Deep learning 2

Parameter sharing: Convolutional and Recurrent neural networks; attention

Daniel Oberski, Robert Bagheri

Today

- Convolutional neural networks
- Recurrent neural networks: one network, applied step by step
  - what you can do with them (one-to-many, many-to-one, …)
  - the recurrence, unrolled in time; encoder–decoder
- Why plain RNNs forget, and how LSTMs remember
- Attention: looking back at every step

Lab (Thursday)

MNIST digits read as a sequence of 28 rows: feed-forward → SimpleRNN → LSTM → LSTM + attention, in Keras (Python)

Reading: ISLR 10; Goodfellow et al. ch. 11

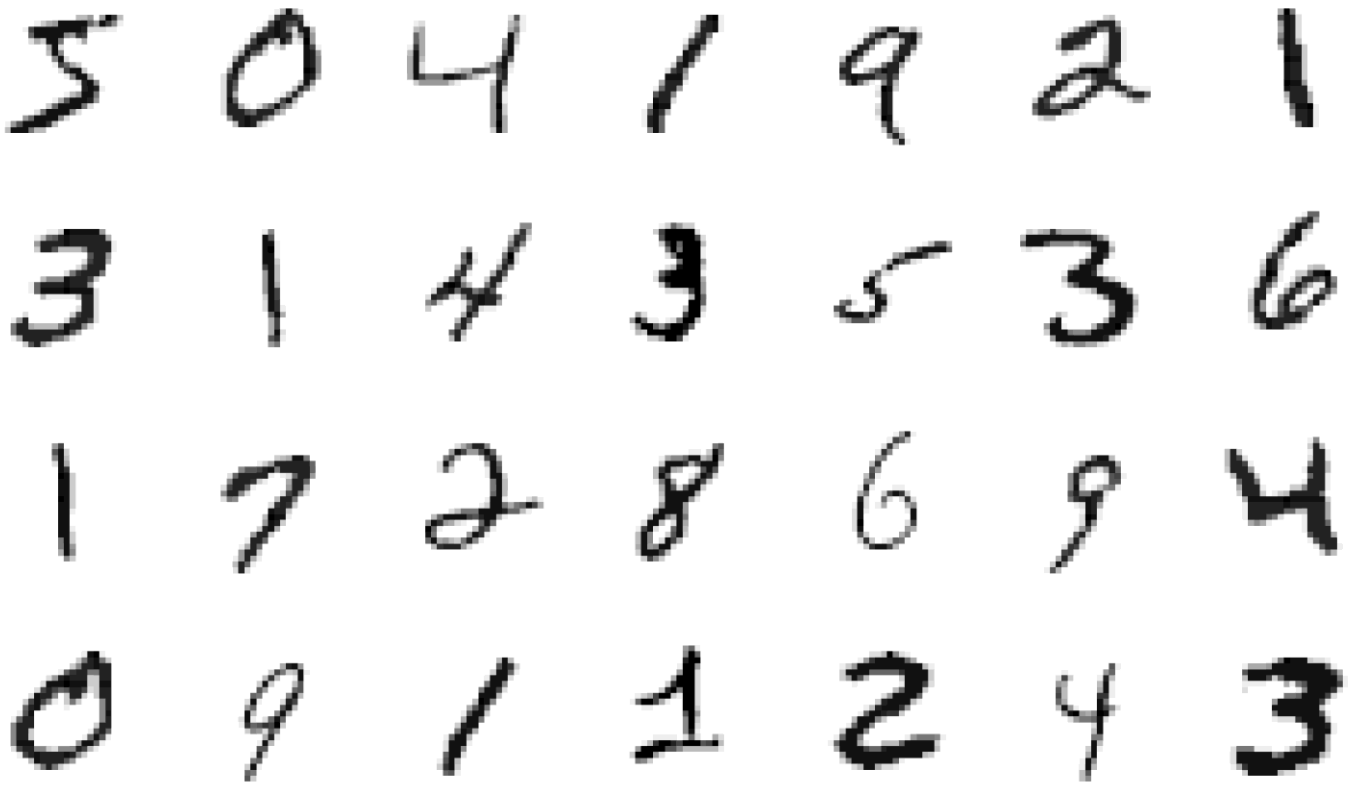
The big idea: **Parameter sharing**

- Feed forward neural networks (FNNs) have a lot of parameters -> unfavorable bias-variance tradeoff?
- With enough data, anything is possible… but “enough” could be more than there is!
- We can often do better by leveraging known structure to reduce the number of parameters
- One method of doing this is **parameter sharing**
- We’ll look at **two** types of neural networks that take advantage of the specific form of a task to do much better than FNN by using parameter sharing

Convolutional neural networks (CNN)

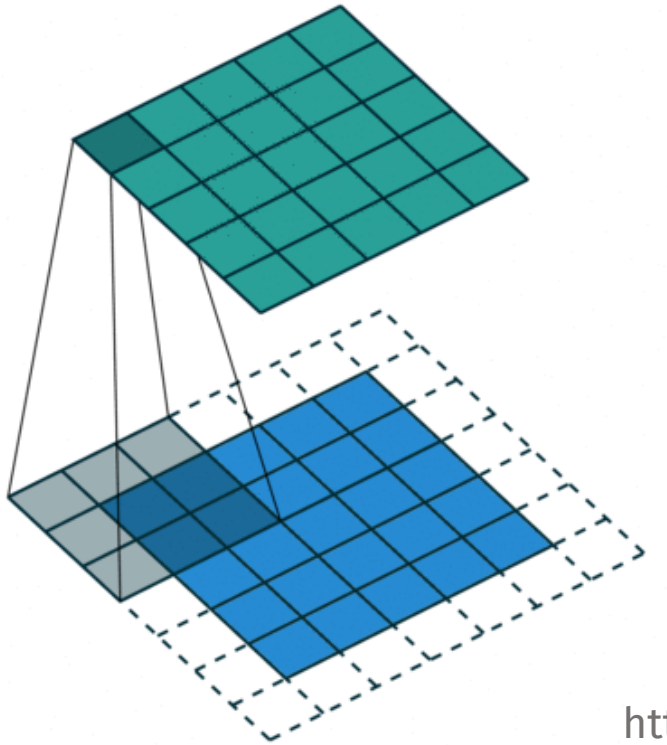
“Hello world” of neural networks

- MNIST (Modified National Institute of Standards and Technology)
- Handwritten digits
- 28 * 28 pixels
- 60 000 training images and 10 000 testing images



What is a convolution

- Convolution is applying a **kernel** (filter) over an image
- Filter defines which **feature** is important in the image



https://github.com/vdumoulin/conv_arithmetic

What is a convolution

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}.$$

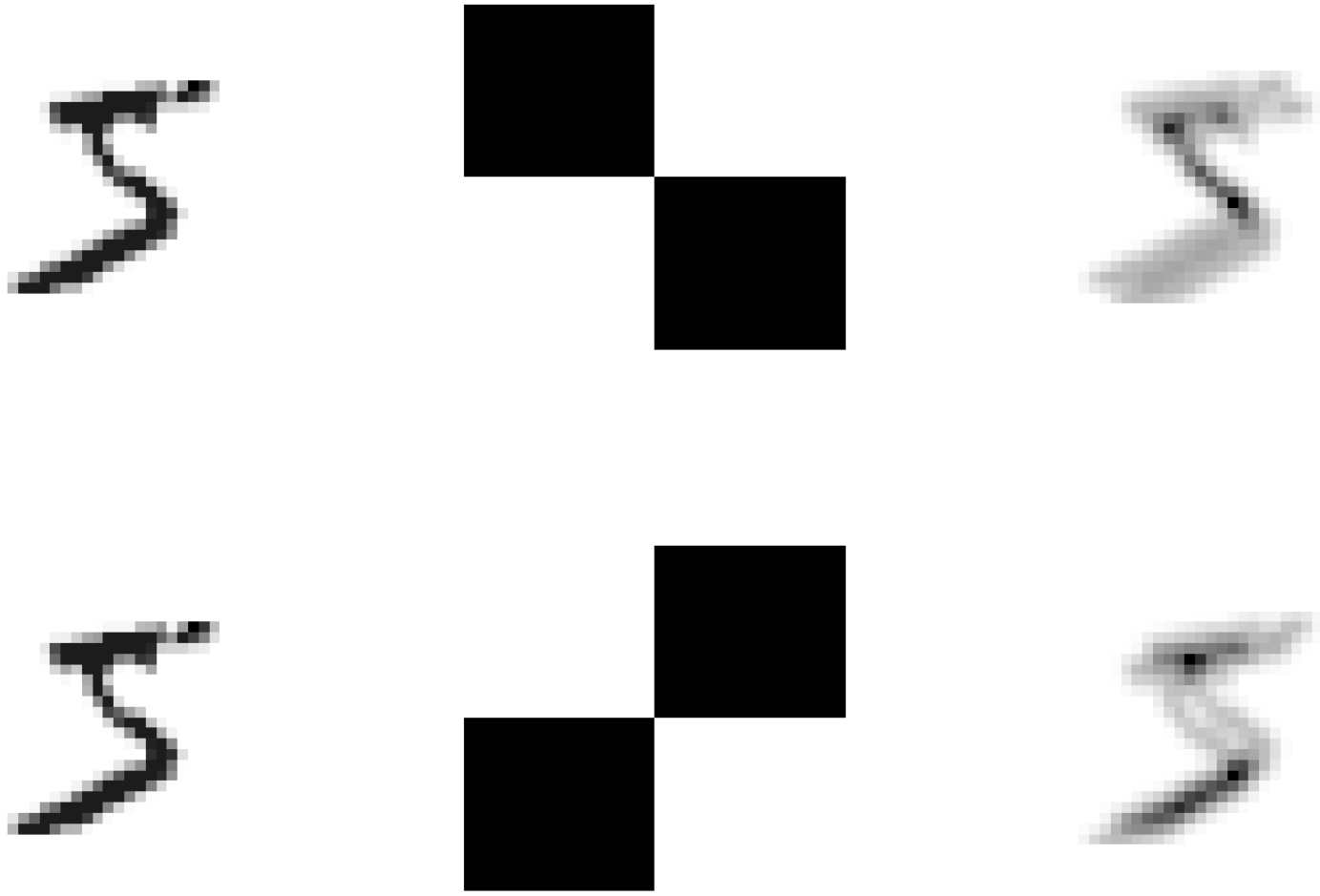
Now consider a 2×2 filter of the form

$$\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

When we *convolve* the image with the filter, we get the result[8]

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}.$$

Detecting diagonal lines with convolution



Convolution layers

- A convolutional neural network is a NN with one or more **convolution layers**
- The parameters / weights in a convolution layer are the elements of the filter
- The filter is **learnt** by the network!

