par(op)

```

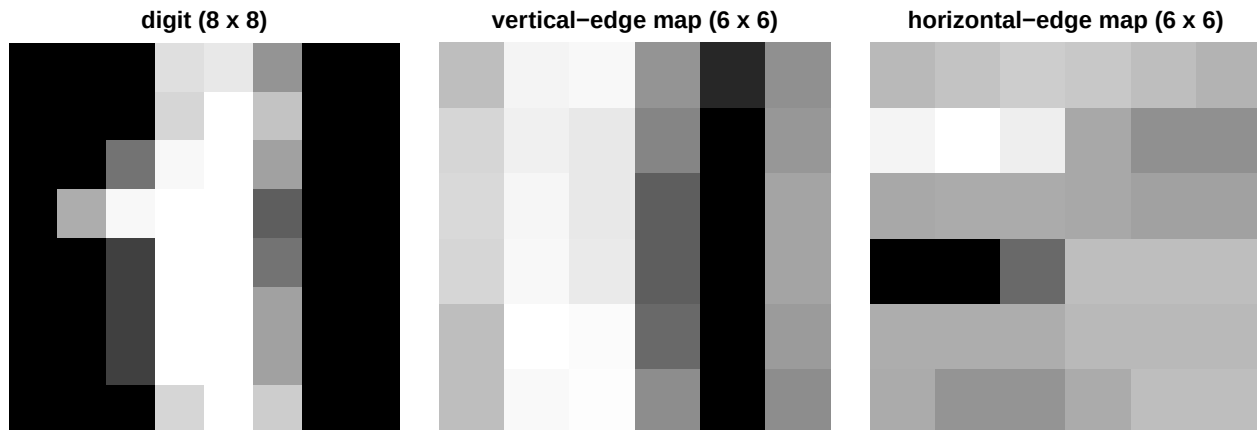


Figure 1: The second digit of the test data (a handwritten 1) and its feature maps under the vertical-edge filter and the horizontal-edge filter. Light squares are large positive sums, dark squares large negative sums.

Exercise 2.6 (Convolution or cross-correlation?). In signal processing, the convolution of X with K is

$$Z_{ij} = \sum_{u=1}^k \sum_{v=1}^k K_{uv} X_{i-u+k, j-v+k},$$

in which the filter is read backwards: as u grows, the row of X shrinks. Equation 1 reads it forwards, and signal processing calls that cross-correlation.

How do the two differ, when both are applied to the same X and K ?

Solution

Solution 2.6. Substituting $u' = k + 1 - u$ and $v' = k + 1 - v$ turns the backwards sum into $\sum_{u', v'} K_{k+1-u', k+1-v'} X_{i+u'-1, j+v'-1}$, which is Equation 1 applied to the filter flipped top to bottom and left to right. So signal-processing convolution is cross-correlation with a flipped filter.

Exercise 2.7 (Convolution or cross-correlation? (part 2)). Use the setup of Exercise 2.6. Does the difference matter when K is learned from data?

Solution

Solution 2.7. When K is learned, the flip changes nothing. Whatever filter training finds under one convention, the flipped filter computes the same feature map under the other, so the two conventions represent the same set of layers and reach the same predictions. Deep learning libraries compute Equation 1 and call it [convolution](#). These notes do the same, and write cross-correlation when the distinction matters [1, Ch. 9].

Exercise 2.8 (Convolution or cross-correlation? (part 3)). Use the setup of Exercise 2.6. Check your answer to Exercise 2.6 on the second digit with K_v .

Solution

Solution 2.8.

2.9 Python

```
def true_conv2d(X, K):
    """The backwards definition: row i-u+k of X is paired with row u of K."""
    k = K.shape[0]
    Z = np.zeros((X.shape[0] - k + 1, X.shape[1] - k + 1))
    for i in range(Z.shape[0]):
        for j in range(Z.shape[1]):
            Z[i, j] = sum(K[u, v] * X[i + k - 1 - u, j + k - 1 - v]
                          for u in range(k) for v in range(k))
    return Z

print(np.allclose(true_conv2d(img_py, K_v), corr2d(img_py, K_v[::1, ::1])))

True

print(np.allclose(true_conv2d(img_py, K_v), corr2d(img_py, K_v)))

False
```

2.10 R

```
true_conv2d <- function(X, K) {
  # The backwards definition: row i-u+k of X is paired with row u of K.
  k <- nrow(K)
  Z <- matrix(0, nrow(X) - k + 1, ncol(X) - k + 1)
  for (i in seq_len(nrow(Z))) {
    for (j in seq_len(ncol(Z))) {
      for (u in seq_len(k)) {
        for (v in seq_len(k)) {
          Z[i, j] <- Z[i, j] + K[u, v] * X[i + k - u, j + k - v]
        }
      }
    }
  }
  Z
}

flip_k <- function(K) K[rev(seq_len(nrow(K))), rev(seq_len(ncol(K)))]
isTRUE(all.equal(true_conv2d(img_r, K_v_r), corr2d(img_r, flip_k(K_v_r))))

[1] TRUE

isTRUE(all.equal(true_conv2d(img_r, K_v_r), corr2d(img_r, K_v_r)))

[1] FALSE
```

The first check is TRUE: the backwards sum equals cross-correlation with the flipped filter. The second is FALSE: with the filter unflipped the two disagree, here by a sign, since K_v flipped is $-K_v$.

i Watch

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

Video 1: Starmer explains convolutional networks for image classification [7].

3 Stride and padding

Exercise 3.1 (Changing the size of the feature map). In Solution 2.6, an 8×8 image gave a 6×6 feature map.

How could the filter visit fewer positions, to get a smaller feature map?

Solution

Solution 3.1. Move the filter more than one pixel between positions. Stepping two pixels at a time visits about a quarter of the positions of a one-pixel step.

Exercise 3.2 (Changing the size of the feature map (part 2)). Use the setup of Exercise 3.1.

How could a feature map keep the image's size, 8×8 , with a 3×3 filter?

Solution

Solution 3.2. Add a border of zeros around the image before sliding the filter. A 3×3 filter then fits at the corner pixels too, with the border covering its overhang.

Definition 3.1 (Stride). The **stride** s of a convolution is the number of pixels the filter moves between neighbouring positions, along the rows and the columns [6]. The convolution in Equation 1 has stride $s = 1$. With stride s , the filter's top-left corner visits rows $1, 1+s, 1+2s, \dots$ and columns $1, 1+s, 1+2s, \dots$

Example 3.1 (Stride 2 on the 1). Apply K_v to the second digit with stride 2.

3.1 Python

```
Z_s2 = corr2d(img_py, K_v, stride=2)
print(Z_s2.astype(int))

[[ 3  42 -45]
 [ 19 29 -48]
 [ 3  45 -48]]

print(np.array_equal(Z_s2, Z_v[:, ::2, ::2]))

True
```

3.2 R

```
Z_s2_r <- corr2d(img_r, K_v_r, stride = 2)
Z_s2_r

[,1] [,2] [,3]
[1,] 3 42 -45
[2,] 19 29 -48
```

```
[3,]    3    45   -48
```

```
identical(Z_s2_r, Z_v_r[c(1, 3, 5), c(1, 3, 5)])
```

```
[1] TRUE
```

The feature map is 3×3 . It keeps every other row and column of the stride-1 map, and so contains 9 of its 36 entries.

Definition 3.2 (Padding). **Padding** with p adds p rows of zeros above and below the image, and p columns of zeros to its left and right, before the filter slides over it [6]. The padded image is $(H + 2p) \times (W + 2p)$. Convolution with no padding ($p = 0$) is called “valid” in software; with stride 1 and $p = (k - 1)/2$ for odd k the feature map is as large as the image, and the software calls it “same”.

Example 3.2 (Padding by one pixel). Apply the 3×3 filter K_v to the second digit with $p = 1$.

3.3 Python

