Programmeringsteknik: NumPy

FÖRELÄSNING 3 2024-03-26

Föreläsning 3

* Första ordningens differentialekvationer
* Lite om system av differentialekvationer / högre ordningens ekvationer
* Mer om kurvanpassning
* Inläsning av data från fil
  * hämtning av data från nätet
* Några funktioner för matriser och vektorer
* Träna eget neuralt nätverk till att känna igen siffror

Att lösa differentialekvationer med Python

* Vi ska lösa differentialekvationer i Python med biblioteket scipy.

$$y'(t) = f(t, y(t))$$
$$y(t_1) = y_1$$

Första ordningens differentialekvationer

- Differentialekvationer har funktionens derivata som en variabel (första ordningen = förstaderivatan)
- På engelska: *initial value problem.*

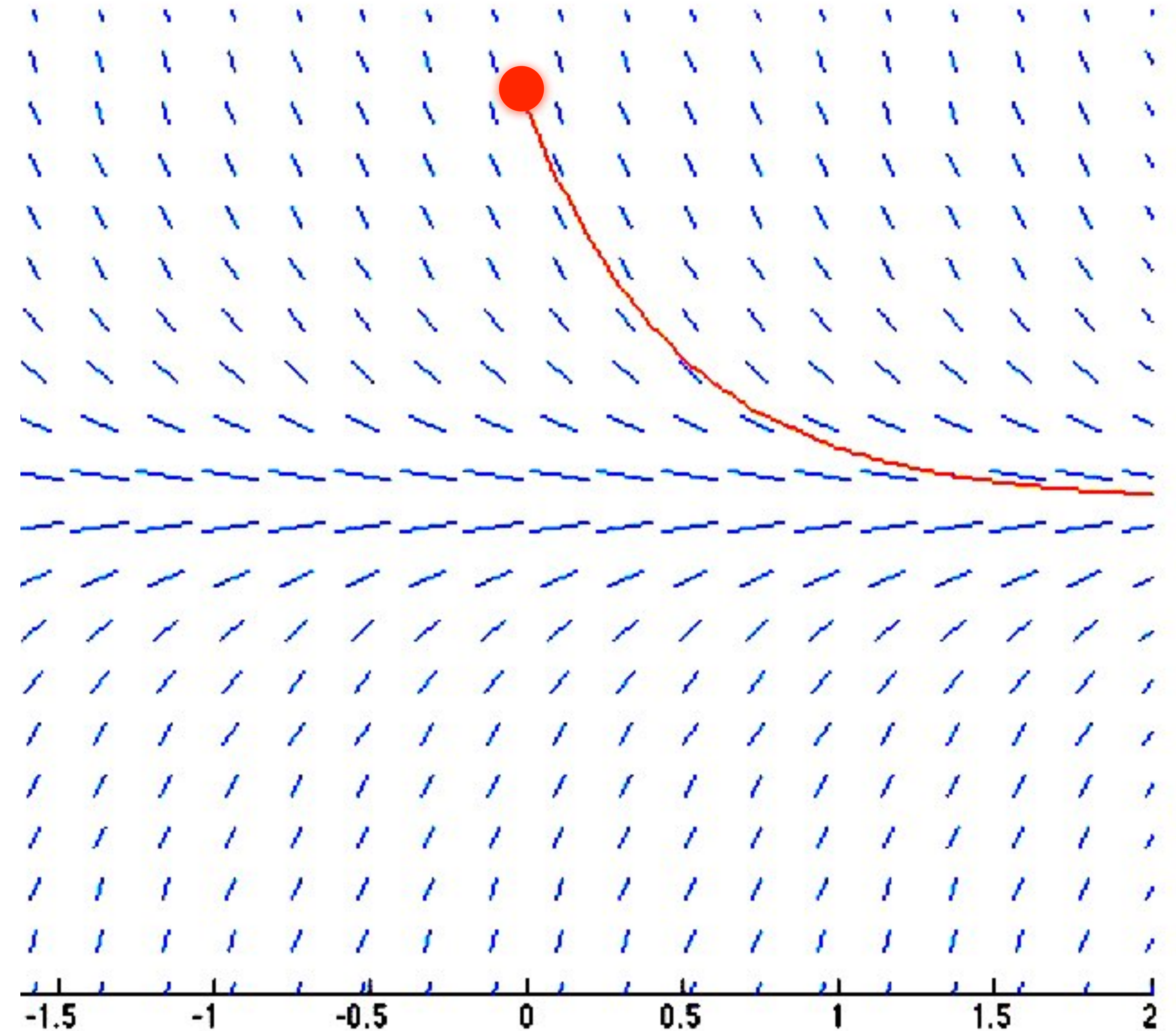
$$y'(t) = f(t, y(t))$$
$$y(t_1) = y_1$$

- Lösningen består av en funktion *y* som beskriver hur systemet utvecklas.
- Analytiska lösningar är svåra, men vi kan få en numerisk uppskattning för funktionen *y(t)*

Exempel från fysiken: riktningsfält

- Derivatan kan ses som ett **riktningsfält**
- Riktningsfältet + startvärdet ger lösningen:

$$y'(t) = -2 \cdot y(t)$$
$$y(0) = 4$$



Eulers metod

- Utgå från positionen givet av startvärdet och ”stega” fram lösningen på riktningsfältet i små steg (`dt`).

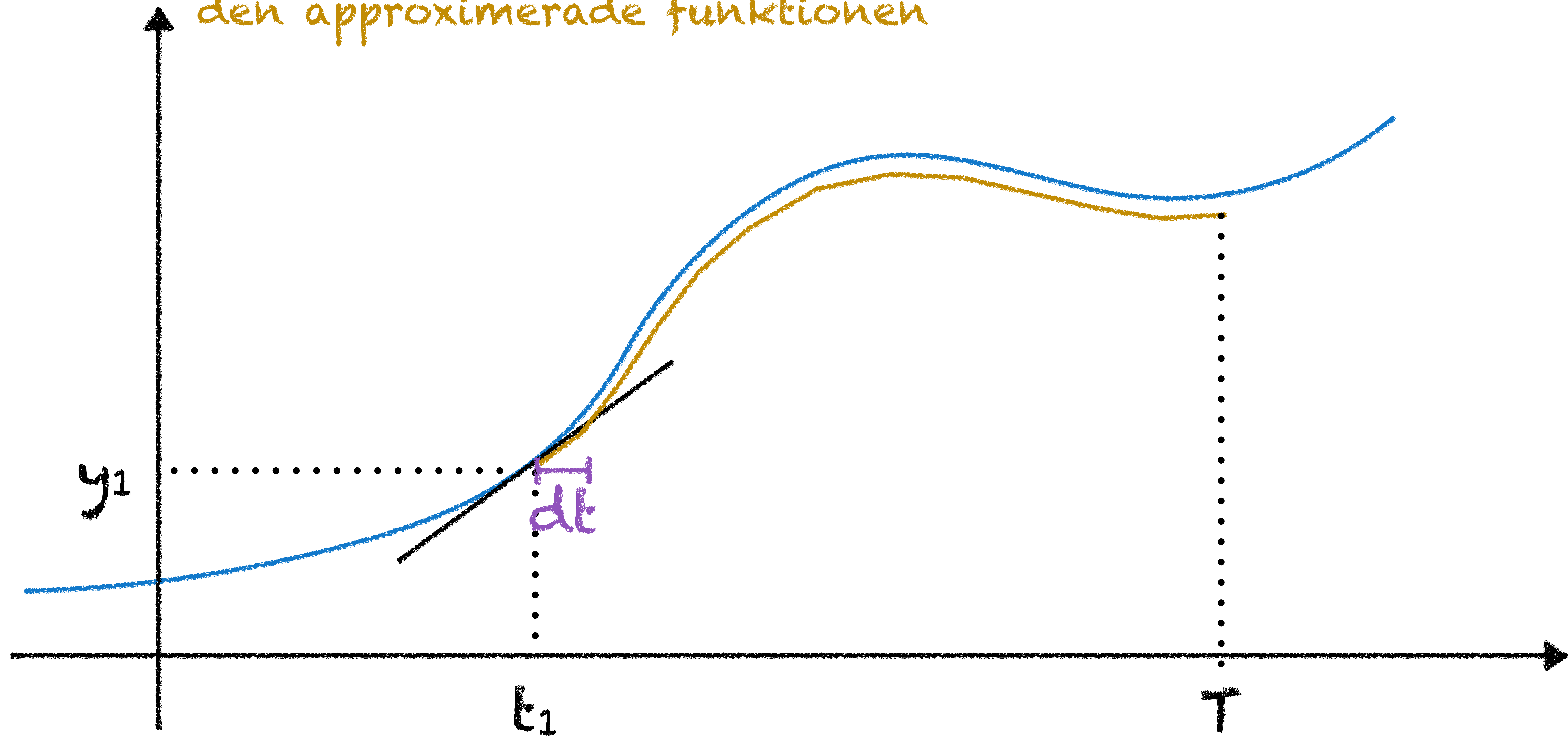
$$y'(t) = \frac{dy}{dt} \rightarrow dy = dt \cdot y'(t)$$

1. Starta från begynnelsevärdet (t_1, y_1): $t_c = t_1$, $y_c = y_1$
2. Iterativt uppdatera nästa värde:

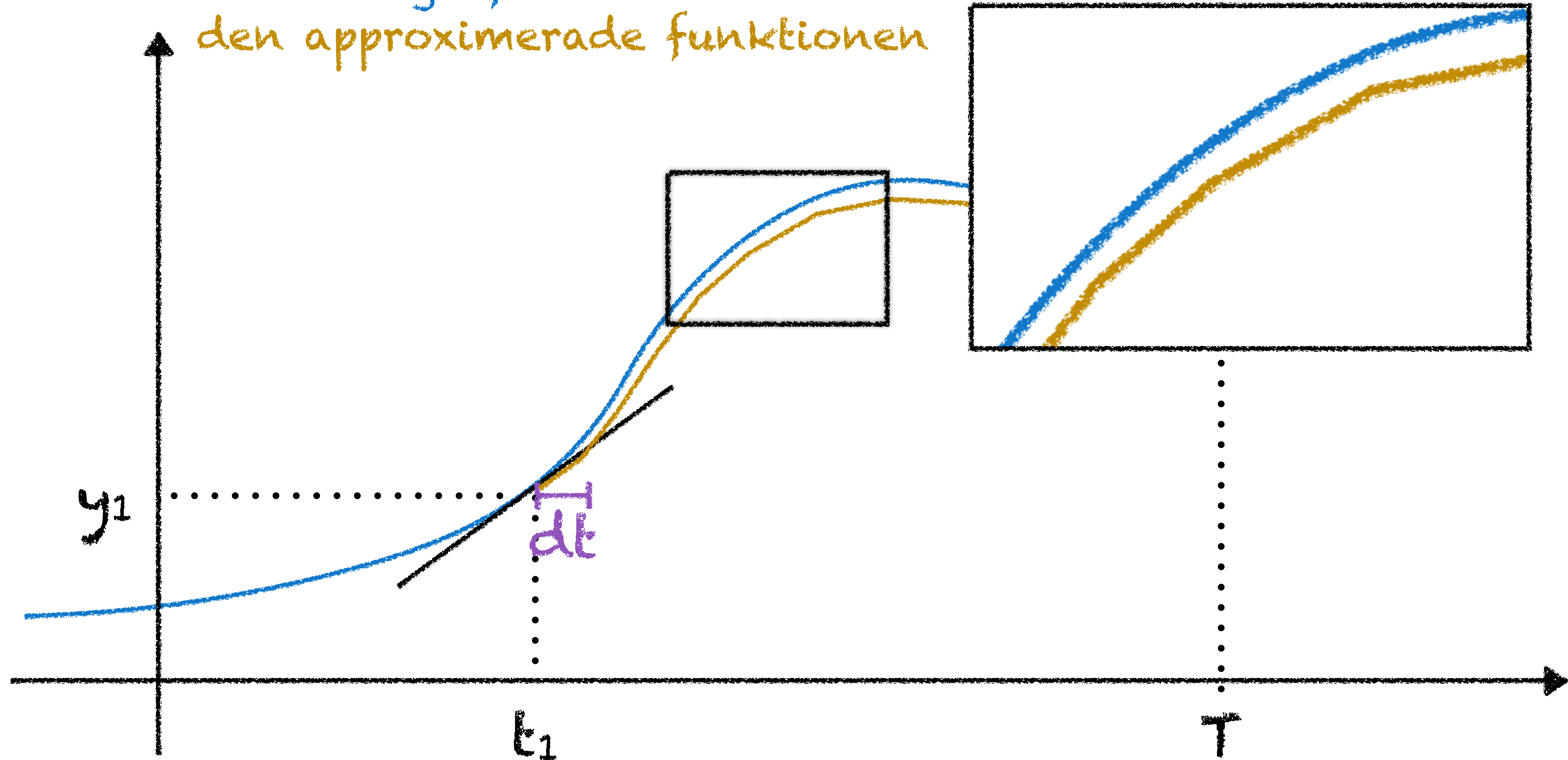
   $y_c = y_c + dt * f(t_c,y_c)$ # f är funktionen för y’

   $t_c = t_c + dt$
3. Upprepa steg 2 så länge $t_c <$ slutvärdet T
4. Vi returnerar vektorn med alla `t`– och `y`–värden vi räknat ut

Eulers metod forts.



Eulers metod forts.



Runge-Kutta

- Standardmetoden heter Runge-Kutta ‘RK45’.
- Den fungerar ungefär som Eulers metod men evaluerar funktionsvärdet i 4 punkter framåt i varje steg och tar ett viktat medelvärde av resultatet.

$y_c = y_c + dt * \mathbf{f(t_c,y_c)}$ ← Ersätts med ett medelvärde av 4 punkter

$t_c = t_c + dt$

Inbyggda funktioner

- `scipy` har en funktion `solve_ivp`
  som löser differentialekvation (ivp = initial value problems)

  Funktionen importeras:

  ```
  from scipy.integrate import solve_ivp
  ```

- Den anropas:

  ```
  solution = solve_ivp(yprime, [x_start,
  x_end], [y_start])
  ```

Exempel

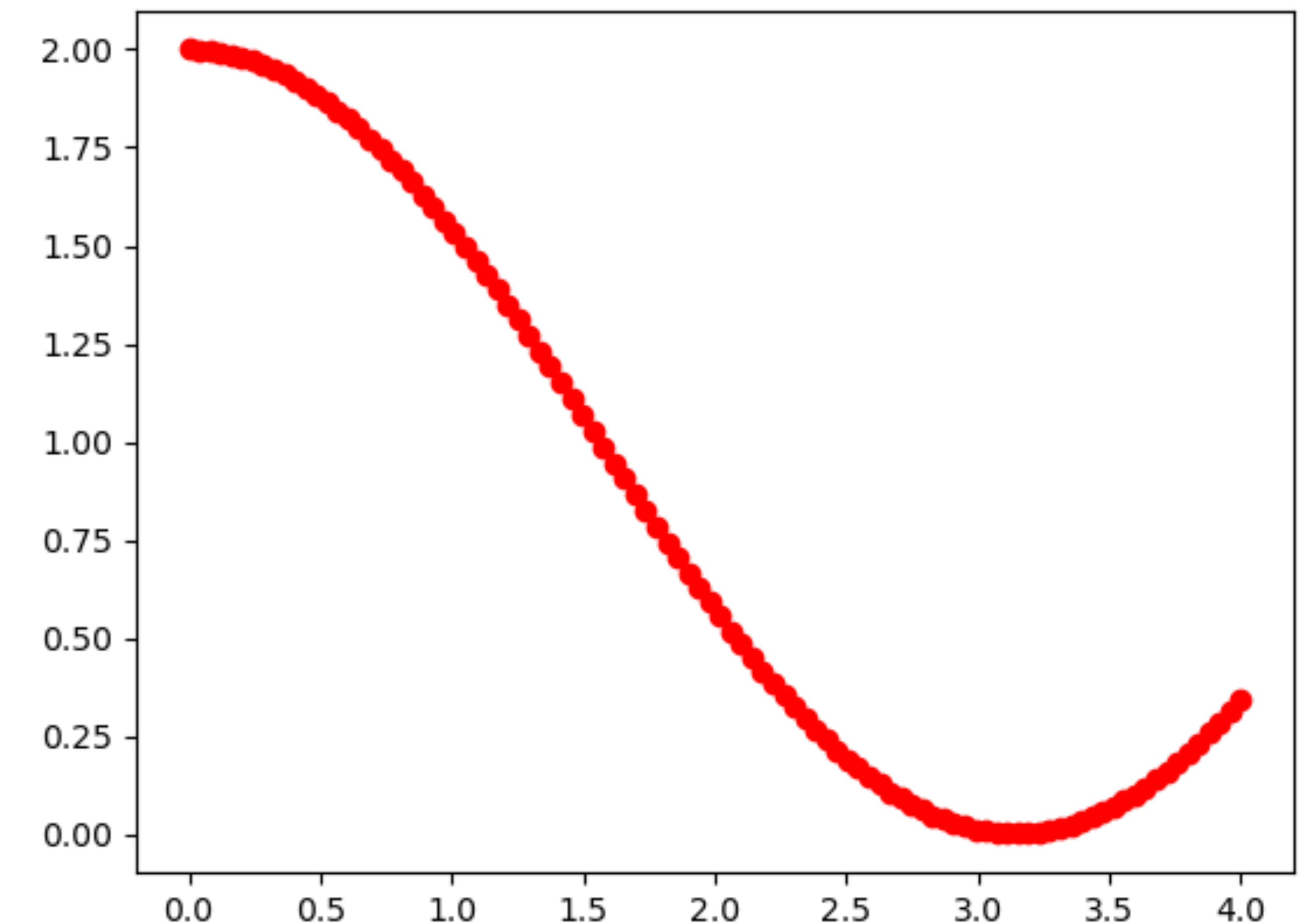
* Vi löser differentialekvationen:

$$y'(t) = -sin(t)$$
$$y(0) = 2$$
$$t = 0..4$$

Exempel: kod

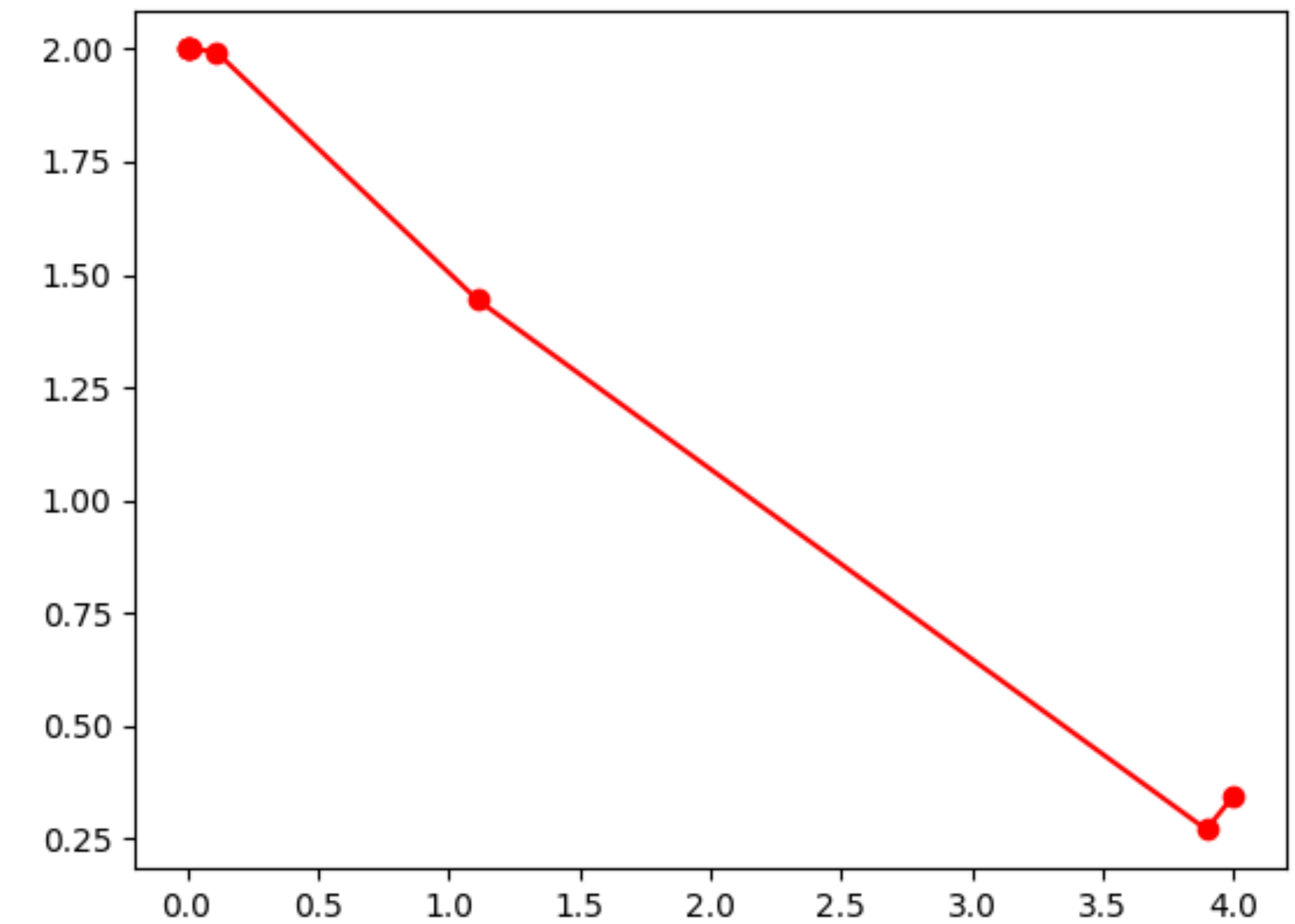
Vi löser differentialekvationen:

```
f = lambda t, y: -np.sin(t)
solution = solve_ivp(f, [0.0, 4.0], [2.], t_eval =
np.linspace(0, 4, num = 100))
t = solution.t
y = solution.y[0]

plt.plot(t, y, 'ro-')
plt.show()
```