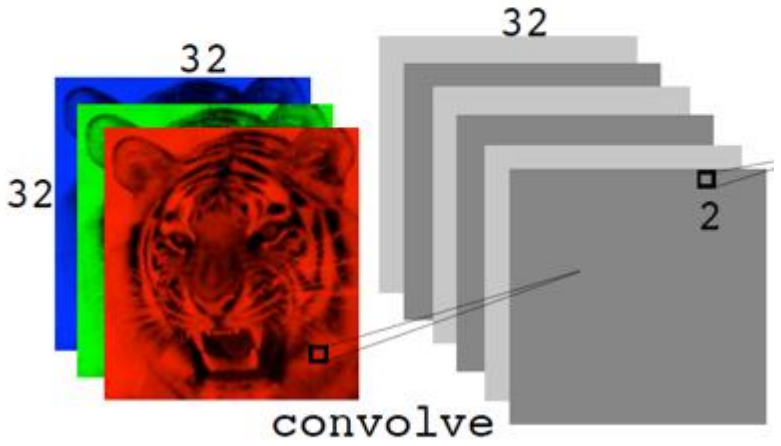


FIGURE 10.8. *Architectu*
Convolution layers are inte
size by a factor of 2 in both

Convolution layers

- In each convolution layer, you can create multiple filters
- Number of parameters is function of:
  - Number of filters (e.g. 6)
  - Size of each filter (e.g. 2x2)
  - NOT the input dimension!
- **Parameter sharing**

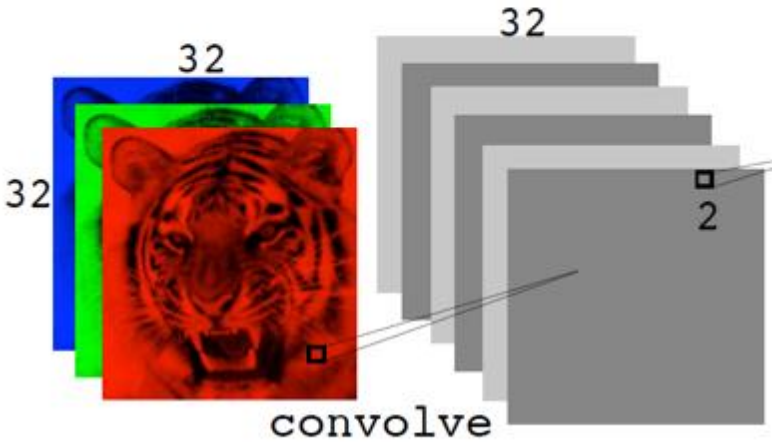


FIGURE 10.8. *Architectu*
Convolution layers are inte
size by a factor of 2 in both

Pooling layer

- Convolution layers are usually followed by a **pooling layer**
- Reduces dimensionality
- **Location invariance**: Robustness against pixel shift / small rotations
- **Max pool** most common

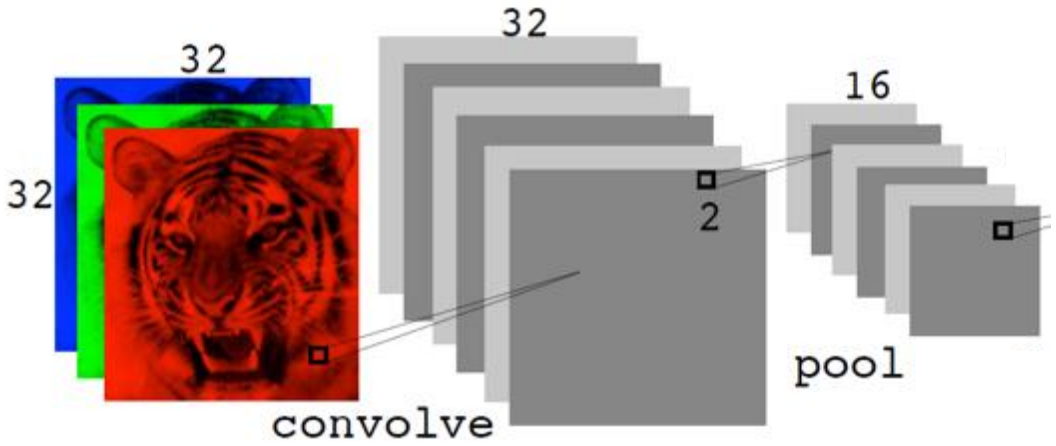


FIGURE 10.8. *Architecture of a de*
Convolution layers are interspersed v
size by a factor of 2 in both dimensio

Pooling layer

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}.$$

Architecture of a CNN

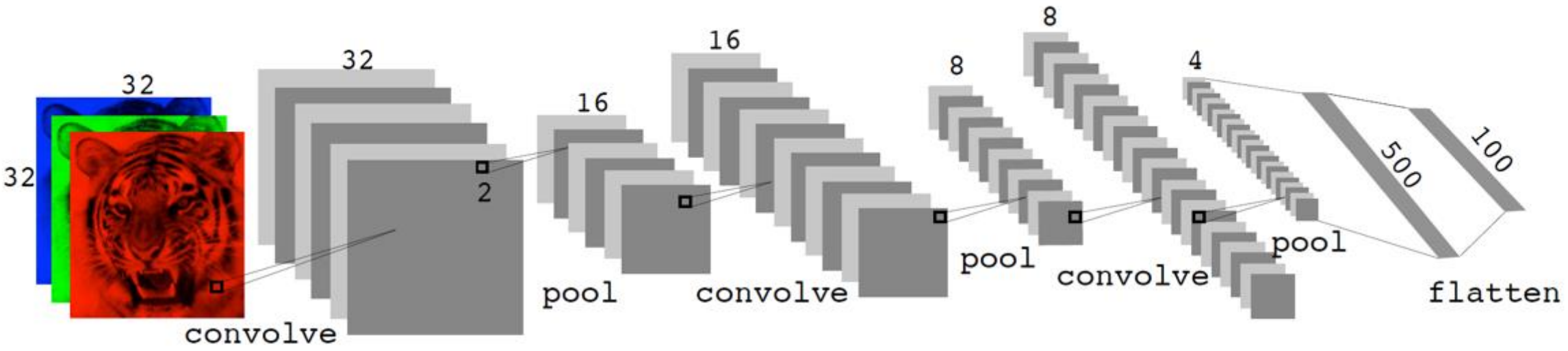


FIGURE 10.8. *Architecture of a deep CNN for the* `CIFAR100` *classification task. Convolution layers are interspersed with* 2×2 *max-pool layers, which reduce the size by a factor of 2 in both dimensions.*

Applying CNN to MNIST

```
model_cnn <-
  keras_model_sequential(input_shape = c(28, 28, 1)) %>%
  layer_conv_2d(6, c(5, 5)) %>%
  layer_max_pooling_2d(pool_size = c(4, 4)) %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(10, activation = "softmax")
```

Model summary

```
summary(model_cnn)
Layer (type)                                 Output Shape                              Param #
=================================================================================================
conv2d_2 (Conv2D)                            (None, 24, 24, 6)                         156
_________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)               (None, 6, 6, 6)                           0
_________________________________________________________________________________________________
flatten_3 (Flatten)                          (None, 216)                               0
_________________________________________________________________________________________________
dense_8 (Dense)                              (None, 32)                                6944
_________________________________________________________________________________________________
dense_7 (Dense)                              (None, 10)                                330
=================================================================================================
Total params: 7,430
Trainable params: 7,430
Non-trainable params: 0
```

