Deep learning

Bernd Porr

January 6, 2023

1 Motivation

We have a dataset $\vec{x}_1, \ldots, \vec{x}_N$ which should be converted via the function

$$\vec{y} = f(\vec{x}) \tag{1}$$

into $\vec{y}$. The goal is to learn this function with the help of examples $\vec{x}_1, \vec{y}_1$; $\vec{x}_2, \vec{y}_2; \ldots$.

The following sections show how learning of f with the help fo examples can be achieved. This is called "inductive learning".

We 1st introduce a linear neuron, then deriving the math for a linear deep network and finally introducing non-linearities into the deep network. This allows to introduce the math in a more gradual way.

2 Linear neuron

Fig. 1A shows a neuron in a single layer (Rosenblatt, 1958). Its activation is $y(n)$ and is the ith neurn in this layer. n is the current time-stamp. The input activity to the neuron $x_j(n) = y_j(n)$ is weighted by ω_{ji} and summed up to:

$$y_i(n) = \sum_j y_j(n) w_{ji} \tag{2}$$

Our function f from the introduction is here a simple linear combination of input activities where ω_{ji} had to be learned to approximate f.

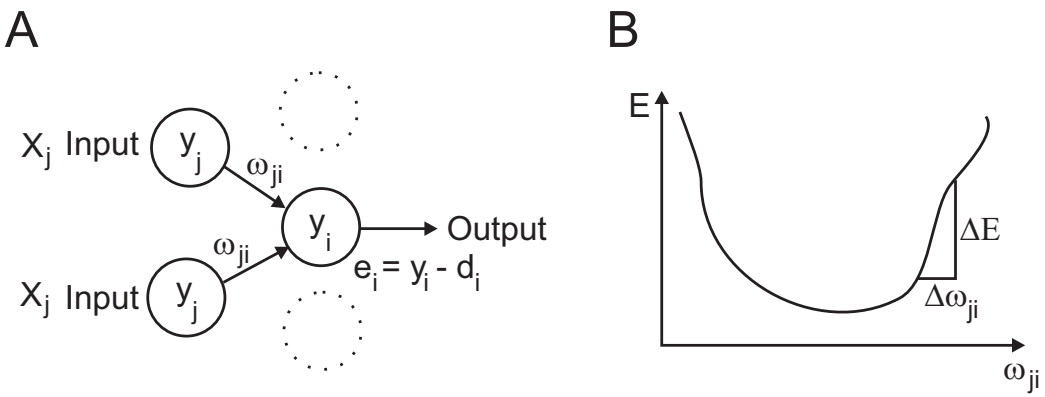

Figure 1: Simple linear neuron

3 The error signal

The task is to learn a function f which converts the input activities to output activities (Eq. 1). This is achived by minimising the error:

$$e_i(n) = y_i(n) - d_i(n) \tag{3}$$

where $d_i(n)$ is the desired output value and $y_i(n)$ the actual output value. The goal is to minimise the square of the error:

$$E = \frac{1}{2}e^2 \tag{4}$$

4 Gradient descent

The central trick here is to take the partial derivative in respect to the weights ω_{ji}

$$\Delta\omega_{ji} = -\mu\frac{\partial E}{\partial\omega_{ji}} \tag{5}$$

$$\omega_{ji} \leftarrow \omega_{ji} + \Delta\omega_{ji} \tag{6}$$

Why does this make sense? Fig. 1B shows the relationship between the error E and the weight ω_{ji}. In this example if the weight is slightly increased the squared error is also increased which is not desirable. However, if increasing ω_{ji} reduces the squared error E then it's a good idea to keep changing the weight in this direction. This approach is called *gradient descent*.