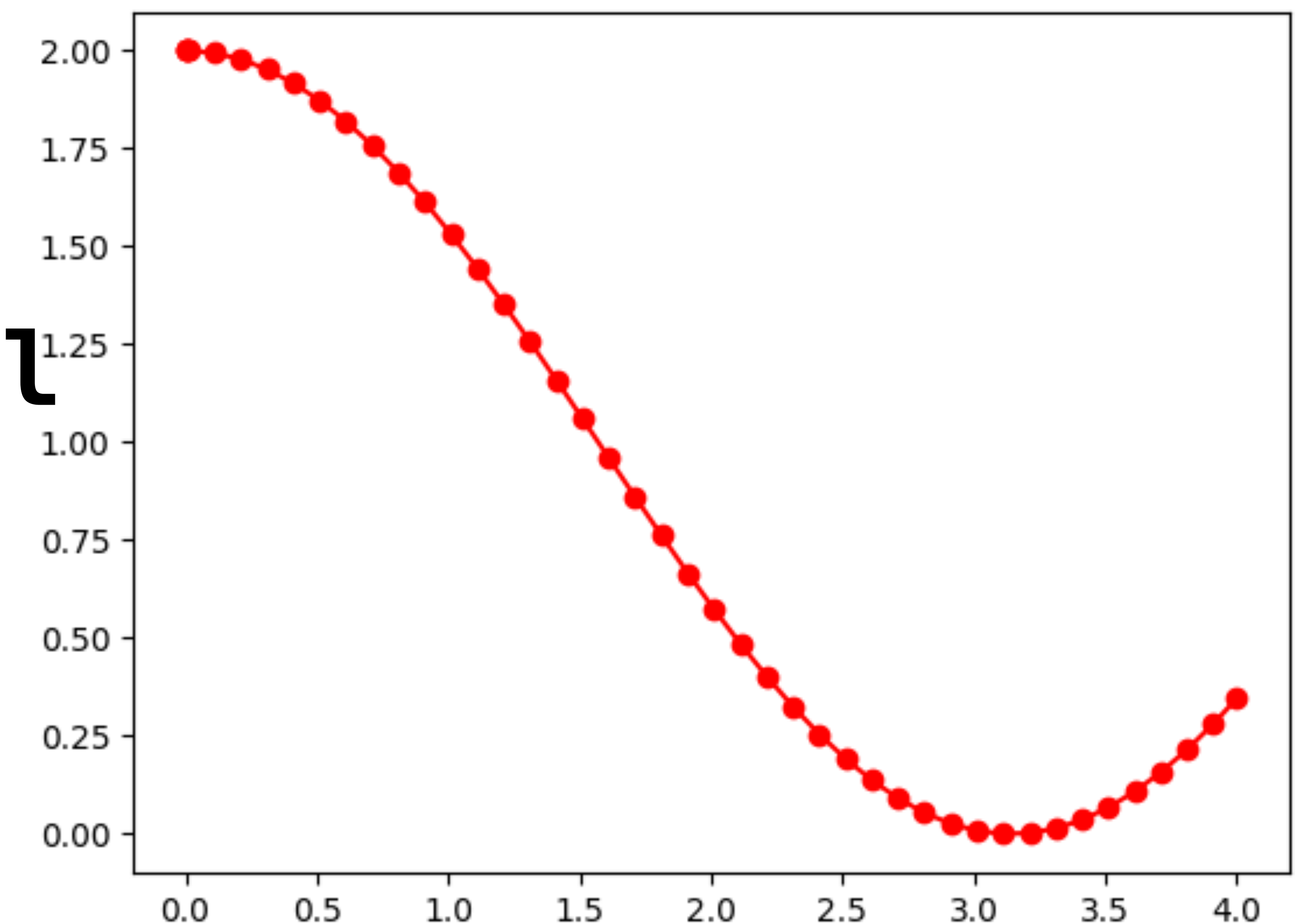
Kommentarer

* Om kurvan får väldigt få punkter kan vi lägga till argumentet `max_step` *eller* argumentet `t_eval` med punkter där funktionen ska evalueras:



1. `solution = solve_ivp(yprime, [0.0, 4.0], [2.], max_step =0.1)`

2. `solution = solve_ivp(yprime, [0.0, 4.0], [2.], t_eval = np.linspace(0, 4, num = 100))`



System av differentialekvationer

* Rävar x och kaniner y
* Om det finns många kaniner så
  äter rävarna dem och förökar sig, fler rävar gör att kaninerna blir färre, då minskar också antalet rävar eftersom maten tar slut, och med färre rävar kan kaninerna bli fler etc …

$$x' = ax - bxy \qquad a = 0.0125$$

$$y' = dxy - cy \qquad b = 0.0002$$

$$x(0) = 75 \qquad c = 0.015$$

$$y(0) = 110 \qquad d = 0.00018$$

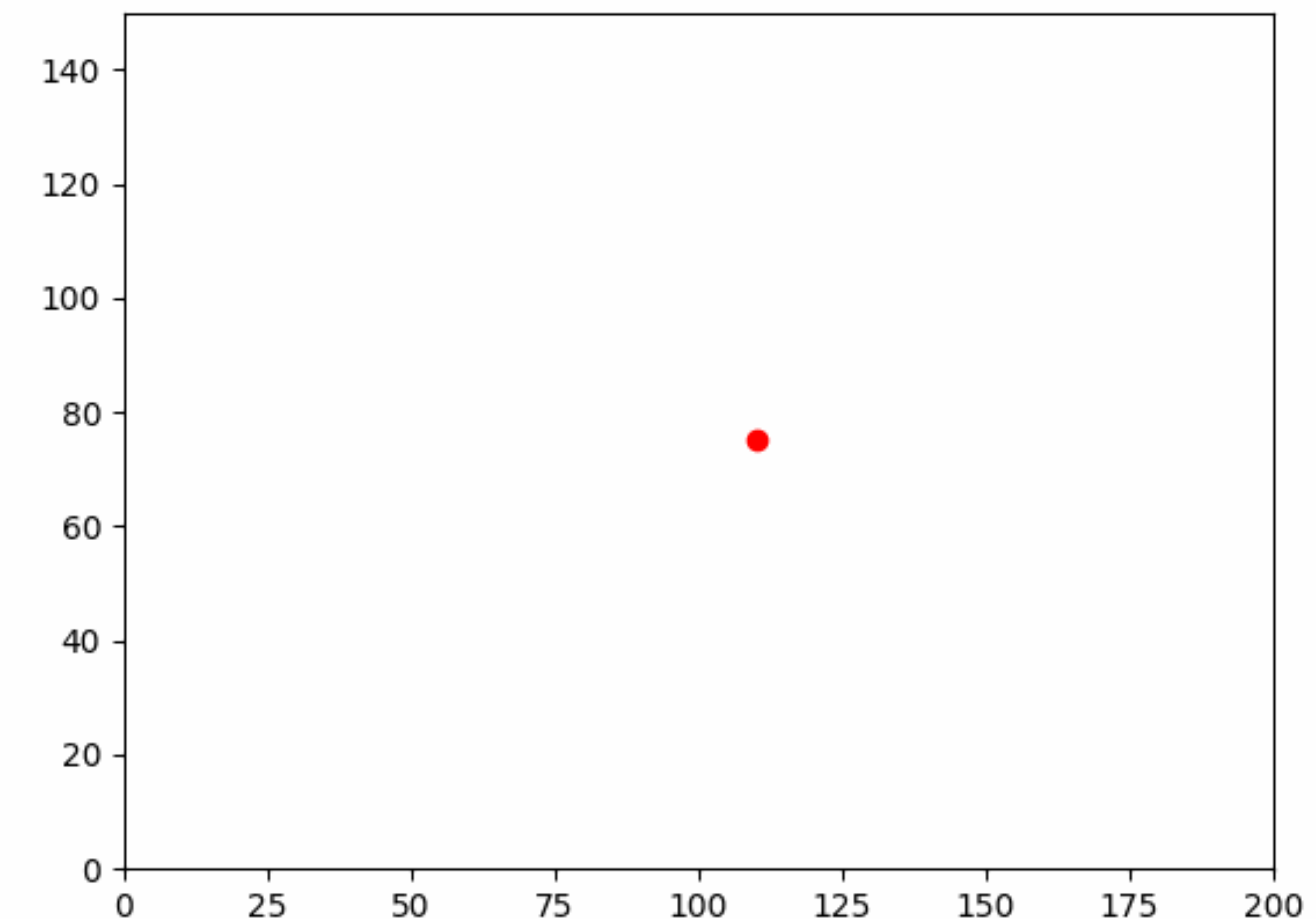
System av differentialekvationer

* `solve_ivp` kan även lösa **system** av differentialekvationer
* Parametern `y0 = [y_start]` kan innehålla många värden.
* `solution.y` är en matris där rad 0 är första funktionen, rad 1 andra osv.
* `solve_ivp` löser alla ekvationerna på en gång

Exempel

* Vår funktion `yprime` returnerar en vektor med nya värden för derivatorna.

```
def yprime(t, yc):
    a, b, c, d = 0.0125, 0.0002, 0.015, 0.00018
    x = yc[0]
    y = yc[1]
    xprime = a * x - b * x * y
    yprime = d* x*y - c* y
    return [xprime, yprime]
```



System av differentialekvationer

* `solve_ivp` kan även lösa **system** av differentialekvationer
* Parametern `y0 = [y_start]` kan innehålla många värden.
* `solution.y` är en matris där rad 0 är första funktionen, rad 1 andra osv.
* `solve_ivp` löser alla ekvationerna på en gång

Högre ordningens differentialekvationer

- Högre ordningens differentialekvationer kan skrivas om till ett system av första ordningens ekvationer

  $$y'' + 7y' - 3y = 0$$
  $$y(0) = 0,\ y'(0) = 1$$

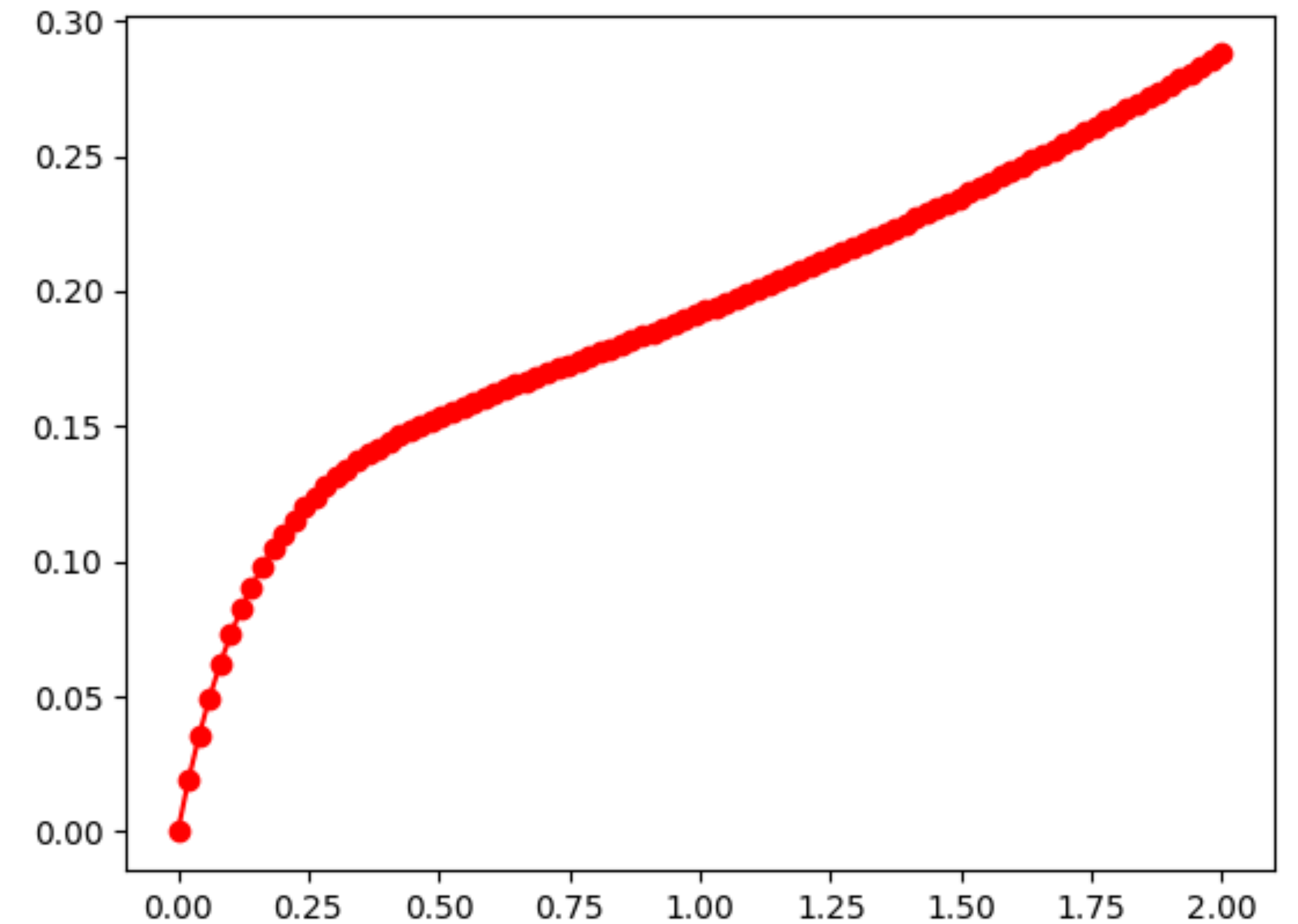
- Inför $y_1 = y,\ y_2 = y'$ vilket ger:

  $$y_1' = y_2$$
  $$y_2' = 3y_1 - 7y_2$$
  $$y_1(0) = 0,\ y_2(0) = 1$$

Exempel

* Då ska vår funktion yprime returnera en vektor med nya värden.

```
def yprime(t, yc):
    y1 = yc[0]
    y2 = yc[1]
    y1prime = y2
    y2prime = 3 * y1 - 7 * y2
    return [y1prime, y2prime]
```



Rep: Anpassa en kurva

* `polynomial.polyfit(x, y, n)` anpassar ett n:te gradspolynom enligt minsta kvadratmetoden.

* `polynomial.polyfit(x, y, 1)` anpassar en linje enligt minsta kvadratmetoden.

x	y
0	6.24
1	4.61
2	3.48
3	2.55
4	2.06
5	1.42
6	1.21
7	0.69
8	0.62
9	0.39
10	0.37

Anpassa kurvor

- Ibland kan vi behöva göra om problemet till linjär form för att sedan anpassa en kurva:

  $y(x) = a \cdot c^x$

Anpassa kurvor

- Ibland kan vi behöva göra om problemet till linjär form för att sedan anpassa en kurva:

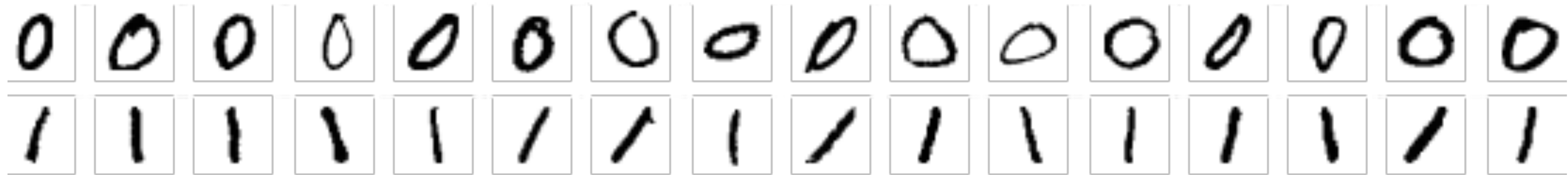
$$y(x) = a \cdot c^x \rightarrow log(y(x)) = log(a) + x \cdot log(c)$$

- Vi kan nu hitta värden för $log(a)$ och $log(c)$ med hjälp av `polyfit`. $a = e^{log(a)}$ eftersom det är den naturliga logaritmen.

Mer om inläsning av data, matriser …

* För att lära oss mer om inläsning av data och funktioner för matriser, ska vi bygga ett eget neuralt nätverk.
* *Det viktiga är inte att förstå hur neurala nätverk fungerar (det finns egna kurser i det) utan att lära sig vilka matrisfunktioner som finns*

Exempel: Neurala nätverk



- Vi vill känna igen handskrivna siffror (bara 0 och 1 för enkelhetens skull).
- Vi har en massa bilder med 28 x 28 pixlar i gråskala 0 (svart) och 255 (vitt). Så varje bild har 784 tal. Varje bild har en *label,* dvs siffran bilden föreställer.
- Vi vill automatiskt känna igen vilken siffra det är.

