Kapitel 1

Maskinlæring

Amanda Le
Andreas Raaskov Madsen

1 Introduktion

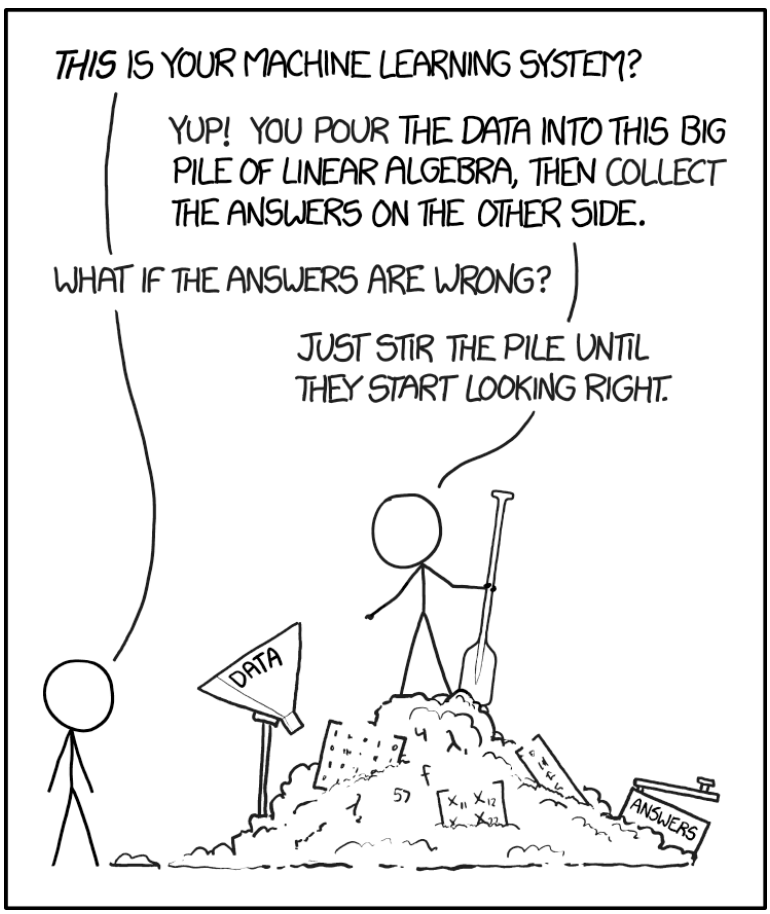
Man kan nogle gange blive træt af at kode og ende med at tænke: "Kunne computeren ikke selv kode det hele for mig?!"", og så er det jo godt, at der findes maskinlæring, da man med maskinlæring kan lære computeren at kode dine funktioner og algoritmer for dig, se eksempelvis: GitHub Copilot. Men hvordan virker det, og hvad er maskinlæring egentlig?

Maskinlæring dækker over en række metoder, hvorved computeren selv kan lære at finde mønstre i store mængder data. Dette kan benyttes til både klassifikation; eksempelvis at klassificere om en person har en bestemt sygdom eller ej, og til at lave forudsigelser om fremtiden; såsom hvordan vejret kommer til at se ud i morgen.

Udgangspunktet for maskinlæring er data - en rigtig stor mængde data - og derfor er maskinlæring også kendt som datavidenskab. I medierne er maskinlæring ofte kendt som kunstig intelligens (AI), og selvom der findes kunstig intelligens, som ikke bygger på maskinlæring, så har de sidste 20 års

fremskridt i kunstig intelligens hovedsageligt været inden for maskinlæring.

I dette forløb vil vi kigge på de to hovedkategorier inden for maskinlæring: Supervised maskinlæring og unsupervised maskinlæring. Vi vil møde eksempler på kendte algoritmer, som bliver benyttet inden for begge hovedkategorier og vi vil kigge på teorien, der danner grundlaget for algoritmerne.



Figur 1.1: XKCD 1838

2 Data

Maskinlæring er en underdel af kunstig intelligens. Kunstig intelligens er ofte kendt som datavidenskab, da det ofte handler mere om data end om algoritmen. Datavidenskabsfolk har et ordsprog: "Det er bedre at have et godt datasæt og en dårlig algoritme, end det er at have en god algoritme og et dårligt datasæt". Data kan i sig selv beskrives som alt, der kan kvantificeres med tal. Eftersom stort set alt kan beskrives med tal, så gør det datavidenskab til en videnskabelig disciplin, der kan have opgaver fra alle andre videnskaber.

