Maskininlärning med Python

Maskininlärning (ML)

- **Vad är maskininlärning?** Låter datorer lära sig mönster från data utan att vara explicit programmerade.
- **Python & ML:** Finns kraftfulla bibliotek som (pandas, pytorch, tensorflow).
- **Baksidan:** Maskininlärning kräver i praktiken stora mängder data för att hitta mönster.

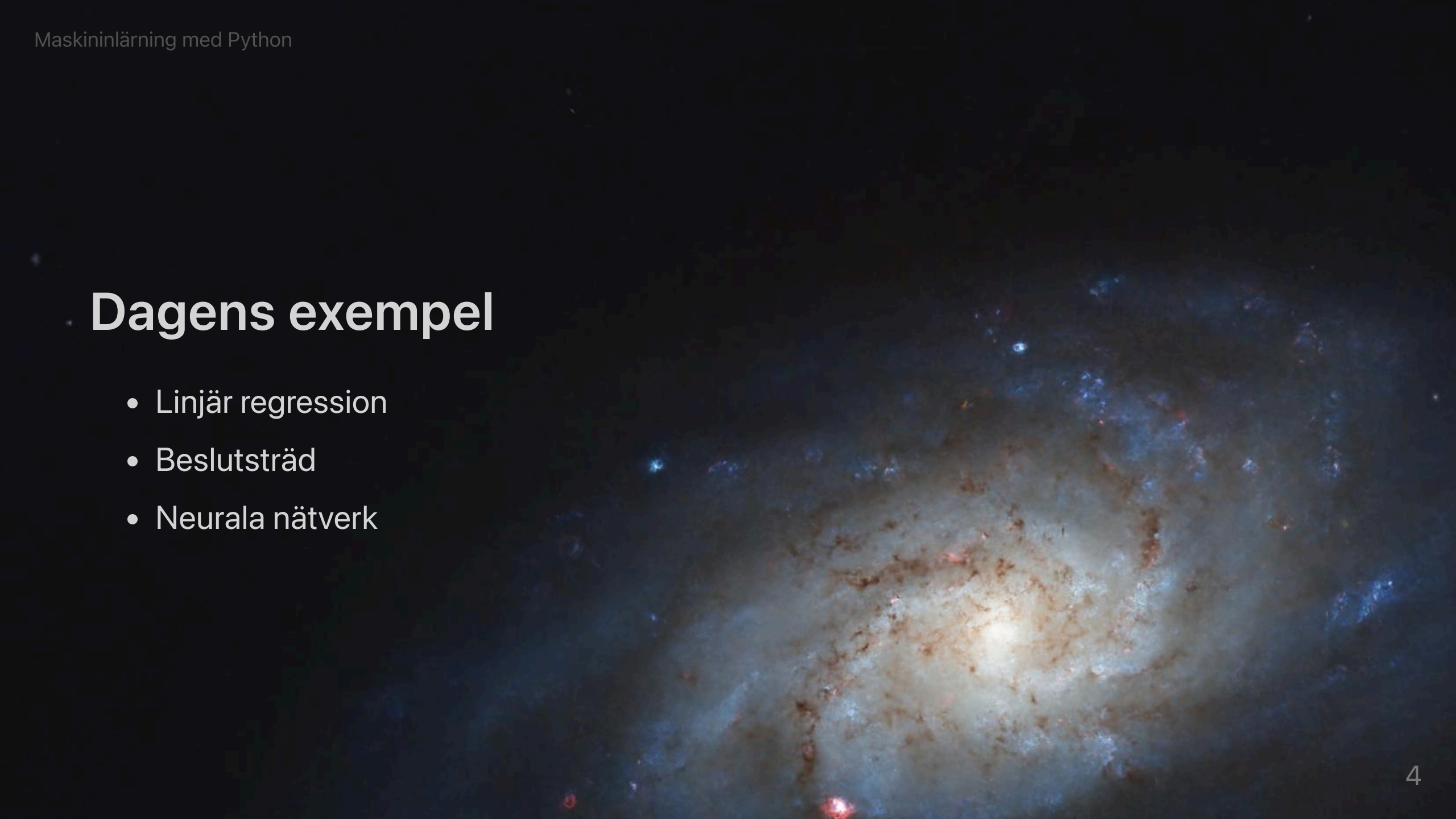
ML (kort kring typer)

- **Övervakad ML**
  - Modellen tränas på ett dataset där både input, X, och korrekt output, y, är kända.
  - Uppmärkt data
- **Oövervakad ML**
  - Ouppmärkt data; inget y-värde finns
- **Förstärkningsinlärning (RL)**
  - "Agent" lär sig genom att prova och få belöningar

Vi kommer skrapa lite på ytan inom *övervakad* ML idag.