Label: 1

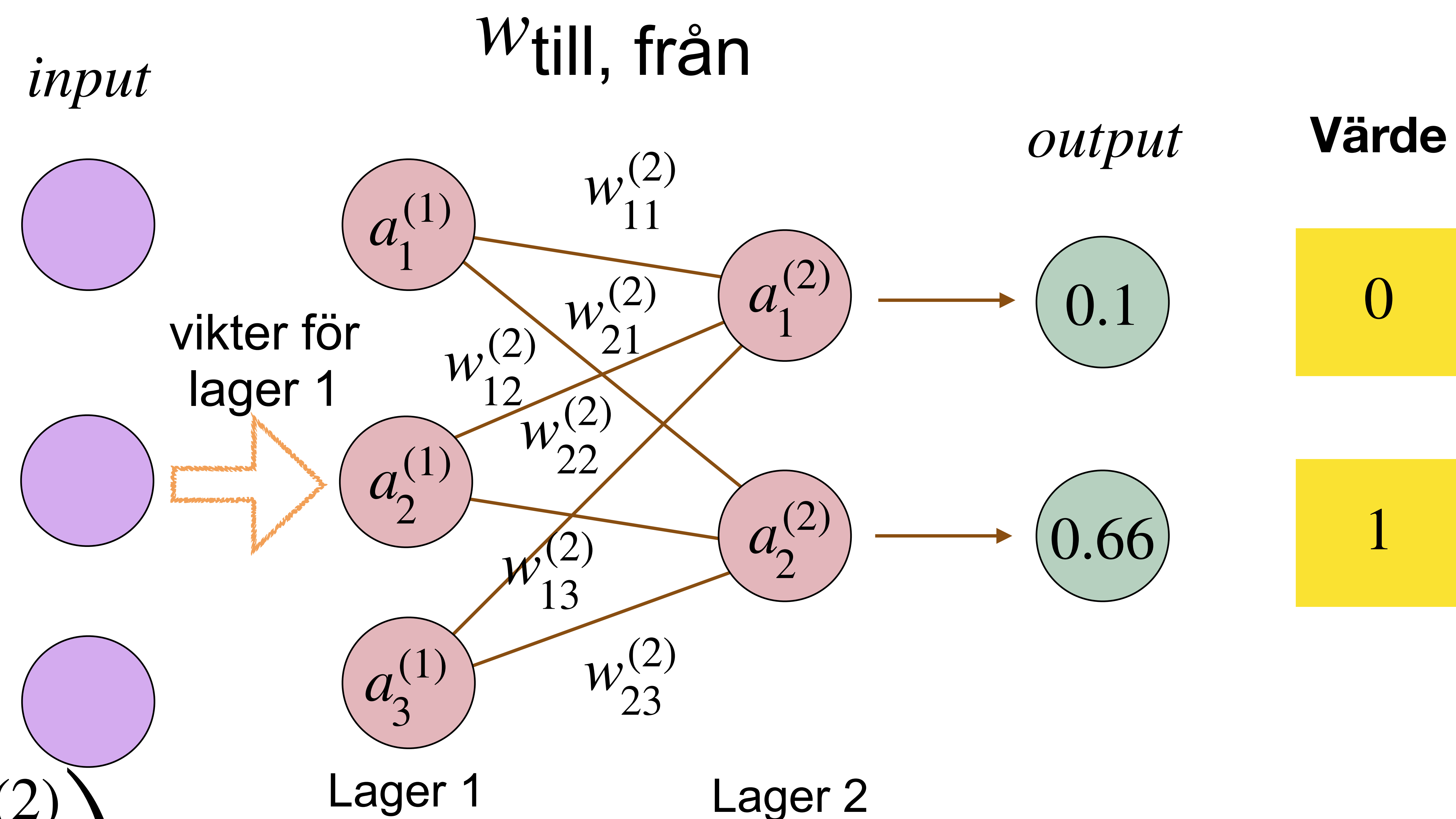
Exempel: Neurala nätverk

* $z_1^{(2)} = \sum_i w_{1i} a_i^1 + b_1^{(2)}$
* Aktiveringsfunktion $a_1 = \sigma(z_1)$

* Detta kan skrivas med hjälp av matriser: $Z = WA + B$

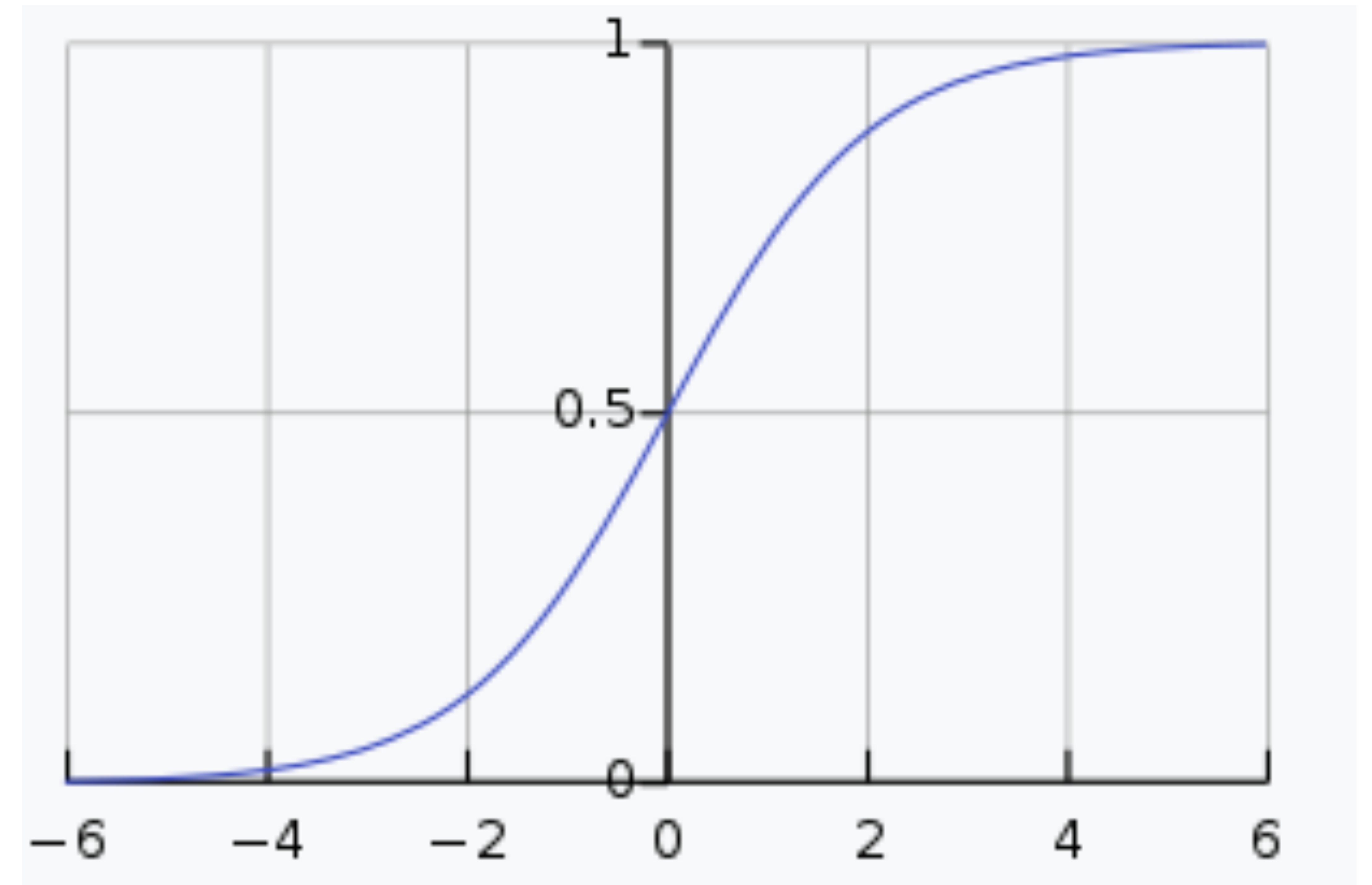
$$\begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} + \begin{pmatrix} b_1^{(2)} \\ b_2^{(2)} \end{pmatrix}$$

och $A_{\text{ut från lagret}} = \sigma(Z)$



Aktiveringsfunktion: sigmoid

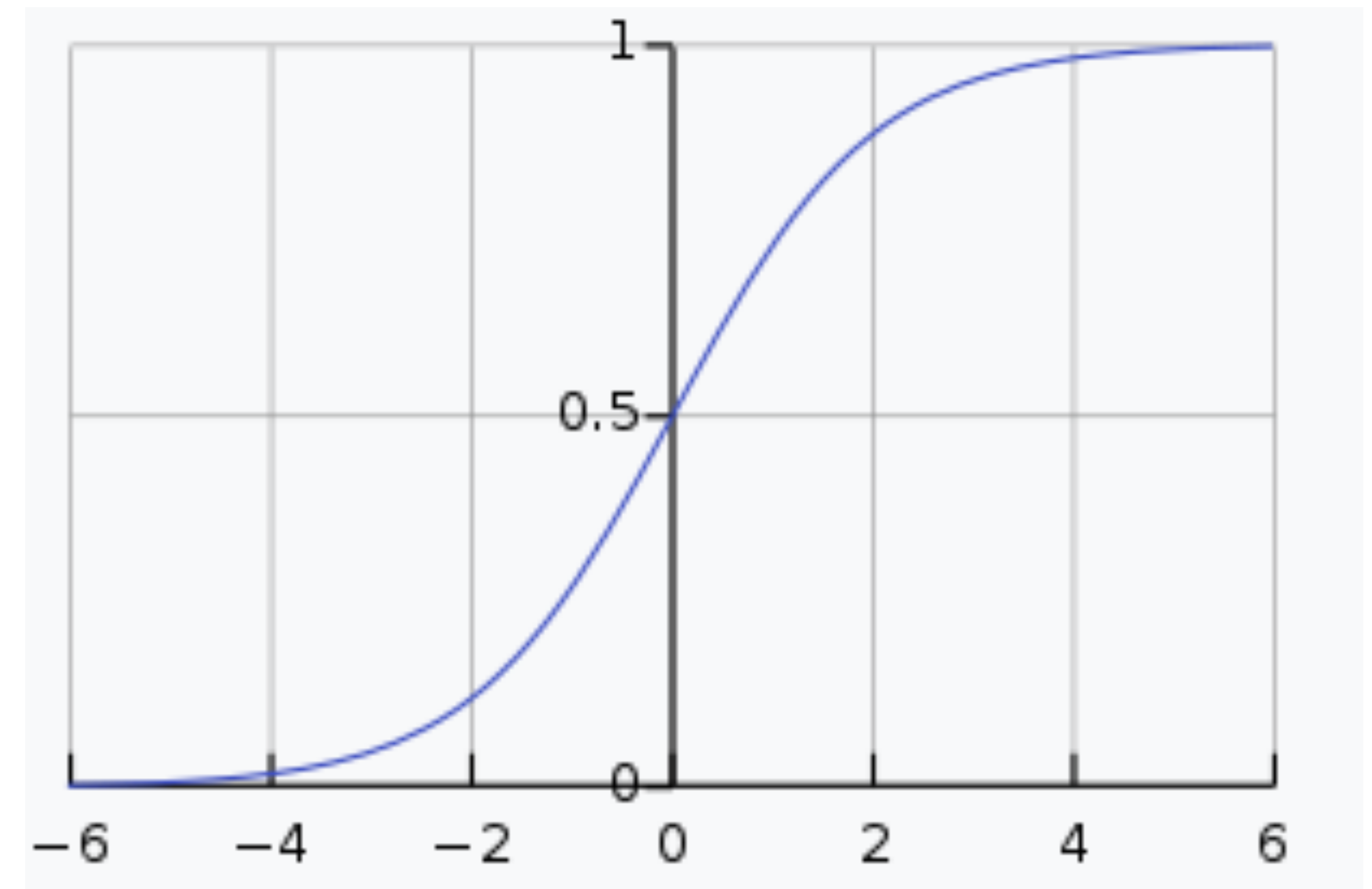
```
def sigma(x):
    return 1/(1 + np.exp(-x))
```



Aktiveringsfunktion: sigmoid

```
def sigma(x):
    return 1/(1 + np.exp(-x))
```

```
def sigma_derivative(x):
    sig = sigma(x)
    return sig*(1-sig)
```



Exempel: Neurala nätverk

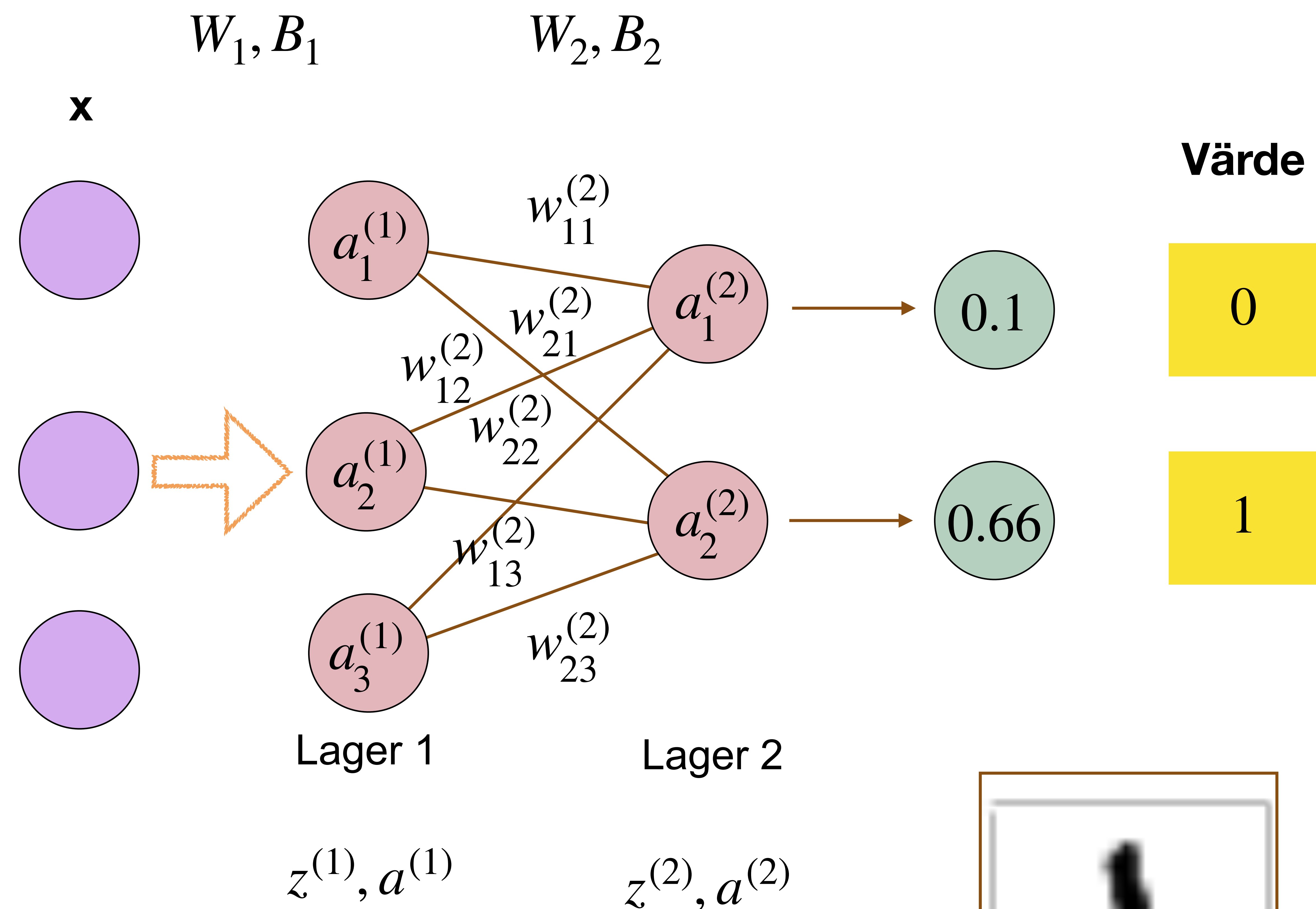
* I varje neuron:
* Inkommande värden:

  $Z = WA + B$

* $A_{\text{ut från lagret}} = \sigma(Z)$

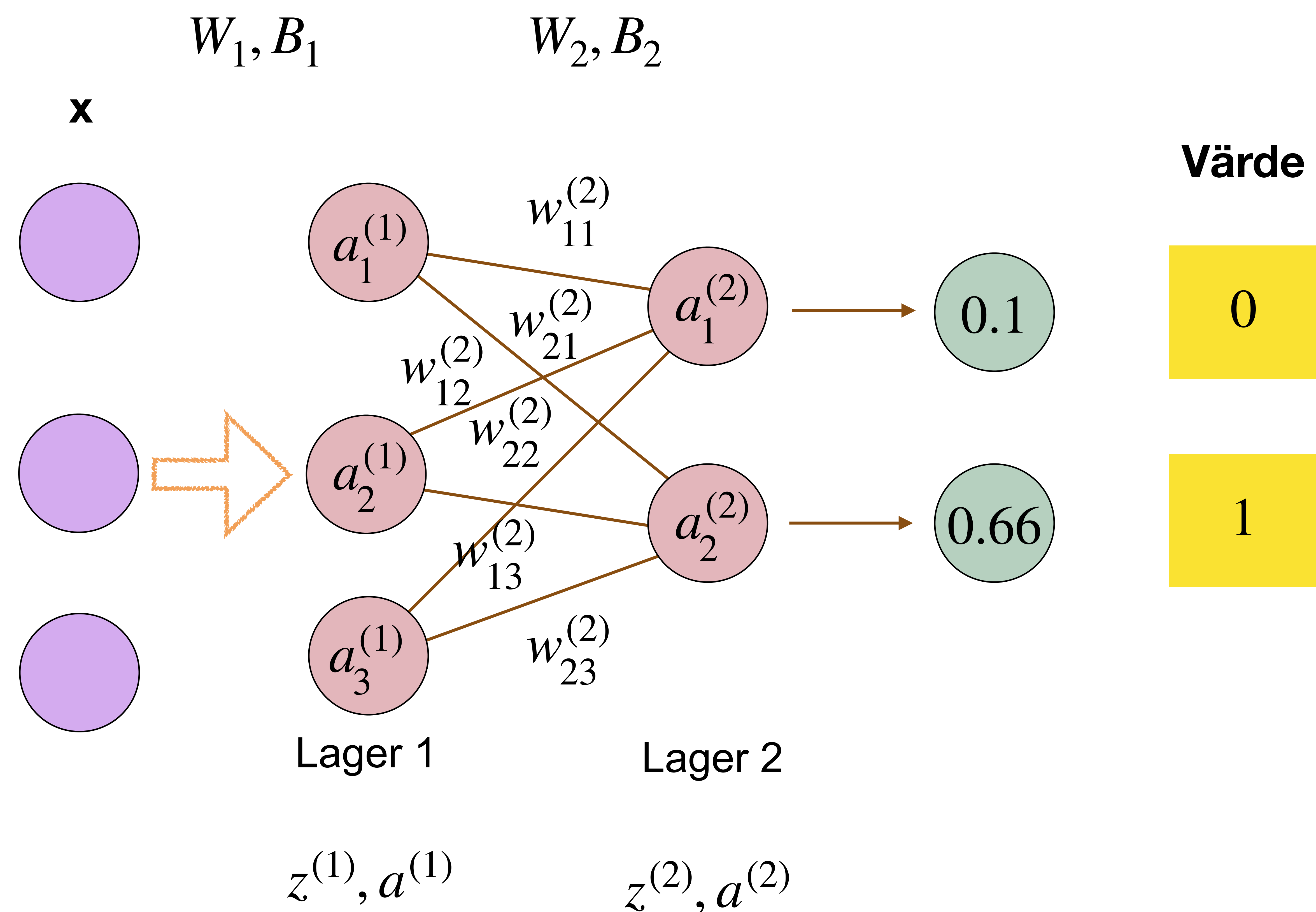
För nätverket till höger kan vi räkna klassificera inputvärden x så här:

```
z1 = np.matmul(W1, x) + B1
a1 = sigma(z1)
z2 = np.matmul(W2, a1) + B2
a2 = sigma(z2)
```