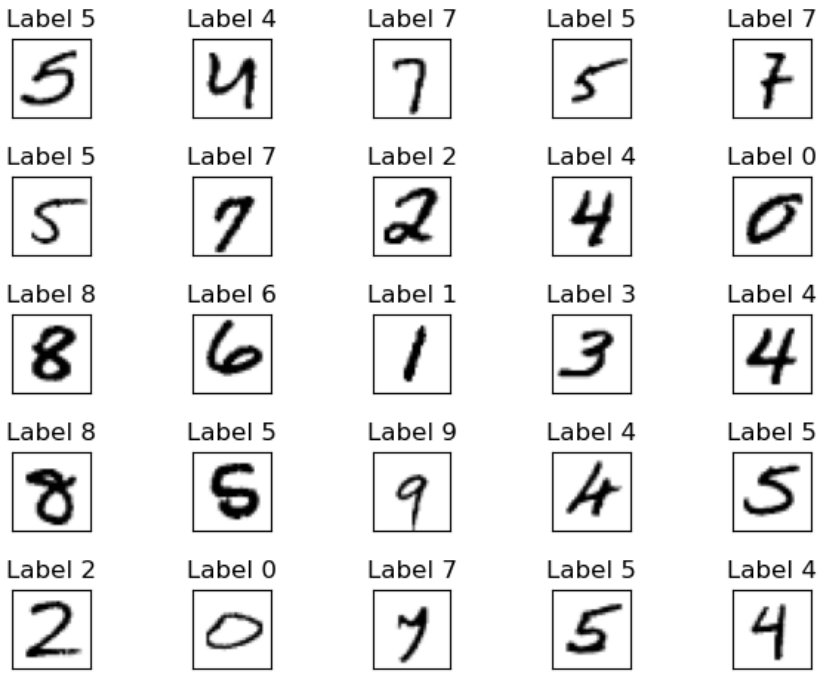
Ét af de datasæt, som vi vil arbejde med her på campen, indeholder næringsdata fra mad serveret på fastfoodkæden Subway's restauranter. Datasættet indeholder mange forskellige attributter, og vi vil fokusere på data om mængden af kulhydrater og mængden af kalorier. Årsagen til, at vi fokuserer på disse to attributter, er, at hver attribut kan ses som en dimension og for at give en intuitiv ide af, hvordan algoritmen virker, så vil vi lave 2D plots. Hvis algoritmen er ordentligt implementeret, så kan den finde sammenhæng i data med flere dimensioner. Dette er en fordel, da man med data i forhold til mennesker typisk arbejder med data i mange dimensioner.

Det andet datasæt, som vi skal kigge på, hedder MNIST og dette datasæt består af billeder af håndskrevne tal.

For at kunne lave maskinlæring på billeder, så skal vi forstå, hvordan, billeder bliver repræsenteret som data i en computer. Billeder i MNIST datasættet har 28X28 pixels og er i sort-hvid. Af denne grund kan et MNIST billede blive repræsenteret med en liste af 784 tal mellem 0 og 255, hvor 0 betyder, at en pixel er helt hvid og 255 betyder, at den er helt sort.

3 Intro til biblioteker

Når man skriver kode, så kan man nogle gange have brug for funktioner, der ikke er en del af den normale Python pakke. I stedet for at skrive disse funktioner fra bunden, så kan man importere andres kode og benytte dem



Figur 1.2: Eksempler af billeder i MNIST datasættet

i sin egen kode. Dette kaldes biblioteker. De mange ekstra biblioteker er én af de største fordele ved at bruge Python.