```
Z_p1 = corr2d(img_py, K_v, pad=1)
print(Z_p1.shape)
```

```
(8, 8)
```

```
print(Z_p1[1:-1, 1:-1].astype(int).tolist() == Z_v.astype(int).tolist())
```

```
True
```

3.4 R

```
Z_p1_r <- corr2d(img_r, K_v_r, pad = 1)
dim(Z_p1_r)
```

```
[1] 8 8
```

```
identical(Z_p1_r[2:7, 2:7], Z_v_r)
```

```
[1] TRUE
```

The feature map is 8×8 , the same size as the image. Its interior 6×6 block equals the unpadded map, and the added outer ring covers the filter positions that overhang the image.

Exercise 3.3 (The size of the feature map). An $n \times n$ image is convolved with a $k \times k$ filter, with padding p and stride s .

How many pixels wide is the padded image?

Solution

Solution 3.3. [Padding](#) adds p pixels on each side, so the padded image is $n + 2p$ pixels wide.

Exercise 3.4 (The size of the feature map (part 2)). Use the setup of Exercise 3.3.

At how many positions along one side can the [filter](#) start?

Solution

Solution 3.4. In Exercise 3.4, the [filter](#)’s first position starts at pixel 1, and its last at a pixel q with $q + k - 1 \leq n + 2p$. Positions are $1, 1 + s, \dots, 1 + ms$ for the largest integer m with $1 + ms \leq n + 2p - k + 1$, so $m = \lfloor (n + 2p - k)/s \rfloor$ and the number of positions is $m + 1$.

Exercise 3.5 (The size of the feature map (part 3)). Use the setup of Exercise 3.3. Evaluate your formula for $n = 8$, $k = 3$ with $(p, s) = (0, 1)$, $(1, 1)$ and $(0, 2)$, and compare with Example 3.1 and Example 3.2.

Solution

Solution 3.5. $(0, 1)$ gives $\lfloor 5/1 \rfloor + 1 = 6$, $(1, 1)$ gives $\lfloor 7/1 \rfloor + 1 = 8$, and $(0, 2)$ gives $\lfloor 5/2 \rfloor + 1 = 3$. These are the sizes of the maps in Solution 2.4, Example 3.2 and Example 3.1.

Theorem 3.1 (Output size of a convolution). *Convolving an $n \times n$ image with a $k \times k$ filter, padding p and stride s gives a feature map that is $n_{out} \times n_{out}$, with*

$$n_{out} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1. \quad (2)$$

For an $H \times W$ image the same formula applies to each side separately [3, Ch. 2].

i Proof

Proof. By Solution 3.3 and Solution 3.4. □

Example 3.3 (Checking the formula against the code). Compare Equation 2 with the shape of `corr2d` output on the 8×8 digit, for filter sizes $k = 2, 3, 5$, paddings 0 to 2 and strides 1 to 3.

3.5 Python

```
import itertools
out_size = lambda n, k, p, s: (n + 2 * p - k) // s + 1
cases_py = list(itertools.product([2, 3, 5], [0, 1, 2], [1, 2, 3]))
ok_py = [corr2d(img_py, np.ones((k, k)), stride=s, pad=p).shape == (out_size(8, k, p, s),) * 2
          for k, p, s in cases_py]
print(len(ok_py), all(ok_py))
```

27 True

3.6 R

```
out_size_r <- function(n, k, p, s) (n + 2 * p - k) %/% s + 1
cases_r <- expand.grid(k = c(2, 3, 5), p = 0:2, s = 1:3)
ok_r <- apply(cases_r, 1, function(case) {
  Z <- corr2d(img_r, matrix(1, case[["k"]], case[["k"]]),
              stride = case[["s"]], pad = case[["p"]])
  all(dim(Z) == out_size_r(8, case[["k"]], case[["p"]], case[["s"]]))
})
c(length(ok_r), all(ok_r))
```

[1] 27 1

The formula gives the shape of the output in all 27 combinations. The formula can also give a size of zero or less, when the filter is larger than the padded image, and then no feature map exists.

i Read

Dumoulin and Visin work through the arithmetic of convolutions^a with diagrams [3].

^a<https://arxiv.org/abs/1603.07285>

4 Sharing parameters

A convolutional layer is a fully connected layer with most of its weights fixed at zero and the rest tied together. The next exercise builds that layer explicitly.

Exercise 4.1 (A convolution is a matrix product on the flattened image). Flatten the 8×8 image into $\tilde{x} \in \mathbb{R}^{64}$ (Definition 1.1), and flatten the 6×6 feature map of Solution 2.6 the same way. Then the feature map is a matrix M times $\tilde{x}$. What shape is M ?

Solution

Solution 4.1. M has one row per feature-map entry and one column per pixel: 36×64 .

Exercise 4.2 (A convolution is a matrix product on the flattened image (part 2)). Use the setup of Exercise 4.1. Which entries of row (i, j) of M are not fixed at zero, and what are their values?

Solution

Solution 4.2. Row (i, j) is row $6(i - 1) + j$ of M . It reads only the nine pixels of the patch with top-left corner (i, j) . Pixel $(i + u - 1, j + v - 1)$ is entry $8(i + u - 2) + j + v - 1$ of $\tilde{x}$, and its weight is K_{uv} . The other 55 entries of the row are zero.

Exercise 4.3 (A convolution is a matrix product on the flattened image (part 3)). Use the setup of Exercise 4.1. Build M for K_v and check that $M\tilde{x}$ is the flattened feature map.

Solution

Solution 4.3. For Exercise 4.3, the code builds M row by row:

4.1 Python

```
M_py = np.zeros((36, 64))
used_py = np.zeros((36, 64), dtype=bool) # entries that hold a filter weight
for i in range(6):
    for j in range(6):
        for u in range(3):
            for v in range(3):
                M_py[6 * i + j, 8 * (i + u) + (j + v)] = K_v[u, v]
                used_py[6 * i + j, 8 * (i + u) + (j + v)] = True
print(M_py.shape, np.allclose(M_py @ X_dig[1], Z_v.reshape(-1)))

(36, 64) True
```

```
print(used_py.sum(), np.unique(M_py[used_py]).size)
```

```
324 3
```

4.2 R

```
M_r <- matrix(0, 36, 64)
used_r <- matrix(FALSE, 36, 64) # entries that hold a filter weight
for (i in 1:6) {
  for (j in 1:6) {
    for (u in 1:3) {
      for (v in 1:3) {
        M_r[6 * (i - 1) + j, 8 * (i + u - 2) + j + v - 1] <- K_v_r[u, v]
        used_r[6 * (i - 1) + j, 8 * (i + u - 2) + j + v - 1] <- TRUE
      }
    }
  }
}
dim(M_r)
```

```
[1] 36 64
```

```
all.equal(drop(M_r %%% X_dig_r[2, ]), as.vector(t(Z_v_r)))
```

```
[1] TRUE
```

```
c(sum(used_r), length(unique(M_r[used_r])))
```

```
[1] 324 3
```

The check returns TRUE: row (i, j) of Mx is entry (i, j) of the [feature map](#), flattened row by row. Of the 2304 entries of M , 324 hold a [filter](#) weight, and those take only 3 distinct values, because K_v has zeros and repeated entries.

Definition 4.1 (Parameter sharing (weight sharing, weight tying)). A layer has **parameter sharing** when one parameter is used at many positions. In a convolutional layer, every output position uses the same filter K and bias b , and each output reads only the $k \times k$ patch under the filter, its **receptive field** [6]. Compared with a fully connected layer on the flattened input, the weight matrix M of Solution 4.2 is sparse, and its nonzero entries are tied together. Parameter sharing is also called **weight sharing** or **weight tying**.

Example 4.1 (Counting parameters for the digits). Count the parameters of three layers that read an 8×8 image. The first is fully connected with 36 units, one for each entry of the 6×6 feature map. The second is a convolutional layer with one 3×3 filter. The third is a convolutional layer with 4 filters.

4.3 Python

```
p_in = 8 * 8
fc_36 = (p_in + 1) * 36 # a weight per pixel and a bias, for each unit
conv_1 = 3 * 3 + 1 # nine weights and a bias
conv_4 = 4 * (3 * 3 + 1)
fc_144 = (p_in + 1) * (4 * 36) # a fully connected layer with 4 maps' worth of units
print(fc_36, conv_1, conv_4, fc_144)
```

```
2340 10 40 9360
```

4.4 R

```
p_in_r <- 8 * 8
fc_36_r <- (p_in_r + 1) * 36      # a weight per pixel and a bias, for each unit
conv_1_r <- 3 * 3 + 1            # nine weights and a bias
conv_4_r <- 4 * (3 * 3 + 1)
fc_144_r <- (p_in_r + 1) * (4 * 36) # a fully connected layer with 4 maps' worth of units
c(fc_36_r, conv_1_r, conv_4_r, fc_144_r)
```

```
[1] 2340  10  40 9360
```

A fully connected layer producing one map's worth of outputs has 2340 parameters; the convolutional layer has 10. With 4 feature maps, the counts are 9360 and 40. The convolutional count does not depend on the image size at all: the same 3×3 filter on a 32×32 image still has 10 parameters, while the fully connected layer's count grows with the product of the input and output sizes.

5 Pooling and channels

Exercise 5.1 (Shrinking a feature map). Keep the positive part of the 6×6 vertical-edge map of Solution 2.6, setting negative entries to 0. Then split it into non-overlapping 2×2 blocks and replace each block with its largest entry. What size is the result?

Solution

Solution 5.1. In Exercise 5.1, the 6×6 map has 3×3 blocks, so the result is 3×3 .

Exercise 5.2 (Shrinking a feature map (part 2)). Use the setup of Exercise 5.1. How many parameters does this step have?

Solution

Solution 5.2. None: taking a maximum has no weights to learn.

Exercise 5.3 (Shrinking a feature map (part 3)). Use the setup of Exercise 5.1. Shift the digit one pixel to the right and repeat. Is the 3×3 result more similar to its unshifted version than the 6×6 map is to its unshifted version? Measure similarity by the correlation^a between the two maps.

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

Solution

Solution 5.3. In Exercise 5.3, the code compares the maps before and after the shift:

5.1 Python