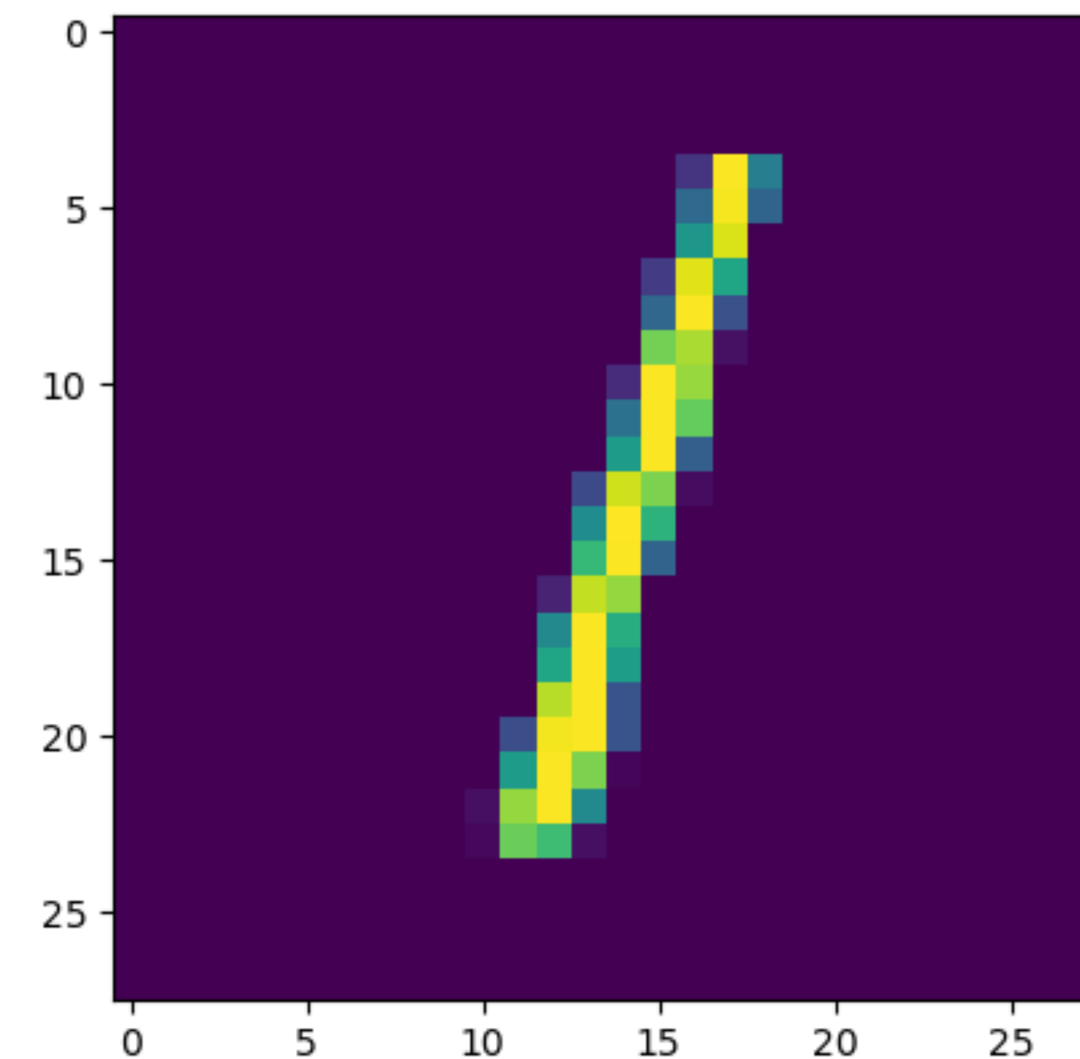
NN: prediktera värdet

```
def forward(x):
    z1 = np.matmul(W1, x) + B1
    a1 = sigmoid(z1)
    z2 = np.matmul(W2, a1) + B2
    a2 = sigmoid(z2)
    return z1, z2, x, a1, a2


def predict(x):
     return forward(x)[-1]
```



NN: prediktera värdet för 1

```
def forward(x):
    z1 = np.matmul(W1, x) + B1
    a1 = sigmoid(z1)
    z2 = np.matmul(W2, a1) + B2
    a2 = sigmoid(z2)
    return z1, z2, x, a1, a2

def predict(x):
     return forward(x)[-1]

result = predict(testdata1)
#[0.17650647 0.85379684]
```



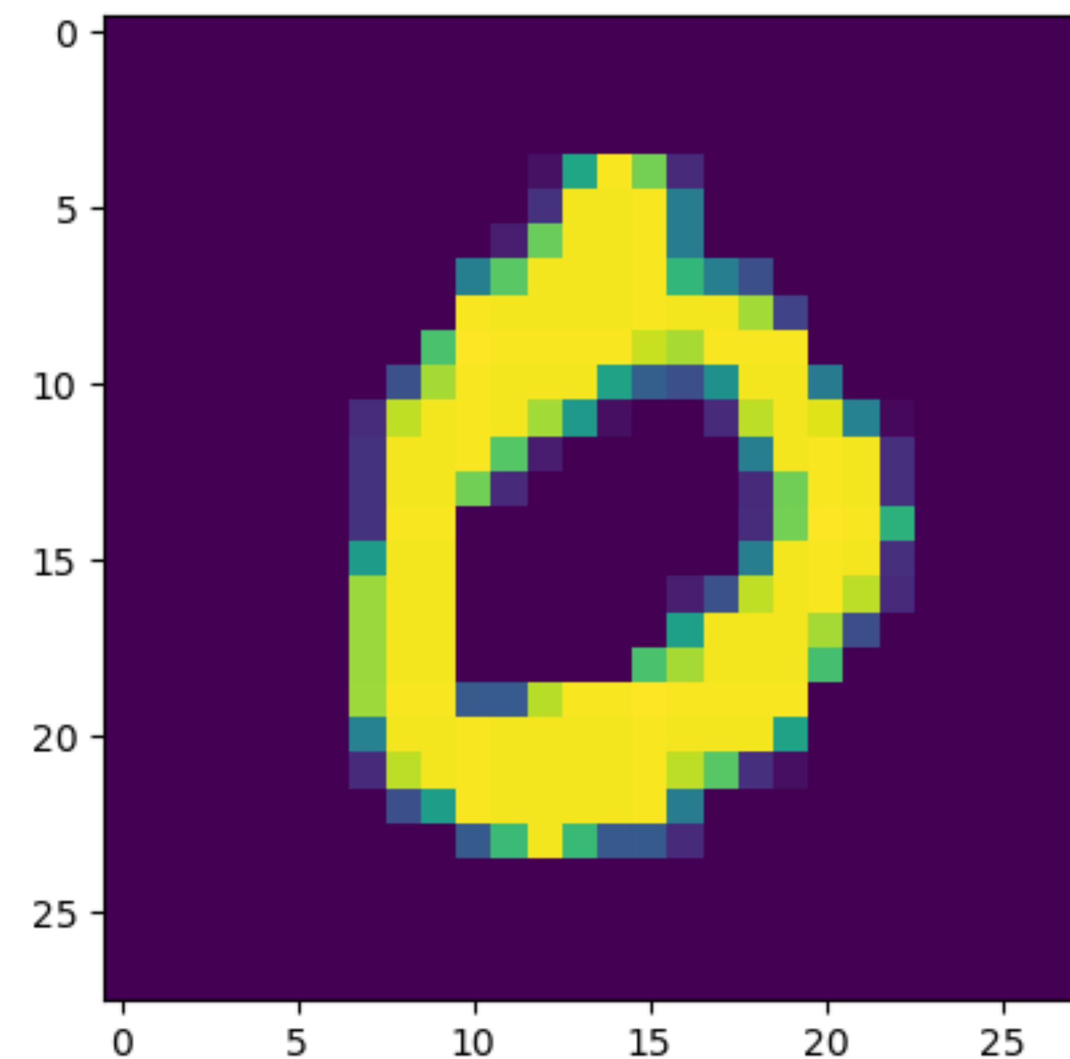
Värde

0.18	0
0.85	1

testdata: 784 värden i en vektor
image: 28 x 28 värden

NN: prediktera värdet för 0

```
def forward(x):
    z1 = np.matmul(W1, x) + B1
    a1 = sigmoid(z1)
    z2 = np.matmul(W2, a1) + B2
    a2 = sigmoid(z2)
    return z1, z2, x, a1, a2

def predict(x):
     return forward(x)[-1]

result = predict(testdata0)
#[0.93890367 0.09069057]
```



Värde

0.94	0
0.09	1

testdata: 784 värden i en vektor

image: 28 x 28 värden

Bygga nätverket

* Nu ska vi bygga nätverket från grunden!
* Vi ska göra följande steg:

1. Hämta all data av bilder och labels
2. Filtrera ut data till 0 och 1.
3. Se till att vår sigma-funktion fungerar
4. Träna vikterna …

Hämta data från nätet

- I labben kommer vi hämta text från en hemsida med `requests`

```
text = requests.get(‘https://cs.lth.se/edaa55/numpy/race').text
#Spara till fil:
with open('race.txt', 'w') as f: f.write(text)
```

Skriva till fil

* `np.savetxt('saved.txt', data)` skriver ut numpy/`arrayen` data till filen. Värdena separeras med ny rad och skriver ut med grundpotensform.

*
```
with open('saved2.txt', 'w') as f:
    f.write(‘hello’)
```
skriver ut en sträng. Argumentet ‘r’ = read och ‘w’ = write

Läsa och skriva till fil

- Mätdata kommer ofta i en fil.
- Den kan läsas in i ren python eller direkt in i en numpy array.

```
with open('test.txt', 'r') as f:
    lines = f.readlines()
    #gör nåt med lines
```

- Med numpy `loadtxt`:

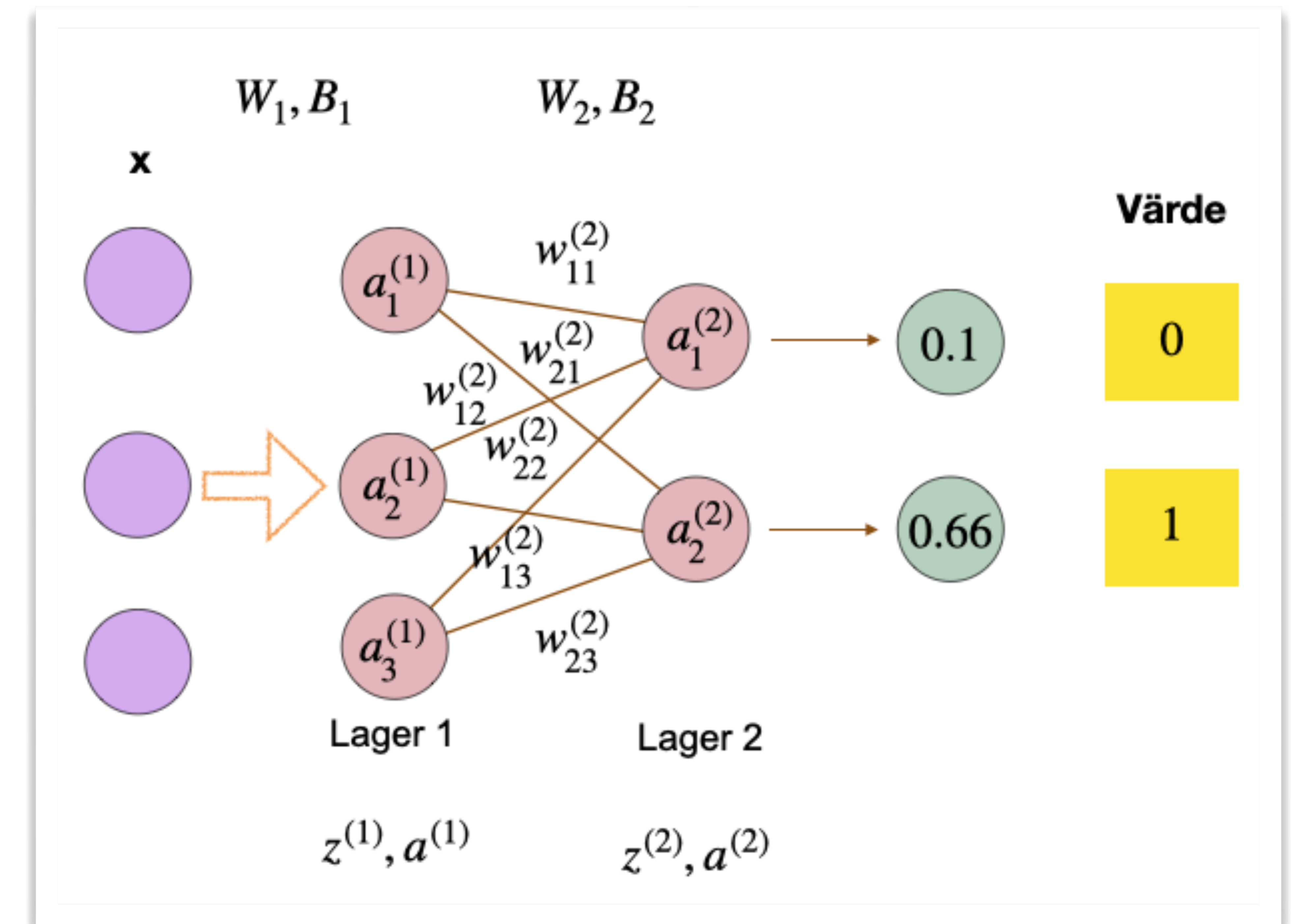
```
data = np.loadtxt('filnamn.txt')
data = np.loadtxt('filnamn.txt', delimiter =
',', dtype = int)
```

Exempel: Neurala nätverk

* I varje neuron:
* Inkommande värden:

$$Z = WA + B$$

* $A_{\text{ut från lagret}} = \sigma(Z)$



Alla bilder och motsvarandesiffror finns online i två filer.
https://fileadmin.cs.lth.se/cs/Education/EDAA55/numpy/trainingdata_images.txt
och
https://fileadmin.cs.lth.se/cs/Education/EDAA55/numpy/trainingdata_labels.txt

Hämta data från nätet: bilder

* Vi kan spara ner datan så här:

```
text = requests.get('https://fileadmin.cs.lth.se/cs/Education/EDAA55/
numpy/trainingdata_labels.txt').text
with open('trainingdata_labels.txt', 'w') as f: f.write(text)
labels = np.loadtxt('trainingdata_labels.txt', dtype= int)

#… och liknande för image_data fast det är inte heltalsvärden.
```

Några funktioner för matriser

- A.T transponerar matrisen
- `np.max(A)` ger största värdet i matrisen.
- `np.min(A)` ger minsta värdet i matrisen.
- `np.sum(A)` summerar alla tal i A
- `np.argmax(A)` ger index för det största elementet.
- `np.roll(v, 1)` roterar vektorn 1 steg bakåt.
- `A.reshape(…)` ändra dimensioner för matrisen.
- `np.where(…)` filtrerar på olika villkor.

Exempel: argmax och sum

```
# Vi vill ha det index med störst värde:

result = predict(testdata)
print('Best prediction:', np.argmax(result))

print(np.sum(result))
```

Exempel: reshape

```
# Vi gör om bilddata till en 28 x 28 bild och
image_data till kolonnvektorer:

images = training_data.reshape(len(training_data), 28,
28)
training_data = training_data.reshape(len(training_data),
784, 1)
```

Några funktioner för matriser

* `B = A >= 2`

```
[[False False False]
 [True False False]
 [ True  True False]]
```

* `B = A[A >= 2]`

```
[2 4 3]
```

$$\begin{pmatrix} 1 & 1 & -1 \\ 2 & 1 & 1 \\ 4 & 3 & -1 \end{pmatrix}$$

Några funktioner för matriser

- Vi kan ta ut alla index för ett villkor:
  `f = v[v > 4]` ger en vektor med alla element större än 4.
- `f = v[`*`villkor`*`]` ger en vektor med alla element som uppfyller villkoret.

Ta ut all data med 0 och 1 forts.

* Datan har bilder på siffror 0 - 9

Vi vill bara ha bilderna på 0 och 1.

```
training_data = training_data[labels < 2]
images = images[labels <2]
labels = labels[labels < 2]
```

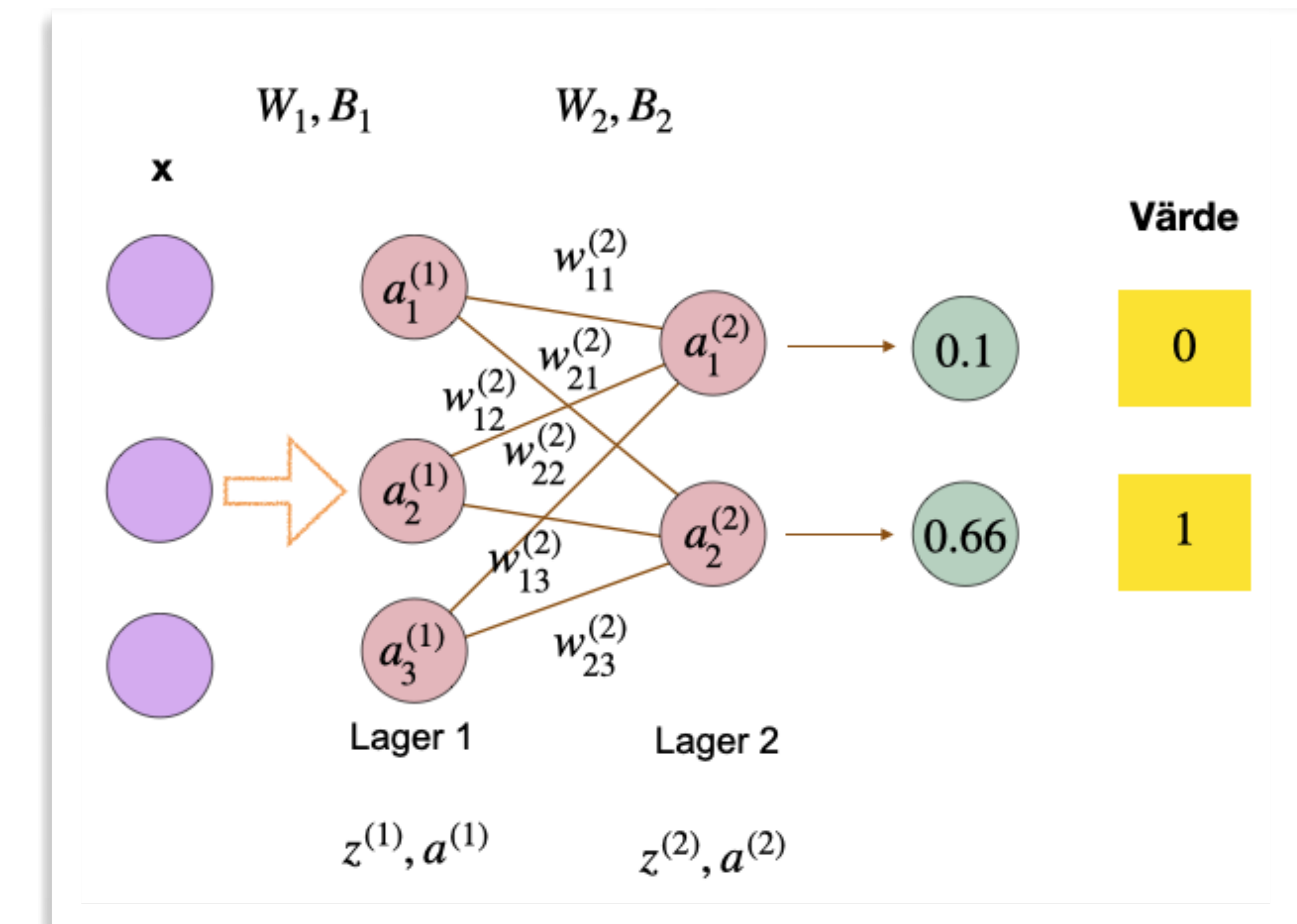
Dela upp i träningsdata och testdata

* Sen delar vi upp datan i träningsdata och testdata:

```
split = 12665
train_X = training_data[0:split]
test_X = training_data[split:]
train_y = labels[0:split]
test_y = labels[split:]
test_images = images[split:]
```

