# *Maskinlæring for økonomer*
# Kompendium i SOK-3023

**Markus J. Aase**
Universitetslektor i matematikk og statistikk

**November 2024**
Handelshøgskolen UiT

*"In the brain, you have connections between the neurons called synapses, and they can change. All your knowledge is stored in those synapses."*
*— Geoffrey Hinton*

# Om kompendiet

*Dette kompendiet handler om maskinlæring, og er designet for økonomistudenter. Kurset forutsetter grunnleggende matematikk, statistikk, og programmeringsferdigheter i Python. Dog, kreves det ikke at studentene har mye kjennskap til maskinlæring og AI fra før. Det er kunnskap jeg håper studentene vil tilegne seg gjennom dette kurset, hvor dette kompendiet, sammen med de fysiske forelesningene vil gi en god innføring til maskinlæring - i teori og praksis.*

*Læring kan være en krevende prosess, men den er også full av muligheter og belønninger. Å forstå maskinlæring vil ikke bare styrke dine analytiske ferdigheter, men også gi deg et fortrinn i dagens og fremtidens arbeidsmarked, der datadrevne beslutninger blir stadig viktigere.*

*La oss begynne reisen inn i maskinlæringens fascinerende verden, lykke til!*

*— Markus J. Aase*

# Innhold

# 1 Introduksjon til maskinlæring

Maskinlæring er et omfattende felt som i stor grad handler om å lage modeller som kan lære fra data, og gjøre prediksjoner eller ta beslutninger basert på dem. I hovedsak kan maskinlæring deles inn i ulike teknikker som har til felles at de forsøker å estimere en underliggende funksjon fra data. Dette inkluderer metoder som er spesifikke for ulike typer data, som språkmodeller for tekst eller bildeklassifisering for visuelle data. I dette kurset vil vi utforske flere av disse teknikkene, med fokus på både teori og praktiske anvendelser.

Vi starter med en innføring i hva maskinlæring (ML) og kunstig intelligens (AI) er, og ser på anvendelser innenfor finans og økonomi. Kurset vil ta for seg hva som skiller prediksjon fra inferens, og gir en forståelse av sentrale konsepter som *reducible* og *irreducible* feil, samt forskjellen mellom regresjon og klassifikasjon. Vi vil også introdusere de matematiske fundamentene som trengs for å forstå ML, som lineær algebra, tensor-operasjoner, sannsynlighetsteori og derivasjon.

Når vi går dypere inn i maskinlæring, kommer vi til å fokusere på nevrale nettverk (NN). Dette omfatter alt fra nettverksarkitektur, som hvordan ulike lag er organisert, til funksjoner som styrer læringsprosessen, slik som tapsfunksjon (loss function) og aktiveringsfunksjoner. Gjennom eksempler som MNIST-datasettet for håndskrevne tall, vil vi se hvordan nevrale nettverk kan trenes for å gjøre prediksjoner. Vi vil også dykke inn i avanserte teknikker, som språkmodeller og bildeklassifisering. Et eksempel på en språkmodell er Whisper, som kan brukes til å transkribere tale til tekst. Slike modeller er basert på dyp læring og viser kraften til nevrale nettverk i naturlig språkbehandling. Når det gjelder bildeklassifisering, vil vi utforske hvordan nevrale nettverk kan trenes til å gjenkjenne objekter i bilder, og vi vil bruke evalueringsmetrikker som confusion matrix, og dens evalueringsmetrikker, for å si noe om hvor god eller dårlig modellen vår er.

Videre ser vi på anvendelser innen tidsserieanalyse, spesielt med fokus på LSTM-modeller (Long Short-Term Memory). Disse modellene er spesielt nyttige når vi skal analysere sekvensielle data, som finansielle tidsserier, der sammenhengen mellom data på ulike tidspunkter er viktig. LSTM-modeller kan håndtere lange avhengigheter i data og blir brukt mer og mer i økonomiske prognoser.

Målet med dette kompendiet er å være et teori-supplement til forelesningene og Jupyter notatene vi går igjennom i Google Colab, for å knytte teori opp mot praksis. Kurset krever derfor aktiv deltakelse i forelesningene, og egenstudium av kompendiet. *Maskinlæring for økonomer*, har som mål å gi masterstudenter i samfunnsøkonomi et *et innblikk* i maskinlæringsverden, hva det kan brukes til og potensielle anvendelser innen samfunnsøkonomien.

# 2 Grunnleggende matematikk og statistikk

For å kunne gjøre noe som helst innenfor maskinlæring, trenger man noen grunnleggende matematikk og statistikk-kunnskaper.

## 2.1 Nødvendig lineær algebra for maskinlæring

Lineær algebra er grunnleggende for mange maskinlæringsalgoritmer. Her er noen nøkkelkonsepter:

- **Vektorer:** En vektor er en ordnet liste av tall som kan representere datapunkter i et flerdimensjonalt rom. En vektor $\mathbf{x}$ i $n$-dimensjoner kan skrives som:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

  hvor $x_i$ er komponentene i vektoren. Vektorer brukes til å representere egenskaper ved datapunkter og utføre beregninger.

- **Matriser:** En matrise er et rektangulært array av tall og kan brukes til å representere datasett og transformasjoner. En matrise $\mathbf{A}$ av dimensjoner $m \times n$ kan skrives som:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

  Matriseoperasjoner som addisjon, subtraksjon og multiplikasjon er viktige for å kombinere og transformere data.

- **Normer:** Normen til en vektor $\mathbf{x}$ er en måling av dens lengde. To vanlige normer er L2-norm (euklidisk norm) og L1-norm:

$$L2\text{-norm: } ||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

$$L1\text{-norm: } ||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i|$$

  Normer brukes i optimalisering for å evaluere og minimere tap. Dette vil vi komme mer tilbake til senere i kompendiumet.

### 2.1.1 Operasjoner med vektorer og matriser

I maskinlæring og datavitenskap er vektorer og matriser grunnleggende byggesteiner. De brukes til å representere data, modeller og beregninger. Her vil vi se nærmere på noen viktige operasjoner som involverer vektorer og matriser.

**Transponering** Transponering av en matrise er en operasjon som bytter plass på radene og kolonnene. Hvis vi har en matrise $A$ definert som:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

da er den transponerte matrisen $A^T$:

$$A^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{pmatrix}$$

Transponering er nyttig i mange sammenhenger, blant annet i beregningene av dot-produktet mellom vektorer og i tilpassingen av data for videre analyser.

**Skalar multiplikasjon** Skalar multiplikasjon er en operasjon der en matrise eller vektor multipliseres med et tall (skalar). Hvis vi har en vektor $x$ definert som:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

og en skalar $k$, så er den resulterende vektoren fra skalar multiplikasjon $k \cdot x$:

$$k \cdot x = \begin{pmatrix} k \cdot x_1 \\ k \cdot x_2 \\ \vdots \\ k \cdot x_n \end{pmatrix}$$

Skalar multiplikasjon brukes ofte til å justere vektene i maskinlæringsmodeller. For eksempel kan det brukes til å justere læringshastigheten i algoritmer som gradient descent, hvor vi oppdaterer vektene i forhold til den gradienten av tapsfunksjonen.

**Matrisemultiplikasjon** Matrisemultiplikasjon er en annen viktig operasjon i maskinlæring. Gitt to matriser $A$ og $B$, der $A$ er en $m \times n$ matrise og $B$ er en $n \times p$ matrise, kan produktet $C = AB$ være definert som:

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Resultatet $C$ vil da være en $m \times p$ matrise. Matrisemultiplikasjon er ofte brukt til å kombinere funksjoner og data i nevrale nettverk. For eksempel kan vi bruke matriser til å representere vektene i et nevralt nettverk, hvor inngangsdataene multipliseres med vektmatrisen for å produsere aktiveringsverdier i det skjulte laget.

**Eksempel:** La oss se på et konkret eksempel med to matriser:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

Her er $A$ en $2 \times 2$ matrise og $B$ er også en $2 \times 2$ matrise. For å finne produktet $C = AB$, bruker vi formelen for matrisemultiplikasjon:

$$C_{11} = (1 \cdot 5) + (2 \cdot 7) = 5 + 14 = 19$$

$$C_{12} = (1 \cdot 6) + (2 \cdot 8) = 6 + 16 = 22$$

$$C_{21} = (3 \cdot 5) + (4 \cdot 7) = 15 + 28 = 43$$

$$C_{22} = (3 \cdot 6) + (4 \cdot 8) = 18 + 32 = 50$$

Dermed får vi produktmatrisen $C$:

$$C = AB = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

I nevrale nettverk kan vi tenke oss at matrise $A$ representerer inputverdiene (for eksempel egenskaper ved et datasett), og matrise $B$ representerer vektene som er lært gjennom treningsprosessen. Produktet $C$ gir oss de aktiverte verdiene som deretter kan brukes i neste lag av nettverket. Dette vil vi komme mer tilbake til i detalj senere i kompendiet.

**Hvorfor lagre data som vektorer og matriser?** Data lagres ofte som vektorer og matriser fordi disse formatene gjør det lettere å utføre matematiske operasjoner og transformasjoner. Vektorer kan representere funksjoner, egenskaper eller observasjoner av datapunkter, mens matriser kan brukes til

å representere grupper av slike datapunkter. Denne strukturen tillater effektiv utnyttelse av lineær algebra, noe som er grunnleggende for algoritmer innen maskinlæring og dyp læring. Videre er mange biblioteker og rammeverk, inkludert TensorFlow, optimalisert for beregninger med matriser og tensorer, noe som gjør databehandlingen raskere og mer effektiv. Lagres data i et vektor- eller matriseformat, muliggjør det parallellbehandling, noe som er essensielt for å håndtere store datamengder og komplekse modeller i maskinlæring [Hull, 2021, s. 20-21].

### 2.1.2 Tensorer og matriser

En **tensor** er en generell betegnelse på en datastruktur som brukes mye innen maskinlæring og numerisk databehandling. Vi kan se på en tensor som en utvidelse av skalarer, vektorer og matriser til høyere dimensjoner. Tensorer gir en fleksibel måte å representere data med flere dimensjoner, som er nyttig i nevrale nettverk og bildebehandling. Vi baserer oss her på en definisjon fra maskinlæringslitteraturen [Goodfellow et al., 2016]:

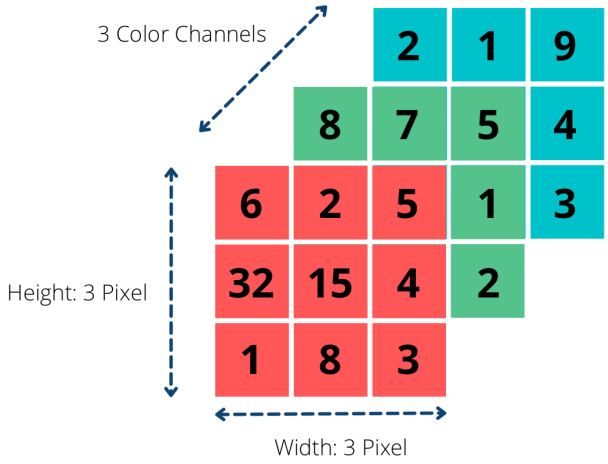
> "In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor."

**Typer av tensorer** En tensor kan betraktes som en generalisering av matriser og vektorer til høyere dimensjoner. Dette er en grunnleggende datastruktur i maskinlæring, spesielt i rammeverk som PyTorch og TensorFlow. Tensorer deles ofte inn etter rang (rank), som beskriver antallet dimensjoner:

- **Skalar** (rank-0 tensor): En enkeltverdi, som et tall 3 eller 7.5. Representert som $\mathbb{R}$.
- **Vektor** (rank-1 tensor): En rekke med verdier, for eksempel en liste som $[1, 3, 5, 7]^T$. Representert som $\mathbb{R}^n$.
- **Matrise** (rank-2 tensor): En tabell med verdier, for eksempel en $\mathbb{R}^3 \times \mathbb{R}^3$ matrise:
$$\begin{bmatrix} 3 & 5 & 7 \\ 2 & 6 & 1 \\ 8 & 4 & 9 \end{bmatrix}$$
Representert som $\mathbb{R}^n \times \mathbb{R}^m$.
- **Rank-3 tensor** og høyere: En flerdimensjonal generalisering av matriser. For eksempel kan en 3D-tensor tenkes som en "kubelignende" struktur med flere matriser stablet oppå hverandre. Representert som $\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p$.

**Når brukes høyere rang (rank) tensorer?**

- Et gråskalabilde (med én kanal) representeres vanligvis som en rank-2 tensor, altså en 2D-matrise av pikselverdier der dimensjonene er høyde × bredde.
- Et fargebilde, som har tre kanaler (rød, grønn og blå), representeres som en rank-3 tensor der dimensjonene er høyde × bredde × kanaler. Se Figur 1




Figur 1: Illustrerer en rank-3 tensor, som ofte er et fargebilde bestående av rød, grønn og blå [Lang, 2022].

Det viktigste å forstå er at når vi snakker om en *tensor* i maskinlæring, refererer **rangen** til antall dimensjoner, og dette må ikke forveksles med dimensjonene til selve matrisen!

## 2.2 Nødvendig kalkulus for maskinlæring

Kalkulus er avgjørende for å forstå hvordan maskinlæringsmodeller blir trent og optimalisert. Viktige konsepter inkluderer:

- **Derivasjon:** Derivasjon er en metode for å finne hastigheten til endring av en funksjon. For en funksjon $f(x)$er den deriverte definert som:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

  Derivasjon brukes i gradient descent-algoritmen for å finne minimum av en tapsfunksjon [Hull, 2021, s. 39].

- **Gradienten:** Gradienter er vektorer av partielle deriverte som viser retningen for den bratteste stigningen av en funksjon. For en funksjon $f(\mathbf{x})$ med flere variabler, hvor $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$, er gradienten:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

  Gradient descent-algoritmen oppdaterer modellparametre i retningen av den negative gradienten for å minimere tapsfunksjonen:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \eta \cdot \nabla f(\mathbf{x}_i)$$

  hvor $\eta$ er læringsraten, og $x_i$ er modellparameter, og $x_{i+1}$ er den oppdaterte modellparameteren.