Numpy

Det mest brugte bibliotek, når man laver datavidenskab, kaldes `numpy`. `numpy` indeholder og giver en masse masse matematiske funktioner, som man også ville kunne finde i ethvert andet CAS værktøj. Måden man importerer `numpy` er via kommandoen:

```
1 import numpy as np
```

"as np" sørger for, at man kan benytte forkortelsen `np` som et synonym for `numpy`. For at få adgang til funktionerne i `numpy` behøver man eksempelvis kun at skrive:

```
1 np.function()
```

i stedet for at skrive: `numpy.function()` hele tiden.
Derudover introducerer `numpy` en særlig type liste, som man kan anvende til lineær algebra. Listen kaldes et numpy array. Et numpy array adskiller sig fra en almindelig Python liste ved, at et numpy array fungerer ligesom en matematisk matrix.
En matrix er et rektangulært skema af tal. Forskellen på et numpy array og en almindelig liste i Python kan ses i følgende eksempel:

Eksempel
[1,2]*2 → [1,2,1,2]
np.array([1,2])*2 → np.array([2,4])

unfML

Vi har lavet nogle hjælpefunktioner således, at i hurtigt kan gå i gang med at lave maskinlæring. Funktionerne ligger i biblioteket unfML og biblioteket kan importeres på følgende måde:

```
1 import unfML
```

Datasættet til maskinlæringsopgaverne kan nu fås ved hjælp af funktionen `unfML.data(opgavenummer)`. For unsupervised learning fås X, for supervised fås X og Y. For at evaluere din model benyttes kommandoen: `unfML.eval(opgavenummer, modeloutput)` eller `unfML.plot(opgavenummer, modeloutput)` for en grafisk illustration. Modellens output vil for K-means være en liste K centrums og for perceptronen vil det være W.

4 Unsupervised maskinlæring og K-means algoritme

Unsupervised maskinlæring dækker over en række metoder til at finde mønstre i data. Metoderne bliver givet en række observationer uden at give et label, der siger, hvad det rigtige svar er, da vi ikke nødvendigvis på forhånd ved, hvad det er vi leder efter. Typiske opgaver i unsupervised maskinlæring er; at klassificere data i forskellige grupper (clustering), finde afvigelser (outlier detection) eller finde sammenhænge, såsom hvis en person køber havregryn, så køber de typisk også mælk (association mining). Vi vil fokusere på en algoritme, K-means, som benyttes til klassificering.

K-means

K-means er en simpel metode til at finde klynger i store mængder data. Her kan en klynge beskrives, som gruppe datapunkter, der ligger tæt på hinanden og derfor hører sammen. Algoritmen starter med at vælge K tilfældige punkter som centrum for klynger; derefter kører algoritmen i et loop, som har to trin.

- Trin 1: K klynger udvælges; for hvert datapunkt udregnes den euklideafstand (Pythagoras' læresætning) til ethvert K centrum. Derefter indsætter man punktet i den klynge, hvis centrum punktet er tættest på.

- Trin 2: For hver klynge udregnes et nyt centrum ved at tage middelværdien af alle punkter.

K-means konvergerer, hvilket vil sige, at på et tidspunkt ændrer resultatet sig ikke længere, selvom man fortsætter loopet. Når dette sker kan algoritmen stoppes, og de resulterende klynger er resultatet af klassificeringen.

Algorithm 1 K-means

```
X = load data
K = antal klustere
punkter,demensioner=np.shape(x)
Center = np.array(random.sample(X,K)) #tilfældige valgte punkter
SidsteCenter = np.empty((K,demensioner))
while Center != SidsteCenter do
    klusters=[ [] for _ in range(K)]
    SidsteCenter = Center.copy()
    for  p in range(punkter) do
        afstand=[]
        for k in range(K) do
            afstand.append(np.sqrt(np.sum((X[p]-center[k])**2)))
        end for
        klusters[np.argmin(afstand)].append(X[p])
    end for
    for k in range(K) do
        center[k,:]=np.mean(klusters[k],axis=0)
```

Opgave 1.1

Implementér K-means algoritmen på Subway datasættet, og plot resultatet fra 3 klynger. For bedre forståelse og debugging kan man plotte datasættet under hver iteration. Efter 3 klynger er plottet; giv plot funktionen argumentet sande=True for at se, hvilke typer mad, der gemmer sig i datasættet.

Opgave 1.2

Få din K-means algoritme til at køre på MNIST datasættet med 10 klynger. Plot de 10 klynger.

Opgave 1.3

Teori: Snak med din sidemand om følgende:

- Giver K-means det samme svar hver gang? Hvorfor/hvorfor ikke?
- Hvad betyder antallet af klynger?
- Hvilke praktiske implikationer kan K-means have?