Förbered data

Gör om *output* från träningen till en kolonnvektor med 1 på det index som siffran föreställer. Detta är en *onehot*-vektor.




```
def onehot(y):
    v = np.array([0, 0])
    v[y] = 1 #[1, 0] för 0 och [0, 1] för 1
    return v.reshape(2, 1)

train_y = np.array([onehot(i) for i in train_y])
```

Filtrering av värden med where

* `np.where(A > 2)` ger alla index i A där värdet är större än 2 dvs (2, 0) och (2, 1) i form av listor på index som ska tas ut.
* `np.where(A > 2, 10, 0)` sätter alla värden större än 2 till 10 och alla andra till 0
* `np.where(A > 2, 0, A)` sätter alla värden större än 2 till 0 och behåller övriga

Exempel: where

`np.where(`*`villkor, om sant, annars`*`)`

Om vi vill filtrera bort alla värden < -500 och större än 500 i en vektor:

```
x = np.where(x < 500, x, 500)
x = np.where(x > -500, x, -500)
```

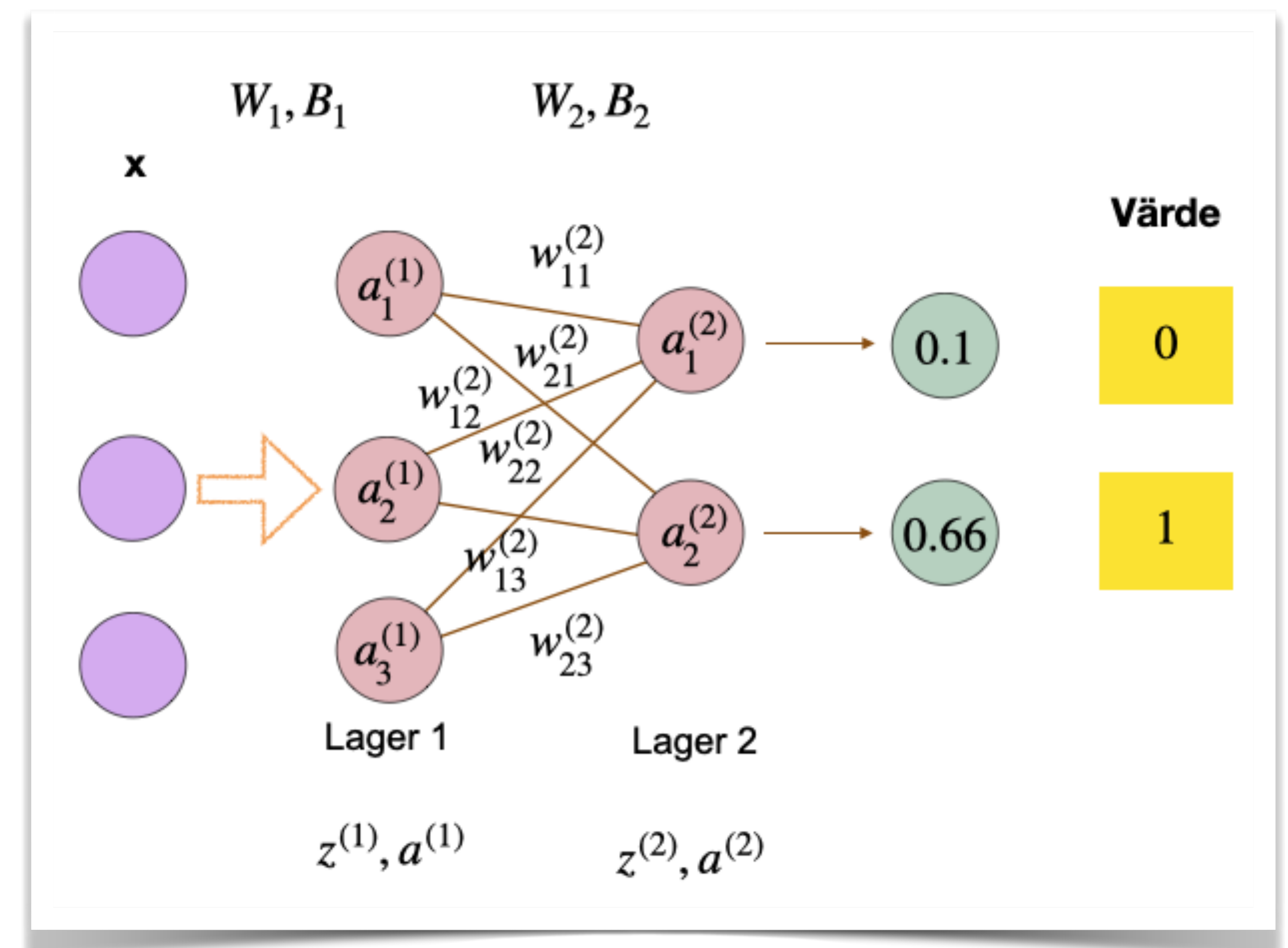
Matriser med slumpvärden

* `D = np.random.rand(2, 3)` #skapar en 2x3-matris med slumpmässiga element mellan 0 och 1.
* `D = np.random.randn(2, 3)` #skapar en 2x3-matris med slumpmässiga element mellan 0 och ±1 med normalfördelning.
* `D = np.random.randint(0, 10, size = (2, 3))` #skapar en 2x3-matris med slumpmässiga heltalselement mellan 0 och 9.
* `v = np.random.randint(-3, 4, size = 10)` #skapar en vektor med slumptal från -3 till 3.

Matriser med slumpvärden

* Skapa ett matriser med slumpmässiga startvikter

```
N1 = 3
N2 = 2
INPUT = 784
W1 = np.random.randn(N1, INPUT)
B1 = np.random.randn(N1, 1)
W2 = np.random.randn(N2, N1)
B2 = np.random.randn(N2, 1)
```



Uppdatera vikterna

Vi uppdaterar vikterna iterativt genom att:

1. Räkna ut felet för **ett exempel** i taget
   i sista lagret med minsta kvadratmetoden.
2. Räkna ut derivatan för varje vikt och b-värde genom att gå bakåt genom nätverket med hjälp av **kedjeregeln**. Detta kallas *backpropagation*.
3. Addera förändringarna från några exempel (en **batch**) och ta ett litet steg i **motsatt** riktning av derivatan. Detta kallas *gradient descent.*
4. Gå flera gånger genom alla träningsexempel.

Uppdatera vikterna

Vi uppdaterar vikterna iterativt genom att:

1. Räkna ut felet för **ett exempel** i taget
   i sista lagret med minsta kvadratmetoden.
2. Räkna ut derivatan för varje vikt och b-värde genom att gå bakåt genom nätverket med hjälp av **kedjeregeln**. Detta kallas *backpropagation*.
3. Addera förändringarna från några exempel (en **batch**) och ta ett litet steg i **motsatt** riktning av derivatan. Detta kallas *gradient descent.*
4. Gå flera gånger genom alla träningsexempel.

Beräkna felet i sista lagret

y är det riktiga värdet för träningsexemplet.

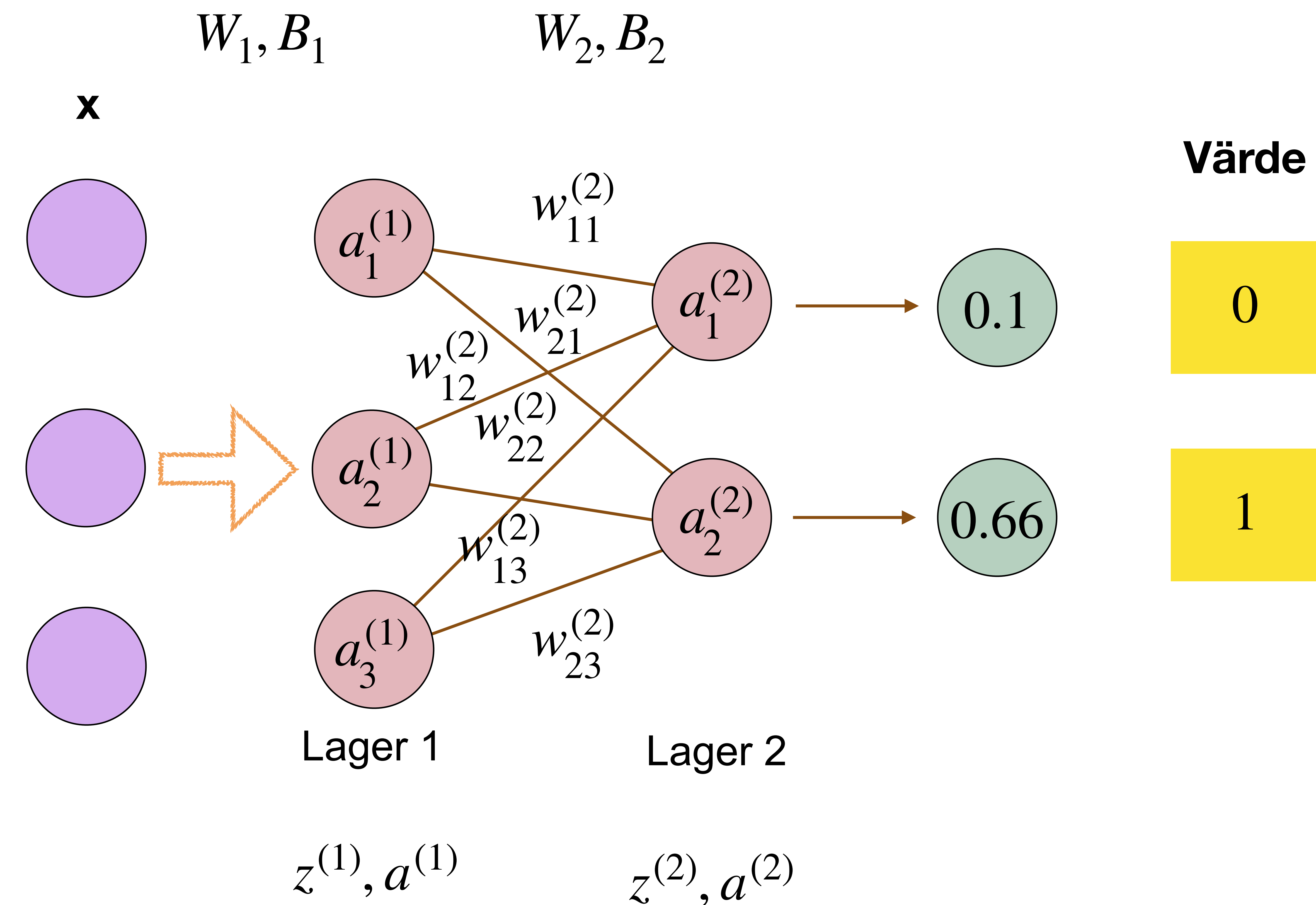
Felet C för ett exempel

$$C = (a^{(2)} - y)^2 = \left(\begin{bmatrix} 0.1 \\ 0.66 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right)^2$$

Den negativa derivatan

$$\frac{dC}{da^{(2)}} = -2(a^{(2)} - y) = -2 * \left(\begin{bmatrix} 0.1 \\ 0.66 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} -0.2 \\ 0.68 \end{bmatrix}$$

Vi vill ändra vikterna så att a_1^2 minskar lite och a_2^2 ökar mycket.



Kedjeregeln

För att veta hur vi ska ändra vikterna använder vi kedjeregeln (inre derivata gånger yttre derivata).

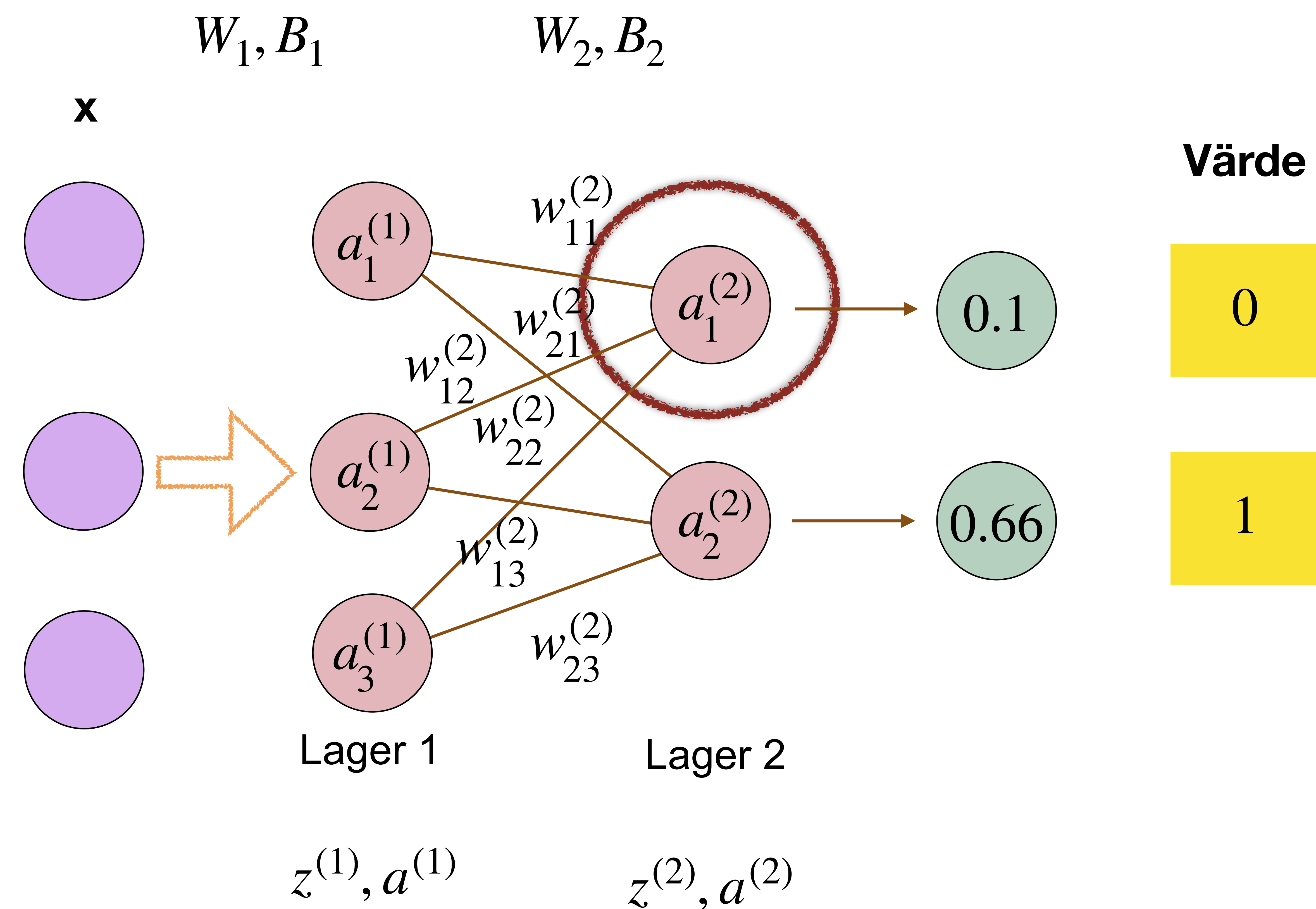
$$y = f(g(x))$$

$$y' = f'(g(x)) \cdot g'(x)$$

$$x \rightarrow z^{(1)} \rightarrow a^{(1)} \rightarrow z^{(2)} \rightarrow a^{(2)} \rightarrow C$$

Det går att skriva derivatorna explicit, t ex:

$$\frac{dC}{dz_1^{(2)}} = \frac{dC}{da_1^{(2)}} \cdot \frac{da_1^{(2)}}{dz_1^{(2)}}$$



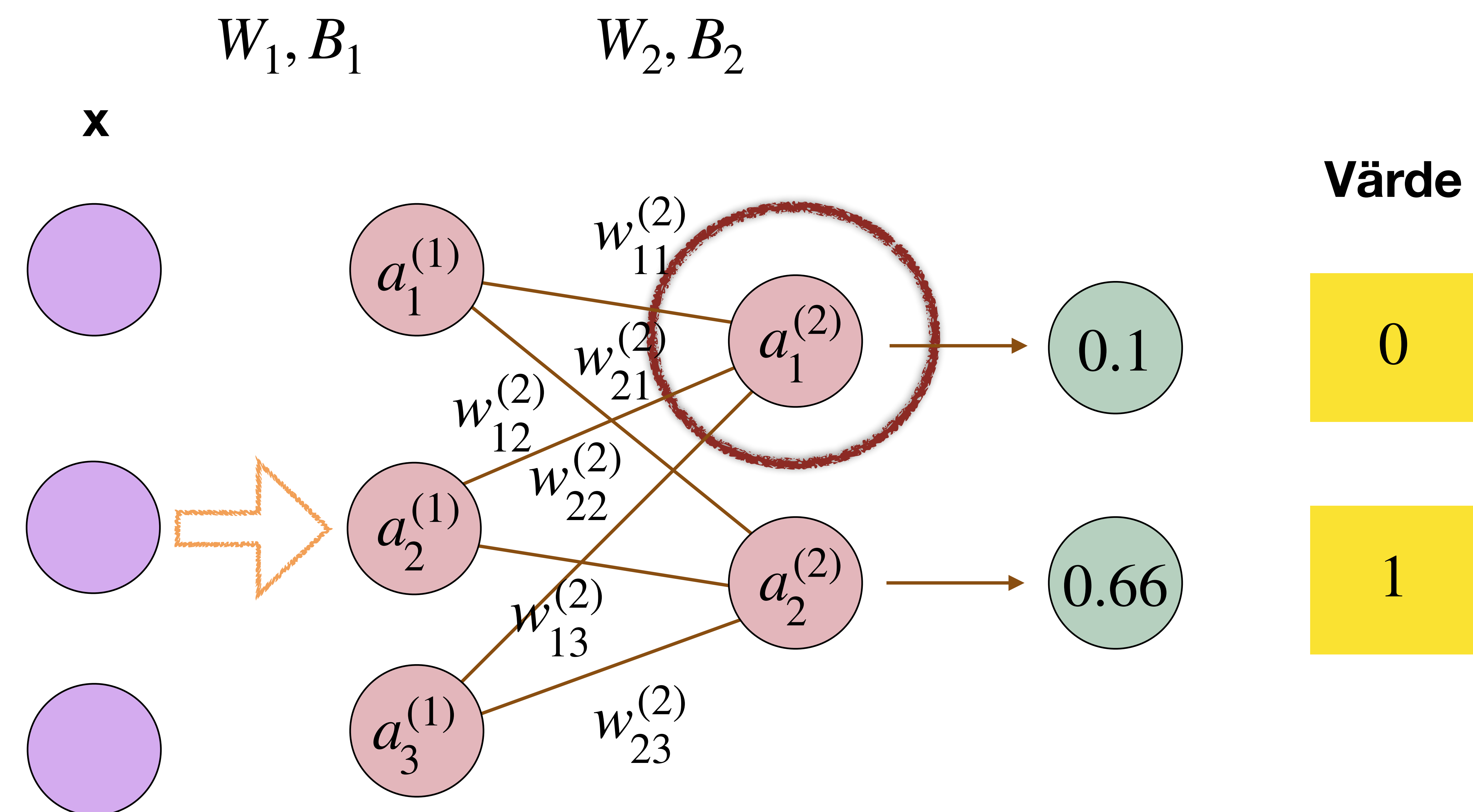
Kedjeregeln

För att veta hur vi ska ändra vikterna använder vi kedjeregeln (inre derivata gånger yttre derivata).

$$y = f(g(x))$$

$$y' = f'(g(x)) \cdot g'(x)$$

$$x \rightarrow z^{(1)} \rightarrow a^{(1)} \rightarrow z^{(2)} \rightarrow a^{(2)} \rightarrow C$$