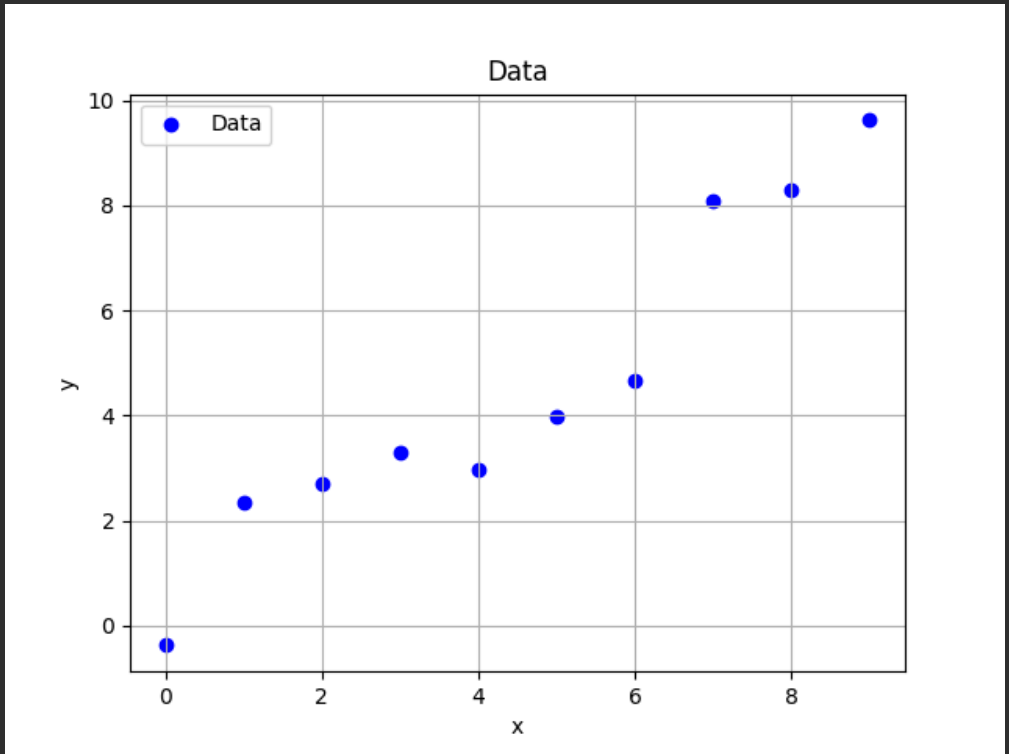
Dagens exempel

- Linjär regression
- Beslutsträd
- Neurala nätverk

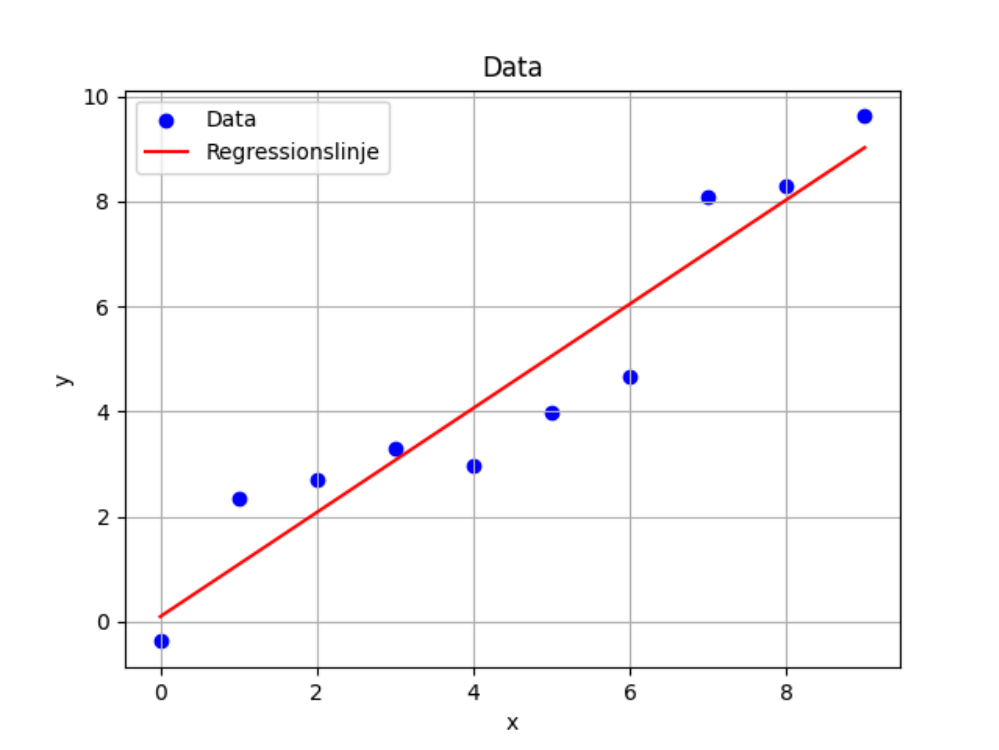


Exempel för introduktion av koncept

- **Linjär regression**: Hitta samband mellan datapunkter



- **Linjär regression**: Hitta samband bland datapunkterna.
- Vilket samband?
  - Linje? Annan kurva?



Datan ovan genererades med följande uttryck:

```
x = np.linspace(0, 10, NN, endpoint=True)
y = np.array([x[ii] + 3 * (-.5 + np.random.rand()) for ii in range(NN)])
```

Därför rimligt att anta ett linjärt förhållande, eller hur?

- Hur hittar vi linjen?
  - Räta linjens ekvation: $y = kx + m$, där k är linjens lutning och m är skärningspunkten med y-axeln

Kör

```
k, m = np.polyfit(x, y, 1)
print(f'k = {round(k,3)}, m = {round(m,3)}')
```

vilket ger `k = 0.993, m = 0.093`

- Nu har vi data för `x = 0, 1, 2, 3, ...`
- Vi vill nu veta `y` för `x=3.5`, dvs "osedd data"
- Använd vår enkla ML-modell och stoppa in x-värdet:

```
prediction = k * x + m
print(f"Prediktion för x={3.5}: {prediction}")
```

vilket ger `Prediktion för x=3.5: 3.568`

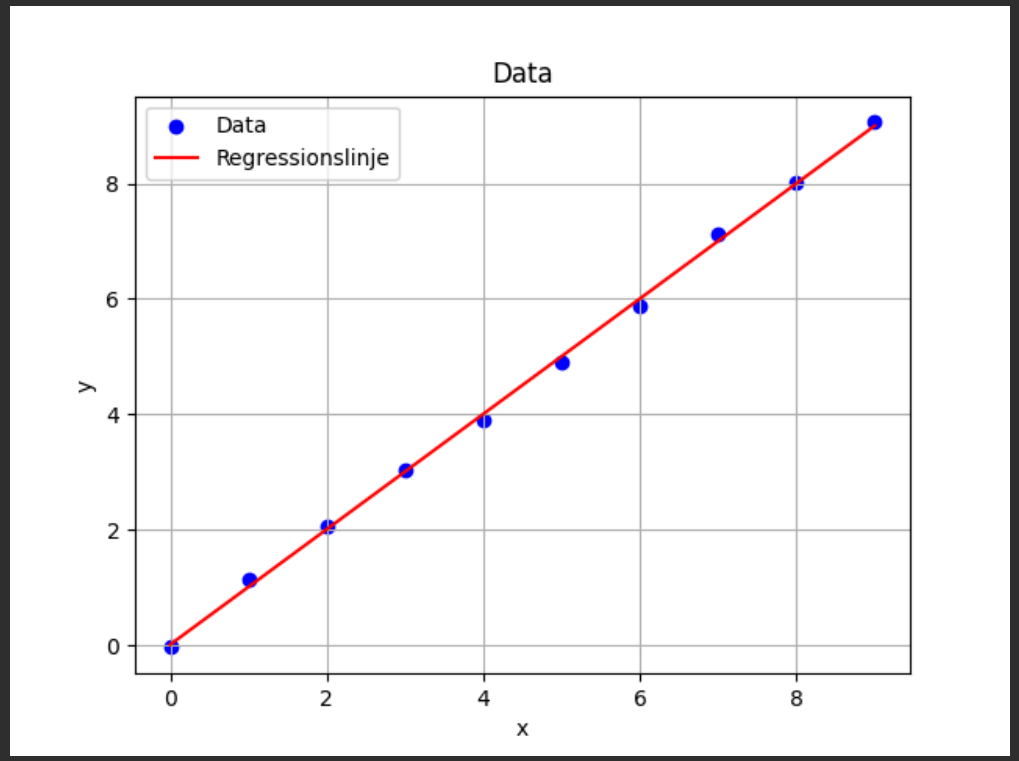
Hur blir det med "bättre" data (mindre slumpmässighet)?

```
y = np.array([x[ii] + .3 * (-.5 + np.random.rand()) for ii in range(NN)])
```

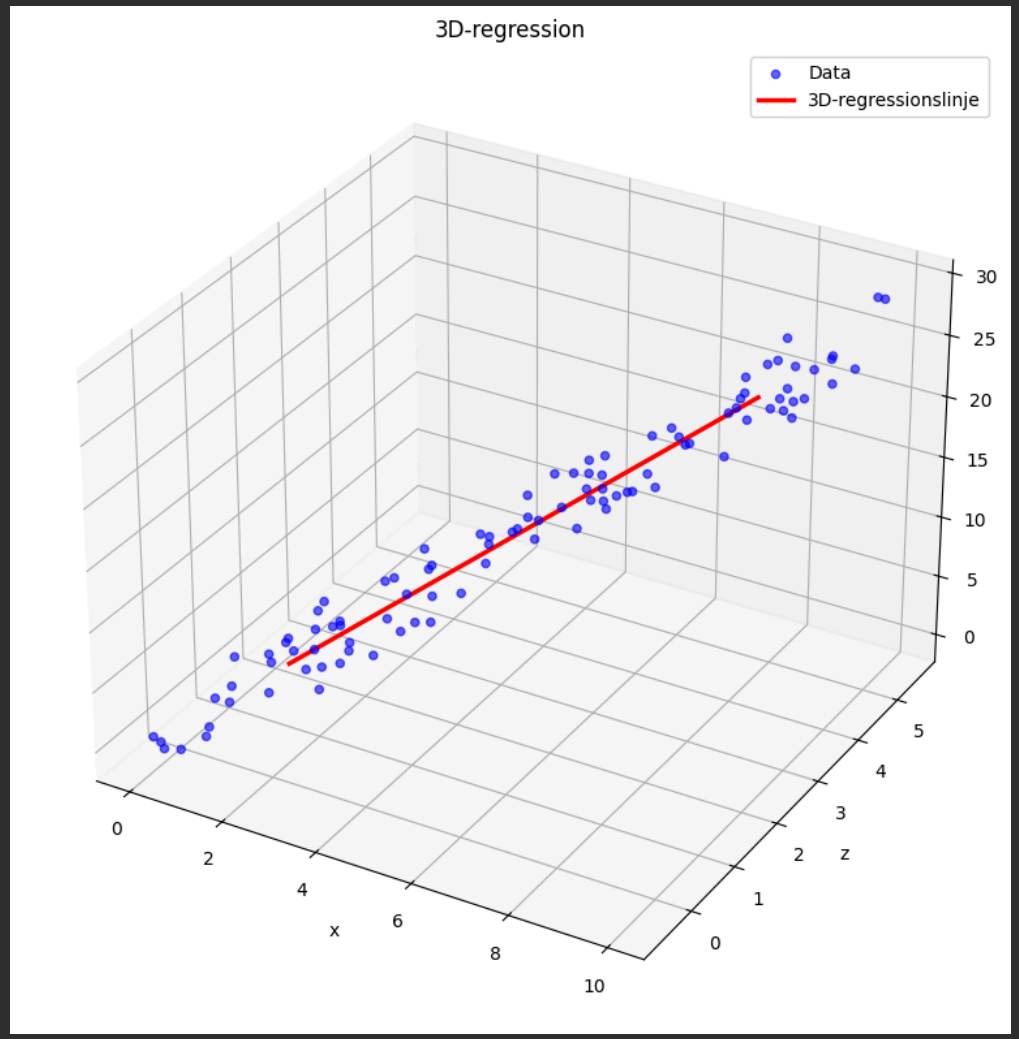
Detta ger linjärregressionen `k = 0.999, m = 0.009`

alltså sambandet $y = 0.999x + 0.009$, vilket ju ligger mycket nära $y = x$.