Compare to feed-forward model

```
summary(model_dff)

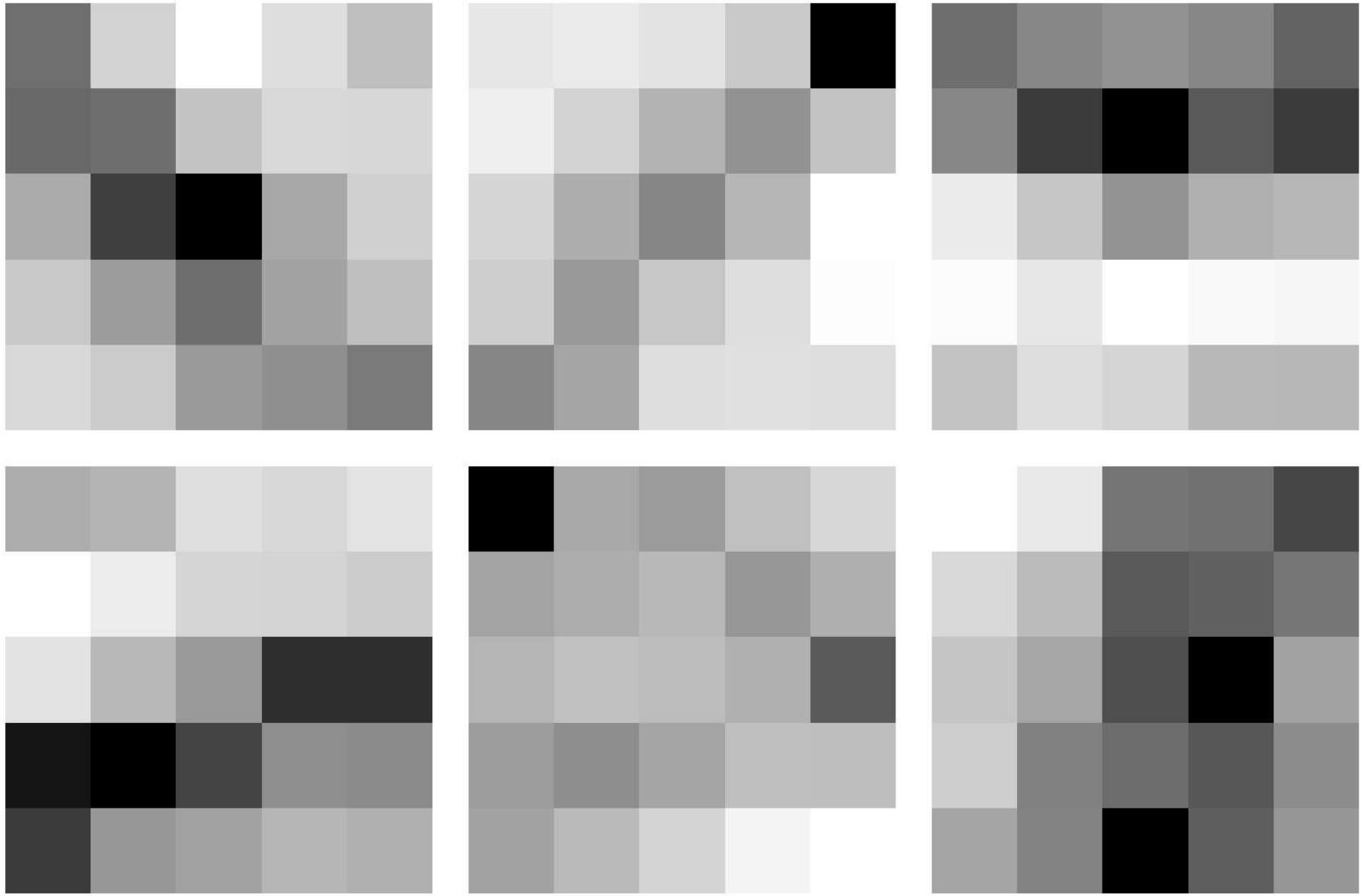
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               200960
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
_________________________________________________________________
```

What about the features?

Unboxing the black box

- Extract the weights of the convolution layer to find the features (filters) that were learnt
- Apply the filters to some example images to get an idea of which features are discriminative for the different numbers

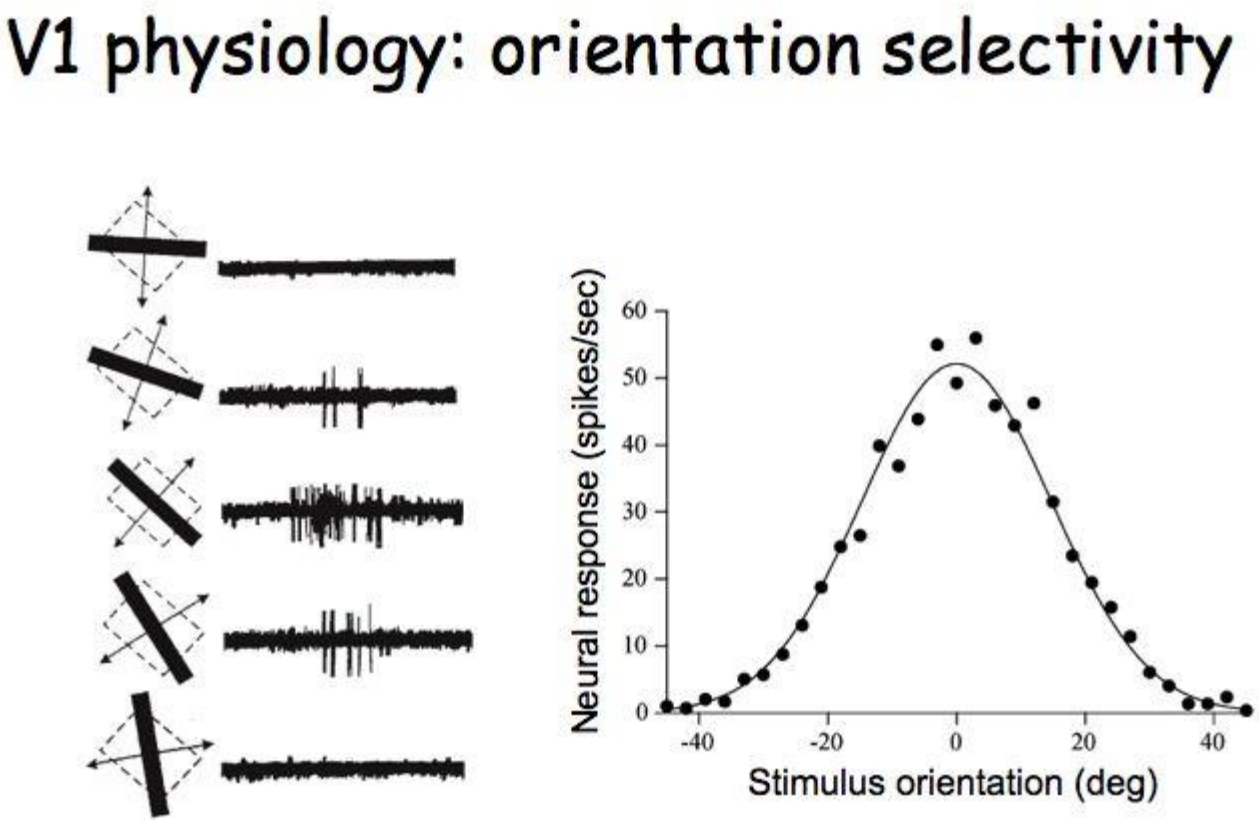
(There are other advanced methods, like layerwise relevance propagation, shapley values, ...)



Similarity to visual brain area

These learnt filters are similar to monkey visual area 1 (V1) neuron sensitivities

http://www.cns.nyu.edu/~david/courses/perception/lecturenotes/V1/lgn-V1.html



Hubel & Wiesel, 1968

Pretrained CNNs

- Download the convolutional layer weights from existing neural network trained on many images
- Apply them to your own images
- Result: a feature vector per image
- Use these feature vectors as input dataset for any kind of supervised task
- This can work really well!

Summary: CNN

- Convolution = applying kernel (filter) over an image
- CNNs employ convolution layers
  - Parameter sharing
  - Feature detection
- Followed by pooling layers
  - location invariance
- Use pretrained networks as a quick proxy

Recurrent neural networks (RNN)

Sequences need a different kind of network

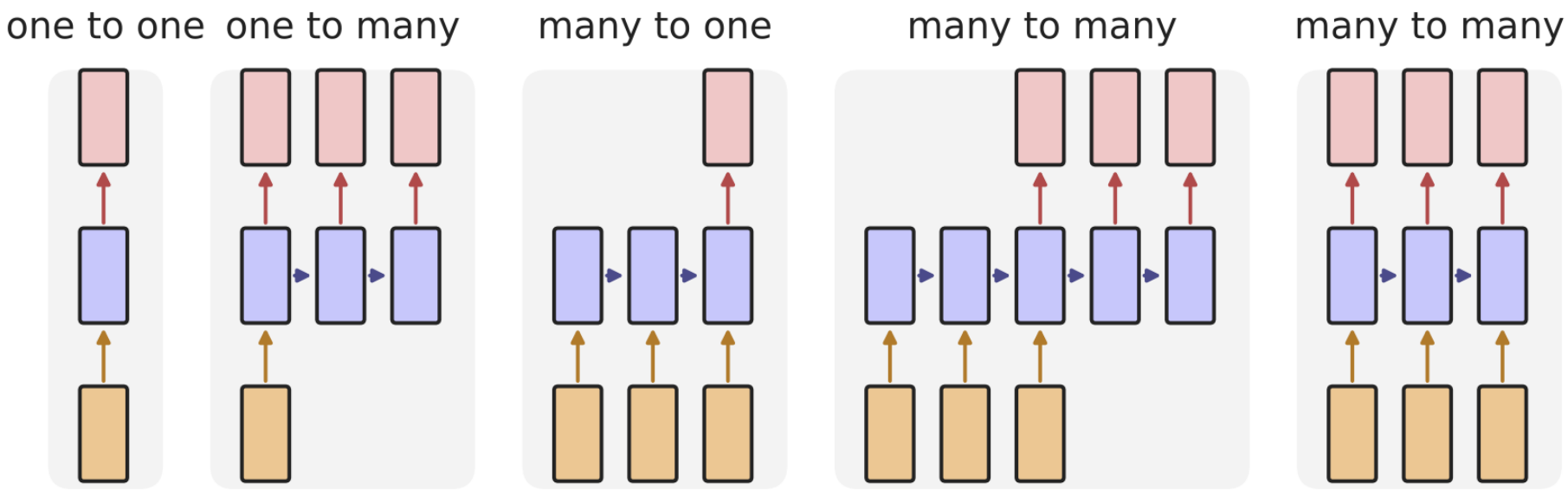
- So far: inputs of a fixed size (28 × 28 pixels, p features)
- Many data come as sequences of varying length: text, speech, sensor readings, a patient's visits, transactions
- *"I went to Nepal in 2009" vs "In 2009, I went to Nepal"*
  - a feed-forward net has separate weights for every position: it would need to learn the same rule at every position
- **Recurrent neural network (RNN): apply the same small network to each element in turn, carrying a state along**
  - parameter sharing across time, like a CNN shares its filter across the image