5 Learning rule for the single layer

We can now derive the learning rule which changes the weights ω_{ji}. We simply insert Eq. 2, 3 and 4 into Eq. 5:

$$\Delta\omega_{ji} = -\mu\frac{1}{2}\frac{\partial(d_i(n)-y_i(n))^2}{\partial\omega_{ji}} \tag{7}$$

$$= -\mu\frac{1}{2}\frac{\partial\left(d_i(n)-\sum_j y_j(n)w_{ji}\right)^2}{\partial\omega_{ji}} \tag{8}$$

$$= \mu\underbrace{\left(d_i(n)-\sum_j y_j(n)w_{ji}\right)}_{-e_i(n)}\cdot y_j(n) \tag{9}$$

$$= -\mu\underbrace{\frac{\partial E}{\partial y_i}}_{-e_i(n)}\underbrace{\frac{\partial y_i}{\partial\omega_{ji}}}_{y_j(n)} \tag{10}$$

$$= \mu\cdot e_i(n)\cdot y_j(n) \tag{11}$$

where $\mu << 1$ is the learning rate or the "step change". The learning rule Eq. 11 is simply a multiplication of the input activity $y_j(n)$ with the error $e_i(n)$ (Widrow and Hoff, 1960).

6 Multi-layer network: error backpropagation and learning rule

Fig. 2 shows now a network with multiple layers. The forward progression of the signals y_j from the input to the output and follows exactly the same recipe as for the single layer network above using Eq. 2. Also the error e_i at the output layer can simply be calculated as the difference between the actual output y_i and the desired output d_i (see Eq. 3). The problem is how to calculate the internal errors e_j und e_k and how they change the hidden weights ω_{kj} (see Eq. 10):

$$\frac{\partial E}{\partial\omega_{kj}} = \underbrace{\frac{\partial E}{\partial y_j}}_{\text{trick!}}\frac{\partial y_j}{\partial\omega_{kj}} \tag{12}$$

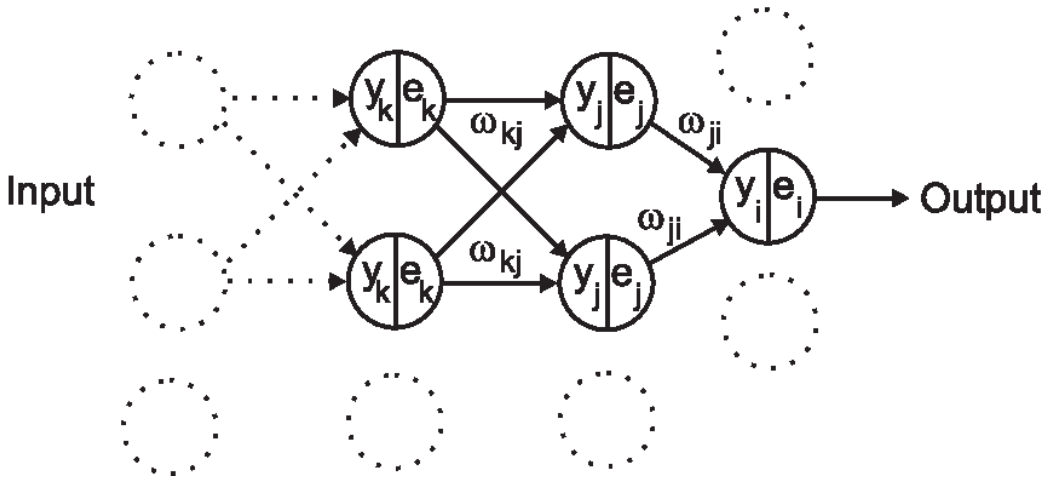

Figure 2: Multi-layer or deep neural network

The central trick here is to express $\frac{\partial E}{\partial y_j}$ with the help of the activities y_i at the output und then to identify the resulting terms with our linear sum Eq. 2 and the chain rule Eq. 10:

$$\frac{\partial E}{\partial y_j} = \sum_i \underbrace{\frac{\partial E}{\partial y_i}}_{-e_i} \cdot \underbrace{\frac{\partial y_i}{\partial y_j}}_{\omega_{ji}} \tag{13}$$

