Lecture 6

Cezar Ionescu

22/06/2019

Administrative

- Homework from 15/06/2019 due now!
- Please complete and hand in the declarations of authorship.

Questions?

Solution to homework from lecture 4

`Ω, Event, p` and `X` as above. Consider the following experiment: `n` elements `e₁, ..., eₙ` are drawn independently from `Ω` and we compute the mean value of `X` for this sample `μₛ = (X(e₁) + ... + X(eₙ))/n`. Then `μₛ` is a random variable whose probability distribution approaches with increasing `n` the normal distribution with mean `μ` and standard deviation `σ/√(n)`.

The theorem states that `μₛ` is a random variable. But a random variable is a function defined in the context of a probability space. What is that probability space here? You will need to specify `Ω', Event', p' : Event' → [0, 1]` and define `μₛ : Ω' → ℝ` as a function in terms of in terms of the given `Omega, Event, p`, and `X`.

Solution to homework from lecture 4

$$\Omega' = (\Omega, \ldots, \Omega) = \Omega^n$$

$$\mathrm{Event}' = \mathbb{P}(\Omega')$$

$$p'(e_1, \ldots, e_n) = p(e_1) * \ldots * p(e_n)$$

$$\mu_s(e_1, \ldots, e_n) = (X(e_1) + \ldots + X(e_n))/n$$

Bayesian learning and neural networks

Given data of the form

$d = \{(x_1, c(x_1)), \ldots, (x_m, c(x_m))\}$

where c is an unknown function, Bayesian learning attempts to find the most probable hypothesis $h \in H$ that can be used to determine c.

When using neural networks, h represents an assignment of the weights and tresholds of the neurons.

Bayesian learning and neural networks

It can be shown (see slides of Lecture 5) that the most probable hypothesis is, under common circumstances, the one that minimises the *error* (or *loss*) function

$$E(w) = \Sigma (c(x_i) - f\ w\ (x_i))^2$$

where $f : Weights \to X \to \mathbb{R}$ is the function implemented by the network.

Hill climbing

Consider the following problem: given a function

`f : ([-5, 5], [-5, 5]) -> ℝ`

find `x, y ∈ [-5, 5]` such that `f(x, y)` is minimal.

We could, e.g., try some values:

```
f(1, 1) = 0.0
f(2, 1) = 0.0
f(3, 1) = 2.0
f(3, 2) = -2.0
```

Hill climbing

We need a way of exploring the domain of `f` more systematically. We can, for example, start in `(0, 0)` and test the corners of the square of side `1` centered in the origin:

```
f(0, 0)       = 5
f(-0.5, -0.5) = 8.25
f(-0.5, 0.5)  = 4.75
f(0.5, 0.5)   = 2.25
f(0.5, -0.5)  = 6.75
```

We can now use `(0.5, 0.5)` as a starting point and continue, perhaps making the side of the square smaller (e.g., `0.9` instead of `1`.

Hill climbing and neural networks

Attempts to use hill climbing to minimise E are usually unsuccessful. That is because the space to be explored is exponential in the number of parameters (for two dimensions we have 4 points, for three dimensions 8, for n dimensions 2^n points).

We need a "guide" to point us in the right direction.

Gradient

For a function `f : X -> ℝ` where `X ⊆ ℝⁿ`, the gradient (if it exists) is the vector

```
∇ f (x) = [D₁ f (x), ..., Dₙ f (x)]
```

where we use `Dᵢ f (x)` instead of the more frequent `∂ f (x) / ∂ xᵢ`

`∇ f (x)` points towards the direction of steepest increase in `f` at `x`.

Gradient descent

If we have information about the gradient of `f`, then we can do **much** better than hill climbing:

```
f(0, 0) = 5
gradf(0, 0) = [-2, -4]
```

So if we move in the direction `[-2, -4]` we should see an increase in `f`; if we move in the opposite direction, a decrease:

```
f(-2, -4) = 37
f(2, 4)   = -3
```

We now move to `(2, 4)` and start again, perhaps choosing a different step size (e.g., `0.9`).