Matematikken bag K-means

K-means virker som en smart algoritme - men det store spørgsmål er vel: Hvorfor virker dette?
Ideen bag K-means er at uddelegere en række observationer $x_1, x_2, \ldots, x_i$ til klynger således, at afstanden til den nærmeste vektor μ_k er minimeret.

Opgave 1.4

Teori: Snak med din sidemand om følgende:

- Hvorfor giver K-means intuitivt mening?

For at forklare hvorfor K-means virker matematisk, så defineres følgende: Lad z_{ik} beskrive, at en observation i tilhører en klynge k. Der gælder således, at hvis $z_{ik} = 1$, så tilhører observation x_i til klyngen k. $z_{ih} = 0$ betyder, at observationen ikke tilhører klynge h, hvor der gælder at $h \neq k$.
Vi kan dermed opstille følgende udtryk, som beregner fejlen i K-means.

$$E = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} \left\| \boldsymbol{x}_i - \boldsymbol{\mu}_k \right\|_2^2 \tag{1.1}$$

Algoritmen går ud på at finde de bedst mulige z_{ik} og μ_k, som vil minimere E

I trin 1 af algoritmen vil man uddele de forskellige datasæt i klynger. Vi har dermed faste μ_k og skal minimere z_{ik}. For at minimere z_{ik} vil vi, for hver observation i, vælge z_{ik}, således at:

$$\sum_{k=1}^{K} z_{ik} \left\| \boldsymbol{x}_i - \boldsymbol{\mu}_k \right\|_2^2$$

minimeres.

Dette kan også opstilles som en funktion, der ser således ud:

$$z_{ik} = \begin{cases} 1 & \text{if } k = \arg\min_h \|\boldsymbol{x}_i - \boldsymbol{\mu}_h\|_2^2 \\ 0 & \text{otherwise} \end{cases}$$

I trin 2 af algoritmen vil man udregne et nyt centrum for hver klynge og her er z_{ik} fast.

$$\nabla_{\boldsymbol{\mu}_k} E = 2 \sum_{i=1}^{N} z_{ik} \left(\boldsymbol{x}_i - \boldsymbol{\mu}_k\right) \tag{1.2}$$

Vi kan isolere μ_k ved at sætte (1.2) lig med 0, da vi gerne vil have, at der ikke er nogen ændring. Vi får dermed følgende:

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} z_{ik} \boldsymbol{x}_i}{\sum_{i=1}^{N} z_{ik}}.$$

Dette udtryk beskriver, at centrum for hver klynge er summen af observationer i klyngen over antallet af observationer i hver klynge; altså gennemsnittet af observationer til klyngen k

Vi kan dermed se, at trin 1 og trin 2 i algoritmen medfører, at E minimeres - og det er derfor K-means virker.

Opgave 1.5

Isolér μ_k i (1.2). Benyt evt. følgende regneregler:

$$\sum_{n=s}^{t} C \cdot f(n) = C \cdot \sum_{n=s}^{t} f(n)$$

$$\sum_{n=s}^{t} f(n) + \sum_{n=s}^{t} g(n) = \sum_{n=s}^{t} (f(n) + g(n))$$

5 Supervised maskinlæring

Supervised maskinlæring minder meget om den måde mennesker lærer på. Algoritmerne vil prøve at gætte, hvad det rigtige svar er. Eksempel i klassificering: Identificere en hund, når man får et billede af en hund. Hvis algoritmen gætter rigtigt, så beholder vi den, som den er, men hvis den gætter forkert, så laver vi en lille justering, der får den til at give et bedre gæt næste gang. Kort sagt kan alle supervised maskinlæringsalgoritmer reduceres til følgende ligning:

$$f(X) = Y$$

Hvor X er input data, og Y er, hvad vi prøver klassificere. Hvis X eksempelvis er et billede, så kan Y være, hvad det er et billede af. $f()$ er en matematisk funktion, som giver Y, når den får X input. Problemet er, at vi ikke kender $f()$, og vi er for dovne til manuelt at formulere den - kort sagt: vi skal lære $f()$.