```
def max_pool(Z, size=2):
    h, w = Z.shape
    return Z.reshape(h // size, size, w // size, size).max(axis=(1, 3))

relu_map = np.maximum(Z_v, 0)
shifted = np.roll(img_py, 1, axis=1) # one pixel right; the last column is empty
relu_map_sh = np.maximum(corr2d(shifted, K_v), 0)
print(max_pool(relu_map).astype(int))

[[38 42  0]
 [41 31  0]
 [48 46  0]]

print(np.corrcoef(relu_map.ravel(), relu_map_sh.ravel())[0, 1],
      np.corrcoef(max_pool(relu_map).ravel(), max_pool(relu_map_sh).ravel())[0, 1])
0.3355417439447914 0.6465797061063564
```

5.2 R

```
max_pool <- function(Z, size = 2) {
  out <- matrix(0, nrow(Z) / size, ncol(Z) / size)
  for (i in seq_len(nrow(out))) {
    for (j in seq_len(ncol(out))) {
      out[i, j] <- max(Z[(i - 1) * size + 1:size, (j - 1) * size + 1:size])
    }
  }
  out
}

relu_map_r <- pmax(Z_v_r, 0)
shifted_r <- cbind(0, img_r[, -8]) # one pixel right; the last column is empty
relu_map_sh_r <- pmax(corr2d(shifted_r, K_v_r), 0)
max_pool(relu_map_r)

      [,1] [,2] [,3]
[1,]   38   42   0
[2,]   41   31   0
[3,]   48   46   0

cor_before <- cor(as.vector(relu_map_r), as.vector(relu_map_sh_r))
cor_after <- cor(as.vector(max_pool(relu_map_r)), as.vector(max_pool(relu_map_sh_r)))
c(cor_before, cor_after)

[1] 0.3355417 0.6465797
```

The correlation^a between the two maps is 0.34 before combining blocks and 0.65 after. A shift of one pixel moves the large responses to a neighbouring column of the map, and the block maximum can keep its value when a response moves within its block.

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

Definition 5.1 (Pooling (subsampling)). A **pooling** layer replaces each $q \times q$ block of a feature map by one number, usually moving the block by q pixels (stride q), so that blocks do not overlap [6]. **Max pooling** takes the largest entry of the block, and **average pooling** takes their mean. Pooling has no parameters. It is also called **subsampling** or downsampling, though downsampling covers other operations too. A $q \times q$ pooling layer turns an $h \times w$ map into an $(h/q) \times (w/q)$ map when q divides h and w .

Example 5.1 (Max and average pooling on one map). Pool the 6×6 positive part of the vertical-edge map with $q = 2$, by maximum and by mean.

5.3 Python

```
avg_pool = relu_map.reshape(3, 2, 3, 2).mean(axis=(1, 3))
print(max_pool(relu_map).astype(int))
```

```
[[38 42  0]
 [41 31  0]
 [48 46  0]]
```

```
print(avg_pool.round(1))
```

```
[[23.5 18.  0. ]
 [29.2 15.  0. ]
 [24.  22.8 0. ]]
```

5.4 R

```
avg_pool_r <- matrix(0, 3, 3)
for (i in 1:3) {
  for (j in 1:3) {
    avg_pool_r[i, j] <- mean(relu_map_r[(i - 1) * 2 + 1:2, (j - 1) * 2 + 1:2])
  }
}
max_pool(relu_map_r)
```

```
      [,1] [,2] [,3]
[1,]   38   42   0
[2,]   41   31   0
[3,]   48   46   0
```

```
round(avg_pool_r, 1)
```

```
      [,1] [,2] [,3]
[1,] 23.5 18.0  0
[2,] 29.2 15.0  0
[3,] 24.0 22.8  0
```

Both outputs are 3×3 . The maximum keeps each block's strongest response, up to 48, and the mean spreads that response over the block, so it is never larger.

A layer usually applies several filters to the same input, and a later layer reads all of their feature maps together.

Definition 5.2 (Channel). A **channel** is one $H \times W$ grid in an array of shape $H \times W \times C$ with C channels. A colour image has $C = 3$ channels (red, green, blue), a greyscale image such as a digit has $C = 1$, and the feature maps produced by F filters form an array with $C = F$ channels. A filter that reads C input channels has shape $k \times k \times C$: it multiplies and adds over the $k \times k$ patch in every channel at once, so it produces one feature map. A convolutional layer with F such filters has $F(k^2C + 1)$ parameters, counting one bias per filter. The number of channels is sometimes called the depth of the array, which differs from the depth of a network.

Example 5.2 (Channels through two convolutional layers). The first layer of the digit network has $F_1 = 4$ filters of size 3×3 on the single channel of the digit. A second convolutional layer with $F_2 = 6$ filters of size 2×2 reads the 4 channels the first layer produces. Count each layer's parameters.

5.5 Python

```
conv_params = lambda k, c_in, f: f * (k * k * c_in + 1)
print(conv_params(3, 1, 4), conv_params(2, 4, 6))

40 102
```

5.6 R

```
conv_params_r <- function(k, c_in, f) f * (k * k * c_in + 1)
c(conv_params_r(3, 1, 4), conv_params_r(2, 4, 6))
```

```
[1] 40 102
```

The first layer has 40 parameters and the second 102. Each of the second layer's filters has $2 \times 2 \times 4 = 16$ weights, since it reads all four channels, and the layer has six of them.

6 A small convolutional network

Exercise 6.1 (A small network for the digits, end to end). Design a network for the 8×8 digits with these layers:

1. a convolutional layer with 4 filters of size 3×3 , stride 1, no padding, followed by ReLU;
2. 2×2 max pooling;
3. flattening;
4. a fully connected layer with 10 outputs, one for each digit, followed by softmax.

Give the shape of the array after each layer.

Solution

Solution 6.1. Layer by layer, Equation 2 gives the shapes: the [convolution](#) maps $8 \times 8 \times 1$ to $6 \times 6 \times 4$, the [pooling](#) gives $3 \times 3 \times 4$, flattening gives a vector of 36 numbers, and the output layer maps it to 10.

Exercise 6.2 (A small network for the digits, end to end (part 2)). Use the setup of Exercise 6.1. Give the number of parameters after each layer.

Solution

Solution 6.2. The [convolution](#) has $4(9 + 1) = 40$ parameters, the [pooling](#) and the [flattening](#) have none, and the output layer has $10 \times (36 + 1) = 370$ parameters.

Exercise 6.3 (A small network for the digits, end to end (part 3)). Use the setup of Exercise 6.1. Pass the second digit through the network with random weights to check the shapes.

Solution

Solution 6.3.

6.1 Python

```

rng_cnn = np.random.default_rng(0)
K_cnn = rng_cnn.normal(scale=0.3, size=(4, 3, 3))      # 4 filters; random, not trained
b_cnn = np.zeros(4)
W_out = rng_cnn.normal(scale=0.1, size=(10, 36))
b_out = np.zeros(10)

maps_cnn = np.stack([np.maximum(corr2d(img_py, K_cnn[f]) + b_cnn[f], 0) for f in range(4)], axis=-1)
pooled_cnn = np.stack([max_pool(maps_cnn[:, :, f]) for f in range(4)], axis=-1)
flat_cnn = pooled_cnn.reshape(-1)
logits_cnn = W_out @ flat_cnn + b_out
probs_cnn = np.exp(logits_cnn) / np.exp(logits_cnn).sum()
print(img_py.shape, maps_cnn.shape, pooled_cnn.shape, flat_cnn.shape, probs_cnn.shape)

(8, 8) (6, 6, 4) (3, 3, 4) (36,) (10,)

print(probs_cnn.sum())

1.0

```

6.2 R

```

set.seed(0)
K_cnn_r <- array(rnorm(4 * 3 * 3, sd = 0.3), dim = c(3, 3, 4)) # 4 filters; random, not trained
W_out_r <- matrix(rnorm(10 * 36, sd = 0.1), nrow = 10)

maps_cnn_r <- array(0, dim = c(6, 6, 4))
pooled_cnn_r <- array(0, dim = c(3, 3, 4))
for (f in 1:4) {
  maps_cnn_r[, , f] <- pmax(corr2d(img_r, K_cnn_r[, , f]), 0) # zero bias, then ReLU
  pooled_cnn_r[, , f] <- max_pool(maps_cnn_r[, , f])
}
flat_cnn_r <- as.vector(pooled_cnn_r)
logits_cnn_r <- drop(W_out_r %*% flat_cnn_r)
probs_cnn_r <- exp(logits_cnn_r) / sum(exp(logits_cnn_r))
list(dim(img_r), dim(maps_cnn_r), dim(pooled_cnn_r), length(flat_cnn_r), length(probs_cnn_r))

[[1]]
[1] 8 8

[[2]]
[1] 6 6 4

[[3]]
[1] 3 3 4

[[4]]
[1] 36

[[5]]
[1] 10

sum(probs_cnn_r)

[1] 1

```

The shapes match: 8×8 , then $6 \times 6 \times 4$, $3 \times 3 \times 4$, 36 and 10, and the ten outputs sum to 1. The weights are random, so the ten probabilities carry no information about the digit; fitting the weights is a job for gradient descent^a and backpropagation.

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

Exercise 6.4 (A small network for the digits, end to end (part 4)). Use the setup of Exercise 6.1. Compare the parameter count with a fully connected network that has one hidden layer of 144 units.

Solution

Solution 6.4.

6.3 Python