**Eksempel på beregning av gradienten:** La oss si vi har en funksjon:

$$f(x_1, x_2) = x_1^2 + 3x_2^2$$

For å finne gradienten, beregner vi de partielle derivatene:

$$\frac{\partial f}{\partial x_1} = 2x_1$$

$$\frac{\partial f}{\partial x_2} = 6x_2$$

Dermed er gradienten:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} 2x_1 \\ 6x_2 \end{pmatrix}$$

**Oppdatering med gradient descent:** Anta at vi starter med initialverdiene $x_1 = 1$ og $x_2 = 2$ og bruker en læringsrate på $\eta = 0.1$. Vi kan beregne gradienten:

$$\nabla f(1, 2) = \begin{pmatrix} 2 \cdot 1 \\ 6 \cdot 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 12 \end{pmatrix}$$

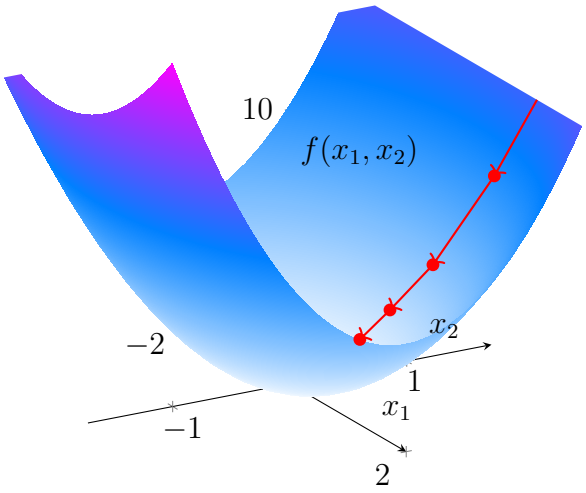
Nå oppdaterer vi parameterne:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}_{i+1} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}_i - 0.1 \cdot \begin{pmatrix} 2 \\ 12 \end{pmatrix}$$

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}_{i+1} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} - \begin{pmatrix} 0.2 \\ 1.2 \end{pmatrix} = \begin{pmatrix} 0.8 \\ 0.8 \end{pmatrix}$$

Så etter én oppdatering har vi fått $x_1 = 0.8$ og $x_2 = 0.8$. Denne prosessen kan gjentas flere ganger til vi konvergerer mot minimum av funksjonen. Denne funksjonen kan være hva vi velger den til å være, ofte bruker vi det vi kaller for en *tapsfunksjon* eller *kostfunksjon*, men det vil vi se nærmere på etter hvert.

Likevel, ta en ekstra titt på det vi akkurat har gjort her, for hvis du forstår dette - er vi på god vei mot å forstå noe av det mest fundamentale i maskinlæring og nevrale nettverk.




Figur 2: Illustrerer gradient decent på $f(x_1, x_2) = x_1^2 + 3x_2^2$ med initielle verdier $(x_1, x_2) = (1, 2)$ og vi ser at punktene går mot minimumspunktet $(0, 0)$.

## 2.3 Nødvendig statistikk for maskinlæring

Statistikk gir verktøyene som trengs for å analysere data og trekke konklusjoner. Nøkkelkonsepter inkluderer:

- **Sannsynlighet:** Sannsynlighet er et mål på usikkerhet og kan representeres som et tall mellom 0 og 1. For en hendelse $A$, er sannsynligheten $P(A)$ definert som:

$$P(A) = \frac{\text{Antall gunstige utfall}}{\text{Totalt antall utfall}}$$

  Sannsynlighetsfordelinger, som normalfordelingen, brukes til å modellere data og gjøre prediksjoner.

- Sannsynlighet er essensielt i maskinlæring fordi mange modeller handler om å forutsi usikkerhet og håndtere støy i data. For eksempel når vi skal forutsi en ukjent verdi $Y$ gitt noen observasjoner $X$, kan vi for eksemepel bruke logistisk regresjon, som bygger direkte på sannsynlighetsmodeller for å gi prediksjoner. Videre er begreper som forventningsverdi, varians og fordelingsfunksjoner sentrale for å modellere hvordan data oppfører seg og for å evaluere modellens ytelse.

### 2.3.1 Forventningsverdi og varians

- **Forventningsverdi:** Forventningsverdien (eller gjennomsnittet) av en stokastisk variabel $X$ er et mål på den sentrale tendensen av variabelen. Den er definert som:

$$E[X] = \sum_{i=1}^{n} x_i P(X = x_i) \quad \text{(diskret variabel)}$$

  eller

$$E[X] = \int_{-\infty}^{\infty} x f(x)\, dx \quad \text{(kontinuerlig variabel)}$$

  Forventningsverdien gir et mål på hva som kan være den forventede verdien fra et datasett.

- **Varians:** Variansen $\text{Var}(X)$ måler hvor mye data sprer seg rundt forventningsverdien. Den er definert som:

$$\text{Var}(X) = E[(X - E[X])^2]$$

Variansen kan også uttrykkes som:

$$\text{Var}(X) = E[X^2] - (E[X])^2$$

En lav varians indikerer at dataene er nær forventningsverdien, mens høy varians indikerer større spredning. Varians er kritisk for å forstå over- og underfitting i modeller.

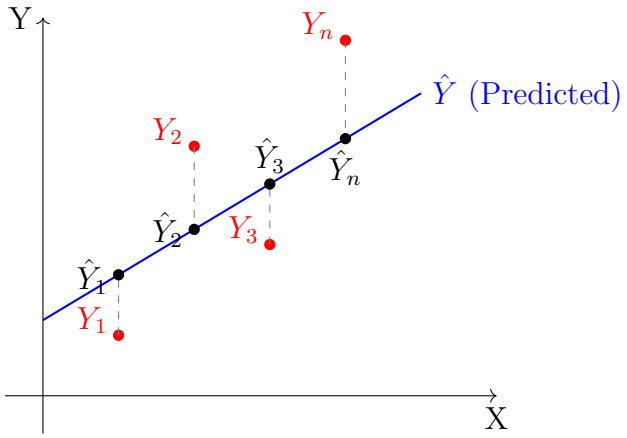
### 2.3.2 Reducible og irreducible error

**Reducible og irreducible error** er to fundamentale konsepter i maskinlæring som beskriver de ulike kildene til feil i en modell. For å forstå disse konseptene bedre, bruker vi matematikk til å forklare dem.

La oss først definere **Mean Squared Error (MSE)**, som er en ofte brukt metrikk for å måle ytelsen til en modell. Hvis en vektor av $n$ prediksjoner er generert fra et utvalg med $n$ datapunkter, der $Y_i$ er den observerte verdien og $\hat{Y}_i$ er den predikerte verdien, kan MSE uttrykkes som:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

MSE representerer gjennomsnittet av kvadrerte forskjeller mellom observerte verdier og predikerte verdier, se Figur 3. Det som er viktig å merke seg her, er at feilene i modellen kan stamme fra to hovedkilder: **reducible error** og **irreducible error** [James et al., 2023, s. 17-18].




Figur 3: Illustrerer $Y_i$ og $\hat{Y}_i$, $i = 1, 2, \ldots, n$ og differansen mellom observasjonene og prediksjonene.

**Prediksjon** er ofte et mål i maskinlæring. Når vi prøver å modellere forholdet mellom den avhengige variabelen $Y$ (den observerte responsen) og uavhengige variabler $X$ (funksjoner), kan vi uttrykke det slik:

$$Y = f(X) + \epsilon$$

hvor $f(X)$ er den "sanne" funksjonen som beskriver forholdet mellom $X$ og $Y$, og $\epsilon$ er feilleddet eller støyen som ikke kan forklares av $X$. Prediksjonen vi gjør fra modellen vår kan uttrykkes som $\hat{Y} = \hat{f}(X)$, hvor $\hat{f}(X)$ er vår tilnærming til den sanne funksjonen $f(X)$.

Et viktig poeng er at det alltid er en viss mengde støy $\epsilon$ i systemet, også kalt **irreducible error**, som vi ikke kan modellere. Dette feilleddet har egenskapene:

$$E[\epsilon] = 0$$

og

$$Var(\epsilon) = \sigma^2$$

Dette betyr at gjennomsnittet av støyen er null, men variansen $\sigma^2$ er positiv, og representerer uforutsigbar variasjon i dataene.

For å forstå hvordan den totale feilen kan splittes i reducible og irreducible komponenter, kan vi se på forventningen til kvadratet av feilen mellom den faktiske verdien $Y$ og den predikerte verdien $\hat{Y}$:

$$E[(Y - \hat{Y})^2] = E[(f(X) + \epsilon - \hat{f}(X))^2]$$

Uttrykket over kan deles opp som:

$$E[(Y - \hat{Y})^2] = E[(f(X) - \hat{f}(X))^2] + E[\epsilon^2]$$

Siden $\epsilon$ er støyen, vet vi at $E[\epsilon^2] = Var(\epsilon)$ (Se teori om varians over, for hvorfor det blir slik). Dermed får vi:

$$E[(Y - \hat{Y})^2] = \underbrace{(f(X) - \hat{f}(X))^2}_{\text{Reducible Error}} + \underbrace{Var(\epsilon)}_{\text{Irreducible Error}}$$

Her ser vi at den totale feilen deles inn i to deler:

$(f(X) - \hat{f}(X))^2$ representerer **reducible error**, som er forskjellen mellom den sanne funksjonen $f(X)$ og vår prediksjon $\hat{f}(X)$. Denne feilen kan reduseres ved å forbedre modellen, for eksempel ved å velge bedre funksjoner, justere hyperparametere eller bruke en bedre egnet modell.

$Var(\epsilon)$ representerer **irreducible error**, som er variansen til støyen $\epsilon$. Denne feilen kan ikke reduseres, da den kommer fra naturlig variasjon i

dataene som ikke kan forklares av modellen. Dette er et ekstremt viktig konsept å ha forståelse for når vi jobber med data og maskinlæring.

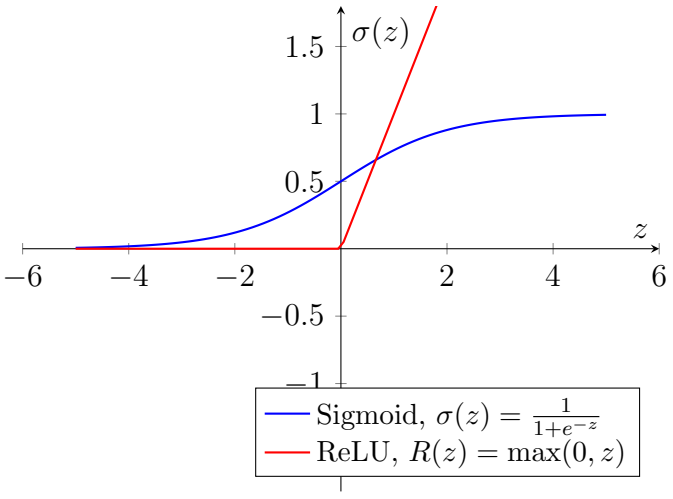
### 2.3.3 Aktiveringsfunksjoner

I nevrale nettverk brukes aktiveringsfunksjoner for å introdusere ikke-linearitet i modellene, slik at de kan lære komplekse mønstre i dataene. Uten en aktiveringsfunksjon vil hvert lag i et nevralt nettverk være en lineær kombinasjon av forrige lag, noe som begrenser modellens evne til å lære ikke-lineære relasjoner.

Det finnes flere typer aktiveringsfunksjoner, og hver har ulike egenskaper som kan være fordelaktige avhengig av oppgaven. Noen av de mest brukte aktiveringsfunksjonene er:

- **Sigmoid-funksjonen**: En S-formet kurve som begrenser utgangsverdiene til å være i spennet $[0, 1]$. Den brukes ofte i utgangslag når vi vil ha en sannsynlighetsverdi mellom 0 og 1, som passer bra for binær klassifisering.
- **ReLU (Rectified Linear Unit)**: En funksjon som lar positive inputverdier passere uendret, men setter negative verdier til null. ReLU er populær i dype nevrale nettverk fordi den gjør det enkelt og raskt å lære, men kan noen ganger føre til "døde" nevroner som slutter å reagere hvis de får for mange nullverdier.

Det er mange aktiveringsfunksjoner man kan velge, men Figur 4 illustrerer grafene til to av de vanligste aktiveringsfunksjonene.




Figur 4: Vanlige aktiveringsfunksjoner: Sigmoid og ReLU.

# 3 Noen maskinlæringsprinsipper

## 3.1 Veiledet og ikke-veiledet læring

Maskinlæring kan i hovedsak deles inn i to hovedkategorier: veiledet læring og ikke-veiledet læring. I tillegg, har vi noe som kalles *reinforcement learning*, men det vil vi ikke gå særlig inn på her.

### 3.1.1 Veiledet læring (supervised learning)

Veiledet læring innebærer at modellen trenes med en datasett som inneholder både inngangsdata (features) og tilhørende korrekte utdata (labels) [James et al., 2023, s. 25]. Målet er at modellen lærer seg å kartlegge en funksjon som kan brukes til å predikere utdataene basert på nye, usette inngangsdata. Et typisk eksempel på veiledet læring er klassifikasjonsproblemer, som når man forsøker å klassifisere blomsterarter basert på fysiske målinger (f.eks. lengde og bredde på kronblad).

En vanlig veiledet læringsmodell er *lineær regresjon*, som forsøker å finne den beste lineære sammenhengen mellom input $X = (x_1, x_2, \ldots, x_n)$ og output $y$. Modellen kan da uttrykkes som:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon$$

hvor $\epsilon$ er støyen eller feilen i modellen, og $\beta_i$ er de ukjente koeffisientene modellen forsøker å estimere.

### 3.1.2 Ikke-veiledet læring (Unsupervised learning)

I ikke-veiledet læring har man kun tilgang til inngangsdataene, og det finnes ingen korrekte utdata eller labels å lære fra. Målet er her å oppdage mønstre eller strukturer i dataene. Et vanlig eksempel er klyngeanalyse (clustering), hvor modellen forsøker å gruppere datapunktene i klynger basert på likhet.

En populær metode innen ikke-veiledet læring er *K-Means Clustering*, hvor målet er å dele et datasett inn i $k$ klynger basert på avstanden mellom datapunkter. K-means algoritmen forsøker å minimere summen av kvadrerte avstander mellom hvert datapunkt og sentrum av sin respektive klynge.

### 3.1.3 Reinforcement learning

*Forsterkende læring* er en annen viktig teknikk innen maskinlæring, hvor en *agent* lærer å ta beslutninger gjennom interaksjon med et miljø. I motsetning til veiledet og ikke-veiledet læring, der modellen lærer fra statiske datasett,

får agenten tilbakemelding i form av belønninger eller straff basert på handlingene den tar. Målet er å maksimere den totale belønningen over tid ved å lære seg en optimal handlingsstrategi.

Selv om forsterkende læring er en kraftfull teknikk, er dette ikke fokus for dette kompendiet, som primært tar for seg **veiledet læring**.

## 3.2 Klassifikasjon versus regresjon

Klassifikasjon og regresjon er to grunnleggende metoder innen maskinlæring, og de har forskjellige mål og metoder for å behandle data.

### 3.2.1 Klassifikasjon

Klassifikasjon refererer til prosessen med å tilordne kategoriske outputdata til inputdata. Målet med klassifikasjon er å forutsi hvilken klasse (eller kategori) et gitt datapunkt tilhører. Dette kan være binær klassifikasjon, hvor det er to klasser, eller flerkategoriklassifikasjon, hvor det er flere klasser.

Eksempler på klassifikasjonsoppgaver inkluderer:

- E-post filtrering (spam eller ikke-spam)
- Diagnose av sykdommer basert på symptomer
- Bildeklassifisering (f.eks. klassifisere bilder som hunder eller katter)

Klassifikasjonsmodeller lærer fra historiske data for å lage en prediksjonsmodell, ofte ved å bruke teknikker som beslutningstrær, nevrale nettverk, logistisk regresjon eller støttevektor-maskiner (SVM). Det handler nemlig om *prediktoren* er *kvantitativ* eller *kvalitativ*.