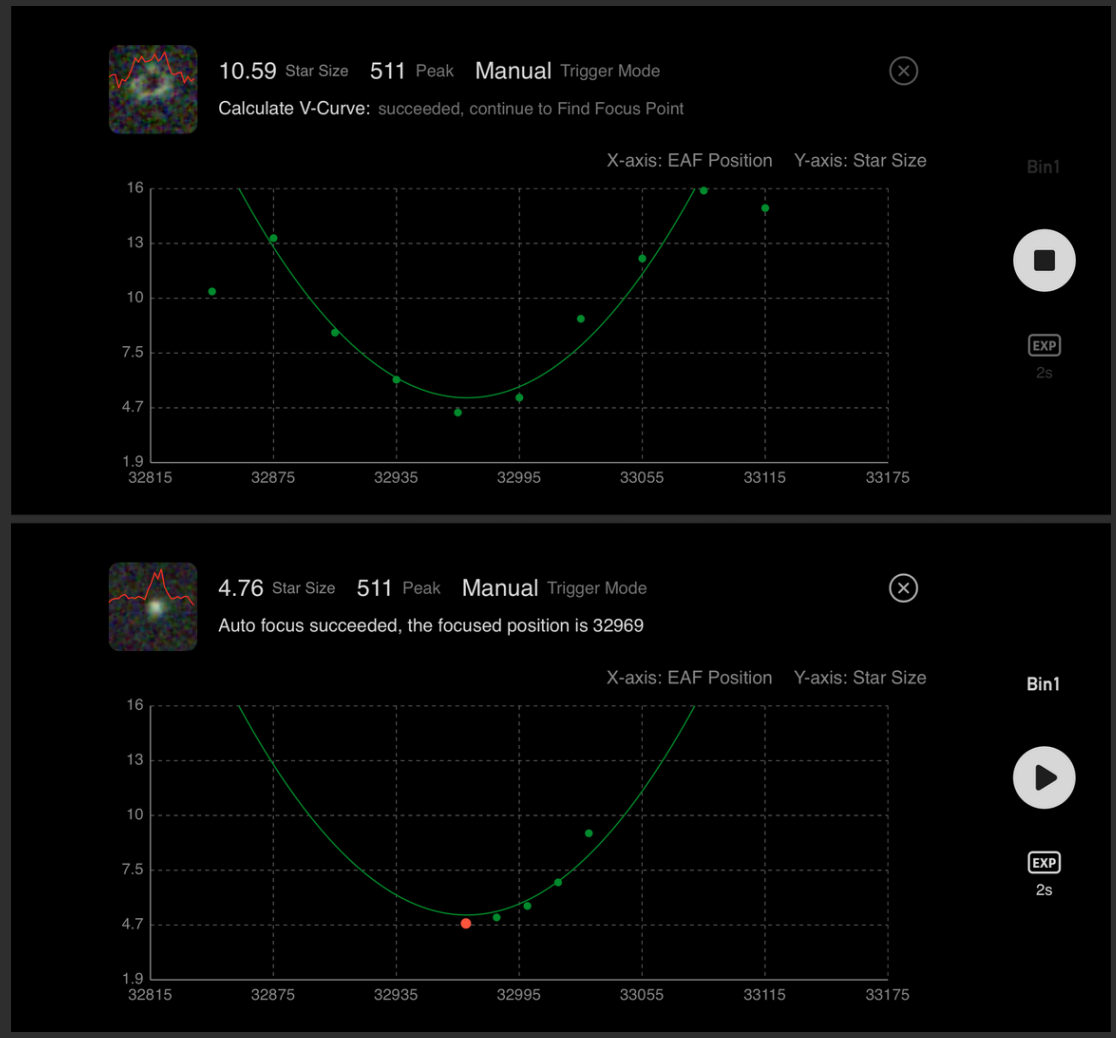
Bra data = bättre maskininlärningsmodell! (Eller, tvärtom, "skit in - skit ut")



Funkar på samma sätt för flerdimensionell data



Verkligt exempel - Autofokus inom astrofoto



Praktiska exempel

- Fastighetsvärdering (boyta, antal rum, område, byggår)
- Efterfrågeprognoser (reklamkampanjer, årstid, veckodag, väder)
- Hälsa, samband mellan livsstil & blodtryck (ålder, BMI, rökning, fysisk aktivitet)
- Ekonomi, löneanalys (utbildning, erfarenhet, kön, ålder)

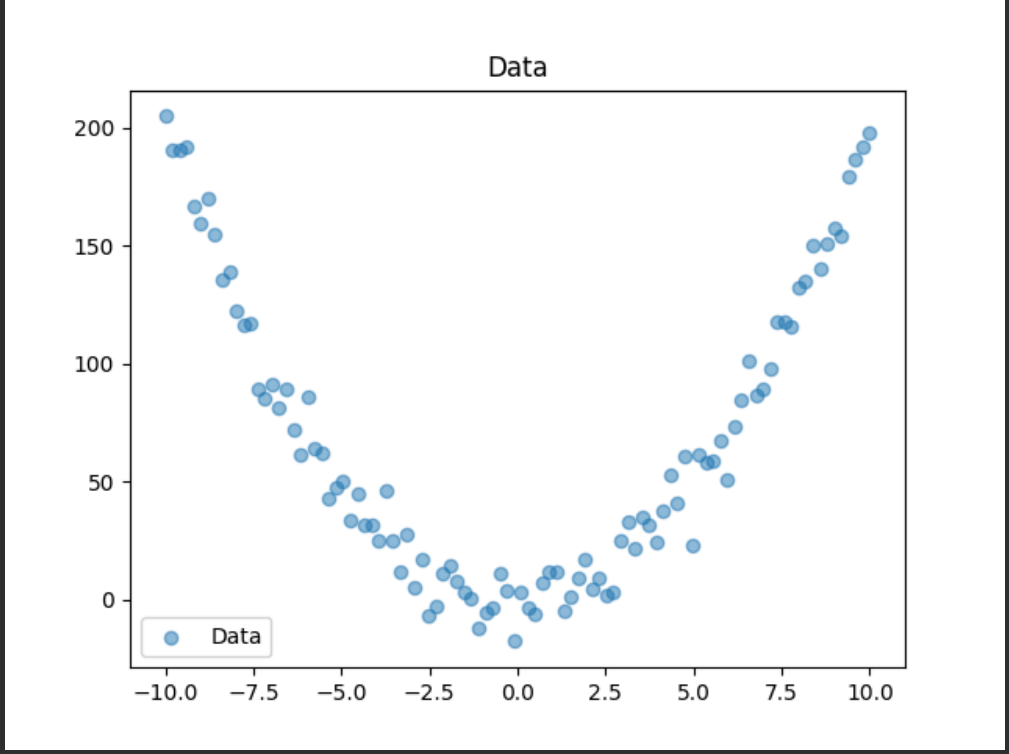
Beslutsträd: En avancerad 20 frågor!