In the lab: an image is a sequence too

a 28 × 28 digit = 28 rows (time steps) of 28 pixel values (features), read top to bottom: $x_1 \ \ldots \ x_{28}$

```
x_train.shape
# (60000, 28, 28)
# = (samples, steps, features)
```

What you can do with an RNN



one to one
the networks so far:
image → class

one to many
image captioning:
image → sentence

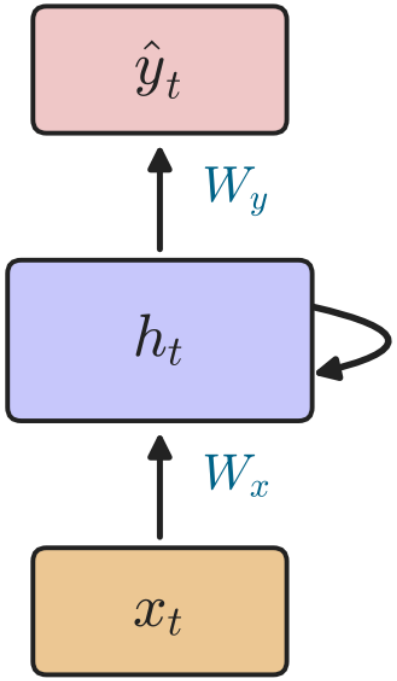
many to one
sentiment: review → score
lab: 28 rows → digit

many to many
translation:
sentence → sentence

many to many
a label per step:
video frames, next word

orange = input, purple = hidden state, red = output

The recurrence: one step, repeated



new state

$$h_t = g\left(W_x x_t + W_h h_{t-1} + b\right)$$

output when needed

$$\hat{y}_t = f\left(W_y h_t + b_y\right)$$

same W_x, W_h, W_y at every step

- h_t: the state, a lossy summary of $x_1 \ \ldots \ x_t$
- the same function and the same weights at every step: that is what "recurrent" means
- compare lecture 1: a hidden layer $h = g(Wx + b)$. An RNN layer is the same, with one extra input: its own previous value
- $g = \tanh$ (or ReLU); $f = \text{softmax}$ for a class, identity for a number

Written out: a plain (“vanilla”) RNN

Lecture 1: a hidden layer

$$h = g(Wx + b)$$
$$\hat{y} = f(W_y h + b_y)$$

An RNN layer: the same, plus its own previous state

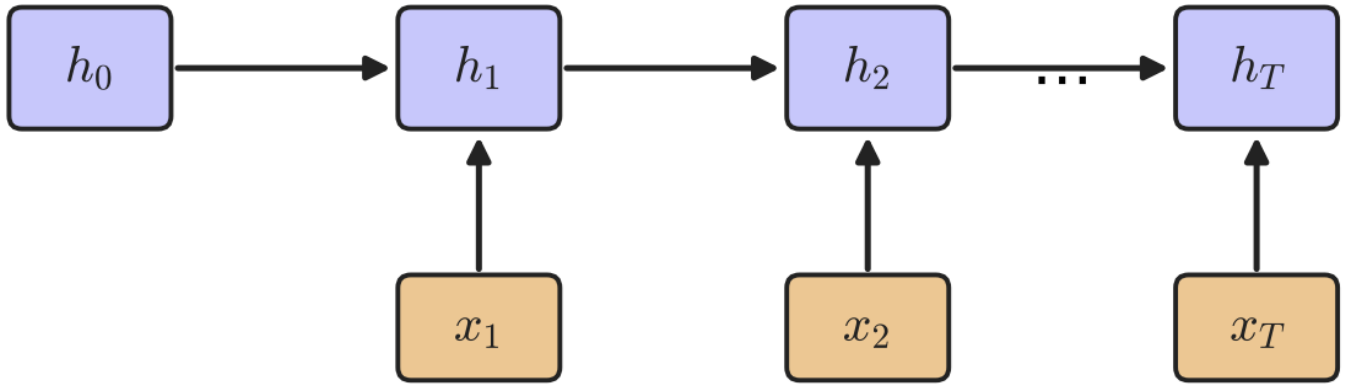
$$h_t = g(W_x x_t + W_h h_{t-1} + b)$$
$$\hat{y}_t = f(W_y h_t + b_y)$$

- W_x: input → state (units × features)
- W_h: state → state (units × units)
- W_y: state → output (classes × units)
- $g = \tanh$, $f = \text{softmax}$; $h_0 = 0$
- **the number of parameters does not depend on the sequence length**

```
keras.layers.SimpleRNN(64)
# 28 features -> 64 units:
# W_x 64x28 + W_h 64x64 + b 64
# = 5,952 parameters
```

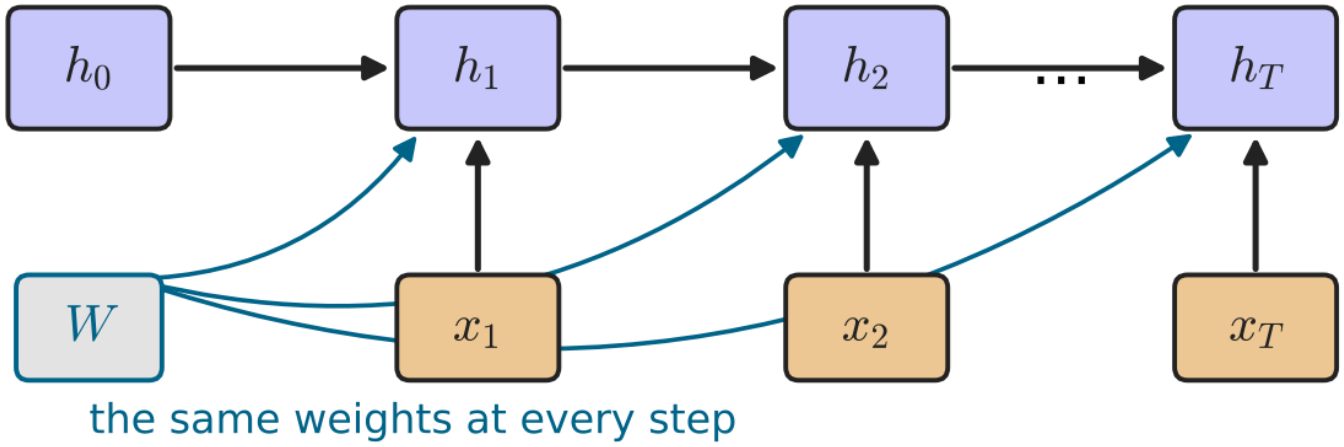
Other books write W_{xh}, W_{hh}, W_{hy} or U, W, V for the same three matrices

Unrolling the RNN in time



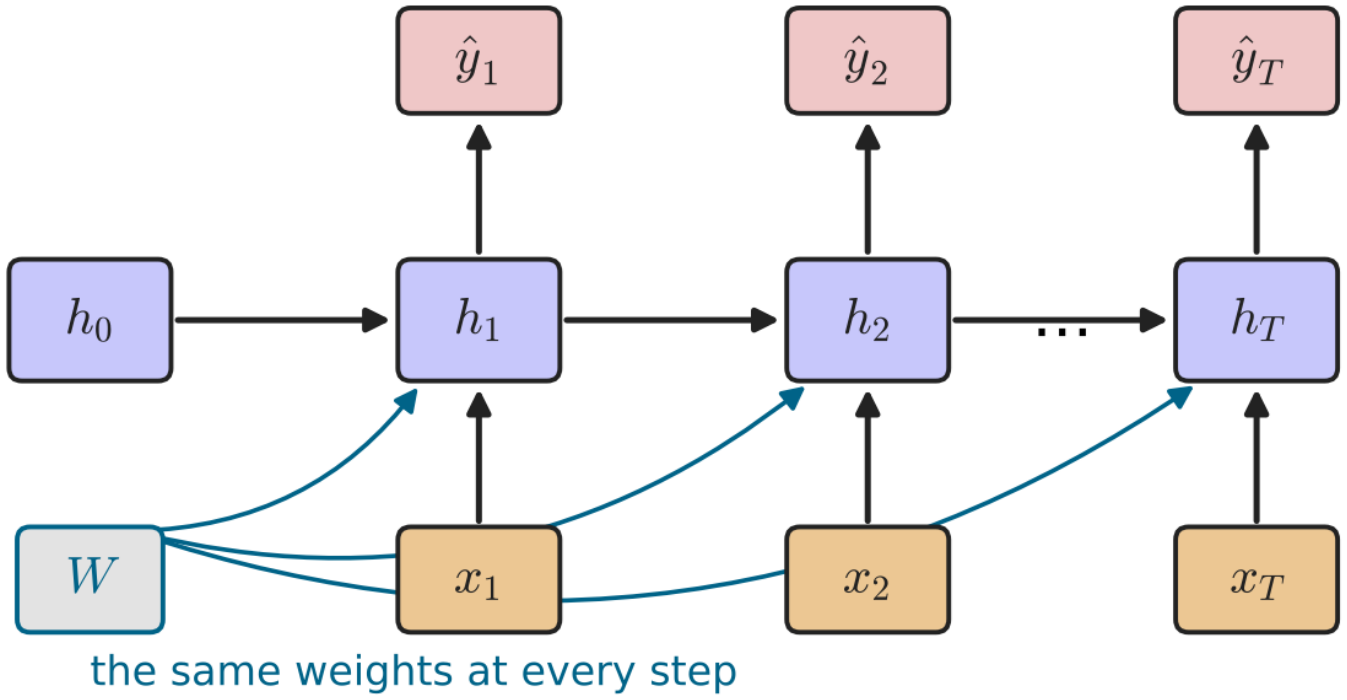
$h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow$ ...: the same step, repeated