### 3.2.2 Regresjon

Regresjon, derimot, er en oppgave som handler om å forutsi kontinuerlige verdier. Målet med regresjon er å modellere forholdet mellom en uavhengig variabel (eller flere variabler) og en avhengig variabel (også kalt prediktor). Dette kan være en enkel regresjon, der man forutsier en enkelt utfall basert på en enkelt input, eller multippel regresjon, der flere inputvariabler brukes til å forutsi en enkelt utfall.

Eksempler på regresjonsoppgaver inkluderer:

- Forutsi huspriser basert på størrelse, beliggenhet og andre faktorer
- Forutsi aksjekurser over tid

- Estimere inntekten basert på utdanningsnivå og erfaring

Regresjonsmodeller bruker metoder som lineær regresjon, polynomregresjon eller mer komplekse modeller som nevrale nettverk for å tilpasse seg dataene.

### 3.2.3 Forskjeller mellom klassifikasjon og regresjon

Den primære forskjellen mellom klassifikasjon og regresjon ligger i outputvariabelen som predikeres:

- **Klassifikasjon**: Outputvariabelen er **kategoriske** (f.eks. klasse A, B eller C).
- **Regresjon**: Outputvariabelen er **kontinuerlige verdier** (f.eks. 10.5, 200.75).

Valget mellom klassifikasjon og regresjon avhenger av problemstillingen og typen data som er tilgjengelig. For å oppsummere, klassifikasjon brukes for å tilordne datapunkter til ulike klasser (kategoriske), mens regresjon brukes for å forutsi numeriske verdier basert på inputdata.

## 3.3 Prediksjon versus inferens

I maskinlæring er det viktig å skille mellom prediksjon og inferens.

- **Prediksjon** refererer til prosessen med å forutsi fremtidige utfall basert på tilgjengelige data, altså historiske data vi har samlet inn og gitt *labels*, altså *merket* utfallet. For eksempel, i en regresjonsmodell kan vi bruke tidligere data til å forutsi verdien av en avhengig variabel $Y$ gitt en uavhengig variabel $X$:

  $$Y = f(X) + \epsilon$$

  hvor $f(X)$ er den prediktive funksjonen, og $\epsilon$ representerer støyen i dataene.

- **Inferens** fokuserer derimot på å trekke konklusjoner om de underliggende prosessene som genererer dataene. Det innebærer å evaluere forholdet mellom variabler og kan inkludere estimering av parameterne i modellen. For eksempel, i en lineær regresjonsmodell kan vi bruke metoder som minste kvadraters estimat for å bestemme parameterne:

  $$\hat{\beta} = (X^T X)^{-1} X^T y$$

  hvor $\hat{\beta}$ er parameterestimatene, $X$ er designmatrisen, og $y$ er vektoren av observerte utfall.

Her, vil vi fokusere mest på prediksjon.

## 3.4 Bias-variance tradeoff

Bias-variance tradeoff er et sentralt konsept i maskinlæring som beskriver balansen mellom to typer feil som påvirker modellens ytelse:

- **Bias** refererer til feil som oppstår når en modell forenkler virkeligheten for mye. Dette kan føre til at modellen ikke fanger opp de underliggende mønstrene i dataene, noe som resulterer i *underfitting*. En modell med høy bias kan for eksempel være en lineær modell som forsøker å tilpasse seg data som har en ikke-lineær struktur.

- **Variance** refererer til feil som oppstår når modellen er for kompleks og fanger opp støy i dataene. Dette kan føre til *overfitting*, der modellen tilpasser seg treningsdataene for godt, men har dårlig generaliseringsevne på nye data. En modell med høy varians kan for eksempel være en kompleks nevralt nettverk med mange lag.

Målet er å finne en balanse mellom bias og varians som minimerer den totale feilen:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

## 3.5 Overfitting og underfitting

**Overfitting** oppstår når en modell er for kompleks og tilpasser seg treningsdataene for mye, noe som resulterer i dårlig ytelse på testdata. Dette kan skje når modellen har for mange parametere eller når det er for lite treningsdata tilgjengelig, se Figur 5. For å identifisere overfitting kan vi se på forskjellen mellom trenings- og testfeil; en modell som presterer mye bedre på treningsdataene enn på testdataene er sannsynligvis overfitted.

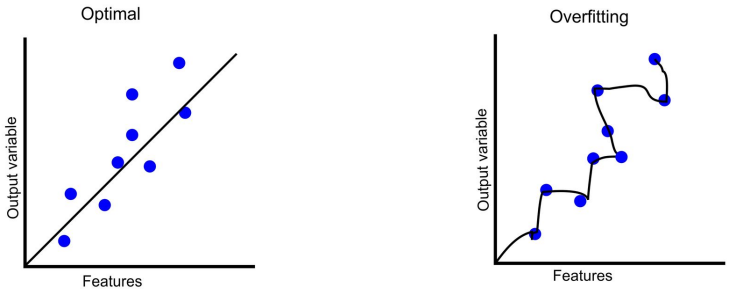
**Underfitting** er det motsatte av overfitting, der modellen er for enkel til å fange opp mønstrene i dataene. Underfitting kan oppstå når man bruker en for enkel modell, som en lineær regresjon på ikke-lineære data, eller når det er utilstrekkelig med trening. Resultatet blir dårlig ytelse både på trenings- og testdata.

For å unngå både overfitting og underfitting, kan man bruke teknikker som:

- Regularisering (f.eks. Lasso, Ridge)

- Kryssvalidering

- Dropout-teknikker




Figur 5: Illustrerer hva en god tilpasning er, og hva overfitting er [Ogbemi, 2021].

- Tidlig stopp under trening av nevrale nettverk
- ... og mange, mange fler

## 3.6 Evaluering av modeller

Evaluering av modeller er avgjørende for å vurdere hvor godt en modell fungerer. Det finnes flere metoder for å evaluere ytelsen til en maskinlæringsmodell.

### 3.6.1 Confusion matrix

Confusion matrix er et verktøy som brukes for å oppsummere hvor god eller dårlig modellen vår er til å klassifisere noe. For et binært klassifiseringsproblem, vil en confusion matrix vise antall sanne positive (TP), sanne negative (TN), falske positive (FP) og falske negative (FN). En typisk 2x2 confusion matrix ser slik ut:

| | Predicted Negative | Predicted Positive |
|---|---|---|
| Actual Negative | $TN$ | $FP$ |
| Actual Positive | $FN$ | $TP$ |

Fra denne confusion matrixen kan vi utlede flere evalueringsmetrikker, som for eksempel:

- **Accuracy**:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision**:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall** (også kjent som Sensitivitet):

$$\text{Recall} = \frac{TP}{TP + FN}$$

Det er mange måter å evaluere en maskinlæringsmodell sin ytelse, og her diskuterer vi bare et fåtall av noen av de mest populære metrikkene. Vit at det finnes mange fler, og hva som egner seg best - jo, det kommer an på dataene dine og problemet du prøver å løse.

### 3.6.2 Accuracy

Accuracy er en av de mest brukte metrikker for å evaluere ytelsen til klassifiseringsmodeller. Den defineres som andelen korrekt klassifiserte tilfeller (både sanne positive og sanne negative) i forhold til det totale antallet tilfeller. Den kan beregnes som følger:

$$\text{Accuracy} = \frac{\text{Antall korrekte prediksjoner}}{\text{Totalt antall prediksjoner}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Selv om accuracy er enkel å forstå, kan den være misvisende i tilfeller der datasettene er ubalanserte. I slike tilfeller kan det være bedre å bruke andre metrikker, som precision og recall.

### 3.6.3 Precision og Recall

**Precision** er et mål på hvor mange av de predikerte positive tilfellene som faktisk er positive. Det er spesielt nyttig i situasjoner der kostnaden for falske positive er høy. Den kan beregnes med formelen:

$$\text{Precision} = \frac{TP}{TP + FP}$$

**Recall** (eller Sensitivitet) er et mål på hvor mange av de faktiske positive tilfellene som ble korrekt identifisert. Det er nyttig i tilfeller der kostnaden for falske negative er høy. Den kan beregnes som:

$$\text{Recall} = \frac{TP}{TP + FN}$$

I praksis er det ofte en avveining mellom precision og recall, kjent som F1-score, som gir en balansert vurdering av begge:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

### 3.6.4 AUC og ROC-kurver

En sentral del av maskinlæring er å kunne evaluere hvor godt en modell presterer, og det finnes flere evalueringsmetoder for dette formålet. Blant de mest brukte evalueringsmetodene for klassifikasjonsmodeller finner vi ROC (Receiver Operating Characteristic) og AUC (Area Under Curve).

- **ROC-kurve**: ROC-kurven er et verktøy for å visualisere ytelsen til en klassifikasjonsmodell, spesielt når målet er å skille mellom to klasser (for eksempel "positiv" og "negativ"). Kurven plotter forholdet mellom *sant positiv rate* (True Positive Rate, TPR) og *falsk positiv rate* (False Positive Rate, FPR) for ulike terskelverdier. TPR representerer andelen korrekte positive prediksjoner blant de faktiske positive, mens FPR representerer andelen feilaktige positive prediksjoner blant de faktiske negative. En ROC-kurve som bøyer seg nært opp mot øvre venstre hjørne (TPR=1, FPR=0) viser en modell med høy prediksjonsevne.
- **AUC-kurve** AUC, eller *Area Under the Curve*, er et numerisk mål for å evaluere modellen basert på ROC-kurven. AUC representerer arealet under ROC-kurven og varierer mellom 0 og 1, der 1 indikerer en perfekt modell, og 0.5 indikerer en modell som ikke presterer bedre enn ren gjetning. En høy AUC-verdi betyr at modellen har god separasjonsevne, altså at den kan skille mellom de to klassene på en pålitelig måte. Generelt sett anses en AUC på 0.7–0.8 som akseptabel, 0.8–0.9 som god, og over 0.9 som utmerket.

ROC-kurven og AUC gir en verdifull innsikt i hvordan modellen presterer på ulike terskelverdier. Ved å analysere ROC-kurven kan man forstå balansen mellom TPR og FPR, og AUC-verdi gir et intuitivt mål på totalytelsen. Disse evalueringsmetodene er spesielt nyttige i situasjoner med skjeve datasett (f.eks. hvor det er mange flere negative enn positive eksempler), da de gir mer innsikt enn enkle nøyaktighetsmål alene.

### 3.6.5 R-Squared

$R^2$, eller **R-squared** (ofte kalt **forklaringsgraden** på norsk), er en statistisk målestørrelse som brukes til å vurdere hvor godt en regresjonsmodell forklarer variasjonen i et datasett. $R^2$-verdien representerer andelen av den totale variasjonen i den avhengige variabelen som kan forklares av modellen.

**Formel for $R^2$** $R^2$-verdien beregnes vanligvis som følger:

$$R^2 = 1 - \frac{\mathrm{SS}_{\mathrm{res}}}{\mathrm{SS}_{\mathrm{tot}}}$$

hvor:

- $\mathrm{SS}_{\mathrm{res}}$ er summen av kvadrerte residualer, dvs. forskjellen mellom de observerte verdiene og de verdiene som er predikert av modellen.
- $\mathrm{SS}_{\mathrm{tot}}$ er summen av kvadrerte avvik mellom de observerte verdiene og gjennomsnittet av de observerte verdiene.

Det kan altså skrives på følgende måte:

$$R^2 = 1 - \frac{\Sigma_i (y_i - f_i)^2}{\Sigma_i (y_i - \overline{y})^2}$$

**Tolking av $\mathbf{R^2}$**

- $\mathbf{R^2 = 1}$: Modellen forklarer all variasjon i dataene perfekt.
- $\mathbf{R^2 = 0}$: Modellen forklarer ingen av variasjonene i dataene (prediksjonene er like gjennomsnittsverdien til den avhengige variabelen).
- **$\mathbf{R^2}$ mellom 0 og 1**: Modellen forklarer en viss andel av variasjonen i dataene, der høyere verdier indikerer bedre tilpasning.

**Begrensninger ved $\mathbf{R^2}$**

- **Ikke alltid et mål på "goodness of fit" i modellen**: En høy $R^2$-verdi betyr ikke nødvendigvis at modellen er god eller generaliserbar; det kan bety at modellen er overtilpasset.
- **Vanskelig i ikke-lineære modeller**: $R^2$ kan være misvisende for komplekse, ikke-lineære modeller der variasjonen ikke forklares særlig bra av enkle regresjonsmodeller.
- **Kan øke med flere variabler**: Å legge til flere forklaringsvariabler i modellen øker ofte $R^2$-verdien, selv om disse variablene kanskje ikke er relevante.

**Justert $R^2$** For å håndtere noen av begrensningene ved $R^2$ brukes ofte en justert $R^2$-verdi. Denne tar hensyn til antall forklaringsvariabler i forhold til mengden data og straffer for overtilpasning, noe som gir en mer pålitelig evaluering av modellens forklaringskraft når mange variabler er involvert.

$R^2$ er nyttig for å vurdere en regresjonsmodells forklaringsgrad, men bør brukes sammen med andre evalueringsmetoder for en fullstendig vurdering av modellens ytelse.

## 3.7 Tapsfunksjon/Kostfunksjon

I maskinlæring er tapsfunksjonen (eller kostfunksjonen) et mål på hvor godt modellen presterer i forhold til de faktiske utfallene. Den kvantifiserer forskjellen mellom de predikerte verdiene ($\hat{Y}$) fra modellen og de faktiske verdiene ($Y$) i datasettet. En vanlig brukt tapsfunksjon for regresjonsproblemer er den *gjennomsnittlige kvadratiske feilen* (eng: Mean Squared Error, MSE), som defineres som:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Her er $n$ antall datapunkter, $Y_i$ er den faktiske verdien, og $\hat{Y}_i$ er den predikerte verdien. Ved å minimere tapsfunksjonen, prøver vi å forbedre modellens nøyaktighet.

### 3.7.1 Gradient descent

Gradient descent er en optimaliseringsalgoritme som brukes for å minimere tapsfunksjoner. Algoritmen fungerer ved å iterativt justere modellparametere (som vekter og biaser i nevrale nettverk, som vi kommer mer inn på senere) i retning av den negative gradienten av tapsfunksjonen. Dette gjør det mulig å finne det "optimale" punktet der tapsfunksjonen er minimal.

Gradienten av en kostfunksjon $C$ er en vektor som inneholder de partielle deriverte av $C$ med hensyn til hver av parametrene, som vekter $\mathbf{w}$. Matematisk kan gradienten skrives som:

$$\nabla C(\mathbf{w}) = \left(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_k}\right)$$

hvor $\mathbf{w} = (w_1, w_2, \ldots, w_k)$ er vektorene med parametere (ofte vekter, eller weights).