This procedure is a variant of *gradient descent*. In "real" gradient descent, the step size η is chosen to minimise

```
f(x - η * D₁ f (x, y), y - η * D₂ f (x, y))
```

Revisiting perceptrons

```
perceptron : (ℝⁿ, ℝ) -> ℝⁿ -> {-1, 1}
perceptron ([w₁,..., wₙ], θ) [x₁, ..., xₙ] = if s ≥ θ then 1
                                              else -1
 where s = w₁*x₁ + ... + wₙ * xₙ
```

We are given

$d = \{(x_1, c(x_1)), \ldots, (x_m, c(x_m))\}$

Can we use gradient descent to determine w_i and θ?

Gradient descent and perceptrons

We can compute `E`, but the problem is that it is not differentiable w.r.t. w_i.

Because `perceptron ([w₁,..., wₙ], θ) : ℝⁿ -> {-1, 1}` is **discontinuous**.

We can approximate the step function required by `perceptron` with a differentiable function. For example:

```
perceptron ([w₁,..., wₙ], θ) [x₁, ..., xₙ] = σ(s)
 where s      =  w₁*x₁ + ... + wₙ * xₙ - θ
       σ(y)   =  1 / 1 + exp(-y)
```

Note that this scales the output to `[0, 1]`.

```
σ'(y) = σ(y) * (1 - σ(y))
```

Gradient descent and perceptrons

More generally

```
perceptron ([w₁,..., wₙ], θ) [x₁, ..., xₙ] = f(s)
 where s     =  w₁*x₁ + ... + wₙ * xₙ - θ
       f(y)  =  ...
```

where `f` is some nicely differentiable function.

Gradient descent and perceptrons

In the following, we consider a perceptron with no θ, since

```
perceptron ([w₁,..., wₙ], θ) [x₁, ..., xₙ] =
perceptron ([w₁,..., wₙ, θ]) [x₁, ..., xₙ, 1]
```

and abbreviate

```
ws = [w₁,..., wₙ]
```

Consider `m = 1` (only one element in the data), so that

```
E(ws) = (c(x) - perceptron ws (x))²
      = (t - per ws (x))²
```

Gradient descent and perceptrons

We can apply gradient descent if we can compute $D_i\ E\ (w_i)$. This is easy:

$$
\begin{aligned}
D_i\ E\ (ws) &= D_i\ (t - \text{per}\ ws\ (x))^2 \\
&= D_i\ (t - f(s))^2 \\
&= 2 * (t - f(s)) * D_i\ (t - f(s)) \\
&= -2 * (t - f(s)) * f'(s) * D_i\ s \\
&= -2 * (t - f(s)) * f'(s) * x_i
\end{aligned}
$$

Notice the similarity with the learning rule from the last lecture.

Stochastic gradient descent

In practice, `m > 1`.

"Proper" gradient descent demands the computation of `E`.

However, this would be too time-consuming, and usually an approximate `E` is computed instead, using a "batch" size between `1` and `m`. This approximate version is known as *stochastic gradient descent*.

More than one layer

What if we have more than one layer? Consider a simple example:

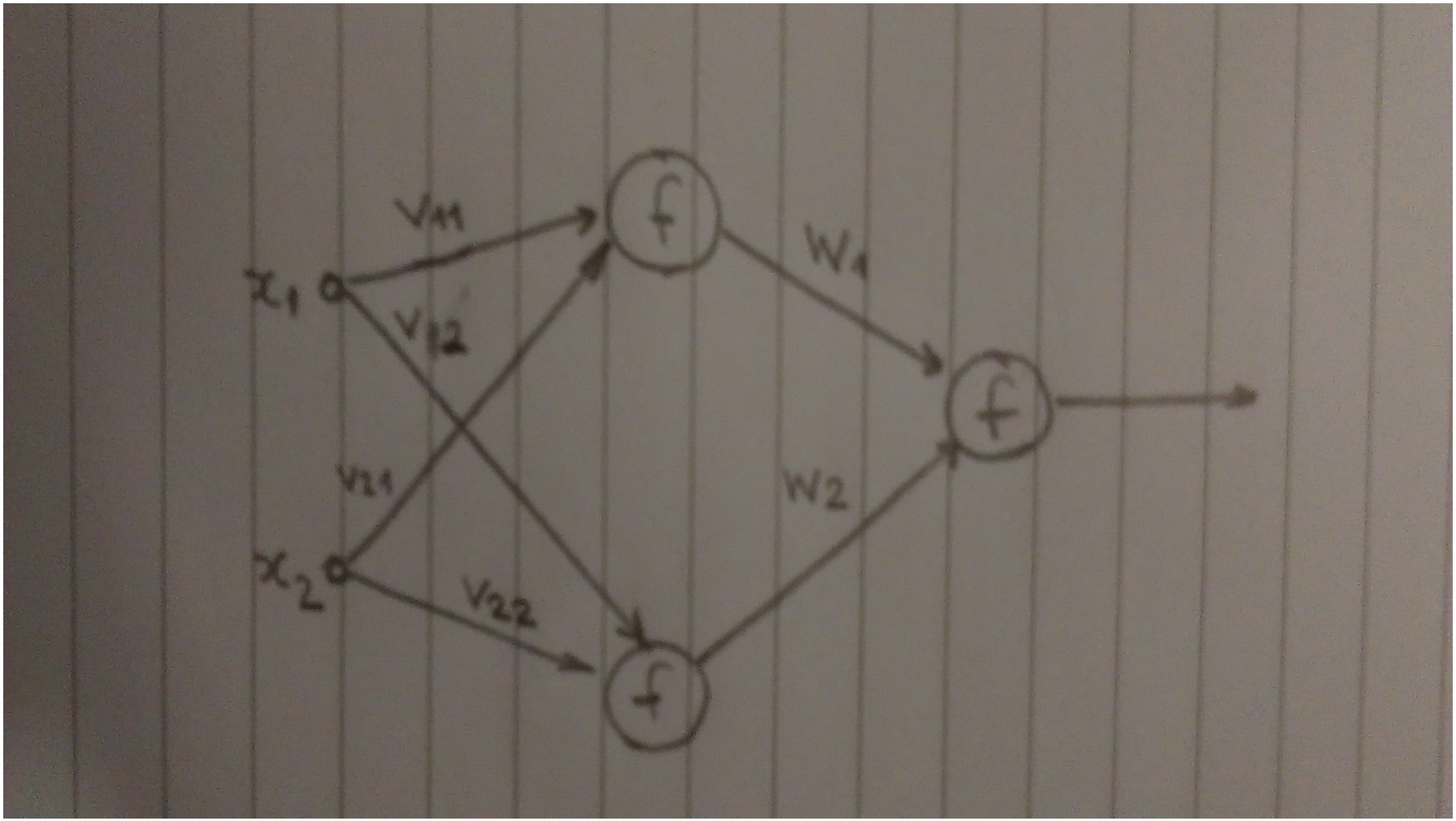

Figure 1: Two layer network

More than one layer

We can compute the error as before

```
E(ws, vs) = (t - per ws [y₁, y₂])²
  where y₁ = per vs₁ [x₁, x₂]
        y₂ = per vs₂ [x₁, x₂]
        vs₁ = [v₁₁, v₂₁]
        vs₂ = [v₁₂, v₂₂]
```

```
D₁ E(ws, vs) =  -2 * (t - f(s)) * f'(s) * y₁
```

But how can we compute, e.g., `D₃ E(ws, vs)`?