Beslutsträd

Ett beslutsträd består av noder som representerar beslut baserade på funktioner i datan, och grenar som leder till nya beslut eller slutliga resultat (blad).

- **Beslutsregler**: Vid varje nod ställs en fråga, t.ex. "Är ålder < 30?". Datan delas upp baserat på svaret.
- **Träning**: Trädet byggs upp *rekursivt* genom att välja den bästa splitten enligt ett mått som minimerar fel (t.ex. Gini impurity, information gain eller MSE).
- **Användning**: Vanligt inom både klassificering (t.ex. "godkänd/underkänd") och regression (t.ex. "förutsäg pris").
- **Tolkbarhet**: En stor fördel är att besluten är lätta att följa och förstå.
- **Risk för överanpassning**: Träd som tillåts växa fritt kan bli väldigt komplexa och överanpassa till träningens data.

Vi ska försöka oss på att träna ett liknande dataset baserat på ett sådant beslutsträd



Skapa en klass för beslutsträdet

- `depth` = Trädets djup. Högt värde ⇒ större noggrannhet men ev. överanpassning (sämre generalisering, jfr överdrivet fokus på detaljer -> missar helheten).
- `split_value` = Värdet där vi delar datan.
- `left / right` = Pekare till vänster och höger subträd (gren).
- `prediction` = Om vi når ett löv, används detta värde som förutsägelse.

```
class DecisionTree:
    def __init__(self, depth=3):
        self.depth = depth           # Maxdjup för trädet
        self.split_value = None      # Värdet där vi delar datan
        self.left = None             # Vänster gren
        self.right = None            # Höger gren
        self.prediction = None       # Förutsägelse om vi är i ett löv
```

- Om vi har nått maxdjupet eller för få datapunkter: beräkna medelvärdet som förutsägelse.

```
def fit(self, X, y, depth=0):
    if depth == self.depth or len(X) < 2:
        self.prediction = np.mean(y)    # Medelvärdet används i löven
        return
```

- Vi delar in datan i två grupper baserat på medianen.

```
    self.split_value = np.median(X)     # Dela datan vid medianen
    left_mask = X < self.split_value
    right_mask = X >= self.split_value
```

- (forts. `fit`) Vi skapar två nya "child nodes" och tränar dem rekursivt.

```
    self.left = DecisionTree(self.depth)
    self.right = DecisionTree(self.depth)

    self.left.fit(X[left_mask], y[left_mask], depth + 1)
    self.right.fit(X[right_mask], y[right_mask], depth + 1)
```

Gör förutsägningar

- Om vi är i ett löv → returnera det förutsagda medelvärdet.

```
def predict(self, X):
    if self.prediction is not None:
        return np.full_like(X, self.prediction)
```

- Vi delar upp X i två delar och förutsäger med hjälp av barnnoderna.

```
    mask = X < self.split_value
    y_pred = np.zeros_like(X)
    y_pred[mask] = self.left.predict(X[mask])
    y_pred[~mask] = self.right.predict(X[~mask])
    return y_pred
```

Skapa och träna trädet

- Skapar syntetisk data där y är en kvadratisk funktion plus slumpmässigt brus.

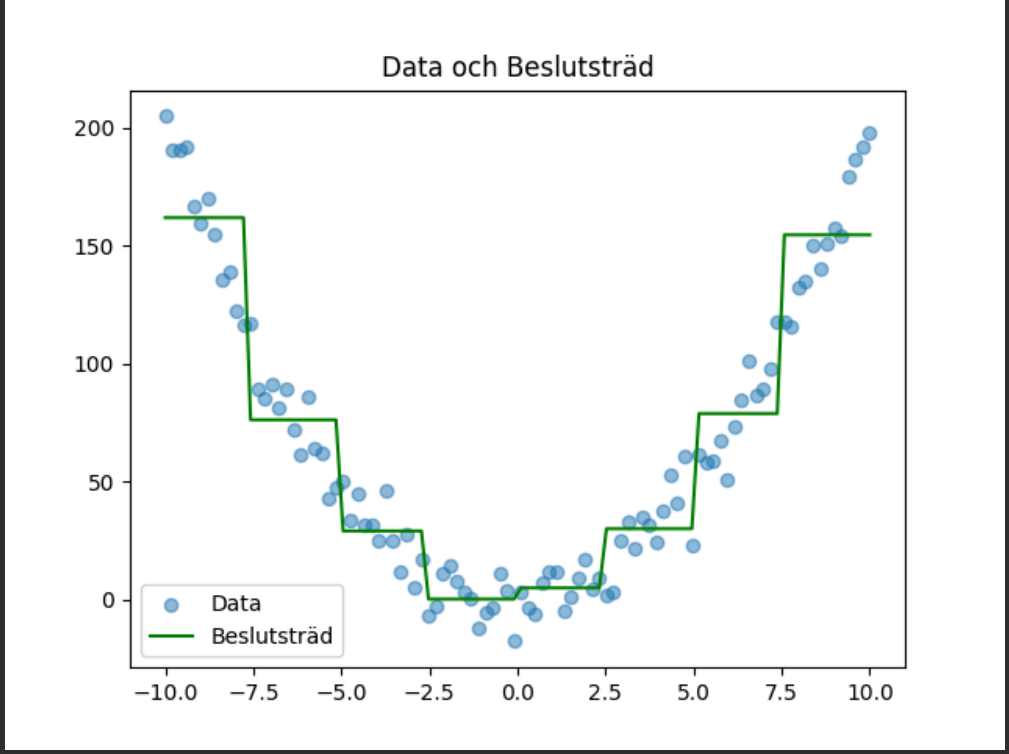
```
np.random.seed(42)
X = np.linspace(-10, 10, 100)
y = 2 * X**2 + np.random.randn(100) * 10
```

- Träna beslutsträd med maxdjup 3

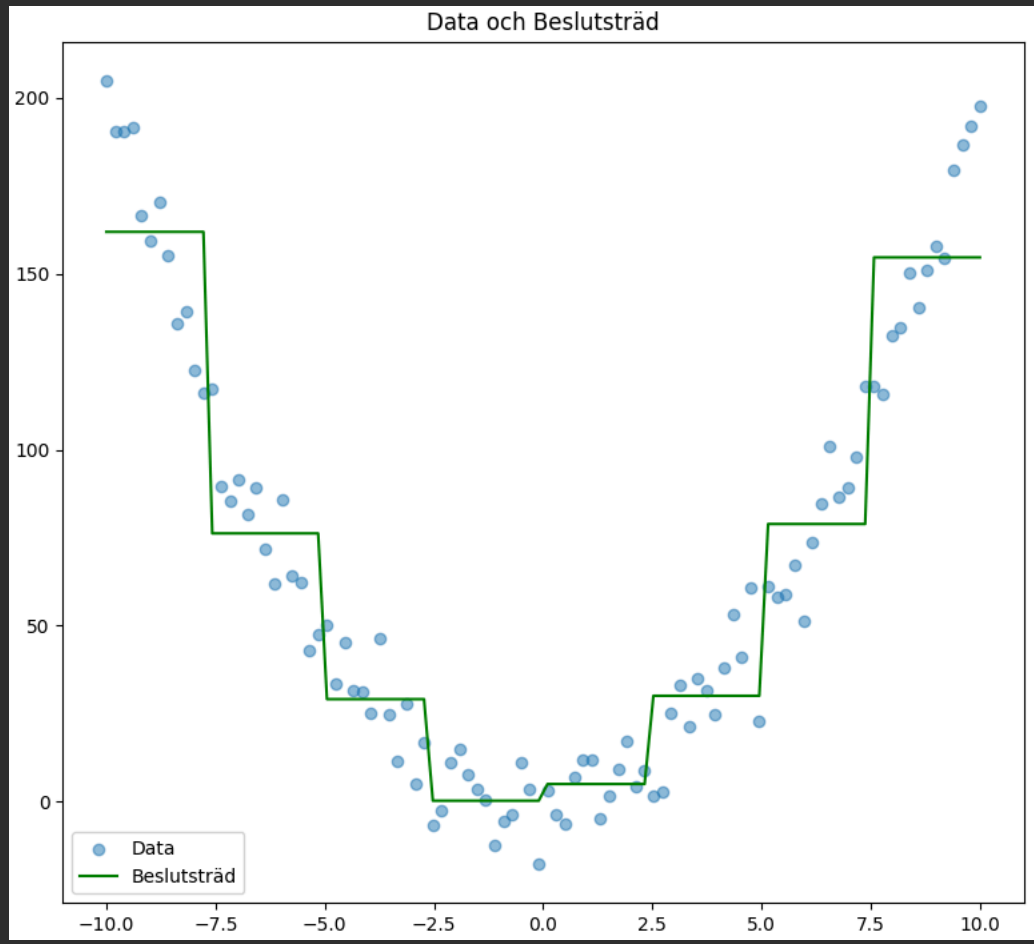
```
tree = DecisionTree(depth=3)
tree.fit(X, y)
y_pred_tree = tree.predict(X)
```

