Lecture 6

Cezar Ionescu

22/06/2019

DEPARTMENT FOR CONTINUING EDUCATION
UNIVERSITY OF OXFORD

Administrative

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

Questions?

Solution to homework from lecture 4

Bayesian learning and neural networks

Given data of the form

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

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) = Σ (xᵢ - f w (xᵢ))²
```

where `f : Weights -> X -> 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 $X \subseteq \mathbb{R}^n$, the gradient (if it exists) is the vector

$\nabla f(x) = [D_1 f(x), \ldots, D_n f(x)]$

where we use $D_i f(x)$ instead of the more frequent $\delta f(x) / \delta x_i$

$\nabla f(x)$ 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 ηis chosen to minimise

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

Revisiting perceptrons

```
perceptron : (ℝ^n, ℝ) -> ℝ^n -> {-1, 1}
perceptron ([w_1,..., w_n], θ) [x_1, ..., x_n] = if s ≥ θ then 1
                                                             else -1
 where s = w_1*x_1 + ... + w_n * x_n
```

We are given

```
d = {(x_1, c(x_1)), ..., (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
       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
       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 `Dᵢ E (wᵢ)`. This is easy:

```
Dᵢ E (ws)  =  Dᵢ (t - per ws (x))²
           =  Dᵢ (t - f(s))²
           =  2 * (t - f(s)) * Dᵢ (t - f(s))
           =  -2 * (t - f(s)) * f'(s) * Dᵢ s
           =  -2 * (t - f(s)) * f'(s) * xᵢ
```

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:

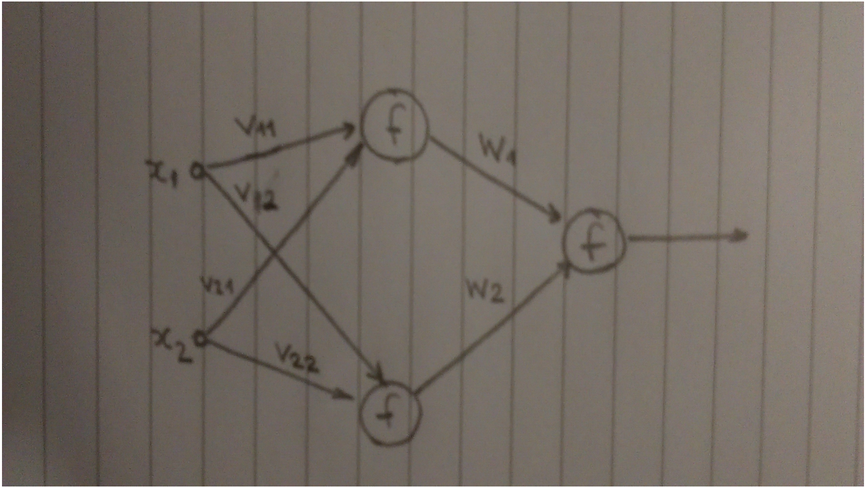

Figure 1: Two layer network

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

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

Homework

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

```
v11 = 1, v12 = -1, v21 = -1, v22 = 1, w1 = 1, w2 = 2
```

Assume the data is `([-1, 1], 0)`. How does the backpropagation algorithm adjust `v11`?