Unrolling the RNN in time



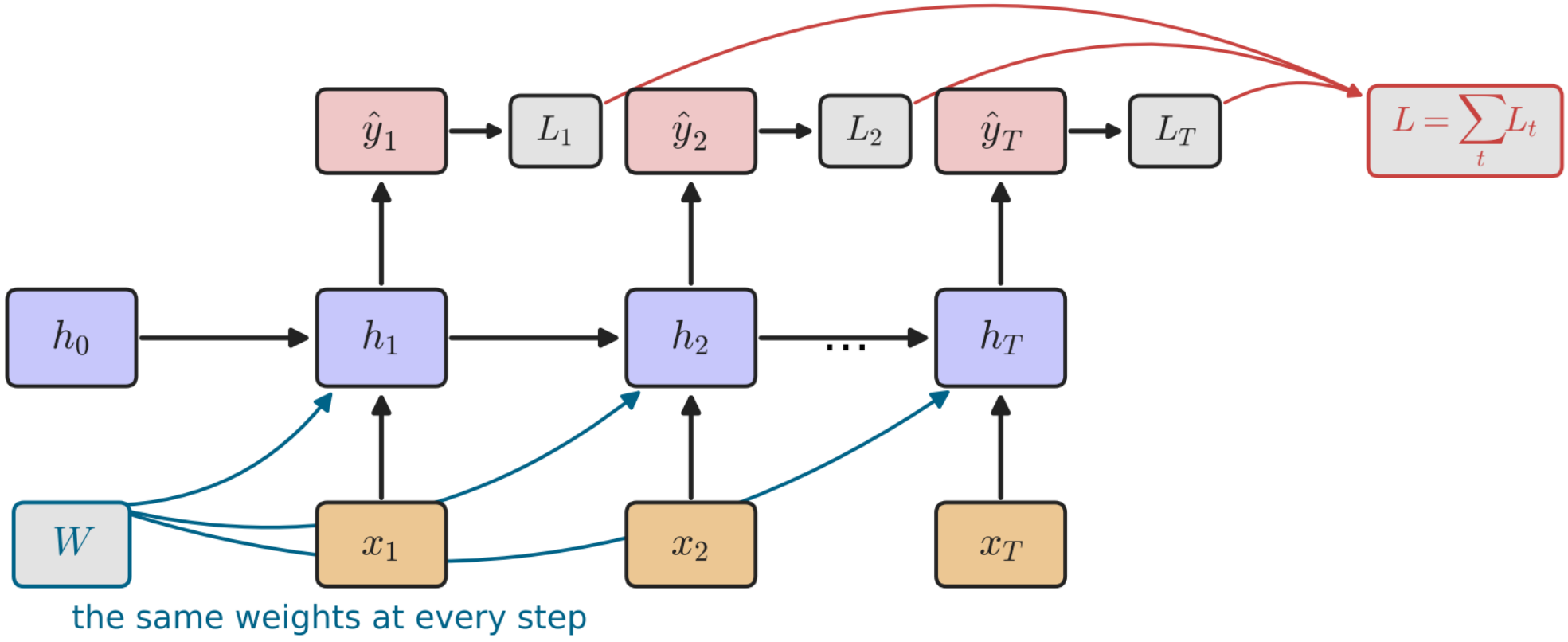
one set of weights $W = (W_x, W_h, b)$, reused at every step

Unrolling the RNN in time



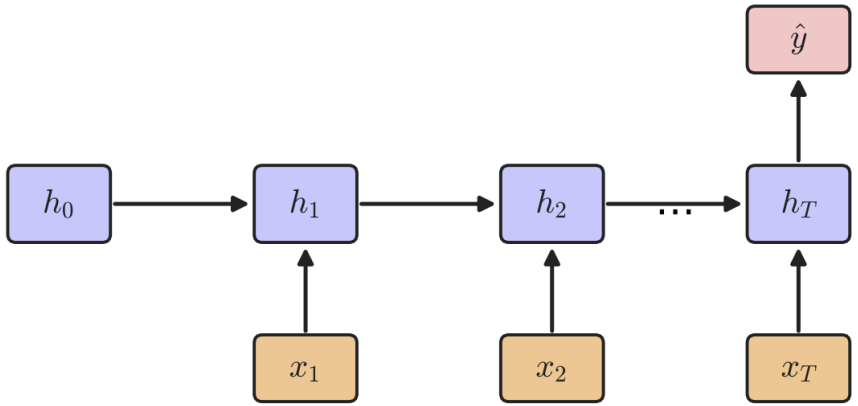
an output wherever we want one (here: every step)

Unrolling the RNN in time



a loss per output; total loss $L = \sum_t L_t$

Many to one: a decision after the whole sequence



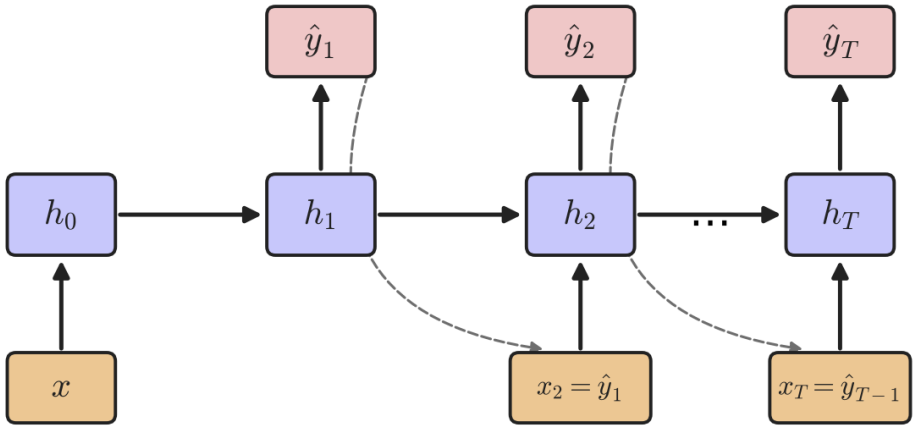
one output,
after the whole sequence
(sentiment; a digit from its 28 rows)

The lab, exercise 3

28 rows in, one digit out: the softmax reads only the last state h_{28}

```
model = keras.Sequential([
    keras.layers.Input(shape=(28, 28)),        # 28 steps x 28 features
    keras.layers.SimpleRNN(64),                # returns h_T only
    keras.layers.Dense(10, activation="softmax")])
```

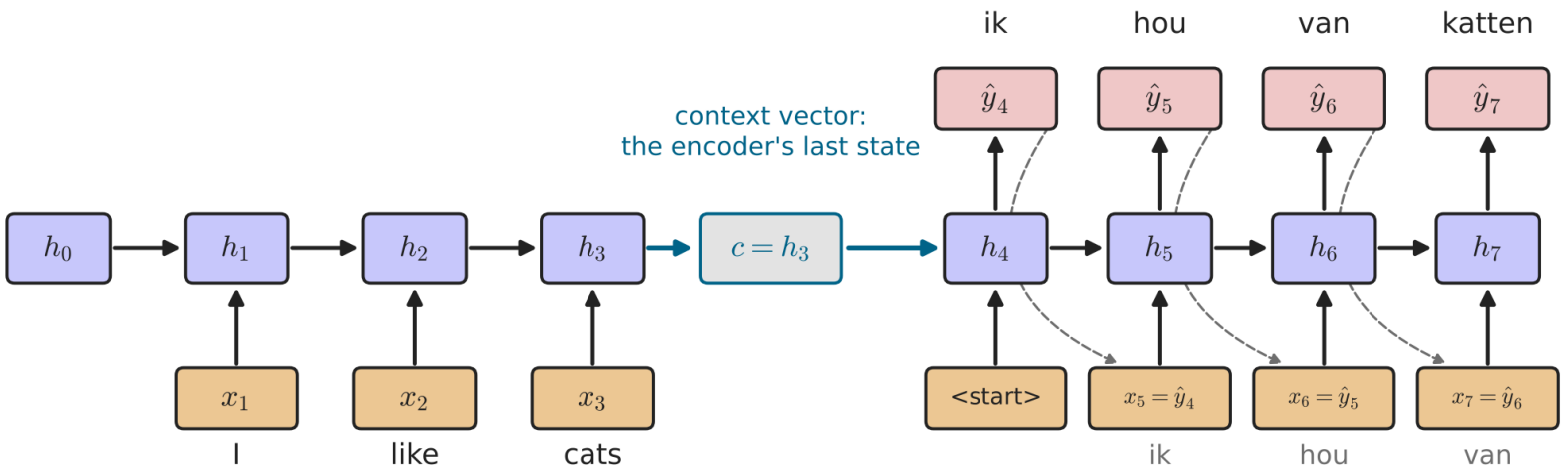
One to many: generate a sequence



one input, a sequence out
(image → caption)
dashed: each output becomes the next input

- one input (an image, or just a start signal)
- each output is fed back in as the next input: the network continues its own sequence
- image captioning, music, text generation