Backpropagation

$$
\begin{aligned}
D_3\ E(ws, vs) &= D_3\ (t - per\ ws\ [y_1(v_{11}, v_{21}), y_2(v_{12}, v_{22})])^2 \\
&= D_3\ E(ws, y_1, y_2) * D_1\ y_1 \\
&= D_3\ E(ws, y_1, y_2) * D_1\ f(z) \\
&= D_3\ E(ws, y_1, y_2) * f'(z) * D_1\ z \\
&= D_3\ E(ws, y_1, y_2) * f'(z) * x_1
\end{aligned}
$$

Homework

Consider a two-layer perceptron as in Figure 1, with `f(y) = y, f'(y) = 1` (such a perceptron is called *linear*). Suppose we start with

$v_{11} = 1, v_{12} = -1, v_{21} = -1, v_{22} = 1, w_1 = 1, w_2 = 2$

Assume the data is `([-1, 1], 0)`. How does the backpropagation algorithm adjust v_{11}?

Other architectures

- recurrent neural networks (Hopfield 1982, Schmidhubber 1992, others in between)
- convolutional neural networks (partly already Rumelhart et al. 1986)
- capsule networks (Hinton 2017)

Recurrent neural networks

- Neurons have a state which is fed back to the input.
- This is an implementation of memory.
- RNNs could, in theory, implement any computable function, but in practice are limited.
- Gradient descent problematic (`vanishing gradient''), leading to LSTM (`Long Short-Term Memory'').

Convolutional neural networks

- Architecture addresses two important problems with traditional feed-forward networks:
- loss of local or temporal order in inputs
- combinatorial explosion of weights