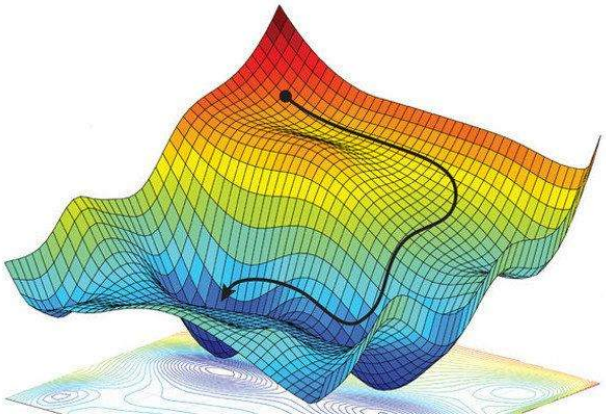
Oppdateringen av parametrene i gradient descent skjer ifølge følgende formel:

$$\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \eta \cdot \nabla C(\mathbf{w}_i)$$

Her er $\eta$ læringsraten, som bestemmer hvor store skritt vi tar i retning av gradienten. Å velge en passende læringsrate er viktig, da en for høy rate kan føre til *overshooting*, mens en for lav rate kan føre til treningsprosessen tar lang tid.

Å minimere tapsfunksjonen er essensen i maskinlæring, spesielt i nevrale nettverk der vi har mange vekter og biaser. Ved å bruke gradient descent til å oppdatere disse parametrene kan vi lære komplekse mønstre i dataene og dermed fange opp sammenhenger vi mennesker aldri kunne funnet!

**Stochastic gradient decent** Stokastisk Gradient Descent (SGD) er en optimaliseringsmetode som ofte brukes i maskinlæring for å trene modeller, spesielt nevrale nettverk. I motsetning til tradisjonell gradient descent, som oppdaterer vektene basert på hele treningsdatasettet, oppdaterer SGD vektene ved å bruke bare ett enkelt eksempel (eller en liten batch) om gangen - og denne velges ut tilfeldig.



Figur 6: Illustrerer stokastisk gradient decent, hvor vi oppdaterer vektene våre for å konvergere mot et minimum [Mishra, 2023].

Prosessen kan beskrives i følgende trinn:

1. For hvert treningspunkt $(\mathbf{x}, y)$, beregnes gradienten av tapsfunksjonen $L$ med hensyn til vektene $\mathbf{w}$:

$$\nabla L(\mathbf{w}) = \frac{\partial L}{\partial \mathbf{w}}$$

2. Vektene oppdateres ved hjelp av gradienten:

$$\mathbf{w} = \mathbf{w} - \eta \cdot \nabla L(\mathbf{w})$$

   hvor $\eta$ er læringsraten, som bestemmer størrelsen på oppdateringene.

3. Prosessen gjentas for alle treningspunkter, noe som gjør at modellen kan lære gradvis og tilpasse seg dataene.

Dermed skiller stokastisk gradient decent (SDG) seg fra *vanlig* gradient decent. Fordi vanlig gradient decent, så oppdaterer modellen vektene sine basert på gradienten regnet ut av HELE treningssettet - dette krever mye datakraft. Mens SDG oppdaterer vektene sine ved å bruke kun én (eller en liten *batch*) av treningssettet av gangen. Dette betyr at hver oppdatering av vektene er basert på en tilfeldig valgt datapunkt fra datasettet. Dette introduserer variasjon i oppdateringen, og kan forhindre å ende opp i *lokale minimum*. Oftest fører dette til raksere konvergering mot minimum, men kan også ha noen implikasjoner. Men dette er et eget fagstoff i seg selv, og et sted må vi stoppe.

**Fordeler** Stokastisk Gradient Descent har flere fordeler:

- **Effektivitet:** Den krever mindre minne og kan håndtere store datasett raskere enn tradisjonell gradient descent.
- **Regularisering:** Den stokastiske biten av oppdateringene kan bidra til å unngå overfitting ved å introdusere "støy" i læringsprosessen.

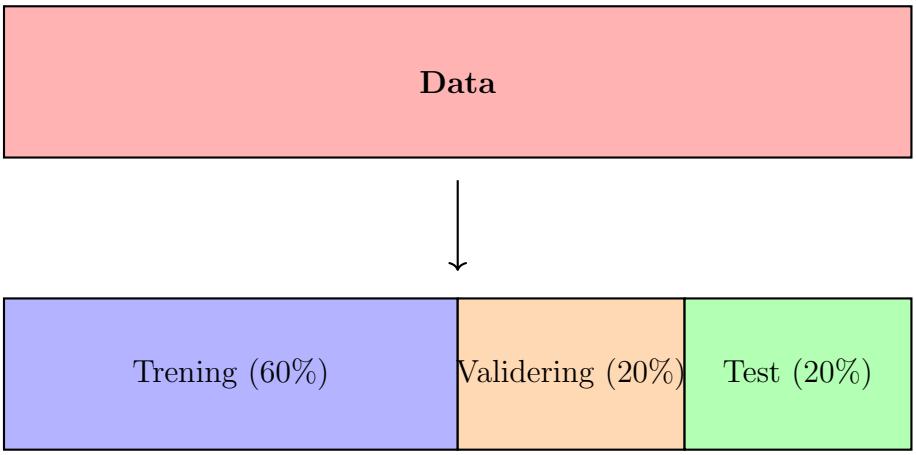
## 3.8 Trenings, validerings og test-sett

Når vi arbeider med maskinlæring, er det viktig å dele dataene våre opp i forskjellige sett for å evaluere modellens ytelse på en pålitelig måte. De vanligste måtene å splitte dataene på er trening og testdata. Treningdata brukes til å trene modellen, mens testdata brukes til å evaluere hvor godt modellen generaliserer til nye, usette data.

Noen ganger er det også fordelaktig å inkludere et valideringssett i tillegg til trening og testdata, se Figur 7. Valideringssettet brukes til å tune hyperparametrene i modellen og gir en indikasjon på hvordan modellen vil prestere på testdataene. Ved å bruke et valideringssett, kan vi redusere risikoen for *overtilpasning* (overfitting), som kan oppstå når modellen tilpasser seg treningdataene for godt og dermed mister evnen til å generalisere.

Treningsprosessen, er den mest krevende prosessen i maskinlæring, og krever da oftest mest data. I Figur 7 er det satt til 60%, mens validering

og test sett er satt til 20% - dette er dog ingen fasit, og opp til brukeren å bestemme disse andelene.




Figur 7: Illustrasjon av hvordan vi kan dele opp data inn i trening, validering og test sett.

# 4 Noen "enkle" maskinlæringsteknikker

I kurset, vil vi fokusere mest på dyplæringsteknikker. Likevel, er det nødvendig å ha en forståelse av enklere maskinlæringsteknikker. Dette er teknikker som ofte kan være svært robuste, da det mest kompliserte ikke nødvendigvis er det beste!

De teknikkene vi kommer til å nevne nå, er mulige teknikker dere kan ta i bruk i kursets prosjektoppgave.

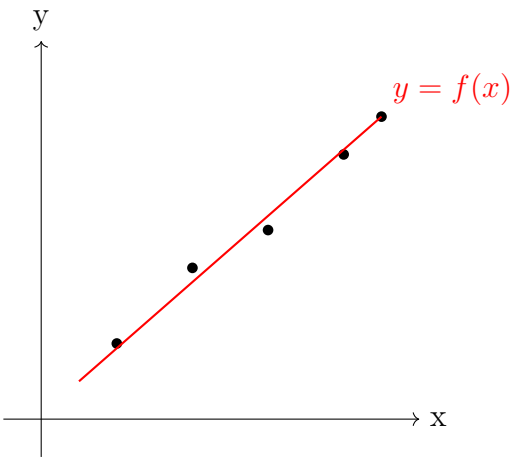
## 4.1 Lineær regresjon

Lineær regresjon er en av de mest grunnleggende og enkle teknikkene innen maskinlæring. Den brukes til å modellere forholdet mellom en uavhengig variabel $x$ og en avhengig variabel $y$ ved å tilpasse en rett linje til dataene. Dette egner seg når den avhengige variabelen er en numerisk verdi, og ikke kategori (da bør man bruke en klassifiseringsmetode). Den matematiske formelen for lineær regresjon er:

$$y = \beta_0 + \beta_1 x + \epsilon$$

hvor:

- $y$ er den avhengige variabelen.
- $x$ er den uavhengige variabelen.
- $\beta_0$ er konstantleddet (intercept).
- $\beta_1$ er stigningstallet (koeffisienten).
- $\epsilon$ er feilleddet.




Figur 8: Illustrasjon av lineær regresjon. De svarte punktene representerer datapunktene, mens den røde linjen viser den beste tilpassede regresjonslinjen $y = f(x)$.

Figur 8 illustrerer hvordan enkel lineær regresjon (eng: *Simple Linear Regression*) fungerer. Den viser en rekke datapunkter, og finner den beste lineære sammenhengen mellom avhengig variabel og den uavhengige variabelen. Når vi har flere uavhengige variabler snakker vi om *multippel lineær regresjon*.

Lineær regresjon er en enkel, men kraftig teknikk for prediksjon og analyse. Den er mye brukt i mange felt, fra økonomi til ingeniørvitenskap, og danner grunnlaget for mer komplekse maskinlæringsmetoder. Likevel, hender det ofte at "det enkle er ofte det beste" - og prøve en lineær regresjon før andre læringsteknikker er en fin trening i forståelse av datasettet man skal prøve å gjøre prediksjoner med.

## 4.2 Logistisk regresjon

Logistisk regresjon er en statistisk metode som brukes for å modellere sannsynligheten for at en bestemt klasse eller hendelse inntreffer, spesielt i tilfeller der utfallsvariabelen er kategorisk (f.eks. ja/nei, 0/1). Den er en type veiledet læring som brukes i klassifiseringsoppgaver.

### 4.2.1 Teoretisk grunnlag

Logistisk regresjon bygger på prinsippet om å bruke en logistisk funksjon (også kjent som sigmoid-funksjon) for å transformere lineære kombinasjoner av inngangsvariabler til en sannsynlighet. Den logistiske funksjonen er definert som:

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

hvor $z$ er en lineær kombinasjon av inngangsvariablene, gitt ved:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Her representerer $\beta_0$ interceptet (konstantleddet) og $\beta_1, \beta_2, \ldots, \beta_n$ er koeffisientene for de respektive funksjonene $x_1, x_2, \ldots, x_n$.

Den predikerte sannsynligheten for at den avhengige variabelen $Y$ er lik 1, gitt $X$, kan uttrykkes som:

$$P(Y = 1|X) = \sigma(z)$$

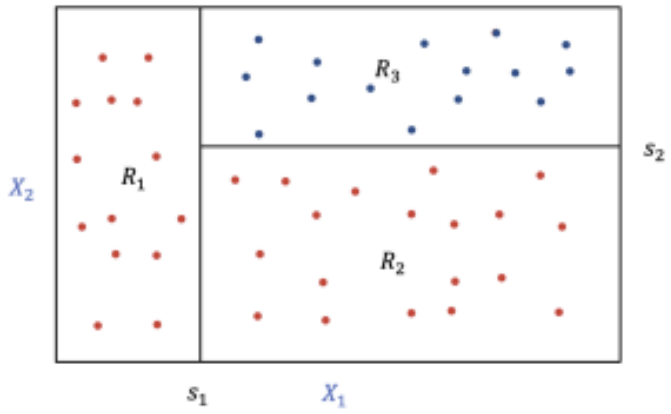
For å forutsi klassen til et datapunkt, bruker vi en terskelverdi (ofte 0.5). Hvis $P(Y = 1|X) \geq 0.5$, klassifiseres datapunktet som tilhørende klasse 1; ellers klassifiseres det som tilhørende klasse 0.

## 4.3 Beslutningstrær

Beslutningstrær er en populær metode innen veiledet læring for både klassifikasjon og regresjon. De fungerer ved å dele opp data i grupper basert på en rekke regler som ligner på menneskelige avgjørelser. Beslutningstrær gir en intuitiv og lettfattelig modell som kan visualiseres, noe som gjør det enklere å forstå hvordan modellen tar avgjørelser. Besultningstrær baserer seg på få antakelser sammenlignet med andre maskinlæringsteknikker, og produserer ofte gode resultat. Spesielt når vi kombinerer de med andre statistiske konsepter, som bootstrapping [James et al., 2023].

### 4.3.1 Splitting av feature Space

Beslutningstrær baserer seg på en teknikk hvor man splitter opp *feature space* i ikke-overlappende regioner. Hvis vi tar en én-dimensjonal vektor med to prediktorer $\mathbf{X} = (X_1, X_2)$, og vi deler opp feature-space inn i $R_1, R_2, \ldots, R_m$ distinkte, ikke-overlappende regioner, se Figur 9. For hver observasjon $X_i$ som faller innenfor en spesfikk region $R_j$ gjør vi *samme* prediksjon for den $X_i$ verdien.




Figur 9: Illustrasjon av feature space.[Aase, 2022].

Det er alt for mange mulige oppdeling av feature space, til at vi kan vurdere alle ulike versjoner. Det ville rett og slett vært for *beregningskrevende*. Én løsning er det som på engelsk kalles *recursive binary splitting*. Dette betyr at den første *splitten* i treet lages ved punkt $s_1$, se Figur 9. Dette lager en *root node*, hvor responsen er enten større eller lik til verdien ved splittelse, altså $s_1$, eller leavere. Da har vi delt opp feature space inn i $R_1 = [X|X_1 < s_1]$ og $R_2 = [X|X_1 \geq s_1]$. Denne $s_1$ er valgt i henhold til et *splitte-kriterie*, som igjen kommer an på om vi har regresjon eller klassifikasjonsproblem foran oss.

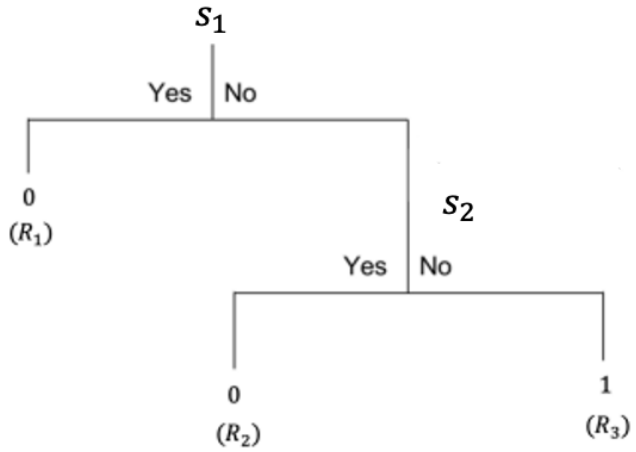
Denne prosessen fortsetter rekursivt ved å splitte partisjonene $R_j$ inn i flere og flere regioner, $j = 1, 2, \ldots$. Hver $R_j$ representerer en *terminal node*, som gir en prediksjon. I Figur 9 er $R_1$ og $R_2$ de røde punktene, mens $R_3$ er de blå punktene.