```
params_cnn = [4 * (3 * 3 + 1), 0, 0, 10 * (36 + 1)]
params_fc = (64 + 1) * 144 + (144 + 1) * 10
print(params_cnn, sum(params_cnn), params_fc)

[40, 0, 0, 370] 410 10810
```

6.4 R

```
params_cnn_r <- c(4 * (3 * 3 + 1), 0, 0, 10 * (36 + 1))
params_fc_r <- (64 + 1) * 144 + (144 + 1) * 10
c(sum(params_cnn_r), params_fc_r)

[1] 410 10810
```

The convolutional network has 410 parameters and the fully connected one has 10,810, about 26 times as many.

Remark (Training a convolutional network)

Remark 6.1 (Training a convolutional network). Backpropagation works on a convolutional network as on any composition of differentiable layers, with two additions. A filter weight K_{uv} is used at every output position, so its gradient is the sum over positions of the gradient each use contributes, as the chain rule gives for a variable that appears many times. Max pooling passes the gradient only to the entry that was largest in its block, since the other entries did not affect the output. The same sum over uses returns for recurrent layers, where a weight is used at every time step.

LeCun et al. [8] used this pipeline of convolution, pooling and a final classifier to read handwritten digits.

7 Recurrent layers

An image has a fixed size, and its neighbours lie in two directions. A series has an order and no fixed length.

Exercise 7.1 (A series as input). The `Smarket` data hold the daily returns (percent change) of the S&P 500 index on 1250 trading days from 2001 to 2005 [5, Ch. 4]. The rows are in time order, and `Lag1` is the previous day's return. Check that the first 30 rows are 30 consecutive trading days, by comparing each `Lag1` with the previous row's `Today`.

Solution

Solution 7.1.

7.1 Python

```
smarket = pd.read_csv("data/smarket.csv")
today_py = smarket["Today"].to_numpy()
first30 = smarket.iloc[:30]
print(smarket[["Year", "Lag1", "Today"]].head(4))
```

```
   Year  Lag1  Today
0  2001  0.381  0.959
1  2001  0.959  1.032
2  2001  1.032 -0.623
3  2001 -0.623  0.614
```

```
print(np.allclose(first30["Lag1"].to_numpy()[1:], first30["Today"].to_numpy()[:-1]))
True
```

7.2 R

```
smarket_r <- read.csv("data/smarket.csv")
head(smarket_r[, c("Year", "Lag1", "Today")], 4)
```

```
# A tibble: 4 x 3
  Year  Lag1  Today
<int> <dbl> <dbl>
1  2001  0.381  0.959
2  2001  0.959  1.03
3  2001  1.03  -0.623
4  2001 -0.623  0.614
```

```
first30_r <- smarket_r[1:30, ]
all.equal(first30_r$Lag1[-1], first30_r$Today[-30])
```

```
[1] TRUE
```

Each row's `Lag1` equals the row above's `Today`, so the rows are consecutive days.
The check passes, so row t is the day after row $t - 1$.

Exercise 7.2 (A series as input (part 2)). Use the setup of Exercise 7.1.

A fully connected layer taking these 30 returns as its 30 inputs would be tied to the length 30. Name two other things it handles badly when the inputs are a time series.

Solution

Solution 7.2. A fully connected layer gives each day its own weights, so it must learn the same pattern again for every position in the window. It also cannot take a series of 31 days when it was built for 30, and, as in Solution 1.8, it is blind to which days are adjacent.

Exercise 7.3 (Carrying a memory along a series). Standardize the daily returns by dividing by their standard deviation in the full data, and call the first three standardized returns z_1, z_2, z_3 . Keep a state of two numbers $\tilde{h}_t$, starting from $\tilde{h}_0 = \vec{0}$, and update it each day by

$$\tilde{h}_t = \tanh(W_h \tilde{h}_{t-1} + w_x z_t), \quad W_h = \begin{bmatrix} 0.5 & -0.4 \\ 0.3 & 0.6 \end{bmatrix}, \quad w_x = \begin{bmatrix} 1.0 \\ -0.5 \end{bmatrix}. \quad (3)$$

These weights are chosen for illustration; none has been fitted. Compute $\tilde{h}_1, \tilde{h}_2, \tilde{h}_3$.

Solution

Solution 7.3. Each step multiplies the previous state by W_h , adds the new input scaled by w_x , and applies $\tanh$ to each coordinate.

7.3 Python

```
z_py = today_py / today_py.std(ddof=1)      # standardize by the full-data sd
W_h = np.array([[0.5, -0.4], [0.3, 0.6]])
w_x = np.array([1.0, -0.5])

def rnn_run(z, W_h, w_x):
    h = np.zeros(len(w_x))
    states = []
    for z_t in z:
        h = np.tanh(W_h @ h + w_x * z_t)
        states.append(h)
    return np.array(states)

print(z_py[:3].round(3))

[ 0.844  0.908 -0.548]

print(rnn_run(z_py[:3], W_h, w_x).round(3))

[[ 0.688 -0.399]
 [ 0.888 -0.452]
 [ 0.076  0.263]]
```

7.4 R

```
z_r <- smarket_r$Today / sd(smarket_r$Today) # standardize by the full-data sd
W_h_r <- matrix(c(0.5, -0.4, 0.3, 0.6), nrow = 2, byrow = TRUE)
w_x_r <- c(1.0, -0.5)

rnn_run_r <- function(z, W_h, w_x) {
  h <- numeric(length(w_x))
  states <- matrix(0, length(z), length(w_x))
  for (t in seq_along(z)) {
    h <- drop(tanh(W_h %*% h + w_x * z[t]))
    states[t, ] <- h
  }
  states
}

round(z_r[1:3], 3)

[1] 0.844 0.908 -0.548

round(rnn_run_r(z_r[1:3], W_h_r, w_x_r), 3)

      [,1] [,2]
[1,] 0.688 -0.399
[2,] 0.888 -0.452
[3,] 0.076  0.263
```

Exercise 7.4 (Carrying a memory along a series (part 2)). Use the setup of Exercise 7.3. What does h_3 depend on?

Solution

Solution 7.4. The state $\tilde{h}_3$ depends on all three returns: z_3 enters directly, z_2 enters through $\tilde{h}_2$, and z_1 through $\tilde{h}_1$ and then $\tilde{h}_2$. The same W_h and w_x are used at all three steps, and the state keeps a length of two however long the series.

Definition 7.1 (Recurrent layer). A **recurrent layer** reads a sequence $\tilde{x}_1, \dots, \tilde{x}_T$ with $\tilde{x}_t \in \mathbb{R}^p$ and keeps a state $\tilde{h}_t \in \mathbb{R}^H$, updated at each step by

$$\tilde{h}_t = f(W_x \tilde{x}_t + W_h \tilde{h}_{t-1} + \tilde{b}), \quad \tilde{h}_0 = \vec{0}, \quad (4)$$

with $W_x \in \mathbb{R}^{H \times p}$, $W_h \in \mathbb{R}^{H \times H}$, a bias $\tilde{b} \in \mathbb{R}^H$ and an activation f applied coordinate by coordinate. The same W_x , W_h and $\tilde{b}$ are used at every step. Predictions are read from the states, such as $g(U\tilde{h}_t)$ at each step or only at $t = T$. The layer in Equation 4 is the simple or Elman recurrent layer [9].

Definition 7.2 (Hidden state (memory)). The **hidden state** $\tilde{h}_t \in \mathbb{R}^H$ of a recurrent layer (Definition 7.1) is the state it keeps and updates at each step by Equation 4. The state $\tilde{h}_t$ is also called the **memory**.

Definition 7.3 (Recurrent neural network (RNN)). A network built from recurrent layers (Definition 7.1) is a **recurrent neural network** (RNN) [10].

Example 7.1 (The hidden state over thirty trading days). Run Equation 3 over the first 30 standardized returns, and again over only the first 10. Figure 2 plots the thirty states.

7.5 Python

```
H30 = rnn_run(z_py[:30], W_h, w_x)
H10 = rnn_run(z_py[:10], W_h, w_x)
print(H30.shape, H10.shape)
```

```
(30, 2) (10, 2)
```

```
print(np.allclose(H30[9], H10[9]))
```

```
True
```

7.6 R

```
H30_r <- rnn_run_r(z_r[1:30], W_h_r, w_x_r)
H10_r <- rnn_run_r(z_r[1:10], W_h_r, w_x_r)
list(dim(H30_r), dim(H10_r))
```

```
[[1]]
```

```
[1] 30 2
```

```
[[2]]
```

```
[1] 10 2
```

```
all.equal(H30_r[10, ], H10_r[10, ])
```

```
[1] TRUE
```

The same layer takes a series of 10 days and one of 30 days, with 6 weights in both cases. The state after day 10 is the same in the two runs, since later days cannot change it.