Loss of spatial information in perceptrons

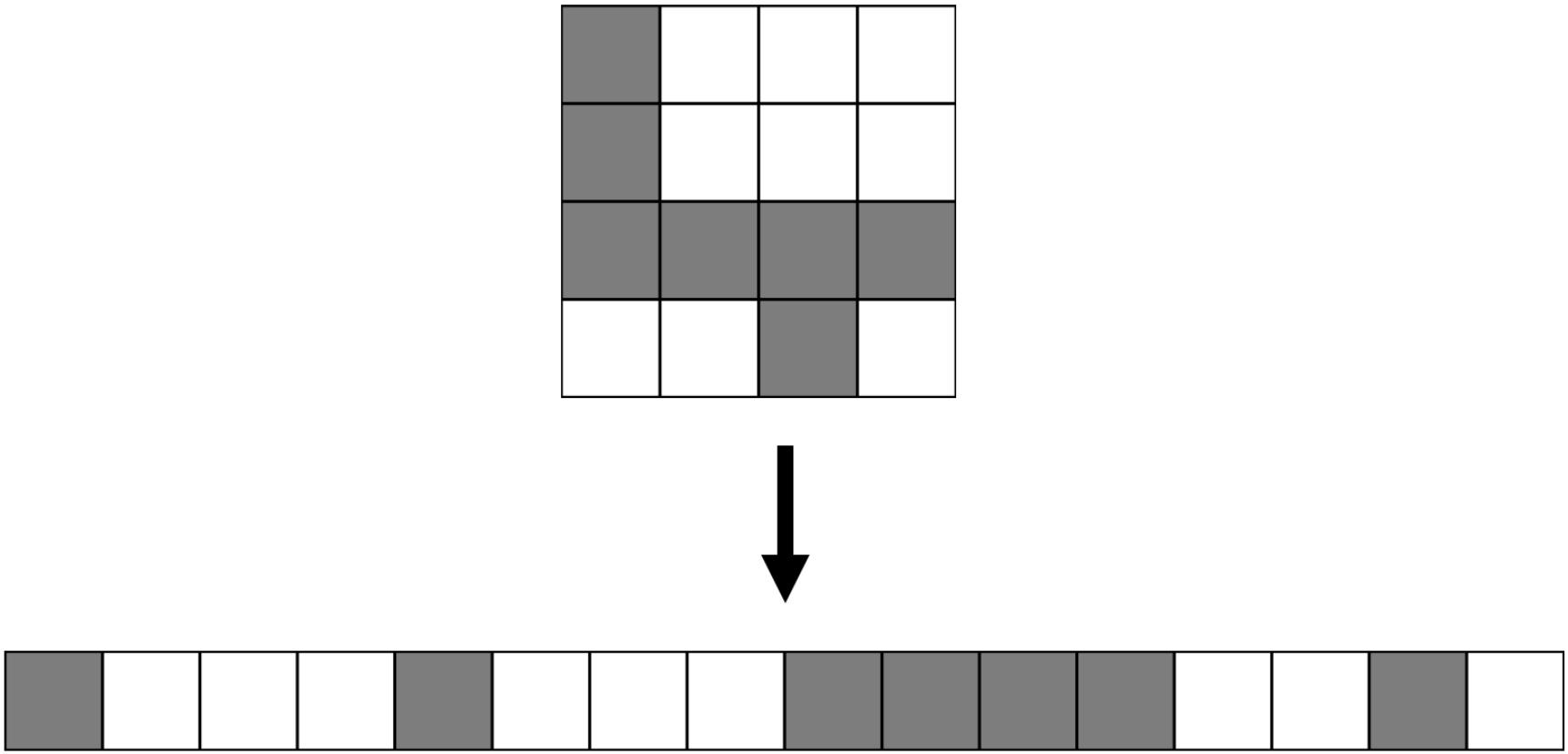

Figure 2: “Linearisation” of an image

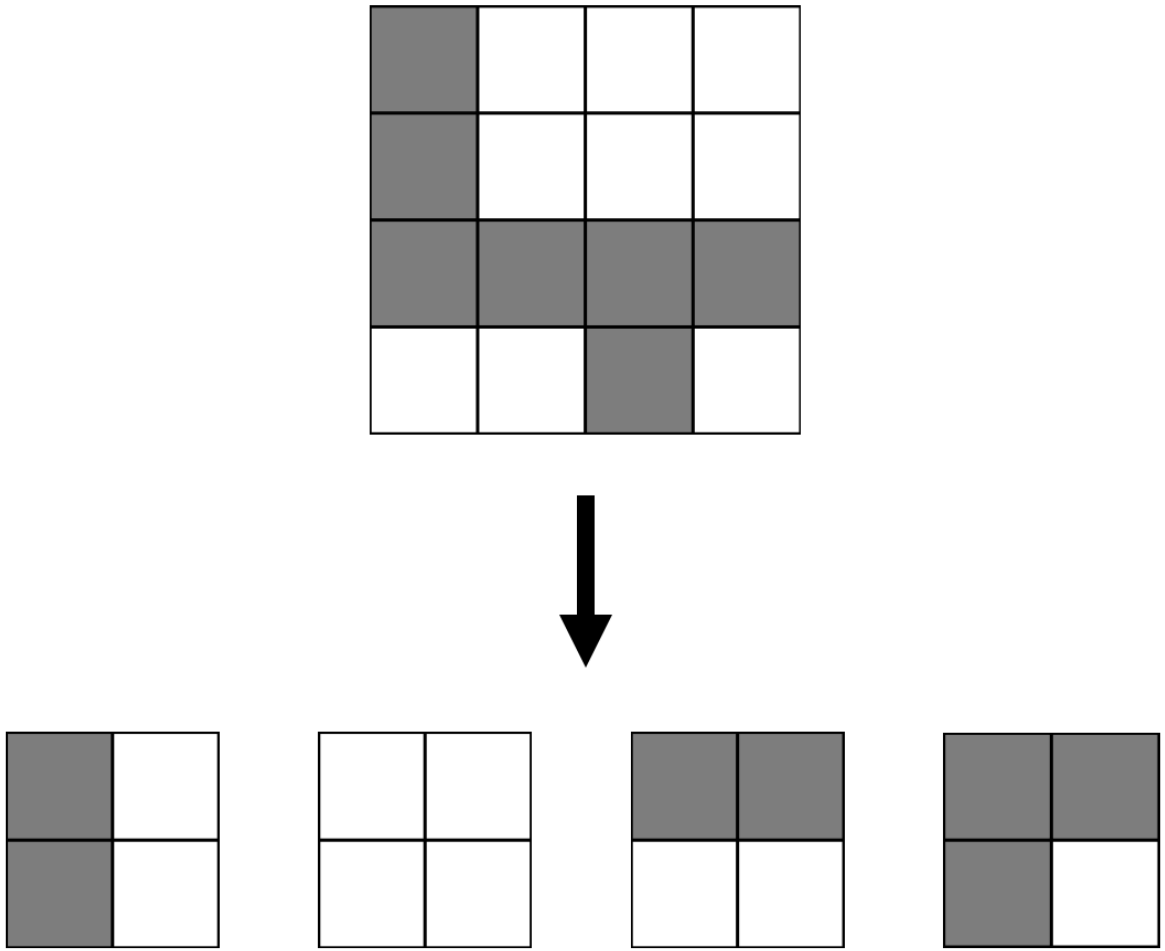

Figure 3: Decomposing an image

Overlapping filters: stride

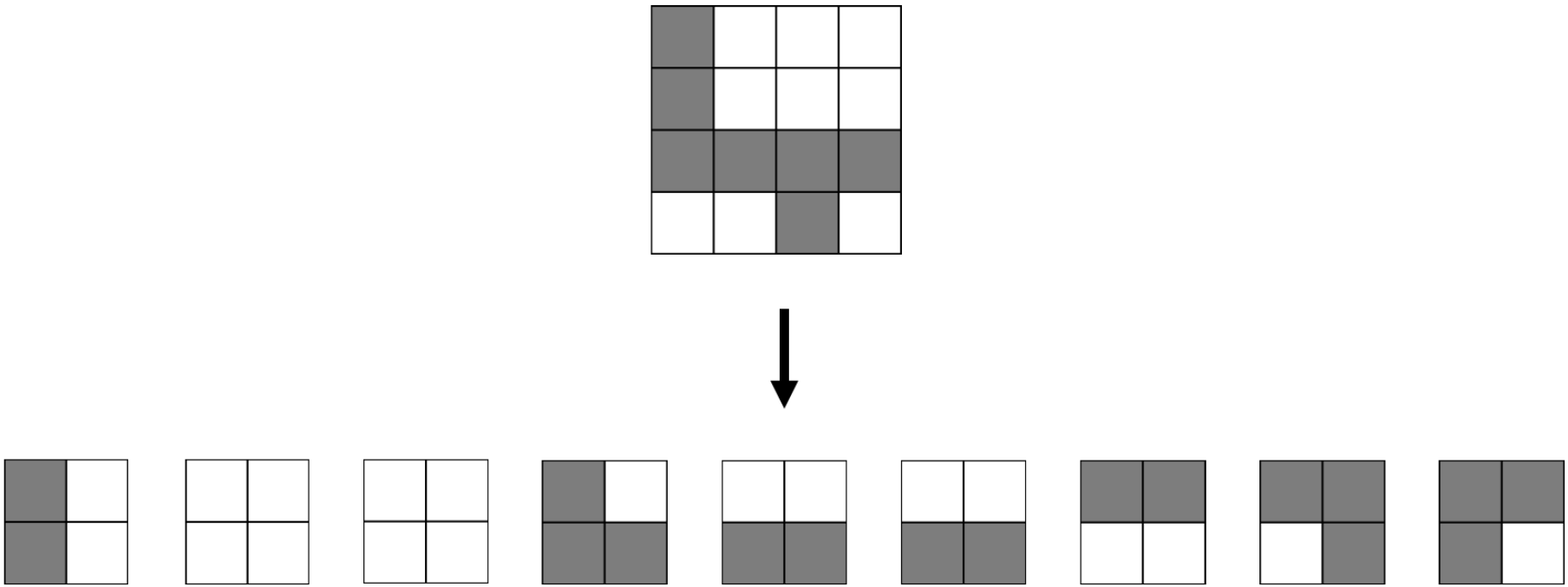

Figure 4: An overlapping decomposition

Convolutional neural networks

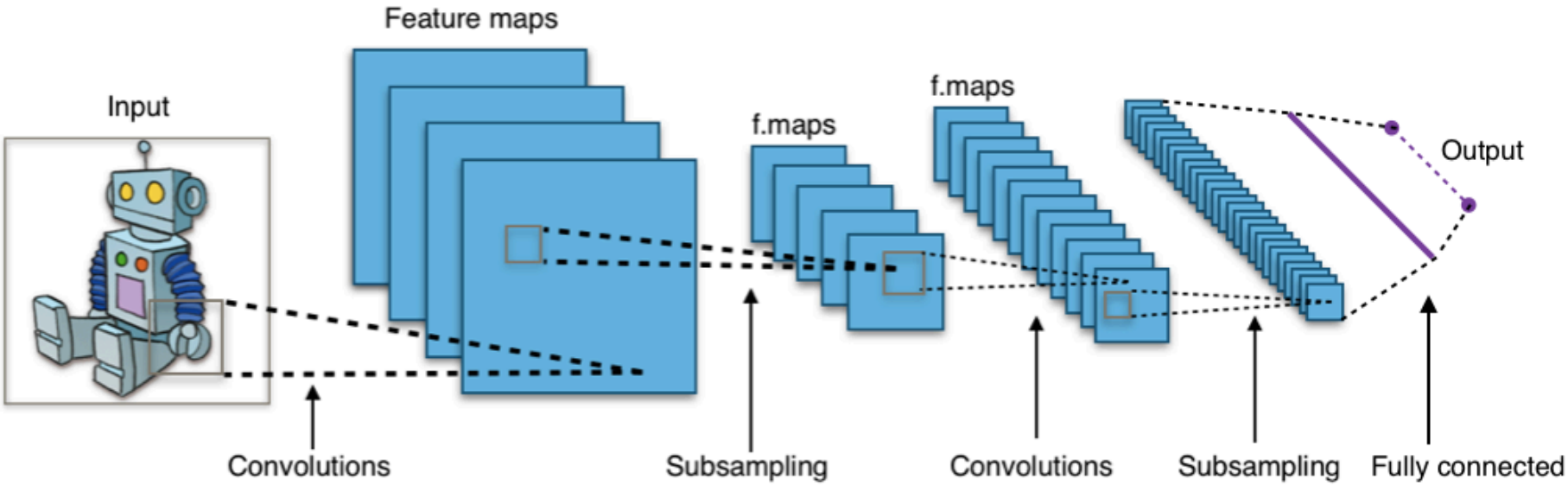

Figure 5: Convolutional neural network [1]

[1] Aphex34 https://commons.wikimedia.org/wiki/File:Typical_cnn.png, https://creativecommons.org/licenses/by-sa/4.0/legalcode}

The surprising success of “deep” networks

- Two layers of weights are sufficient, but not efficient
- Surprisingly, *many* layers turn out to be more efficient
- Pseudo-explanation (Hinton 2005): layers select increasingly high-level functions
- A different pseudo-explanation: neurons similar to logic gates.

Training deep networks

- The problem with many layers is that training is hard (backprop is NP-complete).
- CNNs used to decrease the numbers of weights
- denoising used to extract “features” (compress input space)

Pros and cons of deep neural networks

- Pros
  - very successful (handwriting recognition, speech recognition, automatic translation, policy functions for reinforcement learning, etc.)
- Cons
  - Very opaque
  - Beyond metaphors, not clear why they work (two layers should, e.g., generalise better)

Acknowledgements

Figures on slides *Loss of spatial information*, *Filters*, and *Overlapping filters* by Farah Shamout (Balliol College, Oxford).