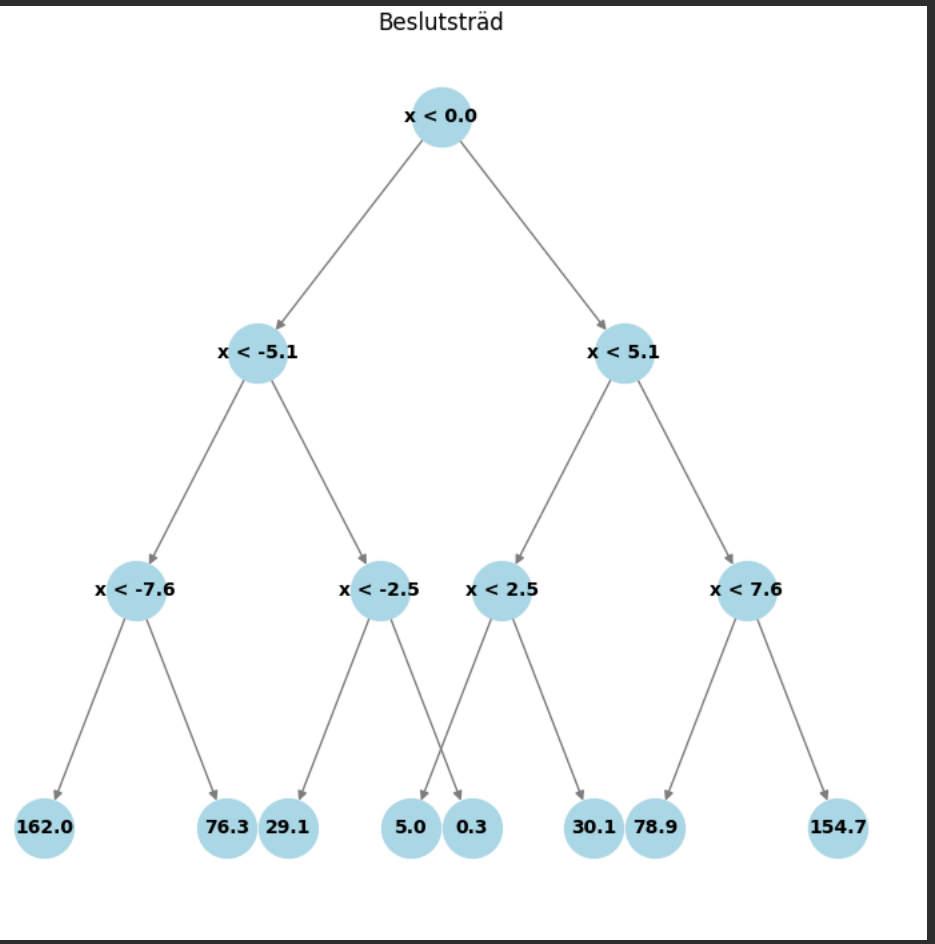
Åskådliggör grafiskt

```
plt.scatter(X, y, alpha=0.5, label="Data")
plt.plot(X, y_pred_tree, color="green", label="Beslutsträd")
plt.legend()
plt.show()
```

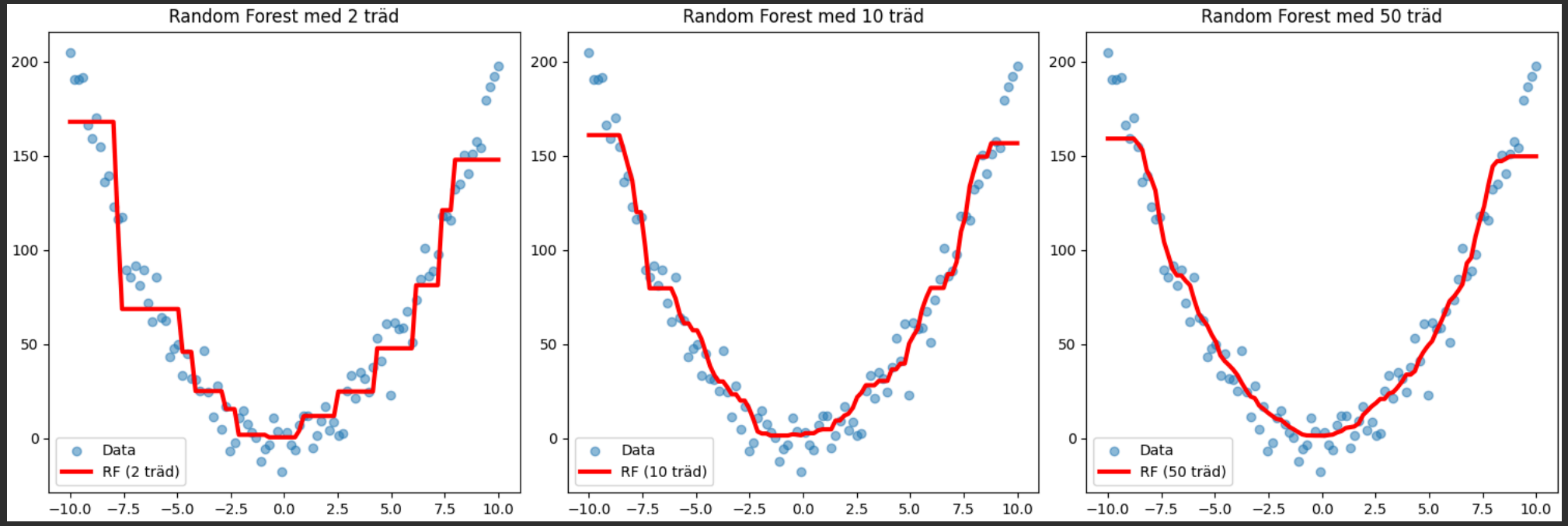


Inklusive "faktiskt" träd





Flera träd = bättre (endast demo)