Encoder–decoder: many-to-one + one-to-many

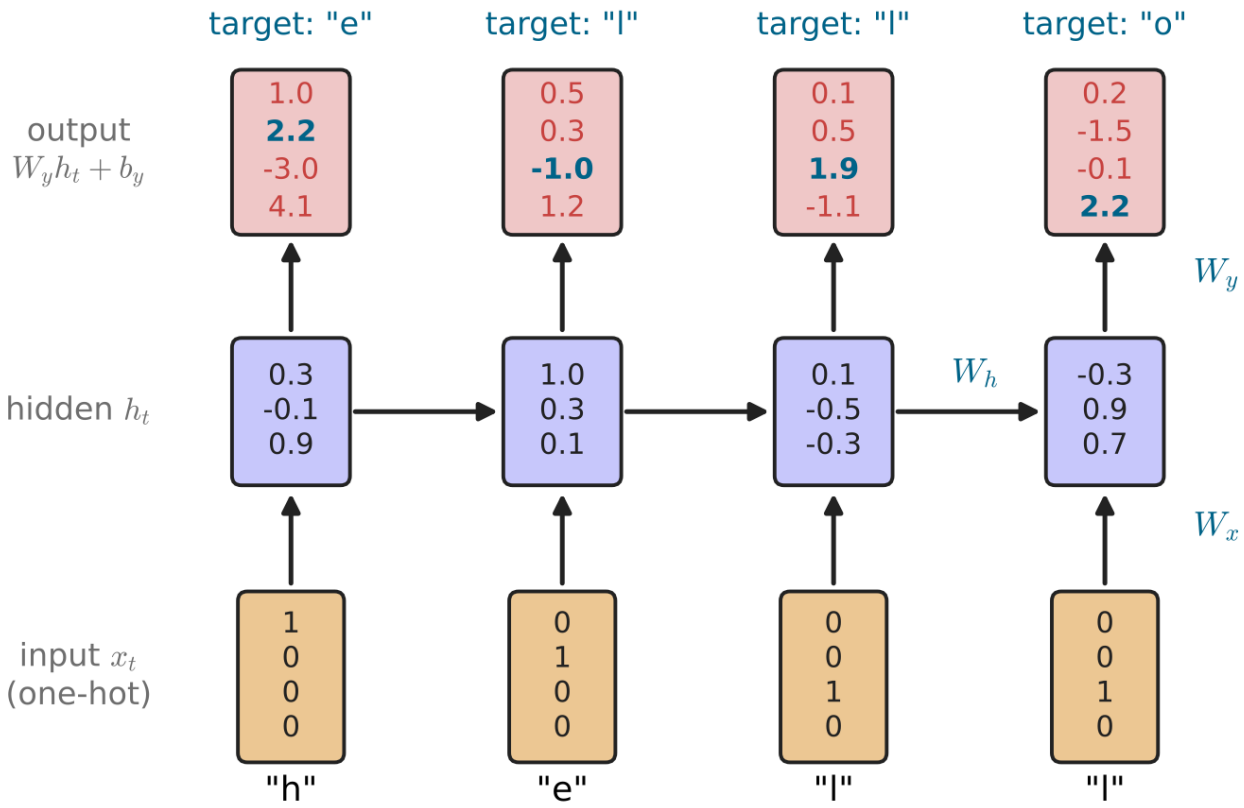


- encoder: the recurrence reads $x_1 \ \ldots \ x_T$; its last state is the context vector $c = h_T$, the summary of the input
- decoder: the recurrence continues from c (so $h_{T+1} = g(W_x \, x_{T+1} + W_h \, c + b)$), one word per step until an end symbol; each word is the next input
- training: cross-entropy per output word; the decoder is fed the correct previous word (“teacher forcing”)

Sequence to sequence (seq2seq)

translation, summarisation, chatbots — output length ≠ input length

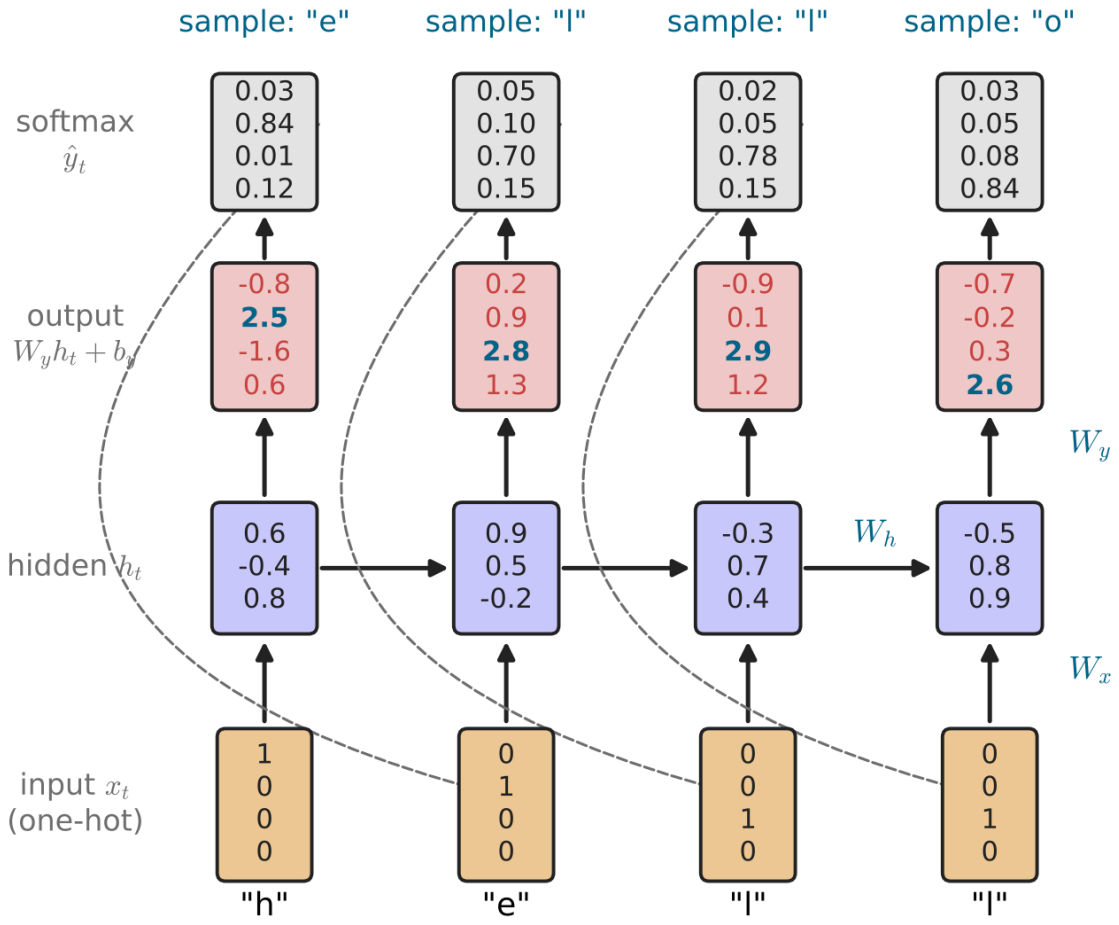
Example: a character-level language model



vocabulary [h, e, l, o]; training word “hello”, early in training

- vocabulary [h, e, l, o]; inputs one-hot
- target at every step: the next character
- scores $W_y\ h_t\ +\ b_y$ → softmax → cross-entropy, which pushes the score of the correct character (blue) up
- early in training the network is still wrong: at step 2 the correct “l” scores −1.0 while “o” scores 1.2 — that is what the loss will fix
- **many to many, with a loss at every step**
- the same recipe, with words instead of characters, is a language model (week 8)

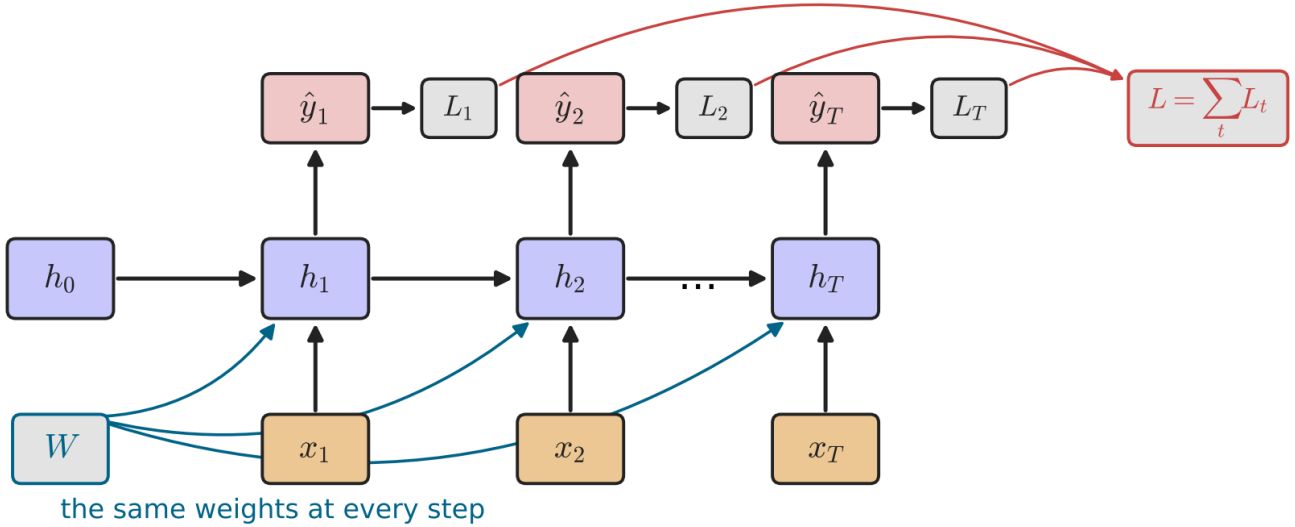
… and generating text from it



vocabulary [h, e, l, o]; after training, generating from “h”