```
# The outer margin leaves room under the lower panel for its legend.
op <- par(mfrow = c(2, 1), mar = c(4, 4, 1, 1), oma = c(3, 0, 0, 0))
plot(1:30, z_r[1:30], type = "h", xlab = "", ylab = "std. return")
```

```

abline(h = 0, lty = 2)
matplot(1:30, H30_r, type = "l", lty = 1, col = c("black", "grey55"), ylim = c(-1, 1.4),
        xlab = "trading day", ylab = "hidden state")
# An empty plot over the whole device places the legend in the outer margin.
par(fig = c(0, 1, 0, 1), oma = c(0, 0, 0, 0), mar = c(0, 0, 0, 0), new = TRUE)
plot.new()
legend("bottom", legend = c("coordinate 1", "coordinate 2"), lty = 1,
      col = c("black", "grey55"), bty = "n", horiz = TRUE)
par(op)

```

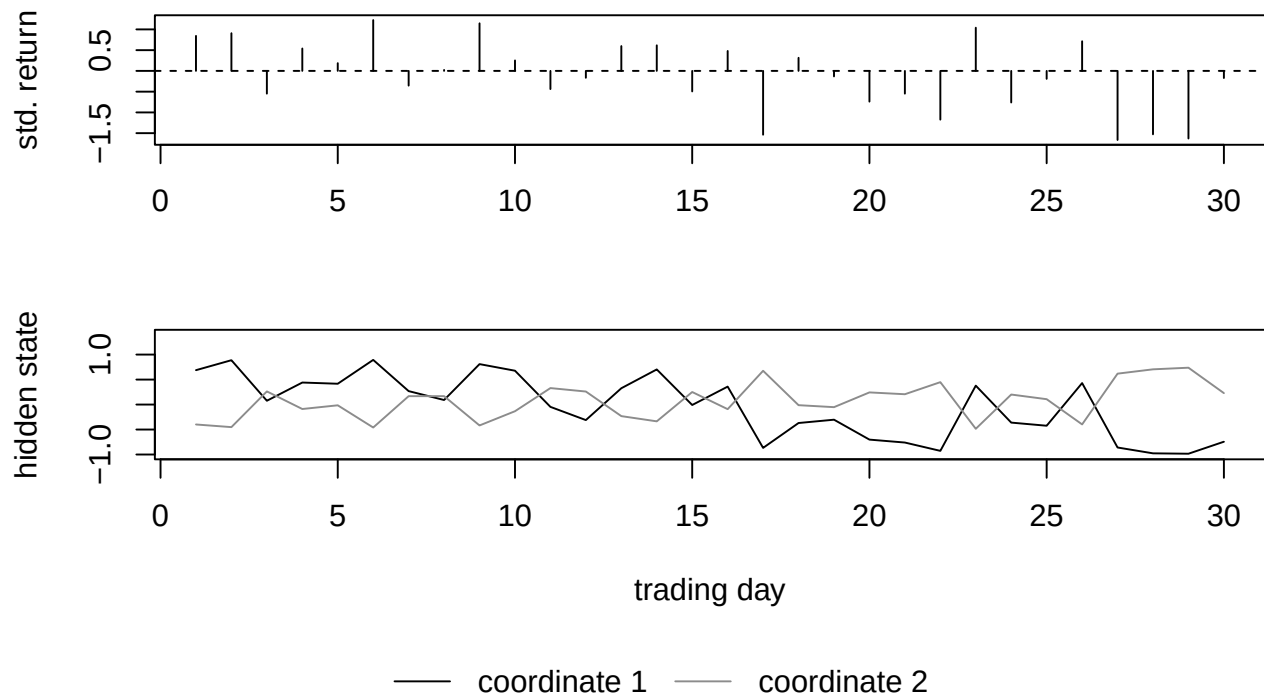


Figure 2: The standardized daily returns of the first 30 trading days of the Smarket data (top), and the two coordinates of the hidden state of the toy recurrent layer (bottom).

Exercise 7.5 (Parameters of a recurrent layer). A recurrent layer has p inputs^a per step and H hidden units.

How many parameters does it have?

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

Solution

Solution 7.5. W_x has Hp entries, W_h has H^2 and $\tilde{b}$ has H , so the layer has $H(p + H + 1)$ parameters.

Exercise 7.6 (Parameters of a recurrent layer (part 2)). Use the setup of Exercise 7.5. Does the count change with the length T of the series?

Solution

Solution 7.6. No: the same W_x , W_h and $\tilde{b}$ are used at every step, whatever T is.

Exercise 7.7 (Parameters of a recurrent layer (part 3)). Use the setup of Exercise 7.5.

How many parameters does a fully connected layer with H units have when it takes all T steps, Tp numbers, as input?

Solution

Solution 7.7. $H(Tp + 1)$, which grows with T .

Definition 7.4 (Unrolling (unfolding through time)). **Unrolling** a recurrent layer over T steps writes it as a feedforward network with T copies of the same layer: copy t takes x_t and the output $\tilde{h}_{t-1}$ of copy $t - 1$, and returns $\tilde{h}_t$. All copies share their parameters (Definition 4.1). Unrolling is also called **unfolding through time**. The unrolled network has depth T , where T is the length of the series.

Example 7.2 (Unrolled over thirty days). A toy recurrent layer has $p = 1$ input^a and $H = 2$ units. Count the weights of its unrolled network over $T = 30$ days, with and without sharing, and for a fully connected layer on the 30 returns.

7.7 Python

```
D_u, H_u, T_u = 1, 2, 30
shared = H_u * (D_u + H_u + 1)
print(shared, T_u * shared, H_u * (T_u * D_u + 1))
8 240 62
```

7.8 R