Praktisk användning

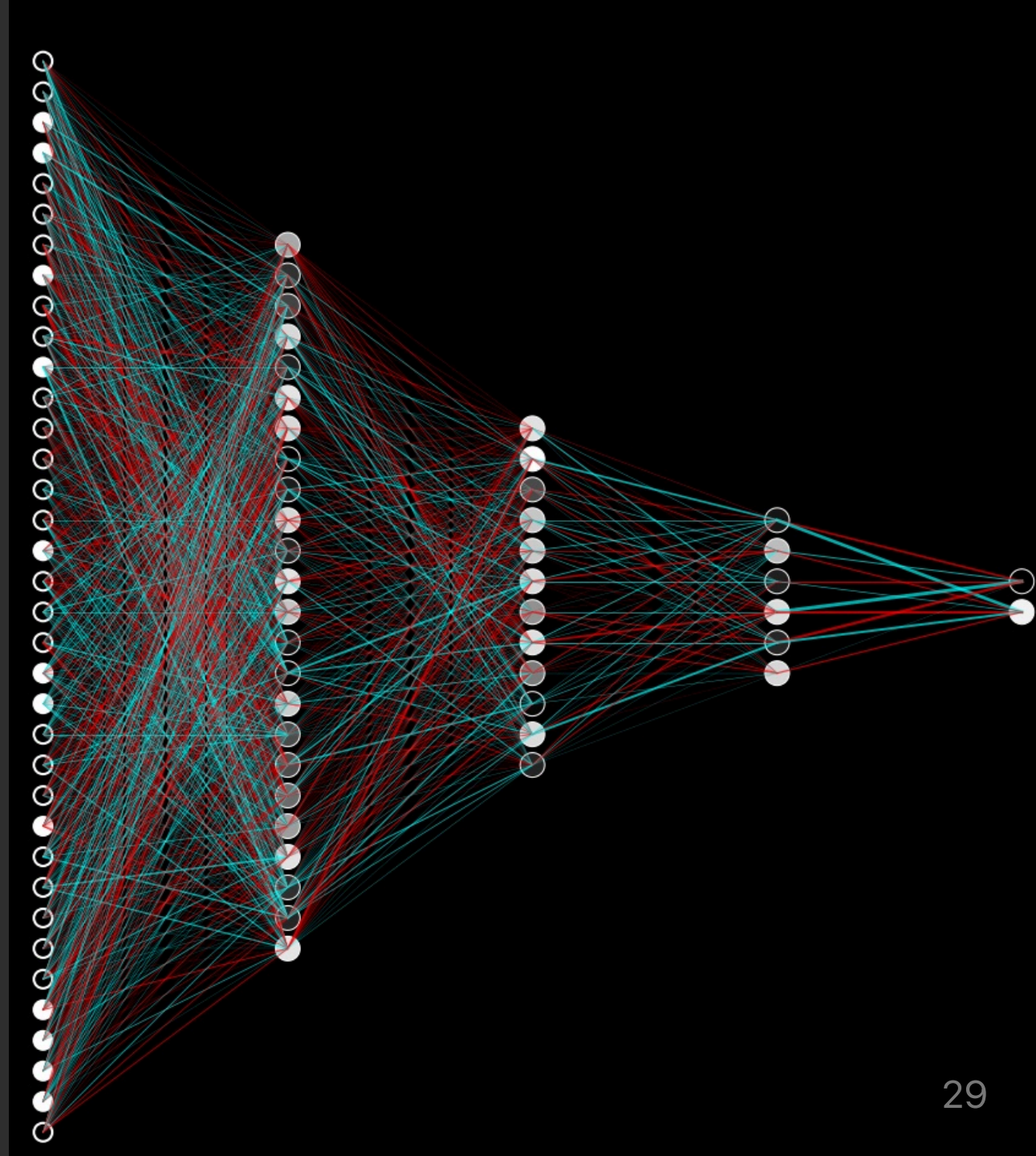
- **Prognoser för fastighetspriser**
  - Finns ett dataset med huspriser (beroende av t.ex. antal rum, boyta, etc.), kan ett beslutsträd hitta enkla regler för att förutsäga pris.
  - 📌 Exempel på regler som ett beslutsträd kan upptäcka:
    - ✅ "Om boytan är över 100 m² → Högre pris"
    - ✅ "Om huset är i Stockholm → Högre pris"

- **Kundsegmentering**
  - Om vi har kunddata (ålder, köphistorik, etc.), kan beslutsträd hjälpa oss att hitta grupper av kunder.
  - 📌 Exempel på regler:
    - ✅ "Om kunden är under 25 och köper teknik → Skicka studentrabatt"
    - ✅ "Om kunden köper för över 5000 kr/mån → Skicka VIP-erbjudanden"

- **Identifiering av bedrägeri i fakturor**
  - Beslutsträd kan hitta ovanliga mönster i fakturor.
  - 📌 Exempel:
    - ✅ "Om en faktura har en summa > 1 000 000 och skickats på en helg → Risk för bedrägeri"
    - ✅ "Om en leverantör har ovanligt många fakturor på samma belopp → Risk för manipulation"

Artificiella neurala nätverk

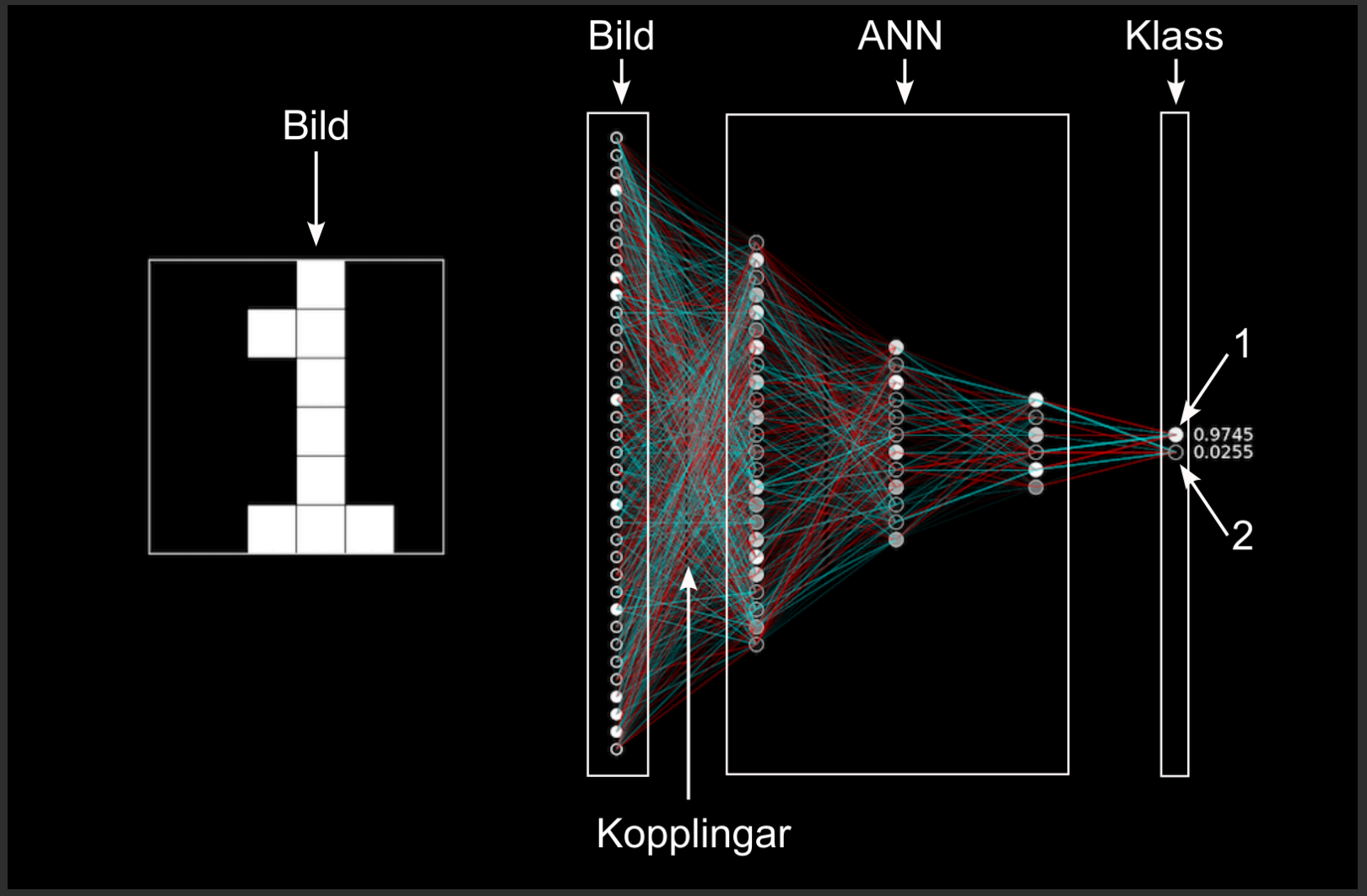
AI:ns "superstjärnor"



Användningsområden

- Självkörande bilar
- Datorseende
- Stora språkmodeller (ChatGPT, Gemini, etc.)