- feed one character, get a probability distribution over the next
- draw a character from it and feed it back in (dashed)
- repeat: the network writes text one character at a time (here after training: the sampled characters spell “hello”)
- trained on enough text: plausible Shakespeare, LaTeX, code — the ancestor of today’s chat models

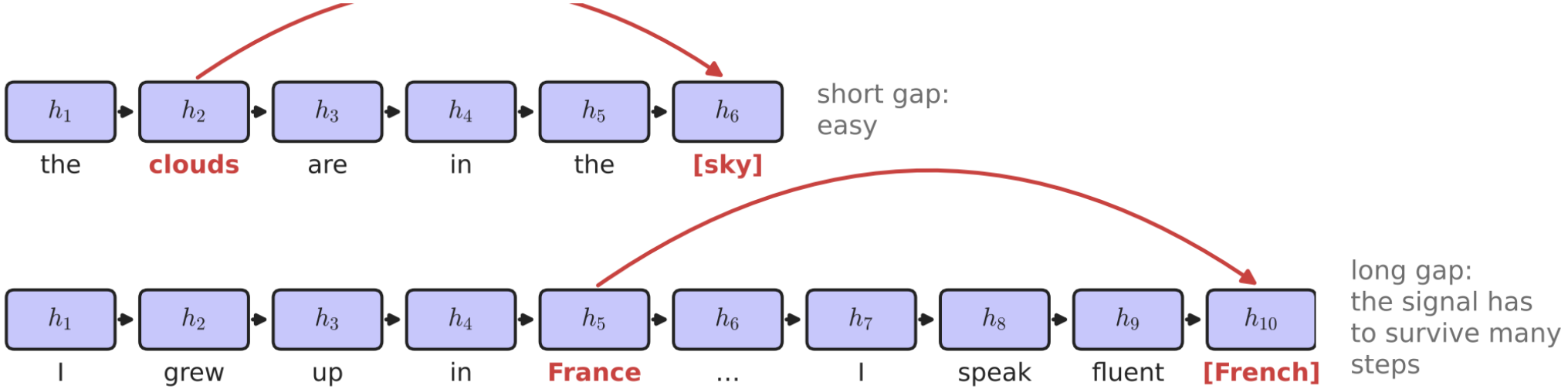
Training: backpropagation through time



- unroll → an ordinary (deep) feed-forward graph
- total loss $L = \sum_t L_t$
- gradient descent with backpropagation, as in lecture 1
- **the same W_h sits at every step: its gradient sums the contributions of all steps, each travelling back through all later steps**
- Keras: compile() and fit() as before

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam", metrics=["accuracy"])
model.fit(x_train, y_train, epochs=5, validation_split=0.2)
```

The problem: plain RNNs forget



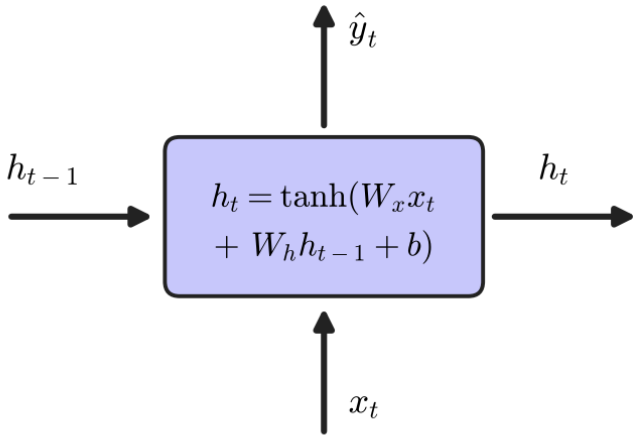
- the gradient from L_t back to h_k is a product of one factor per step (W_h times $\tanh'$): many factors < 1 → it vanishes; > 1 → it explodes
- **so a plain RNN cannot learn to use information from many steps back**

In practice

SimpleRNN: memory of ~10 steps; 28 rows of a digit already strain it (lab, exercises 3–4)

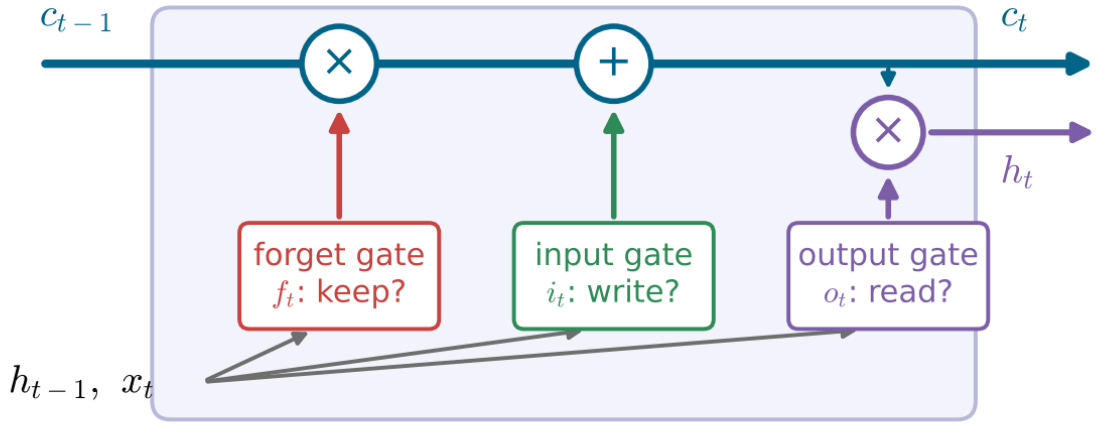
LSTM: a cell with memory and gates

SimpleRNN cell



one state, rewritten at every step

LSTM cell



a memory line c_t that information can travel along unchanged;
three gates (values 0–1) decide what to forget, write and read

- a memory line c_t that passes from step to step untouched, so gradients can travel along it too
- gates (sigmoid, 0–1) learn when to keep, write and read: the network decides what to remember