#### Eksempel

Hvis vi ser på figuren over, kan de røde punktene representere "ikke-svindel", og de blå punktene representerer "svindel". Da kan $X_1$ være hvor mye penger

som er i en transaksjon i Euro. Så si at $s_1 = 4000$ €, da betyr det alt alt til venstre for den vertikale linjen ved $s_1$ betyr "ikke-svindel", som vi kan si er 0. Også kan $X_2$ være antall transaksjoner siste måned, og hvis $s_2 = 3$, vil det betyr at regionen $R_3$ er området hvor transaksjonen har vært **over** 4000€, og denne personen har gjort **flere** enn 3 transaksjoner denne måneden, og dette tilsvarer verdien 1.

Dette kan vi visualisere enklere med et beslutningstre, se Figur 10.




Figur 10: Illustrasjon av et beslutningstre.[Aase, 2022].

**Viktig**

Her gir vi en kort oversikt av hva beslutningstrær er for noe. Det finnes mer teori bak disse læringsteknikkene, og ting er ulikt for regresjon- og klassifiseringsproblemer. Noen stikkord, for den interesserte er:

- Gini index, cross-entropy, deviance
- Cost complexity pruning
- Cross validation
- RSS

### 4.3.2 Pruning av beslutningstrær

Pruning (beskjæring) er en teknikk som brukes for å redusere størrelsen på et beslutningstre ved å fjerne noder som ikke bidrar betydelig til modellens ytelse. Dette er viktig fordi:

1. **Reduksjon av Overfitting**: Store trær kan overtilpasse treningsdataene, noe som resulterer i dårlig generalisering til nye, usette testdata.

2. **Forbedret Modellkompleksitet**: Enkle modeller er ofte lettere å tolke og kan forbedre beslutningstaking i praksis. Et tre med hundrevis av beslutninger er vanskeligere å tolke, enn for eksempel 10.

Pruning kan ses på som en form for regularisering, da det bidrar til å kontrollere kompleksiteten i modellen. Regulariseringsteknikker er avgjørende i maskinlæring for å unngå overtilpasning.

Dette kan vi heldigvis løse på mange måter i Python, som for eksempel

- **Maksimal dybde (max_depth)**: Setter en grense for hvor dypt treet kan vokse. Dette er en av de enkleste regulariseringsmetodene for beslutningstrær. Et grunt tre kan forårsake underfitting, mens et veldig dypt tre kan føre til overfitting, så det er avgjørende å finne en balansert dybde av treet sitt - og dette er jobben til den som utvikler modellen!

- **Minimum antall samples for splitting (min_samples_split)**: Denne hyperparameteren sikrer at en node må ha et minimum antall samples før den kan splittes. Dette kan forhindre små splittelser som fanger støy og dermed fører til en dårligere modell.

- **Maksimalt antall features (max_features)**: Ved å sette en grense for antallet features/variabler som vurderes for splitting, kan du legge til en form for feature-regularisering.

- Det er enda flere. Se dokumentasjon i Python for alle alternativer.

### 4.3.3 Random forests

I forrige kapittel skrøt vi av beslutningstrær og dens intuitive, og enkle tolkning. Som dere kanskje har skjønt til nå, handler mye i maskinlæring og valg av teknikker om en *trade-off*. Det er med andre ord, vanskelig å få *pose og sekk*.

Beslutningstrær har en åpenbar svakhet som er at de ofte kan føre til en høy *varians*. Et forklarende eksempel, er at hvis vi bruker to ulike treningssett, fra samme data, kan de to beslutningstrærne være veldig ulik en annen. Denne variansen kan vi redusere ved å kombinere beslutningstrær med en metode vi kaller *bootstrapping*, der vi bygger flere modeller på ulike, tilfeldige utvalg av treningsdata.

Random Forests er en *ensemble-metode* som bygger på dette prinsippet. Her skaper vi en samling av beslutningstrær (en *skog*) ved hjelp av bootstrapping, og avgjørelsen tas basert på flertallsstemmen fra alle trærne i skogen.

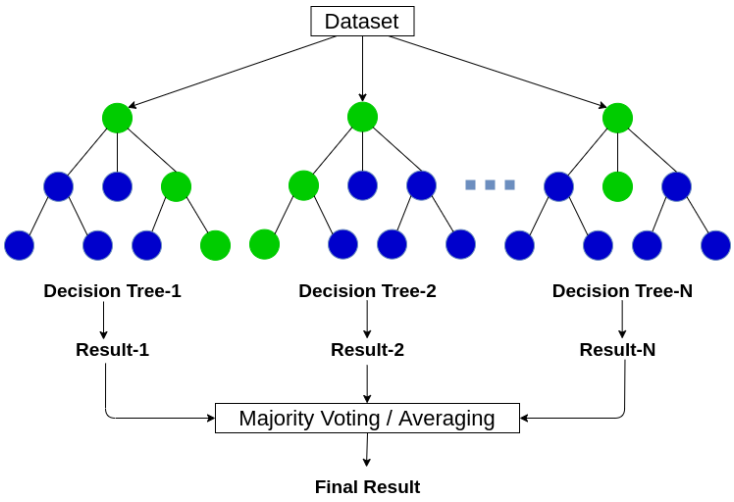
Dette fører til lavere varians og økt robusthet, noe som gjør Random Forests til et effektivt verktøy for både klassifiserings- og regresjonsoppgaver.

**Algorithm 1** Random forest algoritme for klassifiseringstrær

0: Splitt datasettet $\mathbf{X}$ til trening $\mathbf{X}^{\text{train}}$ og test sett $\mathbf{X}^{\text{test}}$.
0: **for** $B$ ganger (antall bootstrap) **do**
0: Bootstrap $\mathbf{X}^{\text{train}} \Rightarrow \mathbf{X}_1^{\text{train}}, \ldots, \mathbf{X}_B^{\text{train}}$
0: **end for**
0: **for** hvert bootstrapped datasett $\mathbf{X}_b^{\text{train}}$ **do**
0: Lag trær basert på $\mathbf{X}_1^{\text{train}}, \ldots, \mathbf{X}_B^{\text{train}} \Rightarrow \hat{f}_1(x), \ldots, \hat{f}_B(x)$ $\{m = \sqrt{p}\}$
0: **end for**
0: Sett sammen beslutningstrærne til $T_B^{\text{RF}} = [\hat{f}_1(x), \ldots, \hat{f}_B(x)]$
0: Prediksjonen/output er *majority vote*, altså det flest trær predikerer. =0

I algoritmen over, er $m = \sqrt{p}$, hvor $m$ er antall prediktorer som vurderes i hver split i treet og $p$ er totalt antall prediktorer. Dette er et vanlig valg, for å bidra til reduksjon av variansen. Hadde vi satt $m = p$ har vi en teknikk som kalles *bagging* [James et al., 2023].

NB: Husk at vi har diskutert **klassifisering** over her, dog selv om regresjon behandles relativt likt, er det ulikheter. Dette dekker vi ikke i kompendiet her, men er nyttig å vite om til prosjektoppgave, masteroppgave eller hvis man kommer over problemstillinger senere - hvor man ønsker å benytte seg av beslutningstrær og *regresjon*.

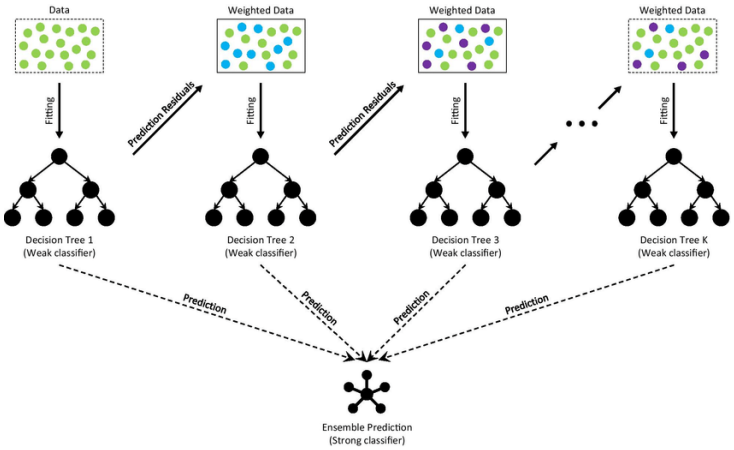



Figur 11: Illustrasjon av random forests.[Curry, 2021].

### 4.3.4 Boosting

Boosting er en statistisk læringsalgoritme som kan anvendes på ulike metoder. Når boosting anvendes på trær, benyttes mange beslutningstrær (klassifikasjon eller regresjon) som kombineres til et mer robust og bedre prediktivt verktøy. På denne måten har boosting et lignende formål som tidligere diskuterte metoder (random forests). Til tross for denne likheten, skiller boosting seg betydelig fra bagging og random forests [Hastie et al., 2009]. I de to sistnevnte metodene vokser trærne uavhengig av hverandre på separate bootstrap-treningsdata (se Figur 11), mens boosting følger en annen tilnærming der trærne bygges sekvensielt, se Figur 12. Med dette menes at hvert tre bruker informasjon fra det forrige treet [James et al., 2023]. Dette er illustrert i Figur 12, der vi harr et initielt tre, og dette oppdateres til at mange "weak classifiers" blir til en "strong classifier".

Vi går ikke inn i detaljer her, men dette er en svært god maskinlæringsteknikk som ofte produserer gode resultater. Ved å bruke boosting, må man også gjøre en del *tuning*, som betyr at man har flere parametere man må justere for å hindre overtilpasning og undertilpasning. I Python, er det veldokumentert hvordan man kan tilpasse ulike parametere som *learning rate*, *antall trær* og *interaction depth*.




Figur 12: Illustrasjon av boosting av beslutningstrær.[Deng et al., 2021].

# 5 Dyplæring

Dyplæring er et omfattende fagfelt innen maskinlæring som bygger på strukturen og funksjonaliteten til nevrale nettverk. Ordet "dyp" refererer til bruken av flere *gjemte lag (hidden layers)* i nevrale nettverk, noe som gjør

dem i stand til å lære komplekse mønstre og representasjoner fra data. Dyplæring er spesielt kraftig for oppgaver som bilde- og talegjenkjenning, samt språkforståelse.

Mens tradisjonelle maskinlæringsmodeller ofte er avhengige av manuell ekstraksjon av funksjoner fra dataene, lærer dyplæringsmodeller disse funksjonene automatisk gjennom lagene i nettverket. Dette gjør dyplæring svært fleksibelt og i stand til å håndtere store mengder ustrukturerte data.

Dyplæring kommer i mange varianter, og vi skal se på noen av de mest anvendte typene:

- *Multiple Layer Perceptron* (MLP): Dette er de "vanlige" nevrale nettverkene, og ligger i grunn for teori om nevrale nettverk, og dette kalles ofte "vanilla" eksempelet av nevrale nettverk, altså det enkleste.
- *Convolutional Neural Networks* (CNNs): Effektive for bilder og visuelle data.
- *Recurrent Neural Networks* (RNNs): Brukes for sekvensielle data som tidssrekker og språk. Vi kommer til å se mest på en type av RNN, kalt **Long Short Term Memory modeller**.
- Språkmodeller: Modeller spesialisert på tekstforståelse og generering, slik som transformer-arkitekturen som brukes i moderne språkmodeller.

## 5.1 Nevrale nettverk

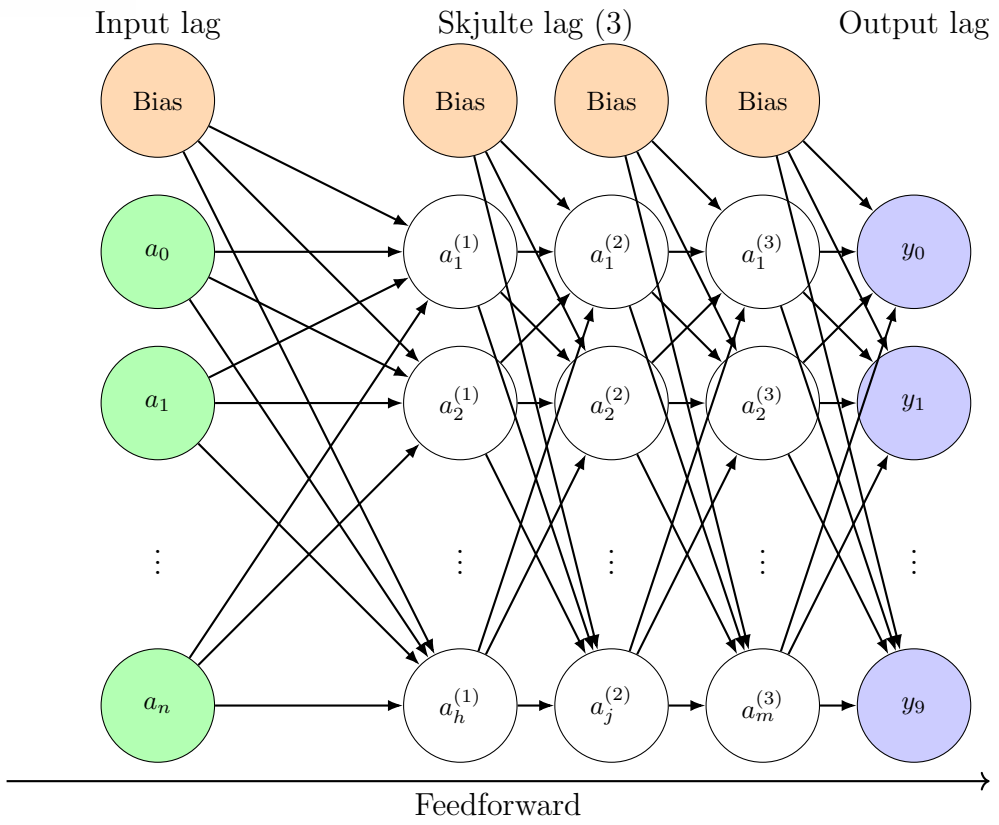
Nevrale nettverk er inspirert av hvordan nevroner fungerer i den menneskelige hjernen. Et nevralt nettverk består av flere lag med "nevroner" eller enheter som hver utfører enkle beregninger og sender resultatene videre til neste lag.

### 5.1.1 Struktur av et nevralt nettverk

Et nevralt nettverk består vanligvis av:

- Et *input layer*, som mottar dataene.
- Et eller flere *hidden layers*, hvor de faktiske beregningene og transformasjonene skjer.
- Et *output layer*, som gir det endelige resultatet av beregningene.

I hvert lag vil nevronene motta inngang fra nevronene i forrige lag, vekte denne inngangen, og sende den videre etter en aktiveringsfunksjon som for eksempel $ReLU$ eller *sigmoid*.




Figur 13: Nevralt nettverk med tre skjulte lag, inkludert inngangs- og utgangslag

## 5.2 Nevrale nettverk

Nevrale nettverk kan sees på som *kjernen* i maskinlæring. I foregående kapitler har vi diskutert teknikker fra de mer *tradisjonelle* læringsteknikkene, som logistisk regresjon, beslutningstrær (og ensemble metoder av disse). La oss nå gå inn i det spennende tema *nevrale nettverk*, hvor vi får bruk for en del av matematikken fra kapittel 2 og 3!