```
D_u <- 1
H_u <- 2
T_u <- 30
shared_r <- H_u * (D_u + H_u + 1)
c(shared_r, T_u * shared_r, H_u * (T_u * D_u + 1))
[1] 8 240 62
```

With sharing, the unrolled network has 8 parameters, as the layer does. Thirty independent copies would have 240. A fully connected layer with two units on all thirty returns has 62, and it fits a series of exactly thirty days only.

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

i Watch

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

Video 2: Starmer explains recurrent neural networks [11].

8 Backpropagation through time

Reuse of the weights across steps is also what makes the gradient hard.

Exercise 8.1 (How the state depends on an earlier state). Consider Equation 4 with $f = \tanh$ and write $\tilde{a}_t = W_x \tilde{x}_t + W_h \tilde{h}_{t-1} + \tilde{b}$. The Jacobian $\frac{\partial \tilde{h}_t}{\partial \tilde{h}_{t-1}}$ is the $H \times H$ matrix whose entry (i, j) is $\partial h_{t,i} / \partial h_{t-1,j}$. Find $\frac{\partial \tilde{h}_t}{\partial \tilde{h}_{t-1}}$.

Solution

Solution 8.1. The derivative of $\tanh$ is $1 - \tanh^2$, so $\tilde{h}_t = \tanh(\tilde{a}_t)$ has the diagonal^a Jacobian $\frac{\partial \tilde{h}_t}{\partial \tilde{a}_t} = D_t$, where $D_t = \text{diag}(1 - h_{t,1}^2, \dots, 1 - h_{t,H}^2)$. Also $\frac{\partial \tilde{a}_t}{\partial \tilde{h}_{t-1}} = W_h$. The chain rule multiplies the two:

$$\frac{\partial \tilde{h}_t}{\partial \tilde{h}_{t-1}} = \frac{\partial \tilde{h}_t}{\partial \tilde{a}_t} \frac{\partial \tilde{a}_t}{\partial \tilde{h}_{t-1}} = D_t W_h.$$

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

Exercise 8.2 (How the state depends on an earlier state (part 2)). Use the setup of Exercise 8.1. Find $\frac{\partial \tilde{h}_T}{\partial \tilde{h}_k}$ for $k < T$.

Solution

Solution 8.2. Because $\tilde{h}_T$ depends on $\tilde{h}_k$ through $\tilde{h}_{T-1}, \dots, \tilde{h}_{k+1}$, the chain rule multiplies one factor for each step:

$$\frac{\partial \tilde{h}_T}{\partial \tilde{h}_k} = \frac{\partial \tilde{h}_T}{\partial \tilde{h}_{T-1}} \frac{\partial \tilde{h}_{T-1}}{\partial \tilde{h}_{T-2}} \dots \frac{\partial \tilde{h}_{k+1}}{\partial \tilde{h}_k} = D_T W_h D_{T-1} W_h \dots D_{k+1} W_h.$$

It is a product of $T - k$ matrices, all built from the one matrix W_h .

Theorem 8.1 (Jacobian of the state of a tanh recurrent layer). *For a recurrent layer with $f = \tanh$ and $\tilde{a}_t = W_x \tilde{x}_t + W_h \tilde{h}_{t-1} + \tilde{b}$, let $D_t = \text{diag}(1 - h_{t,1}^2, \dots, 1 - h_{t,H}^2)$. Then*

$$\frac{\partial \tilde{h}_t}{\partial \tilde{h}_{t-1}} = D_t W_h,$$

and for $k < T$,

$$\frac{\partial \tilde{h}_T}{\partial \tilde{h}_k} = D_T W_h D_{T-1} W_h \dots D_{k+1} W_h.$$

i Proof

Proof. By Solution 8.1 and Solution 8.2. □

Exercise 8.3 (Why the gradient vanishes or explodes). Let $\|\cdot\|$ denote the spectral norm of a matrix, its largest singular value. Use Theorem 8.1. Bound $\|D_t\|$, then bound $\left\| \frac{\partial \tilde{h}_T}{\partial \tilde{h}_k} \right\|$ in terms of $\|W_h\|$.

Solution

Solution 8.3. D_t is diagonal^a with entries $1 - h_{t,i}^2 \in (0, 1]$, so $\|D_t\| \leq 1$. The spectral norm^b of a product is at most the product of the norms^c, so

$$\left\| \frac{\partial h_T}{\partial \tilde{h}_k} \right\| \leq \prod_{t=k+1}^T \|D_t\| \|W_h\| \leq \|W_h\|^{T-k}.$$

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

^b<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^c<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

Exercise 8.4 (Why the gradient vanishes or explodes (part 2)). Use the setup of Exercise 8.3. What does the bound say if $\|W_h\| < 1$?

Solution

Solution 8.4. When $\|W_h\| < 1$, the bound $\|W_h\|^{T-k}$ shrinks geometrically in $T - k$. A change in $\tilde{h}_k$ then has an exponentially small effect on $\tilde{h}_T$, and so a loss that depends on $\tilde{h}_T$ sends exponentially small gradient to the early steps. The gradient vanishes.

Exercise 8.5 (Why the gradient vanishes or explodes (part 3)). Use the setup of Exercise 8.3. If the activation were the identity, so that every D_t were the identity matrix, what would $\frac{\partial h_T}{\partial \tilde{h}_k}$ be?

Solution

Solution 8.5. With the identity activation, $\frac{\partial h_T}{\partial \tilde{h}_k} = W_h^{T-k}$.

Exercise 8.6 (Why the gradient vanishes or explodes (part 4)). Use the setup of Exercise 8.3. With the identity activation, what happens to the norm^a of $\frac{\partial h_T}{\partial \tilde{h}_k}$ if W_h has an eigenvalue of absolute value greater than 1?

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

Solution

Solution 8.6. If W_h has an eigenvalue λ with $|\lambda| > 1$ and eigenvector $\tilde{v}$, then $W_h^{T-k} \tilde{v} = \lambda^{T-k} \tilde{v}$, which grows geometrically, so $\|W_h^{T-k}\| \geq |\lambda|^{T-k}$ and the gradient can explode. With tanh, the factors D_t shrink the product, and the bound of Solution 8.3 does not rule out growth: it only limits it.

Theorem 8.2 (Jacobian of a tanh recurrent layer). *For a recurrent layer with $f = \tanh$ and $k < T$,*

$$\left\| \frac{\partial h_T}{\partial \tilde{h}_k} \right\| \leq \|W_h\|^{T-k},$$

whatever the inputs [12], [13]. If $\|W_h\| < 1$, the gradient of a loss at step T with respect to earlier states vanishes exponentially in $T - k$.

i Proof

Proof. By Solution 8.3 and Solution 8.4. □

Example 8.1 (Gradient norms over thirty trading days). Take four hidden units, with $W_h = \rho Q$ for a rotation matrix Q , so that $\|W_h\| = \rho$ exactly. Drive the layer with the 30 standardized returns, with

input weights $(0.5, -0.3, 0.4, 0.2)$ chosen for illustration. Compute $\left\| \frac{\partial h_m}{\partial h_0} \right\|$ for $m = 1, 5, 10, 20, 30$ and $\rho = 0.5, 0.9, 1, 1.5$, and compare it with the bound ρ^m .

8.1 Python

```
def rot(a):
    return np.array([[np.cos(a), -np.sin(a)], [np.sin(a), np.cos(a)]])

Q4 = np.block([[rot(0.7), np.zeros((2, 2))], [np.zeros((2, 2)), rot(1.9)]])
w_x4 = np.array([0.5, -0.3, 0.4, 0.2])
lags = [1, 5, 10, 20, 30]
rhos = [0.5, 0.9, 1.0, 1.5]

def jacobian_norms(rho, z):
    W, h, J, norms = rho * Q4, np.zeros(4), np.eye(4), []
    for z_t in z:
        h = np.tanh(W @ h + w_x4 * z_t)
        J = np.diag(1 - h ** 2) @ W @ J      # d h_m / d h_0, one factor per step
        norms.append(np.linalg.norm(J, 2))
    return np.array(norms)

norms_py = np.array([jacobian_norms(rho, z_py[:30])[m - 1 for m in lags] for rho in rhos])
bound_py = np.array([[rho ** m for m in lags] for rho in rhos])
fmt = {"float_kind": lambda v: f"{v:.1e}"}
print(np.array2string(norms_py, formatter=fmt))

[[4.9e-01 2.6e-02 6.3e-04 3.1e-07 1.1e-10]
 [8.7e-01 4.1e-01 1.8e-01 2.3e-02 8.7e-04]
 [9.7e-01 6.3e-01 4.9e-01 1.7e-01 1.2e-02]
 [1.5e+00 2.4e+00 4.9e+00 3.6e+00 1.1e+01]]

print(np.array2string(bound_py, formatter=fmt))

[[5.0e-01 3.1e-02 9.8e-04 9.5e-07 9.3e-10]
 [9.0e-01 5.9e-01 3.5e-01 1.2e-01 4.2e-02]
 [1.0e+00 1.0e+00 1.0e+00 1.0e+00 1.0e+00]
 [1.5e+00 7.6e+00 5.8e+01 3.3e+03 1.9e+05]]
```

8.2 R

```

rot_r <- function(a) matrix(c(cos(a), sin(a), -sin(a), cos(a)), nrow = 2)
Q4_r <- rbind(cbind(rot_r(0.7), matrix(0, 2, 2)), cbind(matrix(0, 2, 2), rot_r(1.9)))
w_x4_r <- c(0.5, -0.3, 0.4, 0.2)
lags_r <- c(1, 5, 10, 20, 30)
rhos_r <- c(0.5, 0.9, 1.0, 1.5)

jacobian_norms_r <- function(rho, z) {
  W <- rho * Q4_r
  h <- numeric(4)
  J <- diag(4)
  norms <- numeric(length(z))
  for (t in seq_along(z)) {
    h <- drop(tanh(W %*% h + w_x4_r * z[t]))
    J <- diag(1 - h^2) %*% W %*% J # d h_m / d h_0, one factor per step
    norms[t] <- svd(J)$d[1]
  }
  norms
}

norms_r <- t(sapply(rhos_r, function(rho) jacobian_norms_r(rho, z_r[1:30])[lags_r]))
bound_r <- outer(rhos_r, lags_r, "^")
dimnames(norms_r) <- dimnames(bound_r) <- list(rho = rhos_r, lag = lags_r)
noquote(format(norms_r, digits = 2, scientific = TRUE))

```

	lag				
rho	1	5	10	20	30
0.5	4.9e-01	2.6e-02	6.3e-04	3.1e-07	1.1e-10
0.9	8.7e-01	4.1e-01	1.8e-01	2.3e-02	8.7e-04
1	9.7e-01	6.3e-01	4.9e-01	1.7e-01	1.2e-02
1.5	1.5e+00	2.4e+00	4.9e+00	3.6e+00	1.1e+01