```
keras.layers.LSTM(64)   # instead
of SimpleRNN(64)
# ~4x the parameters; GRU: a
lighter cousin
```

LSTM in four lines (for reference)

gates (each a sigmoid, values in 0–1):

$$f_t = \sigma(W_f\,[h_{t-1}, x_t] + b_f) \qquad \text{forget: keep?}$$

$$i_t = \sigma(W_i\,[h_{t-1}, x_t] + b_i) \qquad \text{input: write?}$$

$$o_t = \sigma(W_o\,[h_{t-1}, x_t] + b_o) \qquad \text{output: read?}$$

memory and state:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_c\,[h_{t-1}, x_t] + b_c)$$

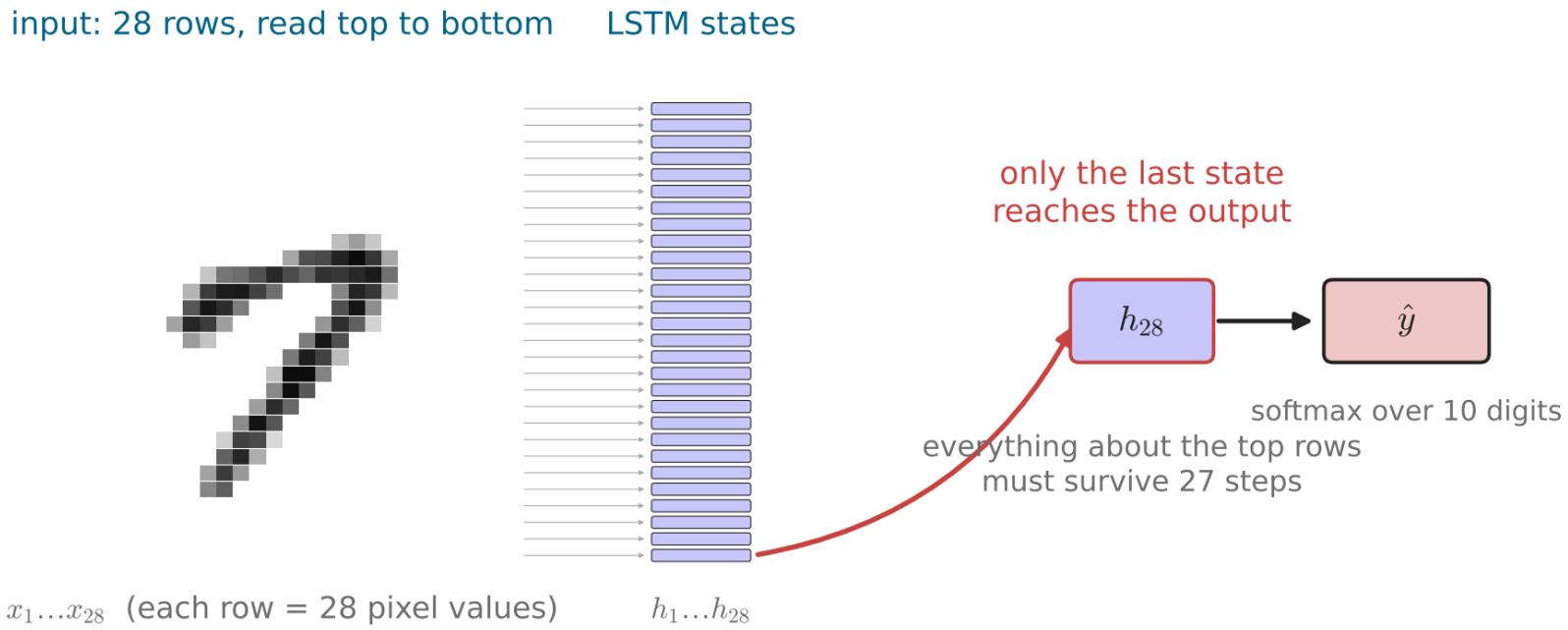
$$h_t = o_t \odot \tanh(c_t)$$

- $\odot$ = element-wise product; $[h, x]$ = the two vectors stacked
- keep (f), write (i), read (o): each gate has its own weights, learned like all the others
- $f = 1, i = 0$**: the memory passes through unchanged — that is the long-term memory**
- don't memorise; remember the picture

Hochreiter & Schmidhuber (1997)

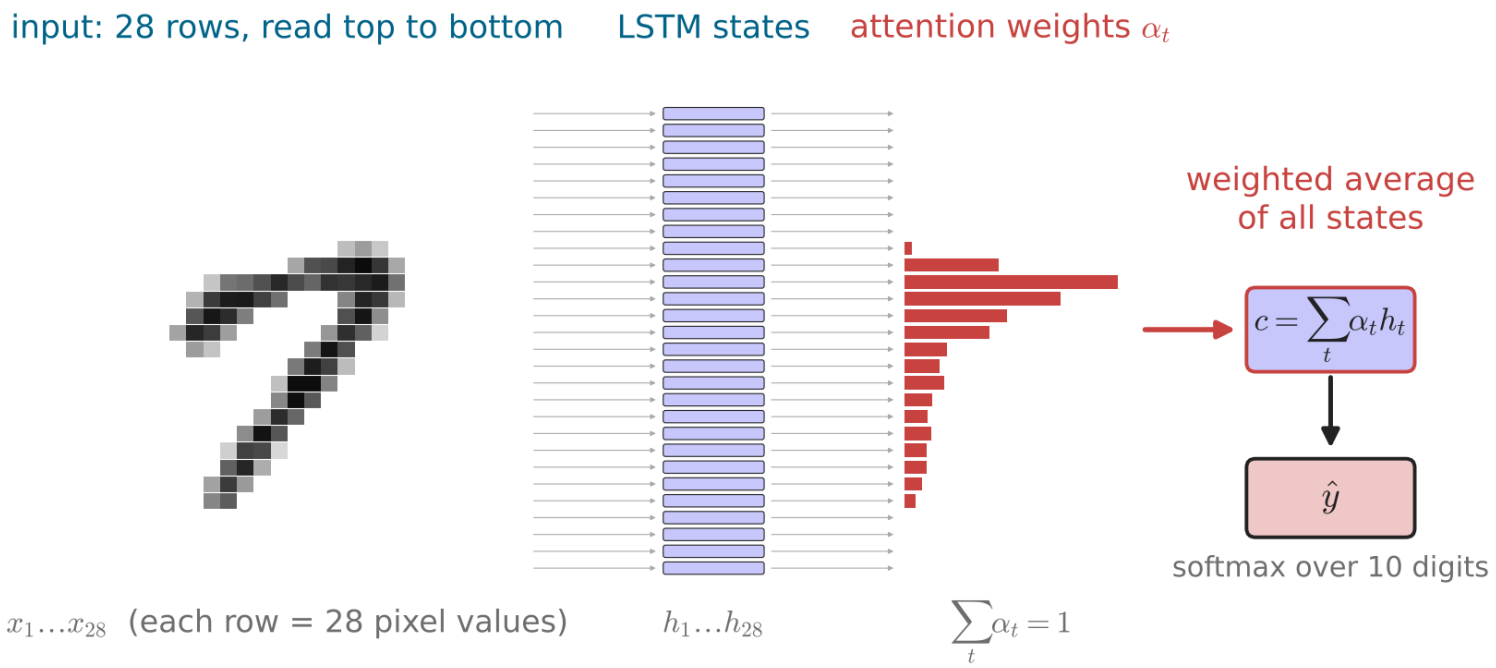
Attention

The bottleneck: everything through one vector



- many-to-one: the output sees only the last state h_T; encoder–decoder: the whole sentence is squeezed into one c. An LSTM helps, but the bottleneck stays

Attention: look back at every step



- keep all states; the output reads a weighted average $c = \sum_t \alpha_t h_t$, with weights the network computes itself ($\alpha_t \geq 0, \sum_t \alpha_t = 1$)
- bonus: plot α to see which rows the network looked at (lab, exercise 8)

How the weights are computed

1. a score per step

$$e_t = v^\top \tanh(W_a\, h_t + b_a)$$

2. softmax across the steps

$$\alpha_t = \frac{\exp(e_t)}{\sum_k \exp(e_k)}$$

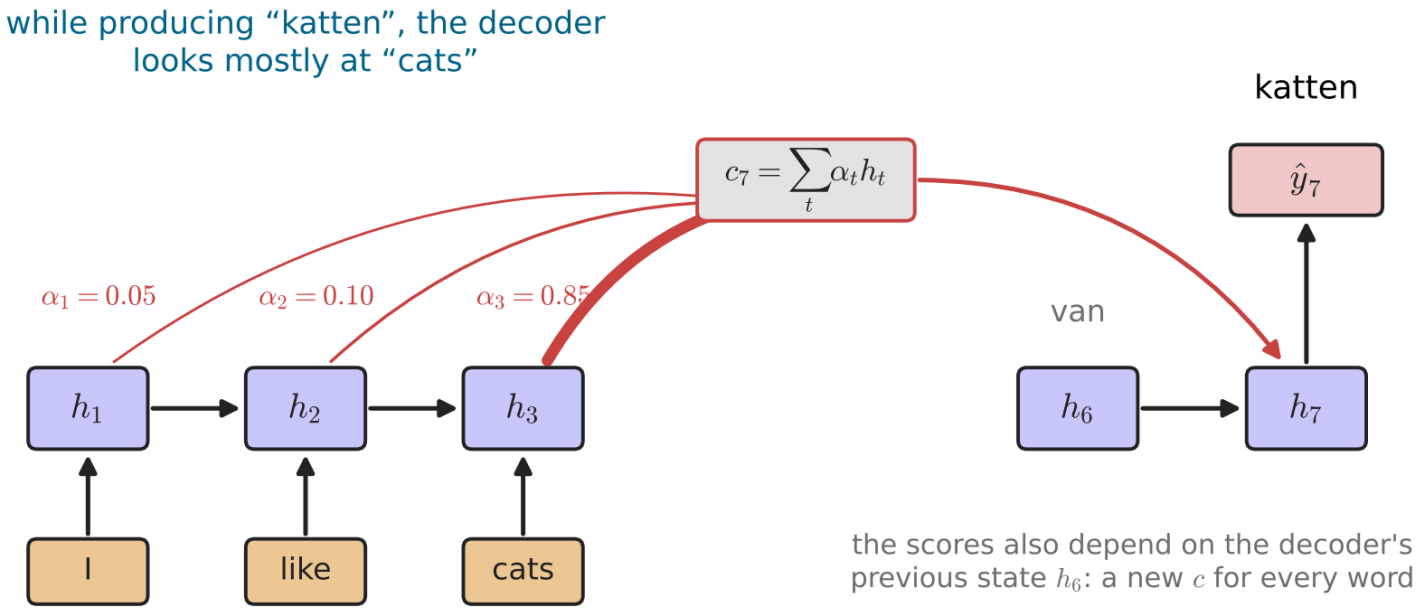
3. weighted average of the states

$$c = \sum_t \alpha_t\, h_t$$

- W_a and v are learned like every other weight; the same ones for every step
- α: positive, sums to 1 — a probability distribution over the steps
- c goes to the softmax instead of h_T
- **"additive" (Bahdanau) attention — the layer in the lab (exercises 6–7)**

```
inputs  = keras.Input(shape=(28, 28))
states  = keras.layers.LSTM(64, return_sequences=True)(inputs)   # all 28 states, not just the last
context, weights = AttentionLayer()(states)                       # c and alpha (the lab's custom layer)
outputs = keras.layers.Dense(10, activation="softmax")(context)
```

Attention in translation: a new context for every word



- the score of h_t now also depends on the decoder’s previous state: $e_t = \text{score}(h_{i-1}, h_t)$
- a new c_i for every output word i: the decoder looks where it needs to
- the weights show the alignment between the sentences
- **attention everywhere, no recurrence at all = the transformer (week 8)**

Bahdanau, Cho & Bengio (2015), Neural machine translation by jointly learning to align and translate

Thursday's lab: four models on the same digits

Feed-forward	flatten the 28 × 28 pixels; Dense(64, relu) → Dense(10)	baseline (ex. 2)
SimpleRNN	28 rows as 28 steps; SimpleRNN(64) → Dense(10)	many-to-one (ex. 3)
LSTM	LSTM(64) → Dense(10); also reading columns instead of rows	memory (ex. 4–5)
LSTM + attention	LSTM(64, return_sequences=True) → attention → Dense(10); plot α per row	attention (ex. 6–9)

Compare test accuracy; look at which rows the attention model looks at — also for a digit it gets wrong

Conclusion

Conclusion: RNNs and attention

- An RNN applies the same small network at every step, carrying a state: $h_t = g(W_x\, x_t + W_h\, h_{t-1} + b)$
- same weights at every step → any sequence length; one-to-many, many-to-one, many-to-many, encoder–decoder
- training = backpropagation through the unrolled graph; the same W_h at every step makes gradients vanish or explode
- LSTM: a memory line plus learned gates (keep, write, read) → long-range dependencies
- attention: instead of one last state, a learned weighted average of all states — and the weights show what the network looked at
- **attention without recurrence = transformers**