### 5.2.1 Arkitektur og matematikken

Nevrale nettverk er bygd opp av lag, kjent som et *Multi-layer Perceptron* (MLP) når det består av flere lag. Hvert lag består av flere *nevroner*, som er matematiske enheter som transformerer dataen den mottar. Disse enhetene beregner en vektet sum av sine innsignaler og sender videre et utgående signal, basert på en aktiveringsfunksjon. La oss se nærmere på strukturen til nevrale nettverk, som kan deles opp i tre hovedtyper lag:

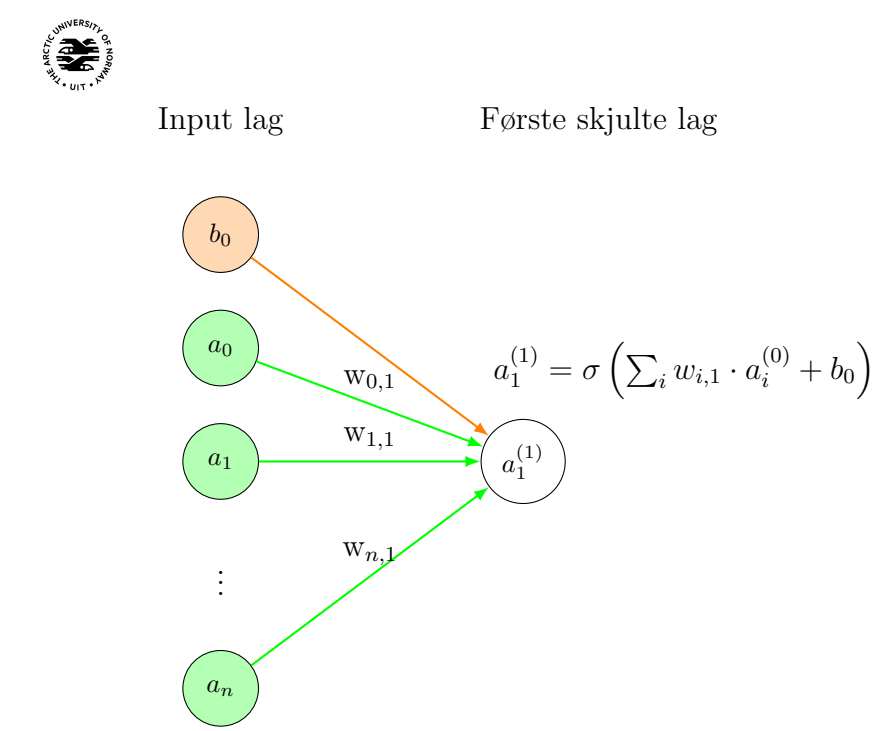
1. **Inngangslag (Input Layer):** Dette første laget mottar dataen direkte. Hvert nevron i inngangslaget representerer én variabel fra datasettet, og dermed tilsvarer antallet nevroner i inngangslaget antallet variabler i datasettet. Inngangslaget utfører ingen beregninger, det er kun et mottakende lag for dataen.

2. **Skjulte lag (Hidden Layers):** Disse lagene er kjernen i nettverket, hvor den faktiske læringen foregår. Skjulte lag består av nevroner som utfører beregninger basert på dataen fra inngangslaget eller forrige skjulte lag. Hvert nevron i et skjult lag mottar input fra alle nevronene i det forrige laget, og resultatet beregnes som en vektet sum av disse innkommende signalene, hvor en konstant bias legges til. Summen sendes så gjennom en *aktiveringsfunksjon*, som for eksempel kan være sigmoid, ReLU eller tanh. Aktiveringsfunksjonen legger til ikkelinearitet i modellen, noe som gir nettverket evnen til å lære komplekse mønstre i dataen.

   En enkel matematisk representasjon for et nevron i skjult lag $l$ kan uttrykkes som:

$$a_j^{(l)} = \sigma\left(\sum_i w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right)$$

   hvor $a_j^{(l)}$ er utgangen (eller aktiveringen) til nevron $j$ i lag $l$, $w_{ij}^{(l)}$ er vekten for forbindelsen fra nevron $i$ i forrige lag $l-1$ til nevron $j$ i lag $l$, $b_j^{(l)}$ er biasen, og $\sigma$ er aktiveringsfunksjonen.




Figur 14: Nevralt nettverk med inngangs- og det første skjulte laget (kun én nevron med vekter), hvor $\sigma$ er en aktiveringsfunksjon.

3. **Utgangslag (Output Layer):** Dette siste laget gir ut nettverkets prediksjon. Antallet nevroner i utgangslaget avhenger av oppgavetypen. For en binær klassifisering vil utgangslaget typisk ha ett nevron som representerer sannsynligheten for én klasse, mens et utgangslag med flere nevroner kan representere forskjellige kategorier i et klassifiseringsproblem med flere klasser.

### Vekter, biaser og alt det der

De to viktigste komponentene som styrer læringen i et nevralt nettverk er *vekter* og *biaser*. La oss gå litt mer detaljert inn på hvordan disse fungerer og hvordan de justeres under treningen.

1. **Vekter (Weights):** Vektene representerer styrken på forbindelsene mellom nevroner i påfølgende lag. I Figur 13, er det en *vekt* mellom hvert eneste nevron fra et lag, til det neste. Figur 14 viser dette, fra input lag til én nevron i det første skjulte laget.

   En høy vekt betyr sterkere forbindelse mellom to nevroner, mens en lav vekt gir svakere forbindelse. Matematisk sett er vektene parametere i en matrise, og de justeres kontinuerlig gjennom læringsprosessen for

å minimere nettverkets kostfunksjon (eller tapsfunksjon). Hver vekt representerer et aspekt av nettverkets forståelse av dataen, og de finjusteres etter hvert som nettverket eksponeres for flere eksempler.

2. **Biaser (Biases):** Bias er en konstant som legges til før inputen går gjennom aktiveringsfunksjonen. Dette gir nettverket mulighet til å justere terskelen for når en node skal bli "aktivert," slik at det kan tilpasse seg mer komplekse mønstre i dataene. Bias gjør at en node i det nevrale nettverket kan forskyve terskelen for aktivering, noe som gir større fleksibilitet. Uten bias ville hver node bare kunne "tenne" når inputverdien passer nøyaktig til vektenes kombinasjon, noe som kan være en begrensning hvis dataene ikke følger denne eksakte sammenhengen. Bias gjør det mulig for nettverket å justere responsen til ulike typer input, selv når de ligger litt utenfor denne grunnlinjen. Det hjelper nettverket å lære mønstre som ikke nødvendigvis starter fra null, men som trenger et lite "dytt" opp eller ned for å fanges opp riktig.

#### "Backpropagation" og oppdatering av vekter og biaser

Når nettverket mottar input og produserer en output, sammenlignes denne outputten med den faktiske verdien for å beregne en *feil*, eller en kost. Denne feilen/kosten brukes deretter til å justere vektene og biasene gjennom en prosess som kalles *bakoverpropagasjon* (eng: *backpropagation*).

Bakoverpropagasjon bruker *kjerneregelen* fra derivasjon for å beregne gradientene til kostfunksjonen med hensyn til alle vekter og biaser. Gradientene representerer hvor mye vektene og biasene bør endres for å redusere feilen. Oppdateringen av en vekt skjer som følger:

$$w_{ij}^{(l+1)} = w_{ij}^{(l)} - \eta \cdot \frac{\partial C}{\partial w_{ij}^{(l)}}$$

der $C$ er kostfunksjonen, og $\eta$ er læringsraten. Læringsraten kontrollerer størrelsen på oppdateringen per iterasjon. Det er viktig å velge en passende læringsrate: en for høy rate kan føre til at vi "hopper over" optimal verdi, mens en for lav rate kan gjøre treningen veldig langsom. Dette har vi diskutert tidligere i kapittel 3.7, se Figur 6.

### 5.2.2 Feed-forward, "fully-connected" nevrale nettverk

Et **feedforward-nettverk** er en type nevralt nettverk der informasjonen strømmer i én retning, fra inngangslaget til utgangslaget, uten noen tilbakemelding (feedback) fra utgangene tilbake til inngangene. Dette betyr at nevronene

i et lag kun påvirker nevronene i neste lag, og det er ingen sykluser i nettverket. Feedforward-nettverk er enkle å implementere og brukes ofte i mange applikasjoner, som klassifisering, regresjon, og mønstergjenkjenning.

Under treningen av et feedforward-nettverk brukes metoder som bakoverpropagering (backpropagation) for å justere vektene og biasene i nettverket. Målet er å minimere feilene, eller minimere *kost-funksjonen*, mellom de predikerte outputene og de faktiske dataene i treningsdataene.

Et **tett lag** (også kjent som et "fullt koblet lag") i et nevralt nettverk er en type lag der hver nevron i laget er koblet til hver nevron i det foregående laget. Dette gjør at informasjon kan strømme fritt mellom lagene, noe som gir nettverket muligheten til å lære komplekse mønstre og sammenhenger i dataene. Figur 13 viser et nettverk som består av tette lag (*Dense layers*.

Hver forbindelse mellom nevronene har en tilknyttet **vekt**, som justeres under treningen av nettverket for å forbedre prediksjonene. Hver nevron har også en **bias**, som er en konstant verdi som legges til før aktiveringsfunksjonen anvendes. Biasen gjør det mulig for nevronene å tilpasse seg bedre til dataene, noe som forbedrer modellens fleksibilitet.

### 5.2.3 Verktøy og biblioteker

I utviklingen av nevrale nettverk er det flere programmeringsspråk og biblioteker som har blitt standardverktøy for forskere og praktikere. De mest populære inkluderer:

- Tensorflow: Utviklet av Google, TensorFlow er et omfattende bibliotek for numeriske beregninger og dyp læring. Det gir høy fleksibilitet og kontroll over modellen, samt muligheter for distribuerte beregninger. TensorFlow inkluderer også Keras, et høyere nivå API som forenkler prosessen med å bygge og trene nevrale nettverk.

- Keras: Et brukervennlig og modulært API som kan brukes med TensorFlow som backend. Det gjør det enkelt å bygge, trene og evaluere nevrale nettverk med lite kode.

- PyTorch er et annet populært bibliotek, utviklet av Meta AI. Det er kjent for sin dynamiske beregningsgraf, som gir mer fleksibilitet ved utvikling av nevrale nettverk, spesielt når det gjelder forskningsformål.

- Scikit-learn: Selv om Scikit-learn ikke er spesifikt designet for nevrale nettverk, tilbyr det enkle modeller og verktøy for forbehandling av data, som kan være nyttige før du trener et nevralt nettverk. Samtidig har det enkle pakker for å splitte data inn i trening, validering og test-data, enkelt å bruke logistisk regresjon, beslutningstrær og så videre.

Dette er for å nevne noen, i kurset kommer vi i hovedsak til å fokusere på bruken av Tensorflow og Keras.

### 5.2.4 Bruksområder i Økonomi

Nevrale nettverk har et bredt spekter av anvendelser innen økonomi og finans. Noen få eksempler på bruksområdene er:

- Risikovurdering: I forsikrings- og finanssektoren brukes nevrale nettverk til å vurdere risiko for investeringer, ved å analysere historiske data, markedsforhold og makroøkonomiske faktorer.
- Svindeldeteksjon: Nevrale nettverk kan brukes for å finne komplekse sammenhenger mellom transaksjoner for å detektere svindel og hvitvasking.
- Bildeklassifisering: Hvis man skal analysere et havneområde (automatisk), kan man bruke nevrale nettverk for bildedeteksjon - for å se om innkommende båter er militære eller ikke-militære.

## 5.3 Whisper-modellen

Whisper-modellen, utviklet av OpenAI, er en automatisk talegjenkjenningsmodell som har vist seg å være svært effektiv i å transkribere og oversette tale fra lydfiler [Radford et al., 2022]. Den er trent på et omfattende datasett (680.000 timer med audio og transkripsjon) som inkluderer et bredt spekter av språk og dialekter, noe som gjør den i stand til å håndtere ulike talestiler og aksenter. Den er overraskende god på norsk, og alle dialekter det medfører.

Arkitekturen til Whisper er basert på transformermodeller, som er kjent for sin evne til å behandle sekvensielle data på en svært effektiv måte. Denne modellen kan transkribere tale i sanntid, noe som gjør den ideell for applikasjoner som videokonferanser, live-oversettelser, intervjuer av kvalitative data og så videre. Den kan også brukes til tale-til-tekst-funksjoner, som lar brukere diktere tekst i stedet for å skrive den.

En av de fremste fordelene med Whisper-modellen er dens robusthet mot bakgrunnsstøy og forstyrrelser, noe som gjør den egnet for bruk i varierte lydmiljøer. Det kommer av at treningsdataene ikke bare er perfekt podkastaudio, men lydfiler skrapet fra internett med bakgrunnstøy.

I tillegg til talegjenkjenning, kan den også oversette mellom flere språk, noe som gir verdifull støtte for kommunikasjon på tvers av språkgrenser. Whisper-modellen representerer et betydelig fremskritt innen automatisk talegjenkjenning, og i dette kurset skal vi få litt *hands on* erfaring med Whisper modellen!

### 5.3.1 Transformer-modell

Vi kommer ikke til å dypdykke i Whisper modellens arkitektur, men det vi vil dekke er at den består av en transformer-modell.

Det er et kraftig form av nevrale nettverk, og det som gjør Whisper modellen unik - er dens *multi-task* format. Det betyr at modellen kan takle flere oppgaver samtidig. Whisper modellen kan altså sjonglere flere ting av gangen, den kan transkribere tale, mens den finner ut hvilket språk det er, markere når enkelt-ord blir sagt og finne ut om det er faktisk tale eller bakgrunnsstøy, samtidig!

I artikkelen til OpenAI, *Robust Speech Recognition via Large-Scale Weak Supervision* [Radford et al., 2022] fant forskerne ut at Whisper-modellen transkriberer så godt som menneskelig nivå! I tillegg, så utkonkurrerte modellen mennesker på noen ting. Likevel, har Whisper noen baksider. Whisper er trent på korte segmenter med tale, og kan slite på times lange foredrag. Men i AI-verden skjer endringer fort, og Whisper har allerede "jobbet rundt" dette problemet, ved at data behandles som biter (eng: *chunks*) av gangen [Murray, 2024].