```
noquote(format(bound_r, digits = 2, scientific = TRUE))
```

	lag				
rho	1	5	10	20	30
0.5	5.0e-01	3.1e-02	9.8e-04	9.5e-07	9.3e-10
0.9	9.0e-01	5.9e-01	3.5e-01	1.2e-01	4.2e-02
1	1.0e+00	1.0e+00	1.0e+00	1.0e+00	1.0e+00
1.5	1.5e+00	7.6e+00	5.8e+01	3.3e+03	1.9e+05

In each table, the rows are ρ and the columns are the lags. Every norm is below its bound: TRUE. For $\rho = 0.5$ the norm falls to $1.1e-10$ by lag 30. Even for $\rho = 1$ it falls, to $1.2e-02$, because the factors D_t shrink the product. For $\rho = 1.5$ the norm is 11 at lag 30, above 1 but far below the bound $1.9e+05$ that the identity activation attains.

Exercise 8.7 (The gradient for the shared weights). Suppose the loss L depends on the layer only through the last state $\tilde{h}_T$. Let $\tilde{g}_t = \frac{\partial L}{\partial \tilde{a}_t}$, with $\tilde{a}_t$ as in Exercise 8.1.

Express $\tilde{g}_t$ in terms of $\frac{\partial L}{\partial \tilde{h}_t}$, and $\frac{\partial L}{\partial \tilde{h}_{t-1}}$ in terms of $\tilde{g}_t$.

Solution

Solution 8.7. Since $\tilde{h}_t = \tanh(\tilde{a}_t)$, the chain rule gives $\tilde{g}_t = D_t \frac{\partial L}{\partial \tilde{h}_t}$. Since $\tilde{a}_t$ depends on $\tilde{h}_{t-1}$ through $W_h \tilde{h}_{t-1}$, it gives $\frac{\partial L}{\partial \tilde{h}_{t-1}} = W_h^\top \tilde{g}_t$. Starting from $\frac{\partial L}{\partial \tilde{h}_T}$, which the loss supplies, these two rules produce $\tilde{g}_T, \tilde{g}_{T-1}, \dots, \tilde{g}_1$ in turn.

Exercise 8.8 (The gradient for the shared weights (part 2)). Use the setup of Exercise 8.7. The matrix W_h is used at every step. Find $\frac{\partial L}{\partial W_h}$.

Solution

Solution 8.8. The loss depends on W_h through each $\tilde{a}_t$, and $(\tilde{a}_t)_i = \sum_j (W_h)_{ij} h_{t-1,j} + \dots$, so

$$\frac{\partial L}{\partial (W_h)_{ij}} = \sum_{t=1}^T \frac{\partial L}{\partial (\tilde{a}_t)_i} h_{t-1,j} = \sum_{t=1}^T (g_t)_i h_{t-1,j}, \quad \text{that is,} \quad \frac{\partial L}{\partial W_h} = \sum_{t=1}^T g_t \tilde{h}_{t-1}^\top.$$

Each step contributes one outer product^a, and the gradient is their sum.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

Theorem 8.3 (The backward recursion and the shared-weight gradient). *Suppose the loss L depends on a tanh recurrent layer only through the last state $\tilde{h}_T$, and let $g_t = \frac{\partial L}{\partial \tilde{a}_t}$. Then*

$$g_t = D_t \frac{\partial L}{\partial \tilde{h}_t}, \quad \frac{\partial L}{\partial \tilde{h}_{t-1}} = W_h^\top g_t,$$

and the gradient for the weight matrix shared across steps is

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^T g_t \tilde{h}_{t-1}^\top.$$

i Proof

Proof. By Solution 8.7 and Solution 8.8. □

Example 8.2 (Checking the backward recursion on a small layer). Take a tanh recurrent layer with $H = 2$ units and $T = 3$ steps, scalar inputs, and the loss $L = \sum_i h_{T,i}$. Compute $\frac{\partial L}{\partial W_h}$ by the recursion of Theorem 8.3, and compare it with a finite difference of the loss:

8.3 Python

```

import numpy as np

rng_g = np.random.default_rng(571)
Wh_g, Wx_g = rng_g.normal(size=(2, 2)) * 0.8, rng_g.normal(size=2)
x_g = rng_g.normal(size=3)

def forward_g(Wh):
    h = [np.zeros(2)]
    for x in x_g:
        h.append(np.tanh(Wx_g * x + Wh @ h[-1]))
    return h

h_g = forward_g(Wh_g)
dL_dh = np.ones(2) # dL/dh_T for L = sum(h_T)
grad_g = np.zeros((2, 2))
for t in range(3, 0, -1):
    g = (1 - h_g[t] ** 2) * dL_dh # g_t = D_t dL/dh_t
    grad_g += np.outer(g, h_g[t - 1])
    dL_dh = Wh_g.T @ g # dL/dh_{t-1}

eps = 1e-6
num_g = np.zeros((2, 2))
for i in range(2):
    for j in range(2):
        d = np.zeros((2, 2)); d[i, j] = eps
        num_g[i, j] = (forward_g(Wh_g + d)[-1].sum() - forward_g(Wh_g - d)[-1].sum()) / (2 * eps)
print(np.max(np.abs(grad_g - num_g)))
4.551213572678847e-11

```

8.4 R

```

set.seed(571)
Wh_g <- matrix(rnorm(4), 2) * 0.8
Wx_g <- rnorm(2)
x_g <- rnorm(3)

forward_g <- function(Wh) {
  h <- list(c(0, 0))
  for (x in x_g) h[[length(h) + 1]] <- tanh(Wx_g * x + Wh %*% h[[length(h)]])
  h
}

h_g <- forward_g(Wh_g)
dL_dh <- c(1, 1) # dL/dh_T for L = sum(h_T)
grad_g <- matrix(0, 2, 2)
for (t in 3:1) {
  g <- (1 - h_g[[t + 1]]^2) * dL_dh # g_t = D_t dL/dh_t
  grad_g <- grad_g + g %*% t(h_g[[t]])
  dL_dh <- t(Wh_g) %*% g # dL/dh_{t-1}
}

eps <- 1e-6
num_g <- matrix(0, 2, 2)
for (i in 1:2) for (j in 1:2) {
  d <- matrix(0, 2, 2); d[i, j] <- eps
  num_g[i, j] <- (sum(forward_g(Wh_g + d)[[4]]) - sum(forward_g(Wh_g - d)[[4]])) / (2 * eps)
}
max(abs(grad_g - num_g))

```

[1] 4.966662e-11

The two gradients agree to the accuracy of the finite difference.

Definition 8.1 (Backpropagation through time). **Backpropagation through time** (BPTT) is backpropagation on the unrolled network (Definition 7.4) [14]. It runs the forward pass over all T steps, storing the states, then passes $\frac{\partial L}{\partial h_t}$ backwards from $t = T$ to $t = 1$ by the rules of Theorem 8.3, and adds each step's contribution to the gradient of the shared weights. The cost grows linearly in T in both time and memory. When the loss also depends on the states at earlier steps, the direct term $\frac{\partial L_t}{\partial h_t}$ is added to $\frac{\partial L}{\partial h_t}$ at each step.

Example 8.3 (Checking BPTT against finite differences). Take the toy layer of Equation 3 on the first 10 days, and the loss $L = h_{10,1}$, the first coordinate of the last state. Compute $\frac{\partial L}{\partial W_h}$ by BPTT, and compare it with a central finite difference in each of the four entries of W_h .

8.5 Python

```

def last_state(z, W_h, w_x):
    return rnn_run(z, W_h, w_x)[-1]

def bptt_grad(z, W_h, w_x):
    states = rnn_run(z, W_h, w_x)
    prev = np.vstack([np.zeros(2), states[:-1]]) # h_{t-1} for each t
    dL_dh = np.array([1.0, 0.0]) # L = first coordinate of h_T
    grad = np.zeros_like(W_h)
    for t in range(len(z) - 1, -1, -1):
        g = (1 - states[t] ** 2) * dL_dh # g_t = D_t dL/dh_t
        grad += np.outer(g, prev[t]) # one outer product per step
        dL_dh = W_h.T @ g # dL/dh_{t-1}
    return grad

grad_bptt = bptt_grad(z_py[:10], W_h, w_x)
grad_fd = np.zeros((2, 2))
for i in range(2):
    for j in range(2):
        E_ij = np.zeros((2, 2))
        E_ij[i, j] = 1e-6
        grad_fd[i, j] = (last_state(z_py[:10], W_h + E_ij, w_x)[0]
                        - last_state(z_py[:10], W_h - E_ij, w_x)[0]) / 2e-6
print(grad_bptt.round(5))

[[ 0.39435 -0.19094]
 [-0.13778 -0.01775]]

print(np.abs(grad_bptt - grad_fd).max())

2.6064077979226496e-11

```

8.6 R

```

bptt_grad_r <- function(z, W_h, w_x) {
  states <- rnn_run_r(z, W_h, w_x)
  prev <- rbind(0, states[-nrow(states), ])      # h_{t-1} for each t
  dL_dh <- c(1, 0)                               # L = first coordinate of h_T
  grad <- matrix(0, 2, 2)
  for (t in rev(seq_along(z))) {
    g <- (1 - states[t, ]^2) * dL_dh             # g_t = D_t dL/dh_t
    grad <- grad + outer(g, prev[t, ])           # one outer product per step
    dL_dh <- drop(t(W_h) %*% g)                  # dL/dh_{t-1}
  }
  grad
}

last_state_r <- function(z, W_h, w_x) rnn_run_r(z, W_h, w_x)[length(z), ]
grad_bptt_r <- bptt_grad_r(z_r[1:10], W_h_r, w_x_r)
grad_fd_r <- matrix(0, 2, 2)
for (i in 1:2) {
  for (j in 1:2) {
    E_ij <- matrix(0, 2, 2)
    E_ij[i, j] <- 1e-6
    grad_fd_r[i, j] <- (last_state_r(z_r[1:10], W_h_r + E_ij, w_x_r)[1] -
                       last_state_r(z_r[1:10], W_h_r - E_ij, w_x_r)[1]) / 2e-6
  }
}
round(grad_bptt_r, 5)

```