Det går att skriva derivatorna explicit, t ex:

$$\frac{dC}{dz_1^{(2)}} = \frac{dC}{da_1^{(2)}} \cdot \frac{da_1^{(2)}}{dz_1^{(2)}} = -2(a_1^{(2)} - y_1) \cdot \sigma'(z_1^{(2)})$$

Ekvationer för att beräkna derivatorna

Sista lagret:

$$\Delta z^{(2)} = -2 \cdot (a^{(2)} - y) * \sigma'(z^{(2)})$$

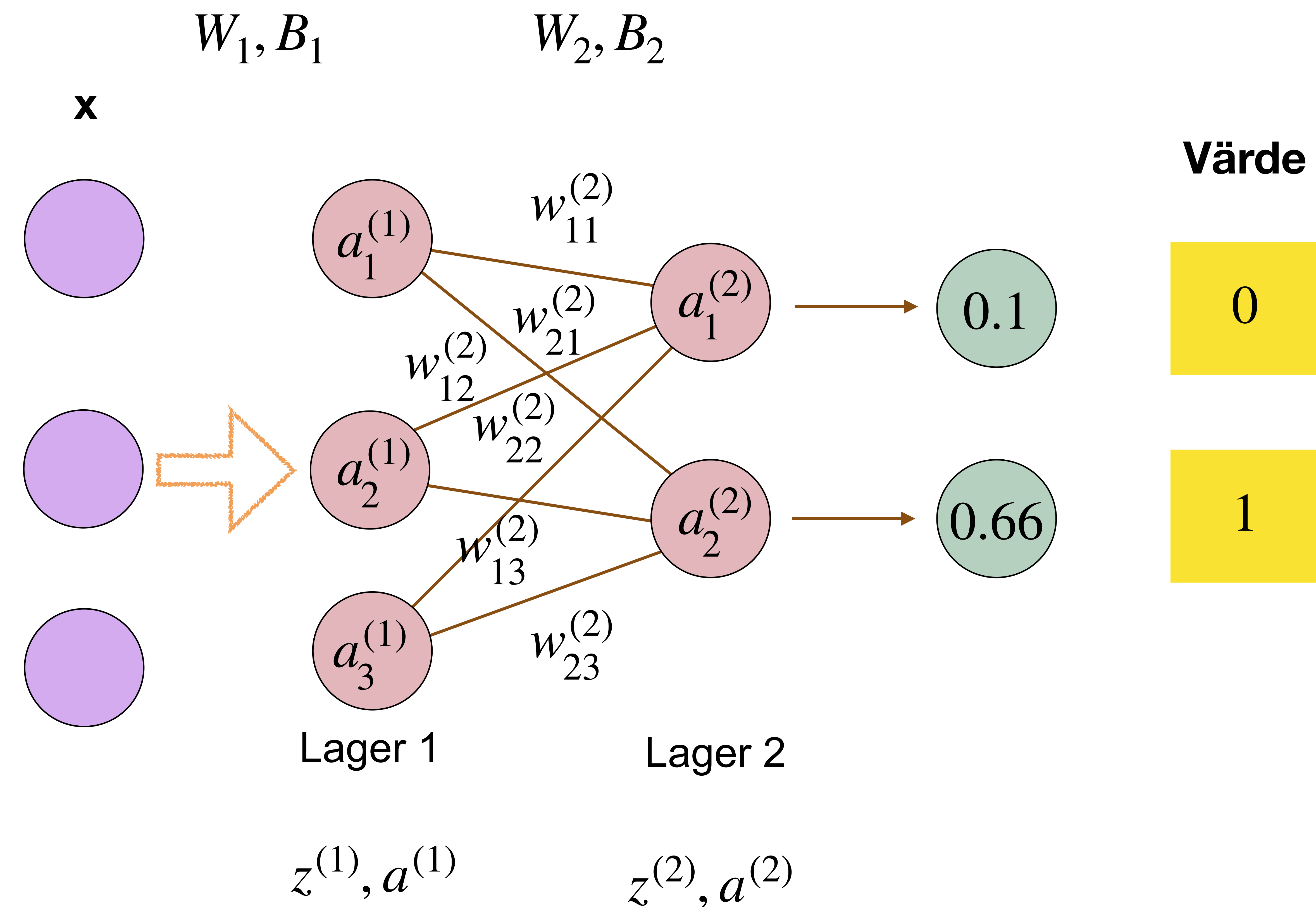
Där $\Delta z^{(2)}$ är vektorn med derivator

I övriga lager:

$$\Delta B^{(l)} = \Delta z^{(l)}$$

$$\Delta W^{(l)} = \Delta z^{(l)} \cdot (a^{(l-1)})^T$$

$$\Delta z^{(l-1)} = \left((W^{(l)})^T \cdot \Delta z^{(l)} \right) * \sigma'(z^{(l-1)})$$



back propagation

```
def backpropagation(y, z1, z2, x, a1, a2):
    deltas_z2 = - 2 *(a2 - y)*sigma_derivative(z2)
    deltas_b2 = deltas_z2
    dw2 = np.matmul(deltas_z2, a1.T) # 2 x 1 mul 1 x 3
    deltas_z1 = np.matmul(W2.T,deltas_z2)*sigma_derivative(z1)
    db1 = deltas_z1
    dw1 = np.matmul(deltas_z1, x.T)
    return dw1, dw2, db1, db2
```

Härledning av ekvationerna

Sista lagret:

$$\Delta z^{(2)} = -2 \cdot (a^{(2)} - y) * \sigma'(z^{(2)})$$

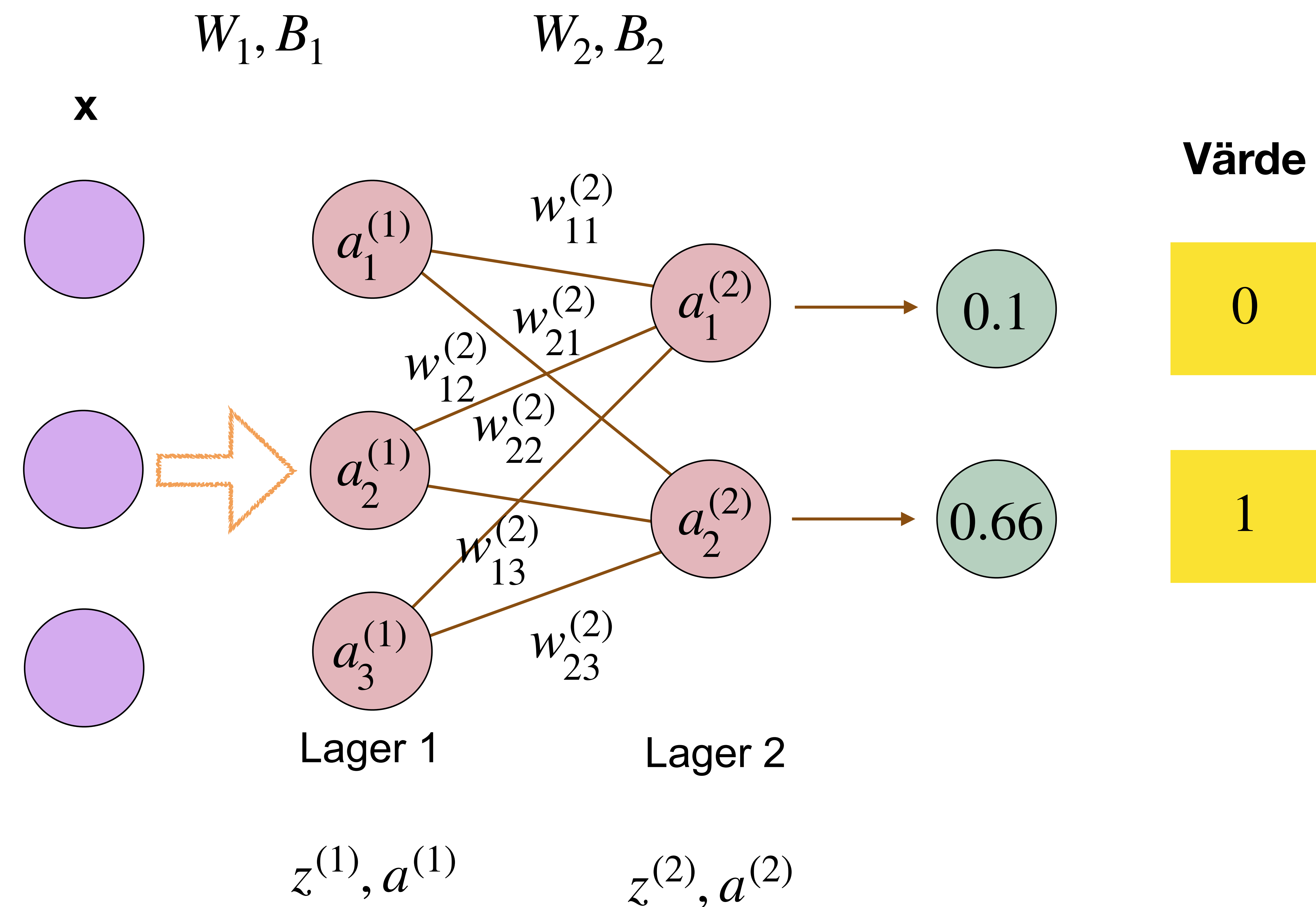
Där $\Delta z^{(2)}$ är vektorn med derivator

I övriga lager:

$$\Delta B^{(l)} = \Delta z^{(l)}$$

$$\Delta W^{(l)} = \Delta z^{(l)} \cdot (a^{(l-1)})^T$$

$$\Delta z^{(l-1)} = \left((W^{(l)})^T \cdot \Delta z^{(l)} \right) * \sigma'(z^{(l-1)})$$



Beräkna felet i sista lagret

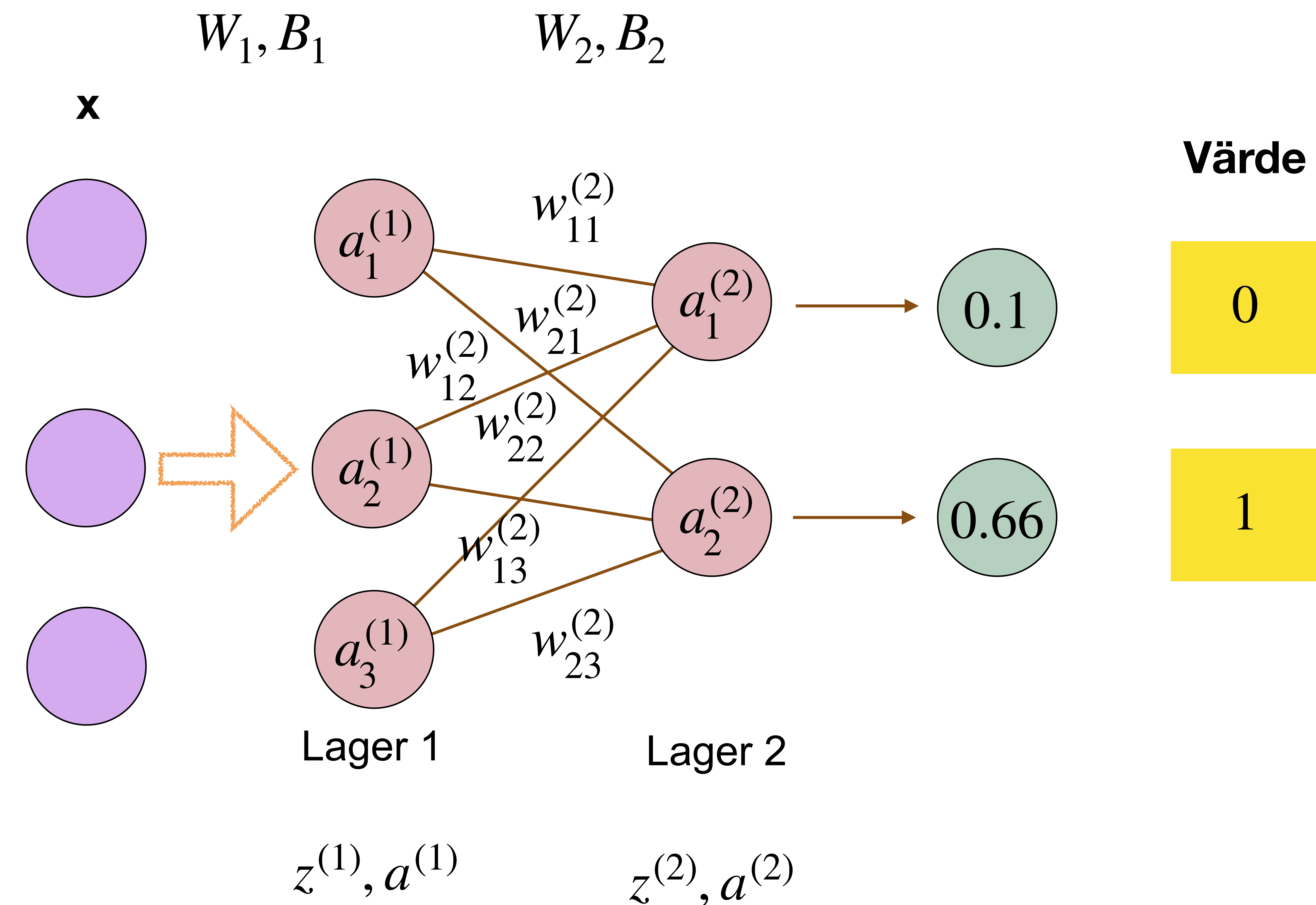
y är det riktiga värdet för träningsexemplet.
Felet C för ett exempel

$$C = (a^{(2)} - y)^2 = \left(\begin{bmatrix} 0.1 \\ 0.66 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right)^2$$

Den negativa derivatan

$$\frac{dC}{da^{(2)}} = -2(a^{(2)} - y) = -2 * \left(\begin{bmatrix} 0.1 \\ 0.66 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} -0.2 \\ 0.68 \end{bmatrix}$$

Vi vill ändra vikterna så att a_1^2 minskar lite och a_2^2 ökar mycket.



back propagation

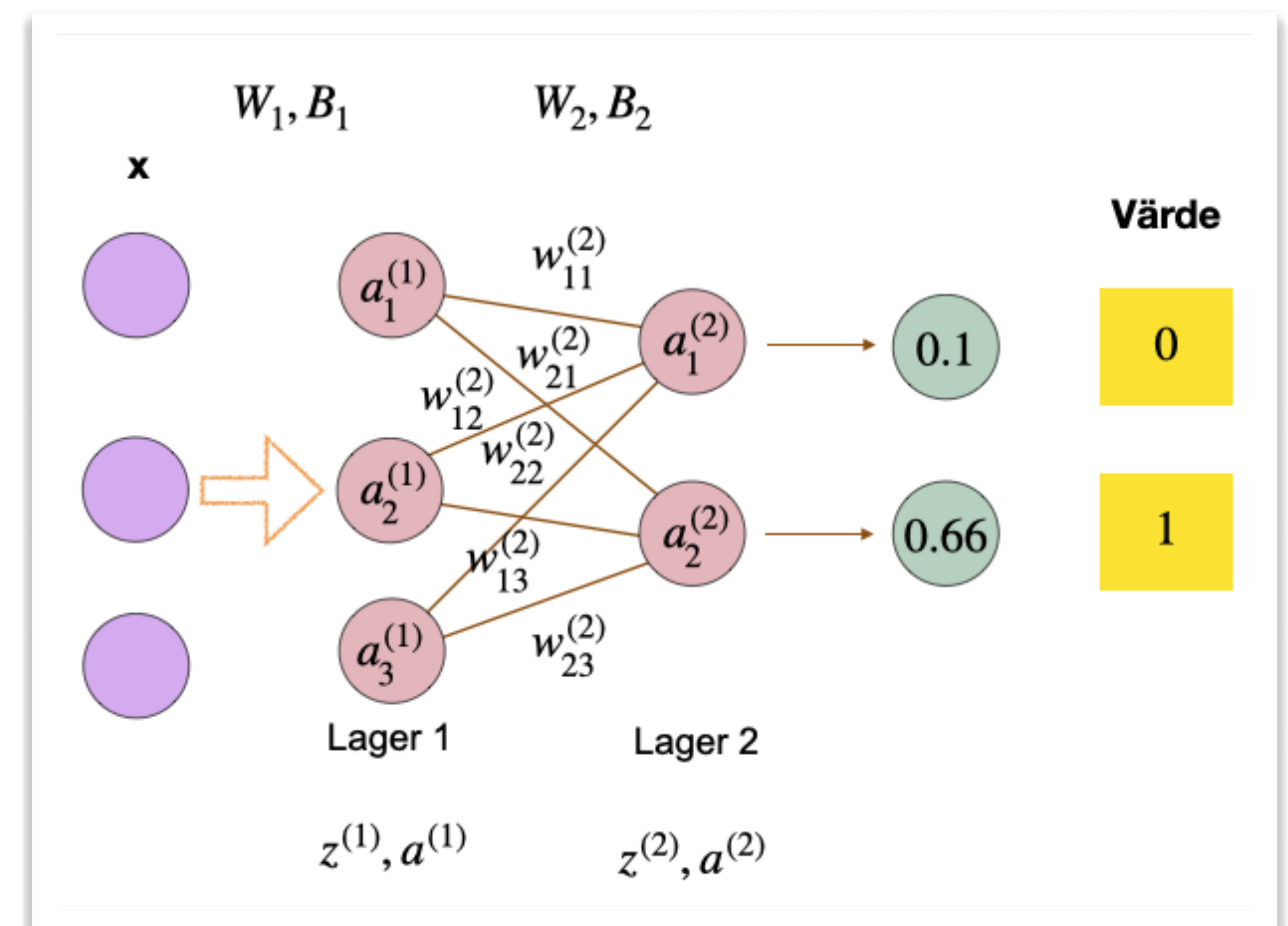
Vektorn med alla derivatorer beräknas genom att multiplicera felet elementvis med sigma-derivatan:

```
def backpropagation(y, z1, z2, x, a1, a2):
    deltas_z2 = - 2 *(a2 - y)*sigma_derivative(z2)
```

Räkna ut ΔB

Derivatorna härleds med kedjeregeln nedan:

$$\Delta B^{(l)} = \Delta z^{(l)}$$



$$z_j^{(2)} = \sum_i w_{ji} a_i^{(1)} + b_j^{(2)}$$

$$x \rightarrow z^{(1)} \rightarrow a^{(1)} \rightarrow z^{(2)} \rightarrow a^{(2)} \rightarrow C$$

$$\frac{dC}{db_j^{(l)}} = \frac{dC}{dz_j^{(l)}} \cdot \frac{dz_j^{(l)}}{db_j^{(l)}} = \Delta z_j^{(l)} \cdot 1$$