### 5.3.2 Personvern

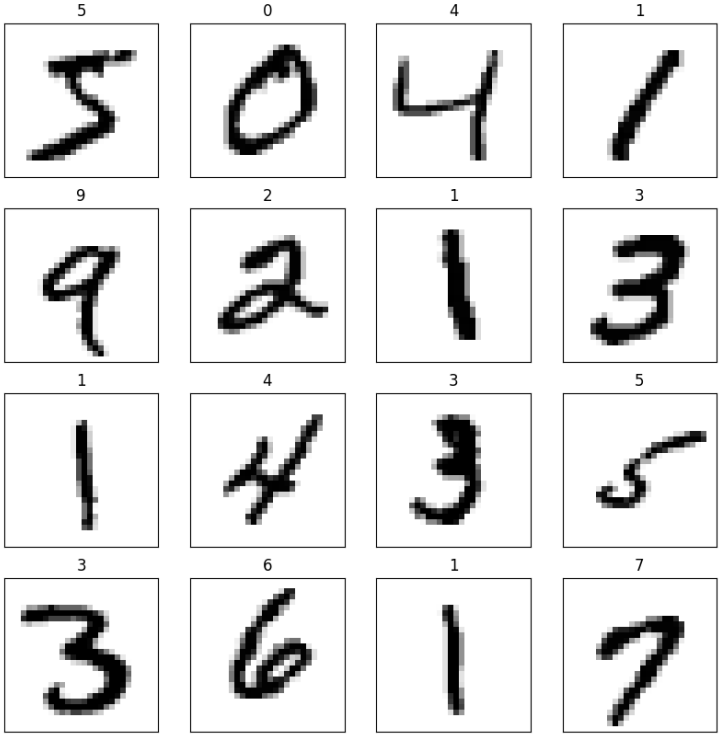
Et problem med modeller på internett som gjør transkribering, er at man må laste opp datamaterialet sitt. Ofte, for eksempel med intervjudata for en masteroppgave, kan data være konfidensielt og GDPR-regler gjør det vanskelig å bruke modeller på nett hvor du laster opp dataene dine.

I dette kurset, vil vi lære hvordan vi kan bruke verktøy som Google Colab, til at filene våre behandles lokalt og aldri forlater Google Colab-området vårt. Bruker man en arbeidsflyt som dette, hvor dataene i tillegg slettes hver gang Colab-området lukkes, kan man bruke transkriberingsverktøy med noen få linjer kode i Python, samtidig som dataene ikke havner utenfor din kontroll.

## 5.4 Bildeklassifisering

Nå har vi sett på nevrale nettverk, ved å diskutere Multi-layer Perceptron. Når vi snakket om bilde-data, bruker vi noe som heter **Convolutional Neural Networks**.

MNIST-datasettet er et populært datasett innen maskinlæring som består av 70 000 håndskrevne sifre (0 til 9), hvor hvert bilde har en størrelse på 28x28 piksler i gråtoner, se Figur 15. Dette datasettet har blitt brukt som en standard for testing av enkle klassifikasjonsmodeller.




Figur 15: Eksempler av håndskrevne tall i MNIST datasettet.

Som vi allerede har sett i de forelesningene, kan en enkel, såkalt "vanilla" multi-layer perceptron (MLP) oppnå ganske gode resultater i å klassifisere disse håndskrevne tallene. MLP-modeller fungerer godt for enkle bilde-datasett som MNIST, ettersom de håndterer bildene som flate vektorer og kan trenes til å gjenkjenne mønstre i pikslene.

Måten vi gjør det på er at bildene består av piksler, i $28 \times 28$. Hvor hver "celle" består av et tall mellom 0 og 255, fordi bildene er lagret i et 8-biters gråskalaformat (sort/hvitt-bilde). I dette formatet betyr:

- **0** representerer ren svart.

- **255** representerer ren hvit.
- Verdier mellom 0 og 255 representerer ulike nyanser av grått, der lavere verdier er mørkere og høyere verdier er lysere.

Når bildene brukes i nevrale nettverk, normaliseres de ofte slik at verdiene ligger mellom 0 og 1. Dette gjøres ved å dele hver pikselverdi på 255. Normalisering kan bidra til at nettverket trener mer effektivt, ettersom det holder input-verdiene innenfor et konsistent skalaområde.

For mer komplekse bildeklassifiseringsoppgaver er det imidlertid nødvendig med modeller som kan utnytte den romlige informasjonen i bildene. Her kommer konvolusjonelle nevrale nettverk (CNN) inn i bildet. I motsetning til MLP bruker CNN spesielle konvolusjonslag som kan fange opp lokale mønstre og teksturer, noe som gjør dem mer egnet til å forstå komplekse bilder der objekter har varierende posisjoner og orienteringer. I de neste kapitlene vil vi se nærmere på hvordan CNN-er gir en betydelig ytelsesforbedring ved å utnytte romlige relasjoner i bildedata.

### 5.4.1 Bilder og deres datastruktur

Bilder, hvertfall digitale bilder, består av et stort antall piksler som hver representerer en liten del av bildet. Hver pixel inneholder fargeinformasjon, og for fargebilder er denne informasjonen vanligvis representert i et RGB-format. RGB står for Rød, Grønn og Blå, som er de tre primærfargene i dette fargesystemet. Hver av disse fargene har en intensitet som vanligvis varierer fra 0 til 255, noe som gir over 16 millioner mulige farger.

Datastrukturen til et digitalt bilde kan forstås gjennom konseptet med tensorer. I denne sammenhengen er et bilde representert som en 3D-tensor, hvor dimensjonene er som følger:

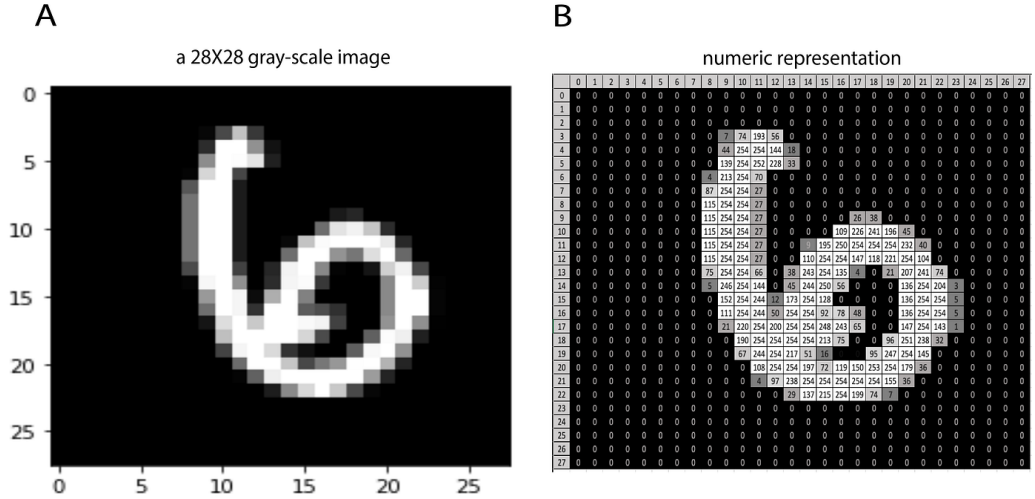
- **Høyde (H)**: Antall piksler i vertikal retning.
- **Bredde (W)**: Antall piksler i horisontal retning.
- **Kanaler (C)**: Antall fargekanaler (for et RGB-bilde er dette 3).

Derfor kan et digitalt bilde med høyde $H$, bredde $W$ og 3 fargekanaler representeres som en tensor med dimensjonene $H \times W \times C$.

### 5.4.2 Convolutional Neural Networks

*Konvolusjonelle nevrale nettverk*, ofte forkortet CNN, er en nettverksarkitektur for dyp læring, og brukes når vi diskuterer bilde-data. Et CNN består av flere lag, derav *dyplæring*, som prosesserer og transformerer et input (som et bilde), og kommer ut i andre enden med et output. Disse CNN'ene kan brukes til bildreprosessering, deteksjon av objekter, men kan også brukes til prediksjon av tekst og lyd.

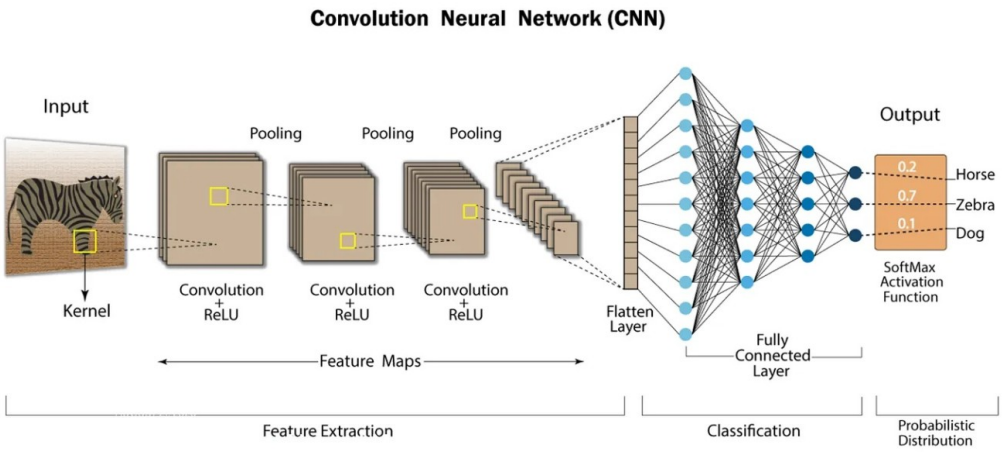
Vi har diskutert fargebilder, som er en rank-3 tensor (3 matriser "oppå hverandre"). Hvis vi ser for oss et grå-skala bilde (rank-2 tensor, med én matrise), se Figur 16. Dette bildet består av 28x28 piksler, hvor hver piksel er representert med 8 bits. Hvor da pikselene kan innta verdier fra 0(sort) til 255 (hvitt).




Figur 16: Illustrasjon av gråskala-bilde [Zammataro, 2022].

CNN får navnet sitt fra den matematiske operasjonen, *konvolusjon*. I CNN, implementeres dette gjennom hva vi kaller på engelsk *feature detector*, *filter* eller en *kernel* - som alle representerer samme konsept. Figur 17 viser et bilde av en zebra, og en kernel. Denne kernel'en, kan sees på som en matrise (som er mindre enn input bildet), som "beveger" seg over bilde - fra øverst til venstre, og jobber seg mot høyre over bildet og deretter går den ned et hakk. Dette gjøres over hele bilde, som gjør at vi får en ny representasjon av input-bildet, et *feature map* av bildet. Hvordan vi spesifiserer denne kernelen, vil bestemme om den kan oppfatte ulike strukturer i bilde, som gjør at datamaskinen kan gjenkjenne hva bilde er for noe. F.eks. kan kernel gjør at vi gjenkjenner ulike kanter, sirkler i tall - som finner hvilke karakteristikk som er typisk for f.eks. tallet 6. Hvor hvert konvulusjonslag, så implementeres

ofte en aktiveringsfunksjon (for å få med ikke-lineære sammenhenger), f.eks. ReLu funksjonen.




Figur 17: Illustrasjon av CNN.

Etter et konvulsjonslag, må man implementere et *pooling*-lag. Det brukes til å *down-sample*, altså nedskalere, feature-map'et vi fikk av konvolusjonslaget. Målet her, er at pooling gjør at vi kvitter oss med det som ikke er "viktig" i bilde, og beholder det som er viktig. Hovedgrunnen til å gjøre dette er å hindre overtilpasning, og gjøre beregningene i senere lag raskere da bildet er mindre etter pooling-laget. F.eks. bruker vi noe som heter *max pooling*, vil vårt gråskala-bilde gå fra 28x28 piksler til 14x14.

Prosessen som er beskrevet over, gjøres igjen og igjen i CNN, for å "få meg seg" mer abstraksjon i nettverket. Til slutt, så vil CNN bestå av et *flatten layer*, som "flater ut" output'et fra pooling-lagene, se Figur 17. Deretter er det *dense* (*tette*) lag, som i MLP nettverk vi diskuterte tidligere som i Figur 13. Hvor man til slutt ender opp med et output-lag, som har som mål å klassifisere bilde som nettverket mates med. Det kan f.eks. være å klassifisere tall mellom 0 og 9 (MNIST datasettet), om det er en hest/zebra/hund, eller om bildet datamaskinen vår tildelt er et militært eller sivilt skip.

Nå har vi gitt en (meget) kort introduksjon til CNN, hvor vi ikke har diskutert all matematikk bak konvolusjonslag, alternativ man kan gjøre ved bruk av kernel, bakpropagering og mer.

## 5.5 Tidsseriemodeller

I matematikk refererer en tidsserie til en rekke datapunkter som er sortert etter tid. Vanligvis er en tidsserie en sekvens av data som er samlet inn ved jevne mellomrom, og er ofte brukt i økonometri og finans. Lukkeprisen for aksje-data, vil være et typisk eksempel på en tidsserie.

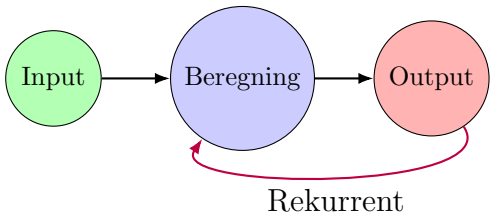
### 5.5.1 Recurrent neural networks

*Recurrent neural network (norsk: Rekursive nevrale nettverk)*, forkortet RNN, er en type nevrale nettverk som er spesielt tilpasset for å arbeide med sekvensiell data. I motsetning til mer tradisjonelle nevrale nettverk (f.eks. MLP), som behandler data i ett enkelt steg, tar RNN-er hensyn til informasjon over flere tidssteg. Dette gjør dem svært nyttige for å analysere og håndtere data som tekst, tale og tidsserier, der rekkefølgen på informasjonen er viktig.

**Visste du at?**

Ordet *rekurrent* betyr **tilbakevendende**. I sammenheng med rekurrente nevrale nettverk refererer det til nettverkets evne til å ta hensyn til tidligere informasjon i dataserien ved å sende informasjon tilbake til seg selv, noe som muliggjør en sekvensiell avhengighet over flere tidssteg.

Hvis vi ser på Figur 18 vil RNN lagre output fra første gang den går igjennom bergnings nevronet, og andre gang - etter den tilbakevendende delen. Dette gjør at at nettverket kan "huske" tidligere steg i sekvensen.

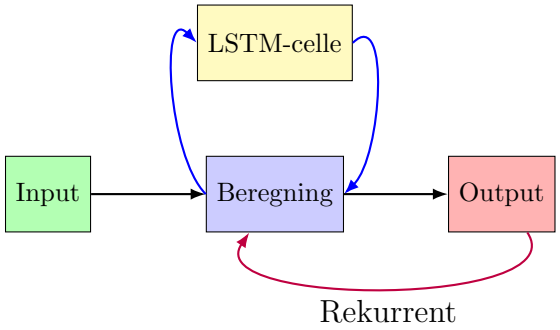



Figur 18: Illustrasjon av et rekurrent nevralt nettverk med nevroner for input, beregning, og output. Rekurrent, eller tilbakevendende tilbakemelding går fra output tilbake til beregning.

### 5.5.2 LSTM-modeller

**Long short-term memory** (LSTM) er en maskinlæringsteknikk som benytter seg av RNN. Som vi diskuterte i delkapittelet over, så kan RNN "huske"

tidligere steg - men dette medfører et problem. RNN kan lide av noe (som på engelsk) kalles *long term dependency problem*, som handler om at når nettverket lagrer mer-og-mer informasjon, kan det slite med å tilegne seg ny informasjon.