Måden vi lærer på er at minimere en objektiv funktion eller en fejlfunktion. Den mest almindelige objektive funktion ser sådan ud:

$$L(X, Y) = \frac{1}{n} \sum_{i}^{n} (f(x_i) - y_i)^2$$

Her ses det tydeligt, at hvis $f(X) = Y$, så er $L(X, Y) = 0$. Dette indikerer, at vi laver en perfekt klassifikation, og at der ikke er nogen grund til at ændre algoritmen. Der bemærkes desuden, at hvis værdien for $f(X)$, og hvad vi ønsker at klassificere Y, har en stor differens, så vil $f(X) - Y$, resultere i et tal, som er stort, hvis fejlen er stor. Hvis målet er at klassificere katte og hunde kan man lade én være 0 og den anden 1. I så fald er $L(X, Y) = 0$, hvis vi gætter rigtigt, og $L(X, Y) = 1$ hvis vi gætter forkert.

Perceptronen

For at give en forståelse af, hvad neurale netværk er, så vil vi fokusere på at lære perceptron algoritmen. Kort sagt kan perceptronen anses som et neutralt netværk med en enkelt neuron.

Definition 5.1. Perceptronen består af ligningen:

$$\hat{Y} = b + \sum_i W_i \cdot x_i$$

W er et antal vægte, som vi ønsker at lære. x_i er en attribut fra vores input data. b er kendt, som en bias. b kan forstås som en ekstra attribut til vores datasæt, som altid er 1. Hvis vi har n attributter, så skal vi lære $n + 1$ vægte, hvor den sidste vægt vil være $b = w(n + 1) \cdot 1$.

Læring

Vi ønsker at finde et antal vægte W, som minimerer vores objektive funktion.

Sætning 5.2. For en perceptron gælder det, at vi kan finde den optimale løsning ved at opstille den objektive funktion og løse udtrykket for W og b:

$$0 = \frac{\partial}{\partial W} \frac{1}{m} \sum_i^m \left(b + \sum_i^n W_j \cdot x_j - y \right)^2$$

Dette vil resultere i et ligningssystem med $n + 1$ ligninger med $m + 1$ ubekendte, hvor n er antal vægte, og m er antal datapunkter. Når man laver mere komplicerede algoritmer kan man have millioner af vægte med hundredtusindvis af datapunkter, hvilket medfører, at man hurtigt løber tør for hukommelse på selv de bedste computere.

Problemet med den optimale løsning er også, at den kan overfitte på træningssættet. Mange algoritmer vil klare sig bedre på et testset, hvis de bliver stoppet før de når deres optimale løsning. Derfor vil vi i stedet implementere en metode kendt som *stokastisk gradient nedstigning.*

Definition 5.3. Update funktionen for vægtenden er givet som:

$$W^{+1} = W \cdot \lambda \cdot \Delta_W$$

λ er kendt som læringsraten. En for lav λ gør, at træningen tager længere tid, hvorimod en for høj λ gør træningen ustabil, og kan forhindre, at træningen

overhovedet sker. Δ_W er gradienten på vægten. Når man træner på et enkelt datapunkt er den givet som:

$$\Delta_W = 2(y - \hat{y}) \cdot x$$

Hvis vi træner på et batch (flere data punkter) har vi:

$$\Delta_W = \frac{2}{n} \sum_{i}^{n} (y_i - \hat{y_i}) \cdot x_i$$

Bevis. Antag, at vi har en perceptron med 1 vægt. I så fald har vi:

$$\Delta_W = \begin{bmatrix} \frac{\partial}{\partial W_1} L(x, y) \\ \frac{\partial}{\partial b} L(x, y) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial w_1} (y - w_1 \cdot x_1 + b)^2 \\ \frac{\partial}{\partial b} (y - w_1 \cdot x_1 + b)^2 \end{bmatrix} \tag{1.3}$$

Vi løser hver enkelt vægt via kædereglen:

$$\frac{\partial}{\partial w_1} (y - w_1 \cdot x_1 + b)^2 = 2 \cdot (y - w_1 \cdot x_1 + b) \frac{\partial}{\partial w_1} (w_1 \cdot x_1 + b))$$

Her ses det, at $(w_1 \cdot x_1 + b) = \hat{y}$, samt at $\frac{\partial}{\partial w_1}(w_1 \cdot x_1 + b) = x_1$. Udtrykket kan derfor reduceres til:

$$\Delta_{W_1} = 2 \cdot (y - \hat{y}) \cdot x_1$$

Biasen er:

$$\frac{\partial}{\partial b} (y - w_1 \cdot x_1 + b)^2 = 2 \cdot (y - w_1 \cdot x_1 + b) \frac{\partial}{\partial b} (w_1 \cdot x_1 + b))$$