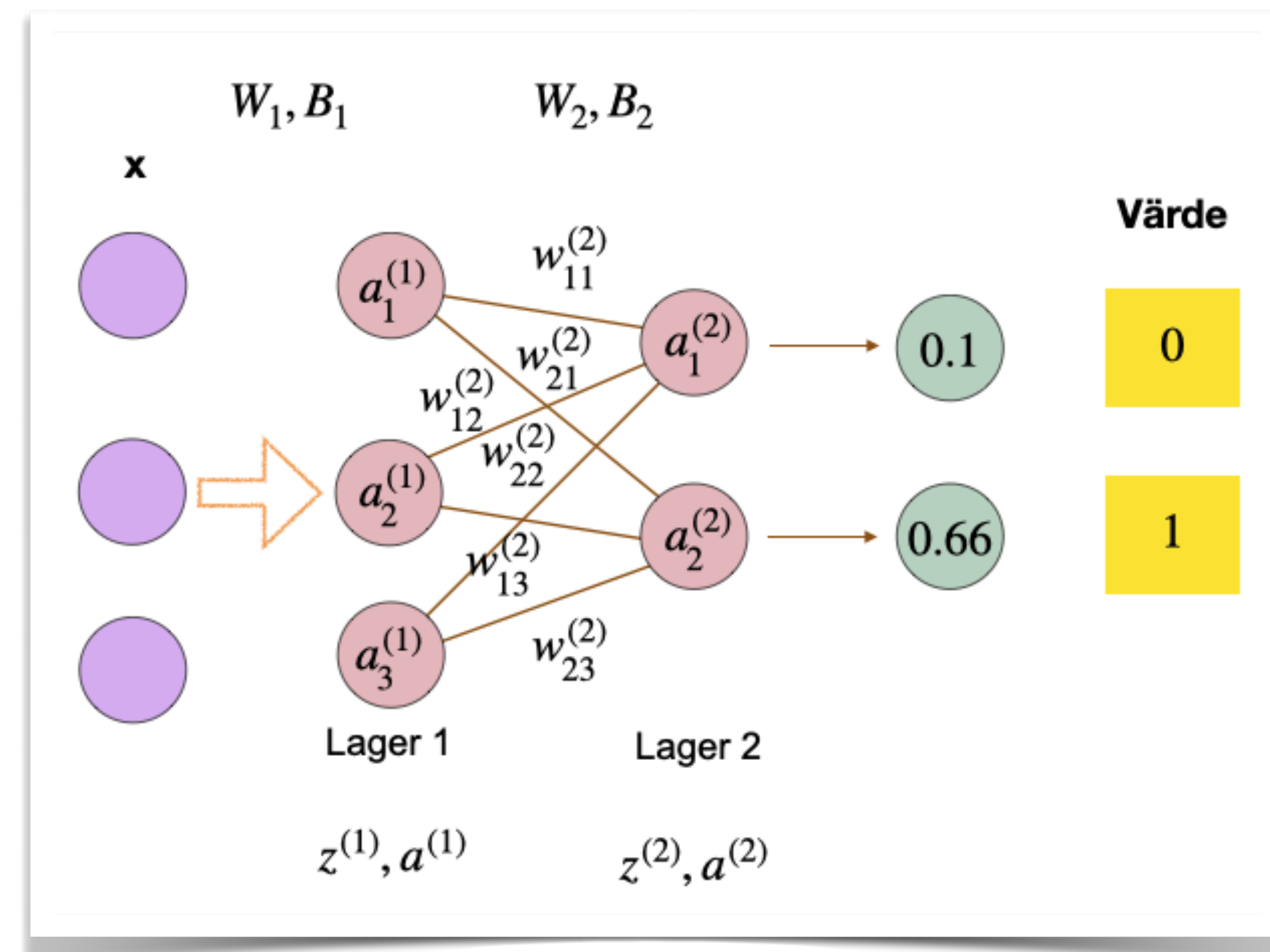
back propagation

```
def backpropagation(y, z1, z2, x, a1, a2):
    deltas_z2 = - 2 *(a2 - y)*sigma_derivative(z2)
    deltas_b2 = deltas_z2
```

Räkna ut ΔW

Kedjeregeln appliceras på varje vikt enligt nedan

$$\Delta W^{(l)} = \Delta z^{(l)} \cdot (a^{(l-1)})^T$$



$$z_j^{(2)} = \sum_i w_{ji} a_i^{(1)} + b_j^{(2)}$$

$$x \to z^{(1)} \to a^{(1)} \to z^{(2)} \to a^{(2)} \to C$$

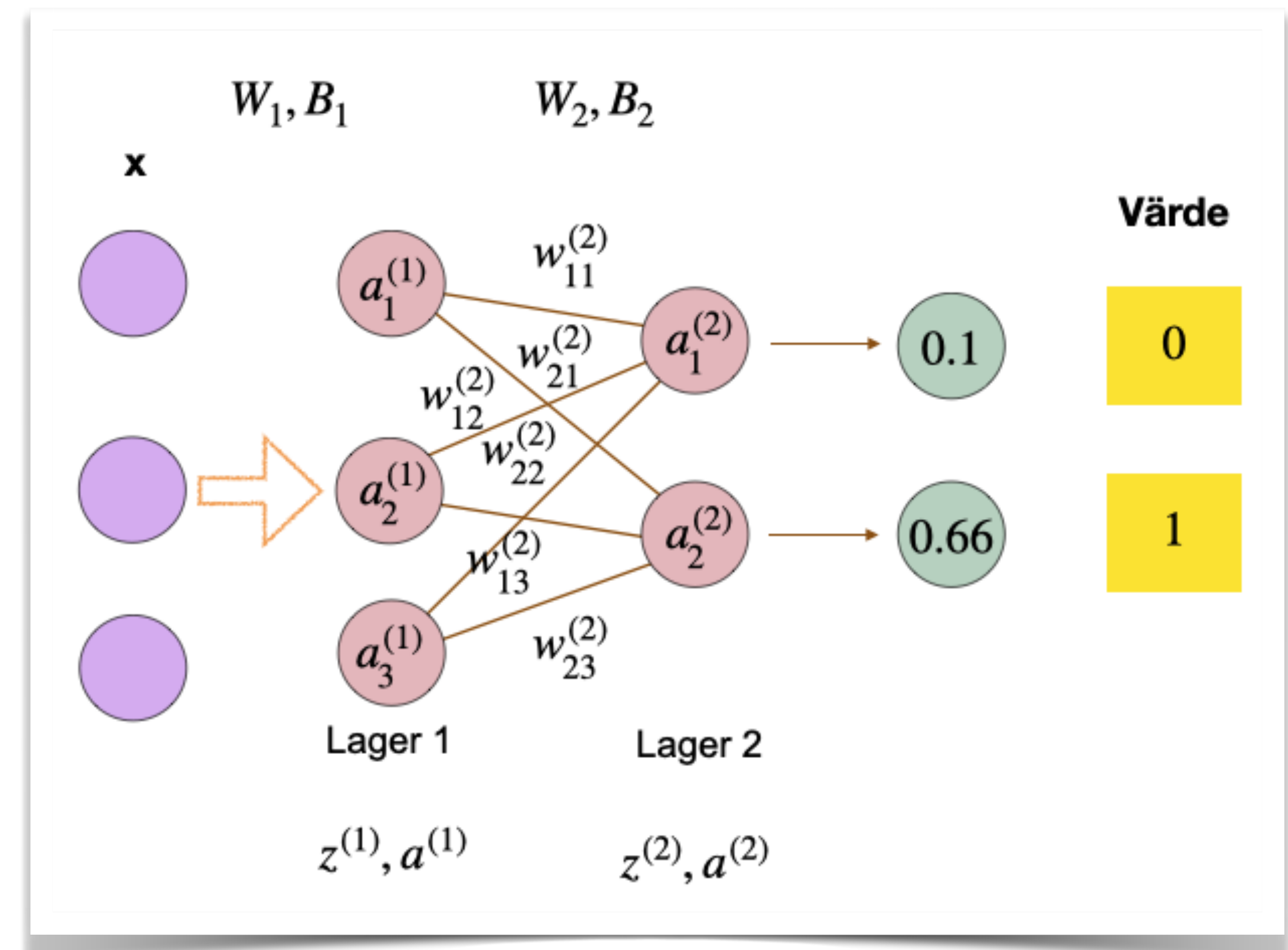
$$\frac{dC}{dw_{ji}^{(l)}} = \frac{dC}{dz_j^{(l)}} \cdot \frac{dz_j^{(l)}}{dw_{ji}^{(l)}} = \Delta z_j^{(l)} \cdot a_i^{(l-1)}$$

Räkna ut ΔW

Kedjeregeln appliceras på varje vikt enligt nedan

$$\Delta W^{(l)} = \Delta z^{(l)} \cdot (a^{(l-1)})^T$$

$$\begin{pmatrix} \Delta w_{11} & \Delta w_{12} & \Delta w_{13} \\ \Delta w_{21} & \Delta w_{22} & \Delta w_{23} \end{pmatrix} = \begin{pmatrix} \Delta z_1^{(2)} \\ \Delta z_2^{(2)} \end{pmatrix} \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}$$



$$z_j^{(2)} = \sum_i w_{ji} a_i^{(1)} + b_j^{(2)}$$

$$x \rightarrow z^{(1)} \rightarrow a^{(1)} \rightarrow z^{(2)} \rightarrow a^{(2)} \rightarrow C$$

$$\frac{dC}{dw_{ji}^{(l)}} = \frac{dC}{dz_j^{(l)}} \cdot \frac{dz_j^{(l)}}{dw_{ji}^{(l)}} = \Delta z_j^{(l)} \cdot a_i^{(l-1)}$$

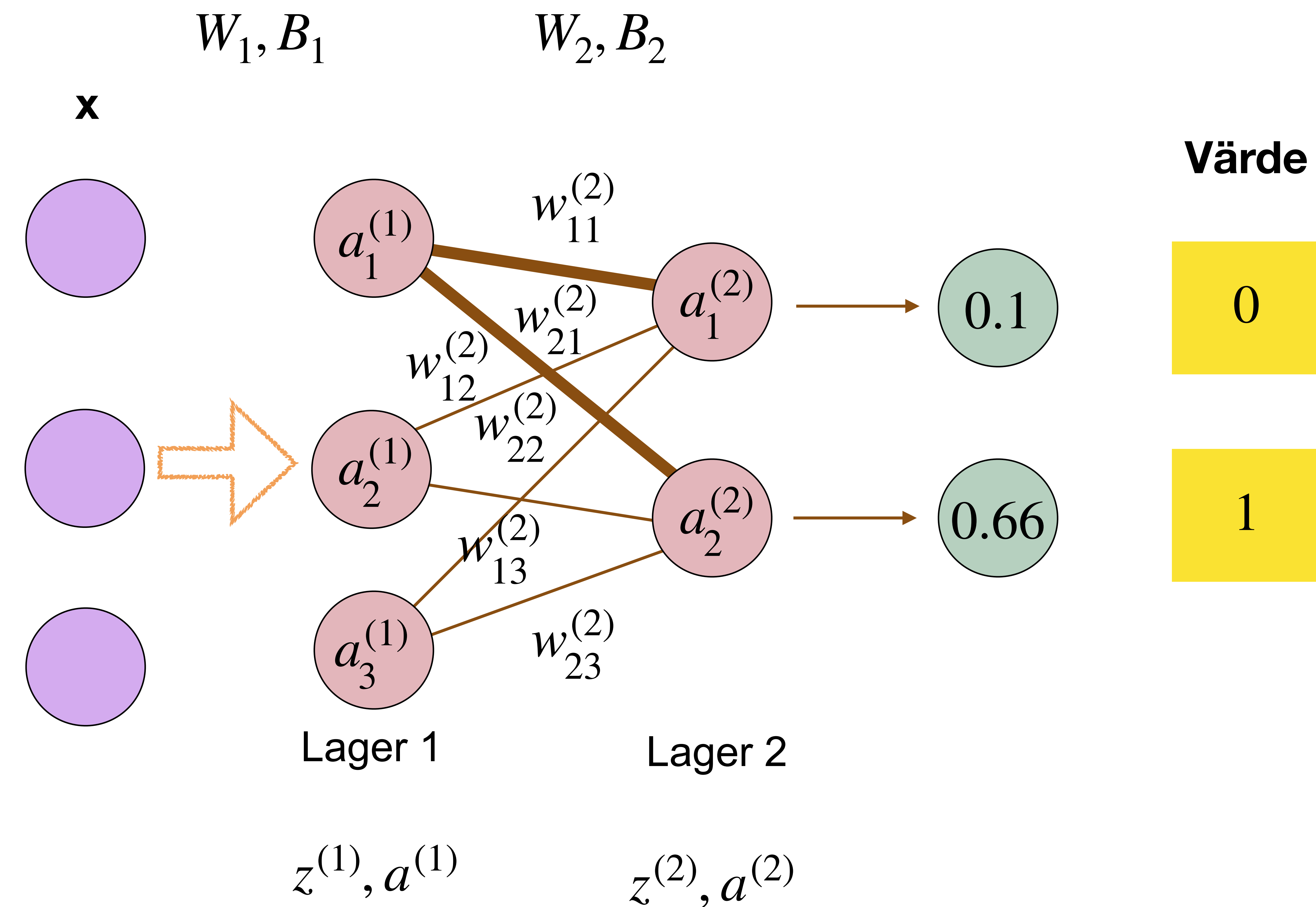
back propagation

```
def backpropagation(y, z1, z2, x, a1, a2):
    deltas_z2 = - 2 *(a2 - y)*sigma_derivative(z2)
    deltas_b2 = deltas_z2
    dw2 = np.matmul(deltas_z2, a1.T)
```

Räkna ut $\Delta z^{(l-1)}$

Varje neuron har flera utgående anslutningar:

$$\Delta z^{(l-1)} = \left((W^{(l)})^T \cdot \Delta z^{(l)} \right) * \sigma'(z^{(l-1)})$$

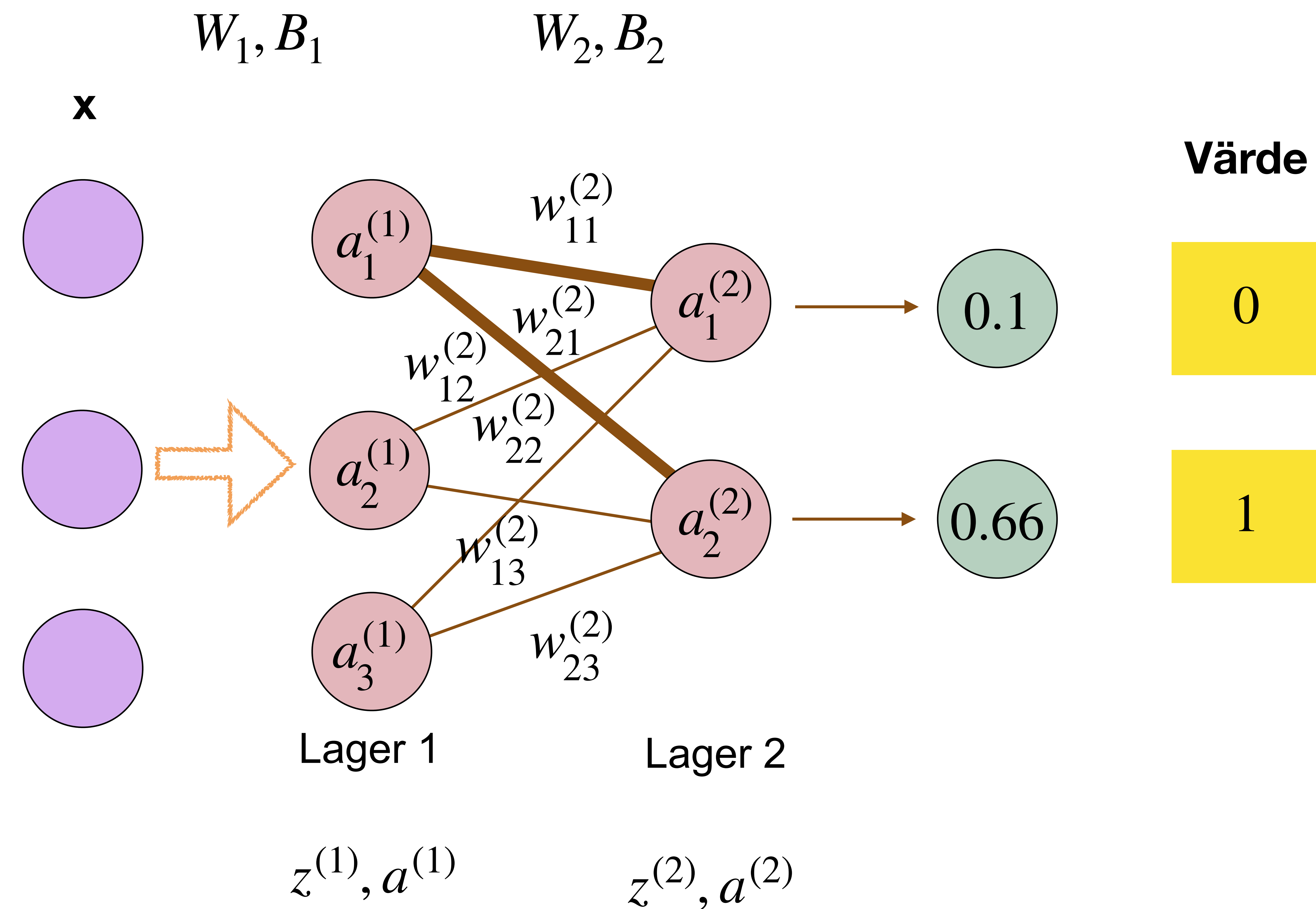


Räkna ut $\Delta z^{(l-1)}$

Varje neuron har flera utgående anslutningar:

$$\Delta z^{(l-1)} = \left((W^{(l)})^T \cdot \Delta z^{(l)} \right) * \sigma'(z^{(l-1)})$$

$$\frac{dC}{dz_1^{(1)}} = \frac{dC}{da_1^{(1)}} \cdot \sigma'(z_1^{(1)})$$



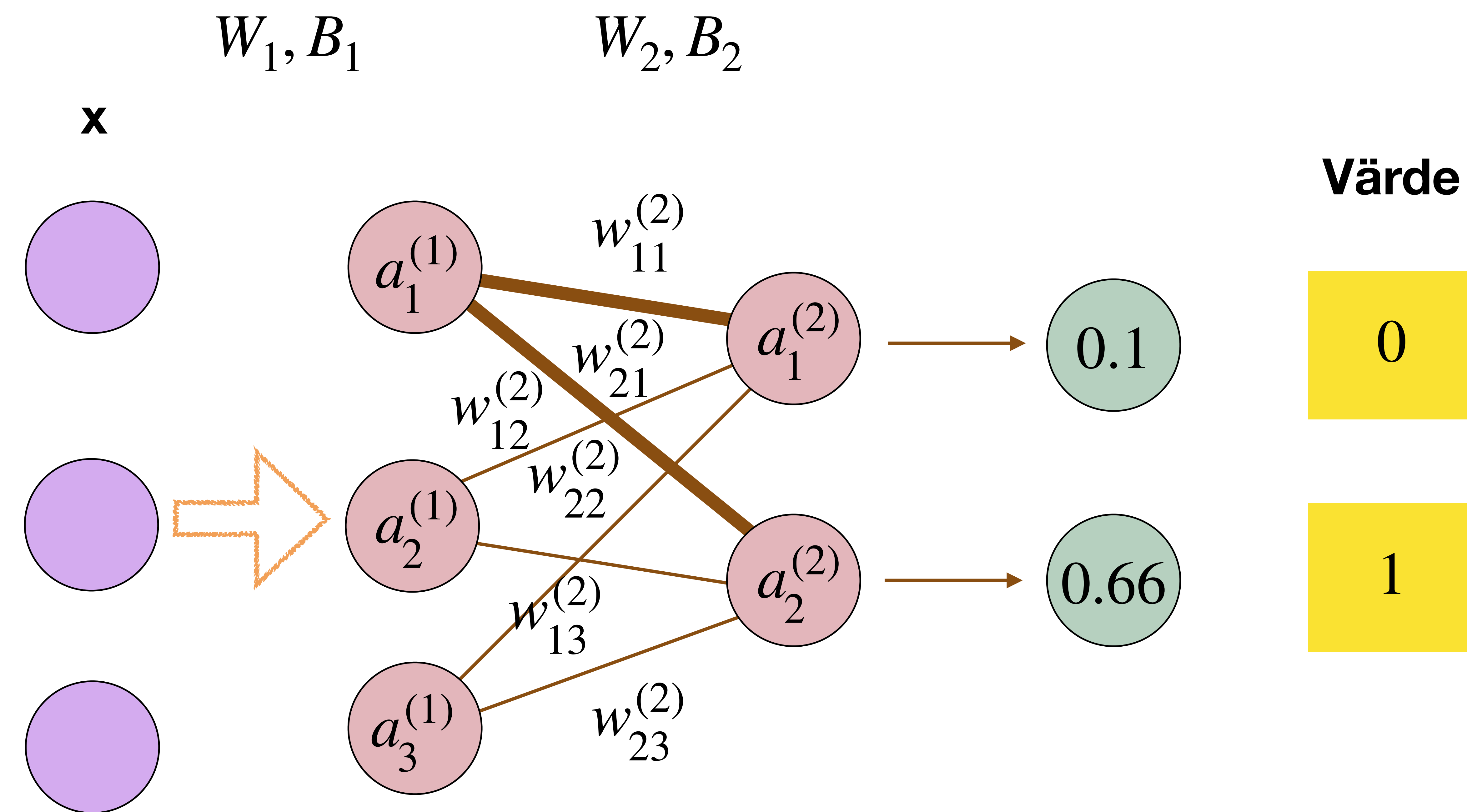
Räkna ut $\Delta z^{(l-1)}$

Varje neuron har flera utgående anslutningar:

$$\Delta z^{(l-1)} = \left((W^{(l)})^T \cdot \Delta z^{(l)}\right) * \sigma'(z^{(l-1)})$$

$$\frac{dC}{dz_1^{(1)}} = \frac{dC}{da_1^{(1)}} \cdot \sigma'(z_1^{(1)})$$

$$\frac{dC}{da_1^{(1)}} = w_{11}^{(2)} \cdot \Delta z_1^{(2)} + w_{21}^{(2)} \cdot \Delta z_2^{(2)}$$