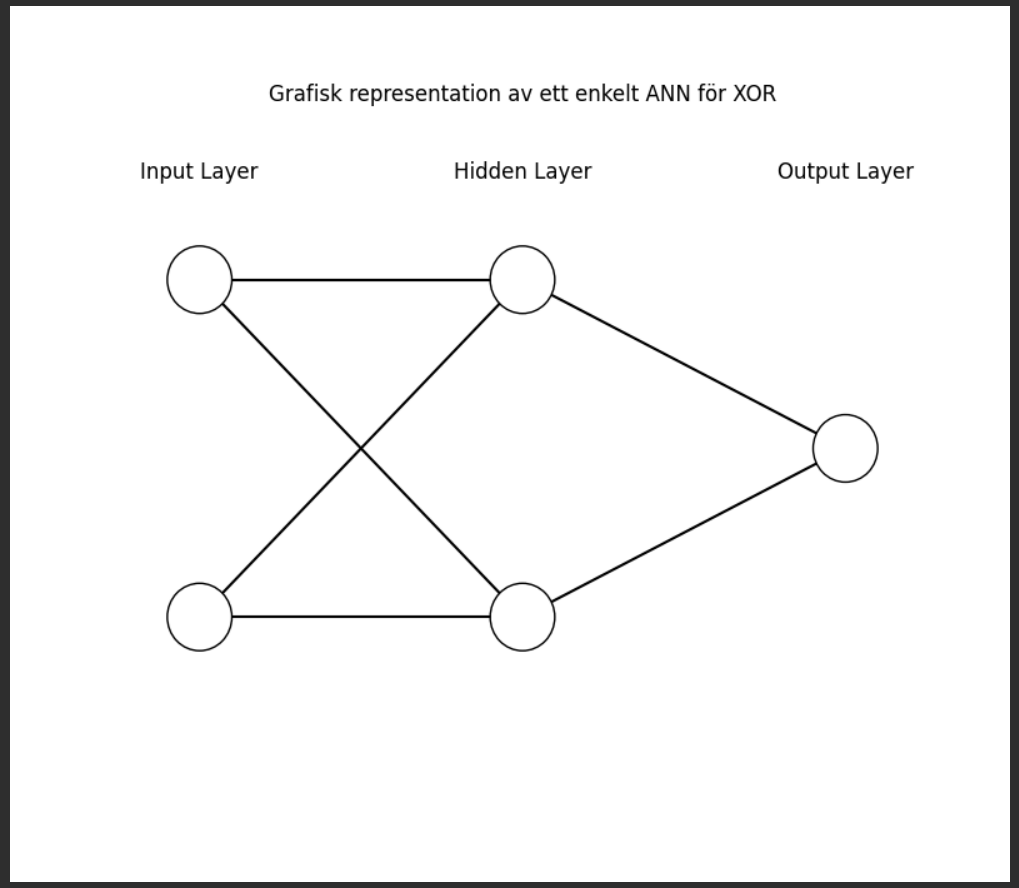
NN i ett nötskal



Grundläggande exempel med NumPy

- Ett neuralt nätverk som lär sig en logisk operator (XOR)
- Poängen är att datorn ska *lära sig en regel som inte är hårdkodad*
    - Detta genom att lära sig från exempeldata (alltså träningsdata)

Grafisk representation av detta exempel-nät



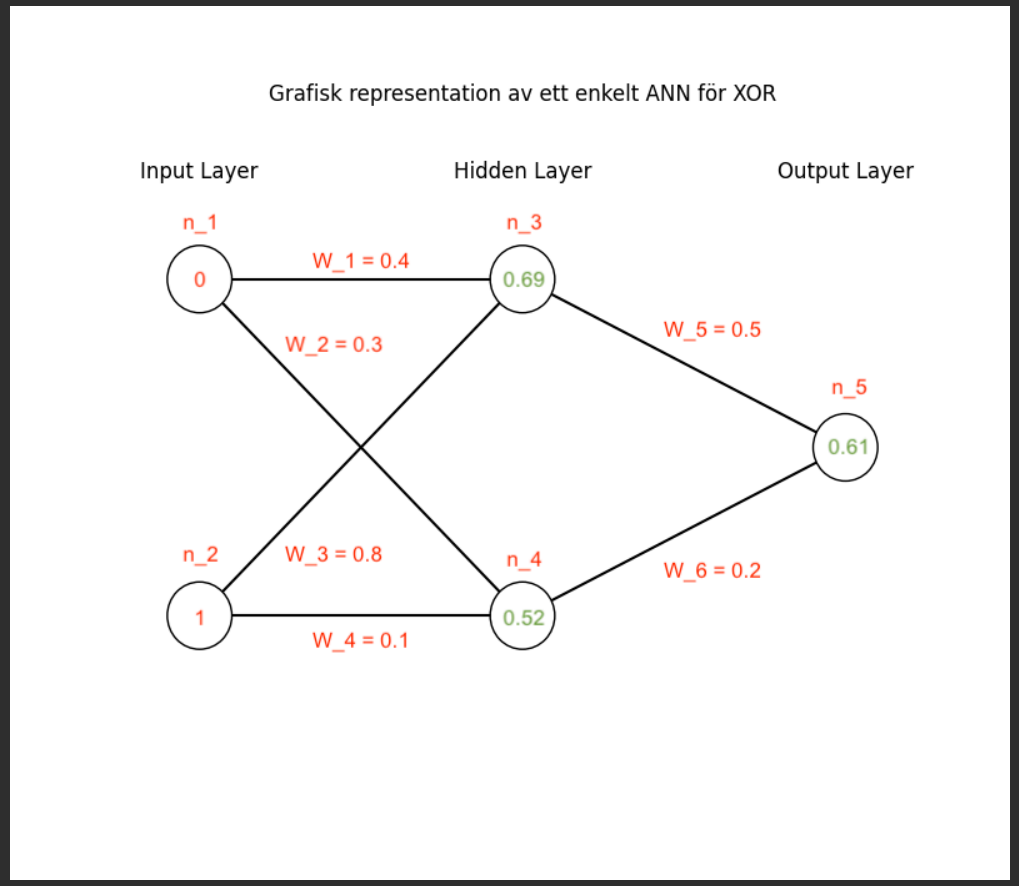
- Det som tränas i ett NN är
  - **Vikter**: Bestämmer hur starkt en nod påverkar nästkommande lager av noder
  - **Biaser**: Läggs till i varje nod för att ge flexibilitet i aktiveringsfunktionen

Definiera träningsdata och facit

- Variabeln `X` innehåller våra indata, i form av par:
  `[0,0]`, `[0,1]`, `[1,0]` och `[1,1]`.
- `y` innehåller de rätta svaren för varje indata enligt XOR:
  För `[0,0]` → `0`
  För `[0,1]` → `1`
  För `[1,0]` → `1`
  För `[1,1]` → `0`

Hur beräknas storheter i ett NN? (initial grov skiss)

Antag följande initiala vikter (bortse från bias just nu)



Hur beräknas storheter i ett NN? (initial grov skiss)

- Antag nätet i bilden ovan
- Antag inmatningen `[0,1]` (vilket ska bli `1` som utmatning enligt ovan)
  Initiera vikter slumpmässigt enligt figuren
- Beräkna $z_3 = n_1 * W_1 + n_2 * W_3 = 0 * 0.4 + 1 * 0.8 = 0.8$
- Beräkna noden n_3's värde med hjälp av den s.k. Sigmoid-funktionen:

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

  - $n_3 = 1/(1 + e^{-z_3}) \approx 0.69$
- Liknande beräkningar ger $n_4 = 0.528$, och $n_5 = 0.61$
- Repetera med övriga tre träningsdata
- Detta kallas framåtpropagering

Denna procedur i NumPy

Ovan skissartade tankegång är faktiskt mycket lättare att visa med kod så nu går vi till botten med det här i NumPy istället.

- Definiera datasettet (träningsdatan):

```
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
```

och "facit":

```
y = np.array([[0],
              [1],
              [1],
              [0]])
```

Initiera parametrarna:

```
input_dim = 2         # Två ingångar
hidden_dim = 2        # Två noder i dolt lager (funkar även med fler!)
output_dim = 1        # En utgång
learning_rate = 0.1 # Hur snabbt inlärningen ska ske
epochs = 10000        # Hur länge vi ska träna
```

Initiera vikterna slumpmässigt:

```
W1 = np.random.randn(input_dim, hidden_dim)      # Vikter mellan inmatning och gömt lager
b1 = np.random.randn(1, hidden_dim)              # Bias mellan inmatning och gömt lager
W2 = np.random.randn(hidden_dim, output_dim)     # Vikter mellan gömt lager och utmatning
b2 = np.random.randn(1, output_dim)              # Bias mellan gömt lager och utmatning
```

Träningsloopen börjar snart. Först ska vi definiera två användbara funktioner.

