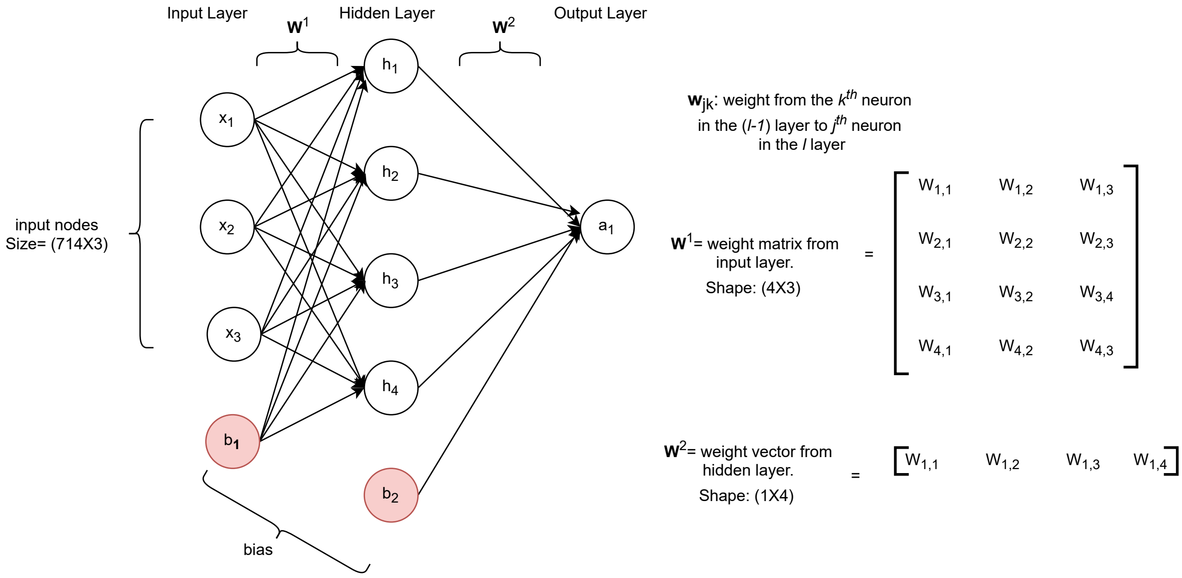
Exercise Sheet 6

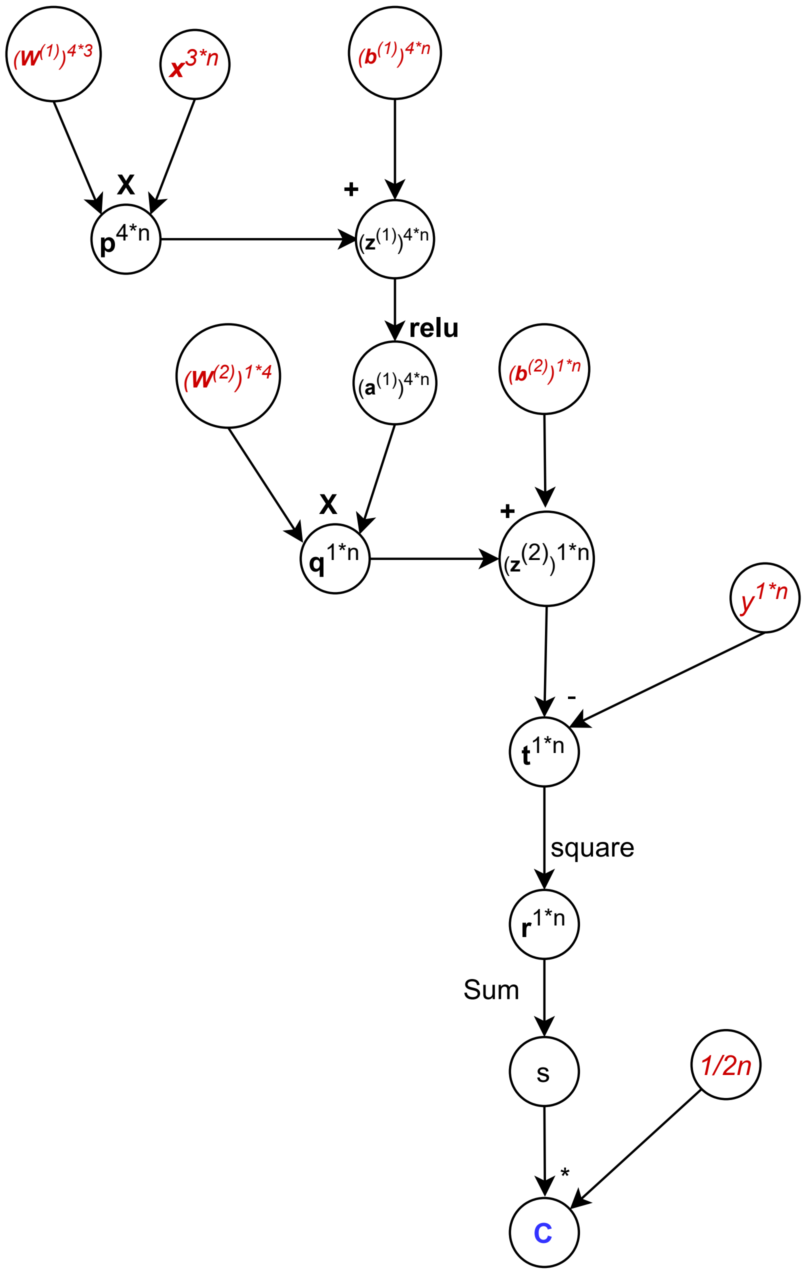
Zena Al Khalili (7009151)
Sangeet Sagar (7009050
{zeal00001,sasa00001}@stud.uni-saarland.de