```

      [,1]      [,2]
[1,]  0.39435 -0.19094
[2,] -0.13778 -0.01775

```

```
max(abs(grad_bptt_r - grad_fd_r))
```

```
[1] 8.157523e-11
```

The two methods agree to 8e-11. BPTT needed one backward pass, whereas the finite differences needed two extra forward passes for each of the four entries of W_h .

i Remark (Repairs for exploding gradients)

Remark 8.1 (Repairs for exploding gradients). When the gradient explodes, gradient clipping rescales it before the update. Clipping does not help a vanishing gradient, which is too small to rescale usefully. Initializing W_h near an orthogonal matrix keeps $\|W_h\|$ near 1 at the start of training, as the initializations of the backpropagation chapter do for fully connected layers. The more lasting fix changed the architecture, as the next section describes.

9 Gates

Exercise 9.1 (A path for memory that does not decay). In Equation 4, the old state $\tilde{h}_{t-1}$ passes through W_h and $\tanh$ before it reaches $\tilde{h}_t$. Design a second vector $\tilde{c}_t \in \mathbb{R}^H$, updated from $\tilde{c}_{t-1}$ and a new candidate u_t , so that the network can choose, coordinate by coordinate, to keep an old value unchanged, erase it, or overwrite it.

Solution

Solution 9.1. Multiply the old value by a number between 0 and 1 that the network computes at each step, and add a new contribution scaled the same way:

$$\tilde{c}_t = \tilde{f}_t \odot \tilde{c}_{t-1} + \tilde{i}_t \odot \tilde{u}_t,$$

where $\odot$ multiplies coordinate by coordinate, and $\tilde{f}_t$ and $\tilde{i}_t$ have all entries in $(0, 1)$. A coordinate of $\tilde{f}_t$ near 1 keeps the old value and near 0 erases it, and a coordinate of $\tilde{i}_t$ near 1 admits the new candidate and near 0 ignores it.

Exercise 9.2 (A path for memory that does not decay (part 2)). Use the setup of Exercise 9.1. What is $\frac{\partial \tilde{c}_t}{\partial \tilde{c}_{t-1}}$ in your design?

Solution

Solution 9.2. Along this path, $\frac{\partial \tilde{c}_t}{\partial \tilde{c}_{t-1}} = \text{diag}(\tilde{f}_t)$, with no weight matrix and no tanh in the way. If the network sets $\tilde{f}_t$ close to 1, the gradient passes through many steps almost unchanged.

Definition 9.1 (Gate). A **gate** is a vector with entries in $(0, 1)$.

Definition 9.2 (Long short-term memory (LSTM)). A **long short-term memory** (LSTM) layer keeps a cell state $\tilde{c}_t$ and a hidden state $\tilde{h}_t$, both in $\mathbb{R}^H$ [15]. Three gates (Definition 9.1) control the cell state:

$$\begin{aligned} \tilde{f}_t &= \text{expit}(W_f \tilde{x}_t + U_f \tilde{h}_{t-1} + \tilde{b}_f), & (\text{forget gate}) \\ \tilde{i}_t &= \text{expit}(W_i \tilde{x}_t + U_i \tilde{h}_{t-1} + \tilde{b}_i), & (\text{input gate}) \\ \tilde{o}_t &= \text{expit}(W_o \tilde{x}_t + U_o \tilde{h}_{t-1} + \tilde{b}_o), & (\text{output gate}) \\ \tilde{u}_t &= \tanh(W_c \tilde{x}_t + U_c \tilde{h}_{t-1} + \tilde{b}_c), & (\text{candidate}) \\ \tilde{c}_t &= \tilde{f}_t \odot \tilde{c}_{t-1} + \tilde{i}_t \odot \tilde{u}_t, \\ \tilde{h}_t &= \tilde{o}_t \odot \tanh(\tilde{c}_t). \end{aligned} \tag{5}$$

The cell state carries memory, and the output gate decides how much of it the hidden state shows. The layer has four weight blocks of the same shape as a simple recurrent layer's, so $4H(p + H + 1)$ parameters. The original LSTM [16] had no forget gate; it was added later [17], and modern libraries include it.

Example 9.1 (How much of the cell state survives thirty steps). Along the cell-state path, the derivative of $\tilde{c}_{30}$ with respect to $\tilde{c}_0$ is the product of the forget gates. Compute it for a coordinate whose forget gate is held at 0.5, 0.9 and 0.99 at every step.

9.1 Python

```
forget = np.array([0.5, 0.9, 0.99])
print(forget ** 30)

[9.31322575e-10 4.23911583e-02 7.39700373e-01]
```

9.2 R

```
forget_r <- c(0.5, 0.9, 0.99)
forget_r^30
```

```
[1] 9.313226e-10 4.239116e-02 7.397004e-01
```

A forget gate of 0.5 leaves 9.3e-10 of the signal, which is a vanished gradient. A gate of 0.99 leaves 0.74. The network learns its gates, so it can hold them near 1 for a coordinate that must remember. This is the direct cell-state path only: the gates also depend on $\tilde{h}_{t-1}$, which adds paths through the recurrent weights.

i Remark (The gated recurrent unit)

Remark 9.1 (The gated recurrent unit). The **gated recurrent unit** (GRU) [18] merges the cell and hidden states and uses two gates instead of three. An update gate $\tilde{z}_t$ and a reset gate $\tilde{r}_t$, both computed like $\tilde{f}_t$ in Equation 5, give

$$\underline{u}_t = \tanh(W\underline{x}_t + U(\tilde{r}_t \odot \tilde{h}_{t-1}) + \underline{b}), \quad \underline{h}_t = (1 - \tilde{z}_t) \odot \tilde{h}_{t-1} + \tilde{z}_t \odot \underline{u}_t.$$

The gate $\tilde{z}_t$ plays the role of both $\tilde{f}_t$ and $\tilde{i}_t$ of the LSTM, tied so that they sum to one. Sources differ on whether $\tilde{z}_t$ or $1 - \tilde{z}_t$ multiplies the old state. A GRU has three weight blocks, so $3H(p + H + 1)$ parameters.

10 Where this goes next

Gated recurrent layers read a sequence one step at a time, so step t cannot start before step $t - 1$ finishes, and information from step 1 reaches step T only through $T - 1$ updates of the state. Attention and Transformers replaces the recurrence: every position reads every other position directly, and all positions are computed at once. Transformers have replaced recurrent networks in most language models, and convolutional layers remain standard for images, audio and other data on a grid.

11 Further reading

- Goodfellow et al. [1], Chapter 9 (convolutional networks) and Chapter 10 (sequence modeling)
- James et al. [2], Chapter 10, which covers convolutional and recurrent networks; James et al. [5] has the Python version

i Watch

Andrew Ng's lectures for DeepLearning.AI, on convolutional networks, cover: convolution as edge detection [19]; padding [20]; strided convolutions [21]; pooling layers [22]; and why convolutions suit images [23].

References

- [1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. Available: <https://www.deeplearningbook.org/>
- [2] 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/>
- [3] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning." 2016. Available: <https://arxiv.org/abs/1603.07285>

- [4] E. Alpaydin and C. Kaynak, “Optical recognition of handwritten digits.” UCI Machine Learning Repository, 1998. doi: 10.24432/C50P49¹.
- [5] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An introduction to statistical learning: With applications in Python*. Springer, 2023. doi: 10.1007/978-3-031-38747-0².
- [6] Wikipedia contributors, “Convolutional neural network — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network
- [7] 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>
- [8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791³.
- [9] J. L. Elman, “Finding structure in time,” *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990, doi: 10.1207/s15516709cog1402_1⁴.
- [10] Wikipedia contributors, “Recurrent neural network — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network
- [11] J. Starmer, “Recurrent neural networks (RNNs), clearly explained.” StatQuest, video on YouTube, 2022. Available: <https://www.youtube.com/watch?v=AsNTP8Kwu80>
- [12] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994, doi: 10.1109/72.279181⁵.
- [13] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *Proceedings of the 30th international conference on machine learning*, 2013, pp. 1310–1318. Available: <https://arxiv.org/abs/1211.5063>
- [14] Wikipedia contributors, “Backpropagation through time — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Backpropagation_through_time
- [15] Wikipedia contributors, “Long short-term memory — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Long_short-term_memory
- [16] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735⁶.
- [17] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with LSTM,” *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000, doi: 10.1162/089976600300015015⁷.
- [18] K. Cho *et al.*, “Learning phrase representations using RNN encoder–decoder for statistical machine translation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing*, 2014, pp. 1724–1734.
- [19] A. Ng, “Edge detection examples.” DeepLearning.AI, Deep Learning Specialization, Course 4; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=XuD4C8vJzEQ>
- [20] A. Ng, “Padding.” DeepLearning.AI, Deep Learning Specialization, Course 4; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=smHa2442Ah4>
- [21] A. Ng, “Strided convolutions.” DeepLearning.AI, Deep Learning Specialization, Course 4; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=tQYZaDn_kSg
- [22] A. Ng, “Pooling layers.” DeepLearning.AI, Deep Learning Specialization, Course 4; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=8oOgPUO-TBY>
- [23] A. Ng, “Why convolutions?” DeepLearning.AI, Deep Learning Specialization, Course 4; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=ay3zYUeuyhU>

¹<https://doi.org/10.24432/C50P49>

²<https://doi.org/10.1007/978-3-031-38747-0>

³<https://doi.org/10.1109/5.726791>

⁴https://doi.org/10.1207/s15516709cog1402_1

⁵<https://doi.org/10.1109/72.279181>

⁶<https://doi.org/10.1162/neco.1997.9.8.1735>

⁷<https://doi.org/10.1162/089976600300015015>