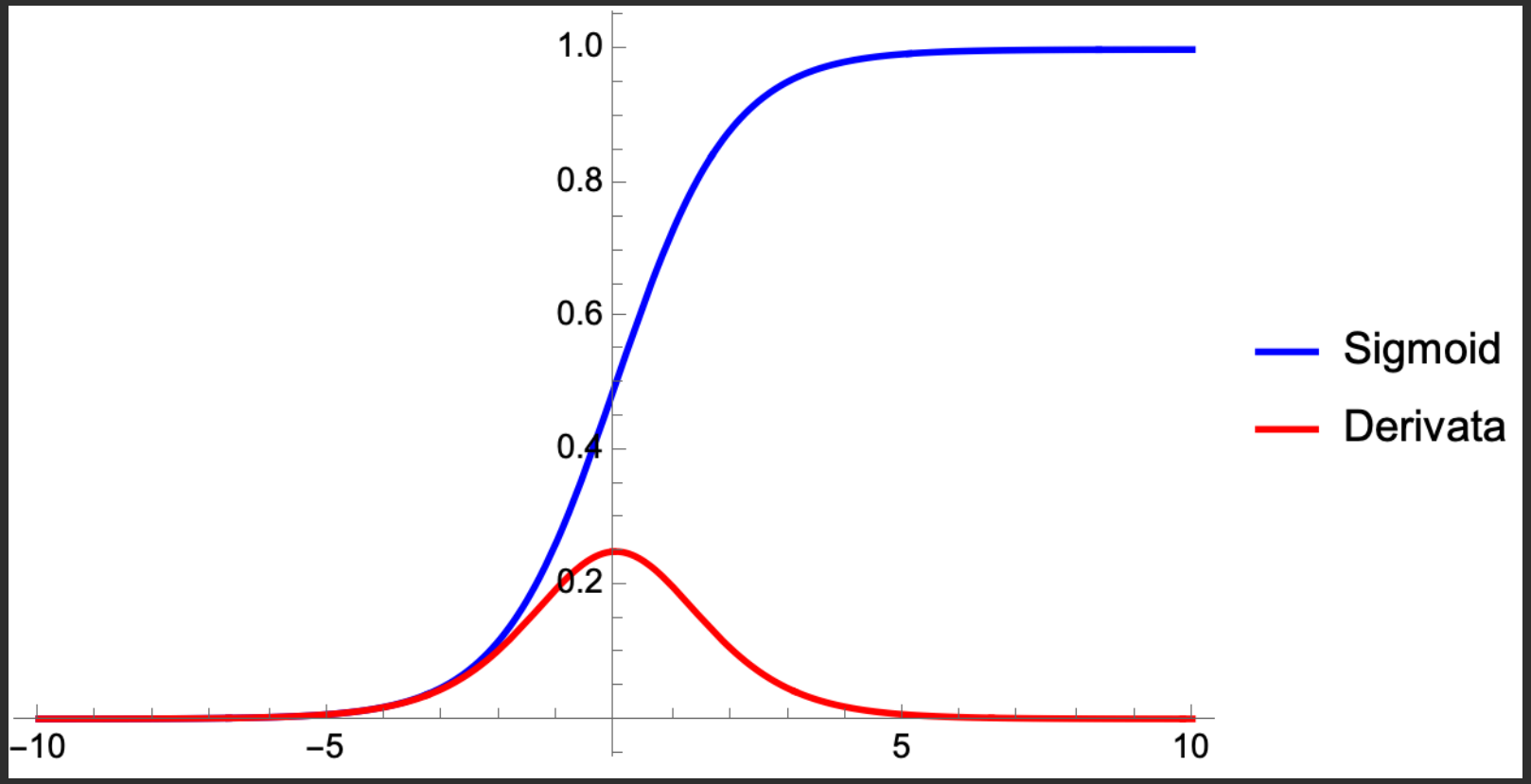
Sigmoid-funktionen (ändrar ett värde till en sannolikhet),

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

samt dess derivata (används under själva inlärningen)

```
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Figur på sigmoidfunktionen och dess derivata:



Sigmoidfunktionen är en av många s.k. aktiveringsfunktioner inom ML.

NU börjar träningsloopen 😎. Vi börjar med den "framåtpropagering" vi skissade på ovan.

```
z1 = np.dot(X, W1) + b1      # Beräkna insignal till dolt lager
a1 = sigmoid(z1)             # Aktivering i dolt lager
z2 = np.dot(a1, W2) + b2     # Beräkna insignal till utgångslagret
a2 = sigmoid(z2)             # Utgången från nätverket
```

Tänk på att detta är matriser! Exempel:

$$z_1 = X \cdot W_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} -0.725 & -0.942 \\ 0.217 & 0.326 \end{pmatrix} = (4 \times 2)\text{-matris}$$

Beräkna fel och medelkvadratfel:

```
error = y - a2
loss = np.mean(error**2)
```

Bakåtpropagering genom beräkning av gradienten (derivatan) av felet med avseende på aktiveringen i utmatningslagret

```
d_a2 = error * sigmoid_derivative(z2)
```

Gradienten av förlustfunktionen med avseende på vikterna i det sista lagret

```
d_W2 = np.dot(a1.T, d_a2)   # .T gör (M x N-matris till N x M-matris)
```

Hur utmatningsneuronens bias ska justeras till nästa träningsloop

```
d_b2 = np.sum(d_a2)
```

En liknande procedur upprepas för det resterande lagret (nästa sida)

Bakåtpropagering genom beräkning av gradienten av felet med avseende på aktiveringen i det mittersta lagret

```
d_a1 = np.dot(d_a2, W2.T) * sigmoid_derivative(z1)
```

Gradienten av förlustfunktionen med avseende på vikterna i det sista lagret

```
d_W1 = np.dot(X.T, d_a1)
```

Hur utmatningsneuronens bias ska justeras till nästa träningsloop

```
d_b1 = np.sum(d_a1, axis=0, keepdims=True)
```

Nu, äntligen, kan vi uppdatera vikterna med de gradienter vi beräknat ovan:

```
W2 += learning_rate * d_W2
b2 += learning_rate * d_b2
W1 += learning_rate * d_W1
b1 += learning_rate * d_b1
```

Dessa vikter och bias går nu in som uppdaterade värden *högst upp i loopen* igen, och hela proceduren upprepas.

- Varje loop brukar kallas `epoch`
  - Tiotusentals epoker (!) kan krävas för tillräckligt bra resultat
  - Här kan vi alltså laborera med parametern `epochs` och se hur många som krävs
- Den andra väsentliga parametern är `learning_rate`
  - Denna säger hur abrupt vi ska uppdatera vikter och bias
  - Typiskt ett lågt värde mycket mindre än ett, men tål att laboreras med

Utvärdering efter träning vid 10000 epoker och learning rate 0.1

```
print(a2)
```

```
[[0.04746178]
 [0.94491401]
 [0.95708922]
 [0.04188052]]
```

dvs, maskinen har lärt sig XOR-operatorn rätt bra med enbart träningsdatan!

Sammanfattning, neurala nätverk

- Skriptet definierar ett nätverk med två lager som tränas på XOR-data.
- Genom att iterativt räkna ut ett svar, jämföra med det korrekta svaret och justera sina vikter, lär sig nätverket att producera rätt output.
- Vi tränar datorn på detta sätt för att demonstrera hur neurala nätverk kan anpassa sig och lära sig komplexa samband genom exponering för exempel, vilket är grunden för maskininlärning.

Denna metod, att lära datorn genom upprepade justeringar baserat på fel, är en central idé inom modern AI och används i många olika applikationer.

Demo, ost-AI

(om det mot förmodan finns tid)