January 5, 2021

Exercise 6.1 - Network scheme



Exercise 6.2 - Computation graph



(W(1))4*3
x3*n
(b(1))4*n
X
+
p4*n
(z(1))4*n
relu
(W(2))1*4
(a(1))4*n
(b(2))1*n
X
+
q1*n
(z(2))1*n
y1*n
-
t1*n
square
r1*n
Sum
s
1/2n
*
C

Exercise 6.3 - Backpropagate

$$\frac{\partial MSE}{\partial w_{ji}} = \frac{\partial MSE}{\partial w^{(1)}} = \frac{\partial c}{\partial s} \times \frac{\partial s}{\partial r} \times \frac{\partial r}{\partial t} \times \frac{\partial t}{\partial z^{(2)}} \times \frac{\partial Z^{(2)}}{\partial q} \times \frac{\partial q}{\partial a^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial z^{(1)}}{\partial p} \times \frac{\partial p}{\partial w^{(1)}}$$

$$\frac{1}{n} \quad \sum^{n} \quad 2t^{1*n} \quad 1*n \quad 1*n \quad W^{(2)\,1*4} \quad 4*n \quad 1 \quad X^{3*n}$$

$$\frac{\partial MSE}{\partial w^{(1)}} = \frac{2}{n} \sum^{n} X^{3*n} \cdot t^{T\, n*1} \cdot W^{(2)\,1*4}$$

$$\frac{\partial MSE}{\partial w_{(1)}} = \frac{2}{n} \sum^{n} X^{3*n} (\hat{y} - y_{target})^{T\, n*1} \cdot W_{(2)}^{1*4}$$

$$\frac{\partial MSE}{\partial w_{kj}} = \frac{\partial MSE}{\partial w^{(2)}} = \frac{\partial c}{\partial s} \times \frac{\partial s}{\partial r} \times \frac{\partial r}{\partial t} \times \frac{\partial t}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial q} \times \frac{\partial q}{\partial w^{(2)}}$$

$$\frac{1}{n} \quad \sum^{n} \quad 2t^{1*n} \quad 1*n \quad 1*n \quad a^{(1)\,4*n}$$

$$\frac{2}{n} \sum^{n} t^{1*n} \quad a^{(1)\,4*n}$$

$$\frac{\partial MSE}{\partial w^{(2)}} = \frac{2}{n} \sum^{n} t^{1*n} \cdot a^{(1)\,T\,(n*4)}$$

$$= \frac{2}{n} \left[t \cdot a^{T}\right]^{1*4}$$

$$\frac{\partial MSE}{\partial w_{kj}} = \frac{2}{n} \left[\delta_k \cdot ReLu(w^{(1)T} \cdot x)\right]$$

$$= \frac{2}{n} \left[(\hat{y} - y_{target}) \cdot ReLu(w^{(1)T} \cdot x)\right]$$

Exercise 6.5 - Implement in PyTorch

- PyTorch basically uses `a.requires_grad_(True)` to track all arithmetic operations which later helps to compute gradient.
- It uses `requires_grad` to hold the value of the gradient, given we have already called `.backward()` (it helps in back-propagation).
- So when we call `backward()`, gradients are collected only for the nodes that have `a.requires_grad_(True)` and `is_leaf` True.
- If we wish to stop the tracking of gradient we simply use `with torch.no_grad()`