Eq. 13 is again substituted into Eq. 12 which gives us:

$$\frac{\partial E}{\partial \omega_{kj}} = -\left(\sum_i e_i \omega_{ji}\right) \frac{\partial y_j}{\partial \omega_{kj}} \tag{14}$$

with

$$y_j = \sum_k y_k(n) w_{kj} \tag{15}$$

and leads to:

$$\frac{\partial E}{\partial \omega_{kj}} = -\left(\sum_i e_i \omega_{ji}\right) \underbrace{\frac{\partial \left(\sum_k y_k(n) w_{kj}\right)}{\partial \omega_{kj}}}_{y_k} \tag{16}$$

$$= -\left(\sum_i e_i \omega_{ji}\right) y_k \tag{17}$$

The change of the internal weight ω_{kj} leads then to:

$$\Delta\omega_{kj} = \mu \cdot y_k \cdot \underbrace{\sum_i e_i \omega_{ji}}_{e_j} \tag{18}$$

where e_j is the internal error and we can now use this recipe to calculate all internal errors. This is called *error backpropagation* which now allows to establish networks with an arbitrary number of layers (Widrow and Hoff, 1960; Rumelhart et al., 1986).

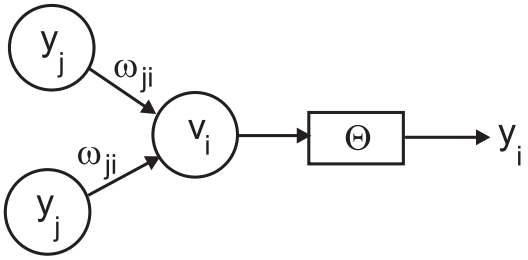

Figure 3: Non-linear neuron

7 Non-linear networks

Only if computations in the separate neurons are non-linear deeper networks make sense. Any linear deep network could always be reduced to a single layer network rendering any deeper network pointless. Consequently, we need to introduce now non-linearities. Fig. 3 shows a non-linear neuron which is the central building block of any deep network where after the linear summation we introduce a non-linearity Θ which we call activation function:

$$y_i(n) = \Theta\left(\underbrace{\sum_j y_j(n) w_{ji}}_{v_i}\right) \tag{19}$$

The crucial question is if the introduction of the activation function changes the learning rule. Luckily not much. Looking at Eq. 10 it becomes directly clear

that the chain rule is simply expanded by another term:

$$\frac{\partial E}{\partial \omega_{ji}} = \frac{\partial E}{\partial y_i} \underbrace{\frac{\partial y_i}{\partial v_i}}_{\Theta'} \frac{\partial v_i}{\partial \omega_{ji}} \tag{20}$$

where Θ' is simply the derivative of the activation function Θ. Consequently the weight change at the output is now calculated as:

$$\Delta \omega_{ji} = \mu \cdot \Theta'(y_i) \cdot y_j \cdot e_i \tag{21}$$

and the internal error as:

$$\Delta \omega_{kj} = \mu \cdot \Theta'(y_j) \cdot y_k \cdot \underbrace{\sum_i e_i \omega_{ji}}_{e_j} \tag{22}$$

Strictly, the activation function needs to be differentiable. However, it turned out that the one way rectifier (Rectifiying Linear Unit = ReLU) works extremely well (Fukushima, 1975):

$$\Theta(v) = \begin{cases} 0, & \text{if } v < 0 \\ v, & \text{otherwise} \end{cases} \tag{23}$$

Its derivative has no definite solution at its origin so that one needs to decide if the derivative is zero there or one. Other popular activation functions are $\Theta(v) = \tanh(v)$ or $\Theta(v) = \frac{1}{1+e^{-v}}$.

References

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biol. Cybern.*, 20(3–4):121–136.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol. Rev.*, 65(6):386–408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Widrow, G. and Hoff, M. (1960). Adaptive switching circuits. *IRE WESCON Convention Record*, 4:96–104.