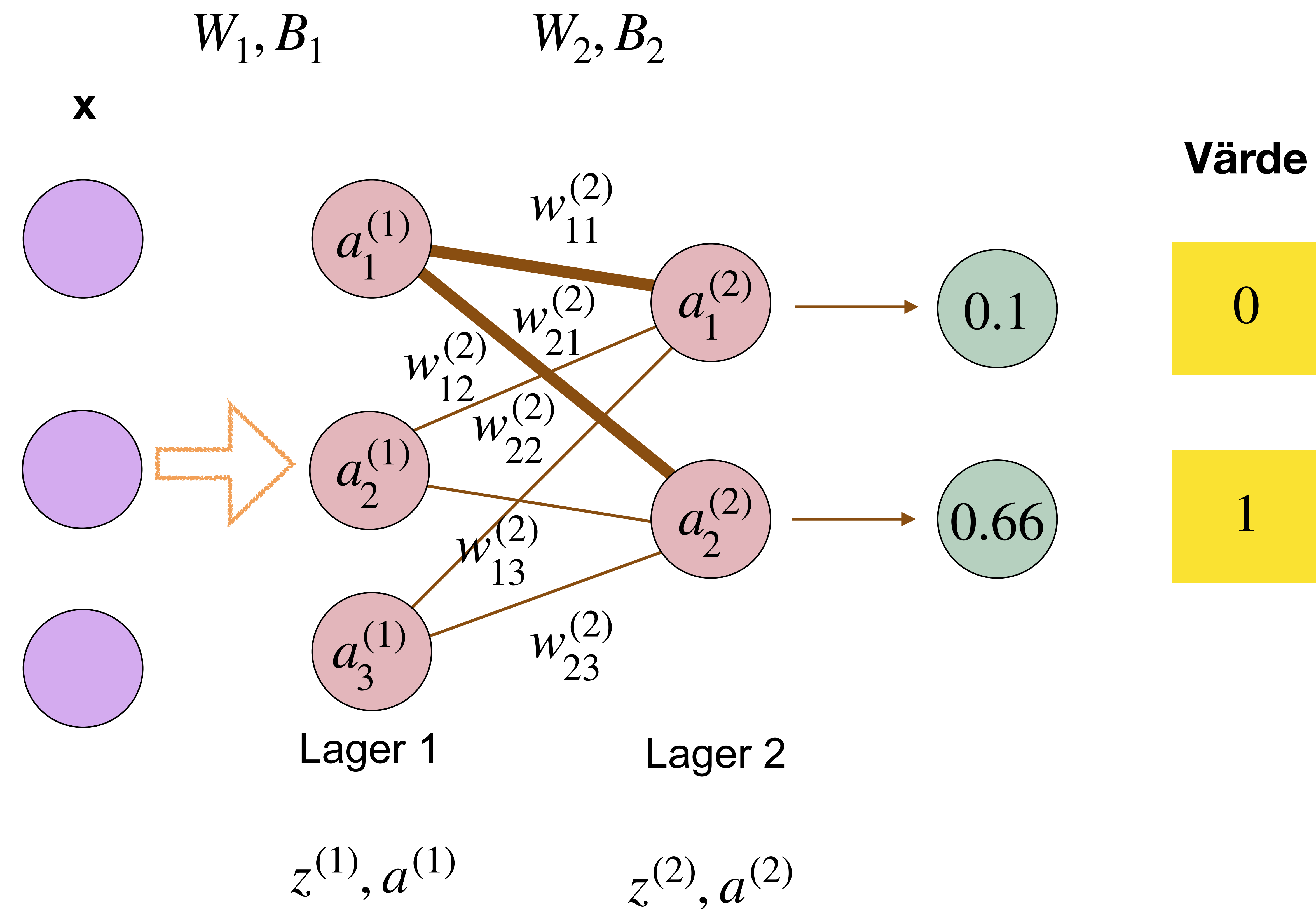


$$\frac{dC}{da_1^{(1)}} = \frac{dC}{dz_1^{(2)}} \cdot \frac{dz_1^{(2)}}{da_1^{(1)}} + \frac{dC}{dz_2^{(2)}} \cdot \frac{dz_2^{(2)}}{da_1^{(1)}}$$

Räkna ut dC/da_1

$$\frac{dC}{da_1^{(1)}} = w_{11}^{(2)} \cdot \Delta z_1^{(2)} + w_{21}^{(2)} \cdot \Delta z_2^{(2)}$$

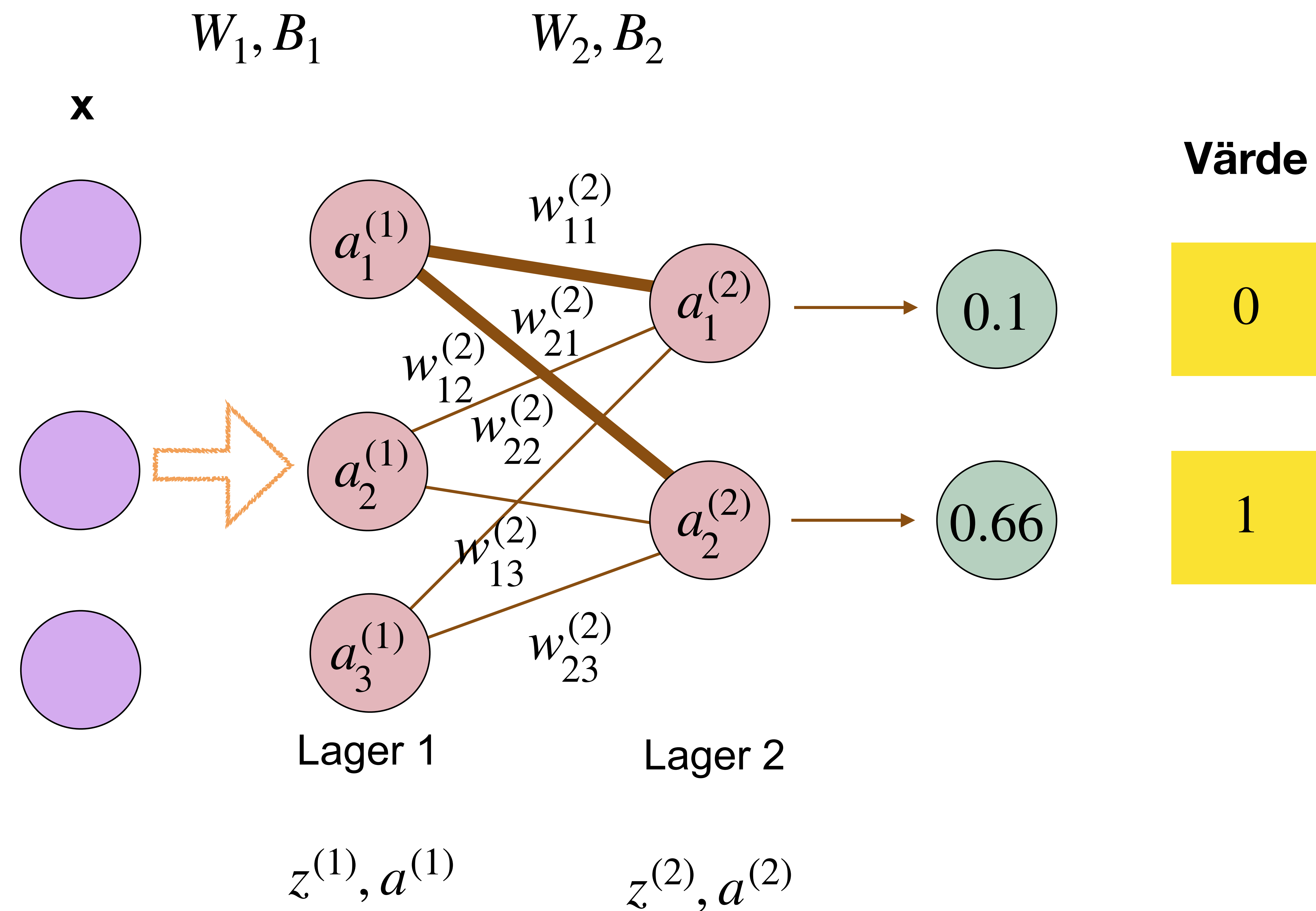
$$\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$



Räkna ut dC/da_1

$$\frac{dC}{da_1^{(1)}} = w_{11}^{(2)} \cdot \Delta z_1^{(2)} + w_{21}^{(2)} \cdot \Delta z_2^{(2)}$$

$$\begin{pmatrix} \Delta a_1^{(1)} \\ \Delta a_2^{(1)} \\ \Delta a_2^{(1)} \end{pmatrix} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix} \begin{pmatrix} \Delta z_1^{(2)} \\ \Delta z_2^{(2)} \end{pmatrix}$$

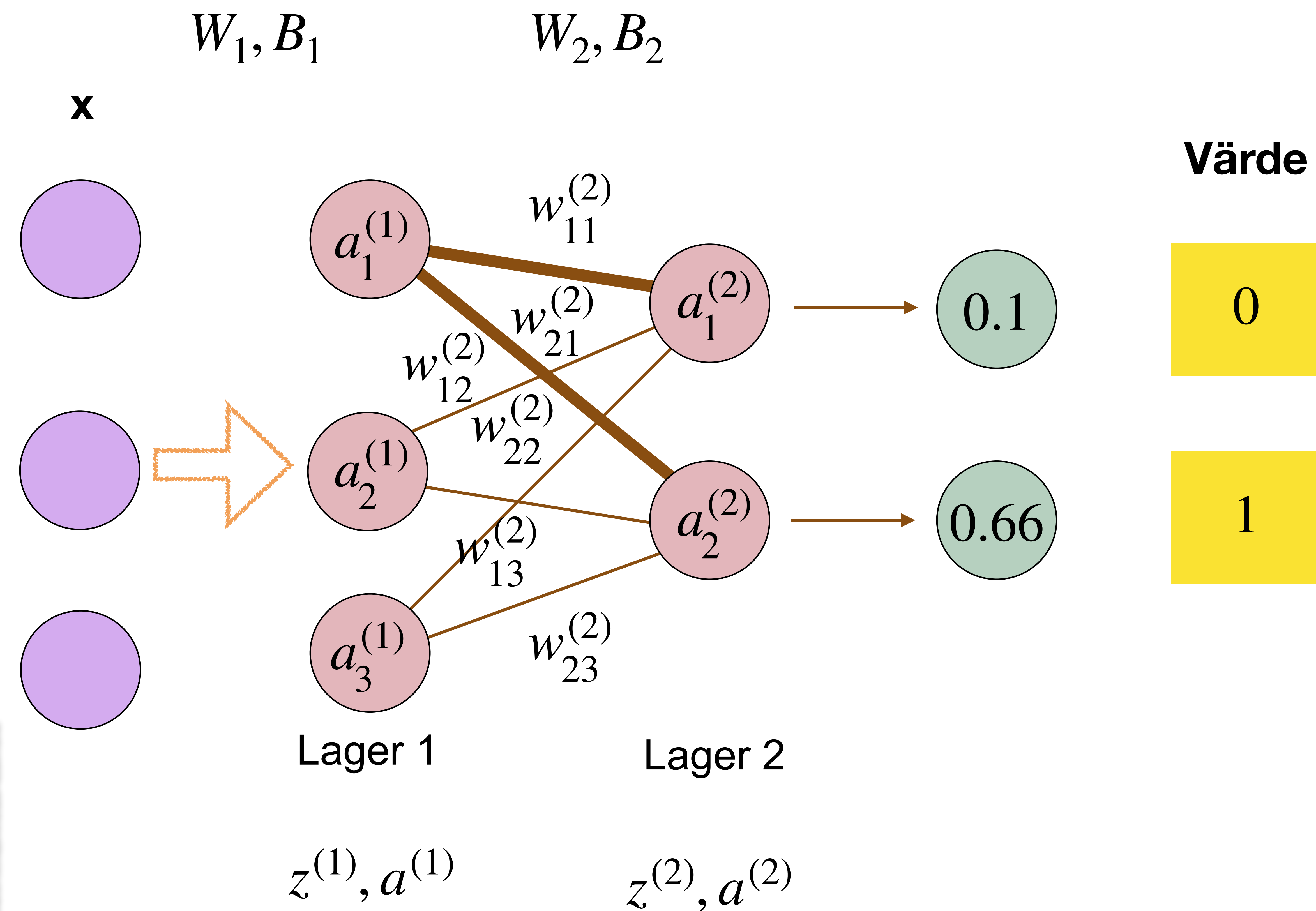


Räkna ut dC/da_1

$$\frac{dC}{da_1^{(1)}} = w_{11}^{(2)} \cdot \Delta z_1^{(2)} + w_{21}^{(2)} \cdot \Delta z_2^{(2)}$$

$$\begin{pmatrix} \Delta a_1^{(1)} \\ \Delta a_2^{(1)} \\ \Delta a_2^{(1)} \end{pmatrix} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix} \left(\begin{pmatrix} \Delta z_1^{(2)} \\ \Delta z_2^{(2)} \end{pmatrix} \right)$$

$$\Delta z^{(l-1)} = \left((W^{(l)})^T \cdot \Delta z^{(l)} \right) * \sigma'(z^{(l-1)})$$



back propagation

```
def backpropagation(y, z1, z2, x, a1, a2):
    deltas_z2 = - 2 *(a2 - y)*sigma_derivative(z2)
    deltas_b2 = deltas_z2
    dw2 = np.matmul(deltas_z2, a1.T) # 2 x 1 mul 1 x 3
    deltas_z1 = np.matmul(W2.T,deltas_z2)*sigma_derivative(z1)
    db1 = deltas_z1
    dw1 = np.matmul(deltas_z1, x.T)
    return dw1, dw2, db1, db2
```

Uppdatera vikterna

Vi uppdaterar vikterna iterativt genom att:

1. Räkna ut felet för **ett exempel** i taget
   i sista lagret med minsta kvadratmetoden.
2. Räkna ut derivatan för varje vikt och b-värde genom att gå bakåt genom nätverket med hjälp av **kedjeregeln**. Detta kallas *backpropagation*.
3. Addera förändringarna från några exempel (en **batch**) och ta ett litet steg i **motsatt** riktning av derivatan. Detta kallas *gradient descent.*
4. Gå flera gånger genom alla träningsexempel.

Träningsloopen

Skapa variabler:

```
#steget vi går i derivatans riktning
step_size = 0.001
#antalet exempel i varje batch
batch_size = 100
#förändringarna i våra variabler
dW1 = np.zeros((N1, INPUT))
dW2 = np.zeros((N2, N1))
dB1  = np.zeros((N1, 1))
dB2 = np.zeros((N2, 1))
```

Träningsloopen forst.

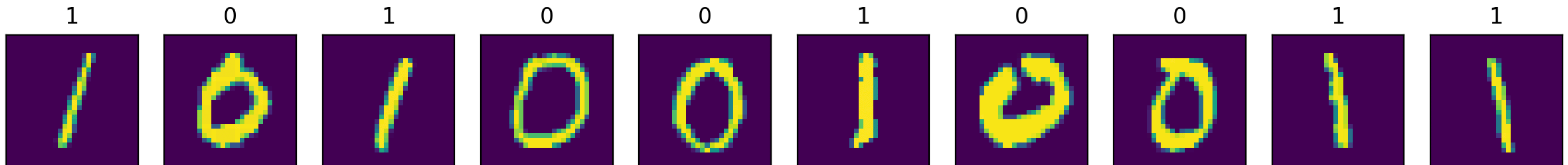
```
for e in range(5):
    print('epoch', e)

    for i in range(len(train_X)):
        x = train_X[i]
        y = train_y[i]

        z1, z2, x, a1, a2 = forward(x)
        dw1, dw2, db1, db2 = backpropagation(y, z1, z2, x, a1, a2)
        dW1 += dw1
        dW2 += dw2
        dB1 += db1
        dB2 += db2
```

Träningsloopen forst.

```
#Var 100:e gång uppdaterar vi vikterna och
#nollställer variablerna
     if i % batch_size == 0:

           W1 += step_size*dW1
           W2 += step_size * dW2
           B1 += step_size * dB1
           B2 += step_size * dB2
           dW1[:,:] = 0
           dW2[:,:] = 0
           dB1[:,:] = 0
           dB2[:,:] = 0
```

Testa några värden!

```
outputs = [predict(x) for x in test_X]
predictions = [np.argmax(result) for result in outputs]

for i in range(10):

    plt.imshow(images[i])
    plt.title('Prediction ' + str(predictions[i]))
    plt.show()
```



Laboration 3

* Ni ska hämta mätdata från nätet. Filtrera bort spikar.
* Beräkna diffekvationer.
* Anpassa kurvor

Övningsuppgift

Gör om nätverket så att det kan känna igen **alla** siffror 0-9.

Tips:

- *Ta bort filtreringen av datan.*
- *Ändra N2, antalet noder i lager 2.*
- *Ändra koden för onehot-vektorn*
- *Öka N1, antalet noder i lager 1, prova N1 = 15.*
- *Öka antalet epoker till 30 (det tar tid att köra!).*
- *Rita upp de 20 första siffrorna och se hur bra det är.*

Nätverket kommer göra många fel. Kan du identifiera alla siffror?
(Med ovan parametrar får jag ca 75% rätt, ni får gärna testa olika parametrar för att hitta något bra).

Exempel: Advent of Code 1 dec 2021

* Läs in mätdata som heltal från en fil (*test.txt*), ett tal per rad.
* Skriv ut antal gånger som värdet ökar från en mätning till en annan.

```
199 (N/A - no previous measurement)
200 (increased)
208 (increased)
210 (increased)
200 (decreased)
207 (increased)
240 (increased)
269 (increased)
260 (decreased)
263 (increased)
```

Antal gånger värdet ökar: 7

Ren python: Advent of Code 1 dec 2021

```
cnt = 0
prev = 10 **30
with open('test.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        value = int(line)
        if value > prev:
            cnt += 1
        prev = value
print(cnt)
```

Numpy: Advent of Code 1 dec 2021

```
import numpy as np

data = np.loadtxt('test.txt')
prev = np.roll(data, 1) #roterar vektorn 1
steg bakåt.
inc = data > prev
cnt = sum(inc[1:]) #det första värdet räknas
inte eftersom det jämförs med det sista.
print(cnt)
```

Tack för idag!