Med $\partial b(w_1 \cdot x_1 + b) = 1$ kan vi reducere til:

$$\Delta_b = 2 \cdot (y - \hat{y})$$

□

Algorithm 2 Perceptron

```
X,Y = load data
W = np.zeros(len[X[0]]+1)
I = antal iterationer
λ = læringsraten
SidsteCenter = np.empty((K,dimensioner))
for  idx, xi in enumuerate(x) do
    prediction = np.dot(W[1:], xi) + W[0]
    update = np.array(λ * (Y[idx] - prediction))
    W[1:] += update * xi
    W[0] += update
```

Opgave 1.6

Implementér perceptron algoritmen på sandwich datasættet med attributterne **kulhydrater** og **mængden af kalorier** til at forudsige, om noget er en salat eller en sandwich.

Opgave 1.7

Implementér perception på MNIST datasættets 784 attributter. Få den til at kende forskellen på 1 og 0.

6 Maskinlæringsbiblioteker

Folk, der laver datavidenskab og maskinlæring, er ofte ikke de bedste programmører. Selvom mange udvikler deres egne algoritmer, så vil de fleste datavidenskabs-job handle om at bruge den rigtige algoritme til at løse et nyt klassificerings problem. Derfor findes der et stort antal biblioteker i Python således, at man kan køre algoritmerne, hvis bare man ved hvordan man bruger dem.

Scikit-learn

Et biblotek, som vi skal arbejde med, er scikit-learn, der et open source og drevet af fællesskabet. Sckikit-learn indeholder alle de algoritmer, som man ville støde på i et introduktionskursus til maskinlæring og mange flere. Dokumentationen til alle algoritmerne kan findes på https://scikit-learn.org/.

Separering af test- og træningsdata

Før at I er klar til at klassificere enhver form for data, er der dog en ting, som vi har glemt at lære jer. Selvom det næsten er det vigtigste, så har vi ikke snakket om, hvordan man separerer trænings- og testdata. Indtil videre har vi lavet algoritmer, som laver en god model af træningsdata. Spørgsmålet er, hvad der sker, når den ser et nyt datapunkt, som den ikke har set før? De fleste algoritmer vil have en meget dårlig performance på deres testsæt, selvom de har en god performance på deres træningssæt. Ironisk nok så vil meget avancerede algoritmer ofte få en dårligere performance på deres testsæt end simple algoritmer grundet *overfitting*. En maskinlæringsalgoritme bør altid vurderes på dens performance på testsættet, eftersom vi ønsker en algoritme, der kan genkende ting, som den ikke har set før.

Sætning 6.1. Som en tommelfinger regel vil man bruge 80% af sin data som træningsdata og 20% som testdata.

Opgave 1.8

Opdél dit MNIST datasæt i et testsæt og et træningssæt via Sci-kit learn.

Opgave 1.9

Find K-means og perceptronen på sci-kit learn og lav øveleserne 1.7 og 1.2 med sci-kit learn.

Opgave 1.10

Find en supervised algoritme på sci-kit learn, som vi ikke har gennemgået. Træn den på dit MNIST træningssæt og test den på testsættet.

Dyb læring

Der findes mange algoritmer, som kan lave supervised maskinlæring, men Neurale Netværk er klart de mest populære grundet deres evne til at lære næsten alt, hvis de er dybe nok og har nok data. Neurale netværk er bygget af forbundne lag med et vist antal neuroner, hvor hver neuron ligner en perceptron. Antallet af neuroner bestemmer, hvor godt netværket er, men hvis der ikke er nok træningsdata, så risikerer du at overfitte.

Perceptronen kan blive set som et neutralt netværk med ét lag. For at lave neurale netværk med flere lag kan vi bruge et bibliotek, som automatisk udregner gradienten. Biblioteker har også mange forskellige typer lag, såsom convolutional lag, der kan bruges på billeder og reccurent lag, som har hukommelse. De to mest populære biblioteker hedder Pythorch og TensorFlow.

Opgave 1.11

Implementér en perceptron via Pythorch. Brug funktionen .backward til at udregne δ_w

Opgave 1.12

Implementér et mindre neutralt netværk via pytorch.