Figur 19: Illustrasjon av et rekurrent nevralt nettverk med en LSTM-celle som påvirker beregningen. Tilbakemelding går fra output til beregning, og fra beregning tilbake til LSTM-cellen.

Figur 19 viser at når et input går inn i beregnings-noden, så skjer det en beregning der (for nå, lar vi dette bare være "en beregning") - som leder til et output. Etter dette, vil beregnings-noden få informasjon fra output-noden (tilbakevendende-delen) fra det forrige steget, input fra det nye steget **og** informasjon fra LSTM-cellen.

### Hva er en LSTM-celle?

En LSTM-celle består av tre deler som vi kaller for *gates*. Det er:

- **Input gate:** Denne gaten bestemmer hvilken informasjon fra den nåværende inputen som skal lagres i cellens tilstand. Den evaluerer inngangsdataene og avgjør hvor mye av informasjonen som skal beholdes.
- **"Glemme" gate (forget gate):** Denne gaten har som oppgave å bestemme hvilken informasjon fra cellens tidligere tilstand som skal glemmes eller skrotes. Den gir modellen mulighet til å filtrere ut irrelevant informasjon, noe som er et problem i tradisjonelle RNN.
- **Output gate:** Denne gaten bestemmer hva som skal sendes videre fra LSTM-cellen til det neste laget i nettverket. Den evaluerer cellens interne tilstand og bestemmer hvilke deler av informasjonen som skal brukes til å generere output.

Alle komponentene i en LSTM-celle får en verdi mellom 0 og 1, hvor 0 betyr at *alt glemmes*, mens 0.5 betyr at noe glemmes og noe tas vare på, og 1 betyr at man tar vare på all informasjonen.

### 5.5.3 Når brukes dette?

LSTM modeller kan brukes til mange ting, som

- Chat-botter, hvor de kan glemme noe informasjon etterhvert, men kanskje må lagre annen informasjon.
- Prediksjon i tidsserie-data, som er det vi kommer til å fokusere mest på i dette kurset når det gjelder LSTM-modeller.
- Tale-gjenkjenning
- og mange, mange fler.

# 6 Så hvorfor er dette interessant for økonomer

Så langt i kompendiet har vi diskutert mange maskinlæringsteknikker, og deres detaljer. Det er viktig å ha en forståelse for fagfeltet, for å kunne anvende det i praksis.

Maskinlæring har blitt stadig viktigere innen økonomi og finans, og teknikker som brukes i nevrale nettverk kan tilby store fordeler for økonomer som jobber med både teoretiske modeller og prediksjon. Mens økonomer tidligere har hatt en tendens til å motstå bruken av maskinlæring på grunn av metodiske forskjeller fra tradisjonell økonometrisk tilnærming, har det skjedd en gradvis aksept for disse metodene [Hull, 2021]. Maskinlæring, med sitt fokus på prediksjon gjennom komplekse, ikke-lineære modeller, tilbyr en kraftig tilnærming som kan **komplementere** (ikke nødvendigvis erstatte) økonometriske metoder.

Hvor dens anvendelse er mange, fra finasielle avgjørelser, deteksjon av svindel/hvitvasking og mange flere samfunnsøkonomiske anvendelser.

# 7 Praktisk bruk i Python

De fysiske forelesningene og seminarene vil være der dere får lære mest om praktisk bruk av maskinlæring i Python. Likevel, vil denne seksjonen vise noen praktiske eksempler med Python kode.

## 7.1 Datasett

En viktig del av maskinlæring er å håndtere dataen på en effektiv måte. Her ser vi på hvordan vi kan laste inn, utforske og prosessere data for bruk i maskinlæringsmodeller, samt hvordan vi kan bruke `scikit-learn` for å dele opp datasettet i trenings- og testsett.

## 7.2 Innlasting av data

Den vanligste måten å håndtere data på i Python er ved bruk av `pandas`-biblioteket, som tilbyr en fleksibel struktur kalt DataFrame. En DataFrame kan inneholde tabulære data, noe som gjør det enkelt å utføre analyser og prosessering. La oss starte med å laste inn et eksempeldata, som vi kan anta ligger i en CSV-fil.

```
1 import pandas as pd
2
3 # Laster inn data fra en CSV-fil
4 df = pd.read_csv("data.csv")
5
6 # Ser på de første radene av dataen
7 print(df.head())
```

I koden over laster vi inn dataen fra en fil kalt `data.csv` og bruker `head()` for å se de første radene. Dette gir en rask oversikt over kolonner og verdier.

## 7.3 Utforske dataen

Etter å ha lastet inn dataen er det viktig å få en forståelse av datasettet. Dette kan innebære å se på kolonnenavn, datatype, og en statistisk oppsummering av numeriske variabler.

```
# Informasjon om datasettets kolonner og datatyper
print(df.info())

# Statistisk oppsummering av numeriske kolonner
print(df.describe())
```

Disse kommandoene hjelper oss med å avdekke eventuelle manglende verdier og om det finnes variabler som må skaleres eller prosesseres ytterligere.

## 7.4 Dataprosessering

I maskinlæring er kvaliteten på dataen ofte avgjørende for modellens ytelse. Dette diskuterte vi tidligere i kompendiet som *shit in, shit out*. Her er noen vanlige steg for å prosessere dataen:

1. **Behandling av manglende verdier**: Ofte mangler data i noen rader eller kolonner, og vi kan enten fylle disse verdiene eller fjerne radene.

```
# Fyller manglende verdier med median
df.fillna(df.median(), inplace=True)

# Alternativt kan vi droppe rader med manglende verdier
df.dropna(inplace=True)
```

2. **Skalering av data**: Mange maskinlæringsalgoritmer fungerer bedre når data er skalert til et lignende område, for eksempel mellom 0 og 1.

```
from sklearn.preprocessing import StandardScaler

# Skalerer en dataframe
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df)
```

3. ***Encoding* av kategoriske variabler**: Hvis datasettet inneholder kategoriske variabler (for eksempel tekstverdier som "Ja" eller "Nei" eller andre kategoriske verdier som "Rød", "Grønn", "Blå"), må disse ofte gjøres om til numeriske verdier for å kunne brukes i maskinlæringsmodeller. Dette kan gjøres på flere måter:

- **One-hot encoding**: Denne metoden lager nye kolonner, én for hver kategori, og fyller disse kolonnene med 1 eller 0 avhengig av om observasjonen tilhører den spesifikke kategorien. Dette er spesielt nyttig når det ikke finnes en naturlig rekkefølge mellom kategoriene. `pandas` tilbyr en enkel måte å gjøre dette på med funksjonen `get_dummies`.

```
import pandas as pd
# One-hot encoding for en kolonne kalt 'kategorie'
df = pd.get_dummies(df, columns=['kategori'])
```

For eksempel, hvis kolonnen `kategori_kolonne` har verdiene `Rød`, `Grønn`, og `Blå`, vil dette lage tre nye kolonner: `kategori_Rød`, `kategori_Grønn`, og `kategori_Blå`, der hver rad vil inneholde 1 i kolonnen for sin kategori og 0 i de andre.

| kategori |
|---|
| Blå |
| Rød |
| Grønn |
| Blå |
| Rød |

→

| kategori_Blå | kategori_Rød | kategori_Grønn |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |

Figur 20: Illusterer hva One-Hot Encoding gjør med 'kategori' kolonnen

- **Label encoding av input**: Noen ganger er det ikke ønskelig å bruke *one-hot encoding*, f.eks. over diskuterte vi det ved fargene blå, grønn og rød. Ved one-hot encoding så opprettes det tre kolonner, som har verdien 0 eller 1, basert på kolonne-navn og om den raden representerer fargen eller ikke. Men hvis vi har data hvor en naturlig ordning eller rangering, kan vi bruke **ordinal encoding**. I kode-eksempelet under bruker vi `sklearn.preprocessing` og `OrdinalEncoder()` for å beholde rekkefølgen i variabelen. "Liten" blir lik 0, "Middels" blir lik 1 og "Stor" blir lik 2.

```python
from sklearn.preprocessing import OrdinalEncoder

# Lager et eksempel-datasett
data = {'størrelse': ['Liten', 'Stor', 'Middels', 'Liten',
                      'Stor', 'Middels', 'Liten']}
df = pd.DataFrame(data)

# Bruker OrdinalEncoder med spesifiserte kategorier og ordning.
encoder = OrdinalEncoder(categories=[['Liten', 'Middels', 'Stor']])

# "Fit and transform" av 'størrelse' kolonnen
df['størrelse'] = encoder.fit_transform(df[['størrelse']])
```

Dette har mange fordeler i maskinlæring, for eksempel bruker vi beslutningstrær vil denne ordningen gjøre at treet kan forså forskjellen på variablene på en måte som "gir mening" og vi bevarer relasjonen mellom variablene. Men, det kan føre til at noen modeller "tror" det er en lineær sammenheng mellom variabler som kan medføre implikasjoner.

- **Label encoding av output**: Hvis man har output-variabler som er kategoriske, kan man benytte seg av `LabelEncoder()` fra `sklearn.preprocessing`, som tilegner et heltall til en kategori. F.eks hvis. en kolonne `df['output']` har verdiene "Kina, "India", "USA" og "Brazil". Vil koden under tilegne en verdi til hvert av landene, som "Brazil" = 0, "India" = 1, "Kina" = 2 og "USA" = 3.

```python
from sklearn.preprocessing import LabelEncoder
# Label encoding for en kolonne kalt 'output'
encoder = LabelEncoder()
df['output'] = encoder.fit_transform(df['output'])
```

Her er det viktig å merke seg at LabelEncoder egner seg å "encode" outputvariabler/målvariabel, ikke input. Legg også merke til at den sorterer i alfabetisk rekkefølge.

## 7.5 Splitte data i trenings- og testsett

En vanlig praksis i maskinlæring er å dele datasettet i et treningssett og et testsett. Treningssettet brukes til å lære opp modellen, mens testsettet brukes til å evaluere ytelsen.

```python
from sklearn.model_selection import train_test_split

# Anta at "y" er målvariabelen/output, og resten er input-funksjoner
X = df.drop("målvariabel", axis=1)
y = df["målvariabel"]

# Del opp i trenings- og testsett
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                    test_size=0.2, random_state=42)
```

## 7.6 Lineær regresjon

Lineær regresjon brukes for å modellere forholdet mellom en avhengig variabel og en eller flere uavhengige variabler. Modellen finner den beste linjen (eng: *best fit*) som passer til dataene. Den generelle formelen for enkel lineær regresjon er:

$$y = \beta_0 + \beta_1 x + \epsilon$$

Her er $y$ den avhengige variabelen, $x$ er den uavhengige variabelen, $\beta_0$ er interceptet, $\beta_1$ er stigningstallet (koeffisienten) og $\epsilon$ er feilleddet.

I Python kan vi bruke `scikit-learn` for å implementere lineær regresjon. Her er et eksempel på hvordan man kan bruke lineær regresjon til å forutsi verdien av en variabel basert på en annen:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Eksempeldata: Uavhengig variabel (x) og avhengig variabel (y)
x = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 5, 4, 5])

# Initialiser og trener vår modell
model = LinearRegression()
model.fit(x, y)

# Prediksjon
y_pred = model.predict(x)

# Visualisering
plt.scatter(x, y, color='blue', label='Data')
```

```
plt.plot(x, y_pred, color='red', label='Lineær regresjon')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()

# Utskrift av koeffisientene
print(f"Skjæringspunkt (beta_0): {model.intercept_}")
print(f"Stigningstall (beta_1): {model.coef_}")
```

I koden over har vi definert et enkelt datasett med én uavhengig variabel $x$ og en avhengig variabel $y$. Vi benytter `LinearRegression` fra `scikit-learn` for å finne den beste lineære modellen. Deretter visualiserer vi resultatene med `matplotlib`, og skriver ut modellens koeffisienter.

## 7.7 Logistisk regresjon

Logistisk regresjon brukes for klassifikasjon, hvor vi ønsker å predikere en binær utfall (typisk eksempel er om noe **skjer** eller **ikke**). Den logistiske funksjonen (sigmoid-funksjonen )konverterer output til å bli mellom 0 og 1:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

Her er $P(y = 1|x)$ sannsynligheten for at hendelsen $y$ skjer gitt $x$, og $\beta_0$, $\beta_1$ er modellens parametre.

Kodeforslag:

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Importerer datasettet (som er brukt tidligere i kurset)
df = pd.read_csv('Fraud.csv')
df.head()

# Kvitter oss med "unødvendige variabler"
df = df.drop(['nameOrig', 'nameDest', 'Unnamed: 0',
              'Date of transaction', 'type', 'branch',
              'Acct type', 'Time of day'], axis = 1)

# Kvitter oss med rader hvor det er NaN-verdier
```

```
df = df.dropna()

# Definerer output-variabelen (altså, dens kolonne-navn)
target = 'isFraud'

# X skal bestå av alle kolonner unntatt output-variabelen
X = df.drop(columns=[target_column])
y = df[target_column]                    # Målvariabelen/output

# Del datasettet i trening og testsett
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                        test_size=0.3, random_state=42)

# Initialiser og tren modellen
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

# Prediksjon på testsettet
y_pred = logreg.predict(X_test)

# Beregn nøyaktighet
accuracy = accuracy_score(y_test, y_pred)

# Skriv ut nøyaktigheten/accuracy
print(f"Accuracy er: {accuracy * 100:.2f}%")
```

I dette eksempelet bruker vi logistisk regresjon for å gjennomføre binær klassifikasjon på datasettet `Fraud.csv`. Vi trener altså en logistisk regresjonsmodell på treningsdataene, og evaluerer den på testsettet for å beregne nøyaktigheten.

## 7.8 Beslutningstrær

I eksempelet under visesr vi hvordan vi lager et beslutningstre, for å klassifisere **svindel** eller **ikke-svindel** fra `Fraud.csv` datasettet.

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Importerer datasettet
df = pd.read_csv('Fraud.csv')
```

```
df.head()

# Kvitter oss med "unødvendige variabler"
df = df.drop(['nameOrig', 'nameDest', 'Unnamed: 0',
              'Date of transaction', 'type', 'branch',
              'Acct type', 'Time of day'], axis = 1)

# Kvitter oss med rader hvor det er NaN-verdier
df = df.dropna()

# Definerer output-variabelen (altså, dens kolonne-navn)
target_column = 'isFraud'

# X skal bestå av alle kolonner unntatt output-variabelen
X = df.drop(columns=[target_column])
y = df[target_column]  # Målvariabelen/output

# Del datasettet i trening og testsett
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                        test_size=0.3, random_state=42)

# Initialiser og tren modellen med DecisionTreeClassifier
dtree = DecisionTreeClassifier(random_state=42)
dtree.fit(X_train, y_train)

# Prediksjon på testsettet
y_pred = dtree.predict(X_test)

# Beregn nøyaktighet
accuracy = accuracy_score(y_test, y_pred)

# Skriv ut nøyaktigheten/accuracy
print(f"Accuracy er: {accuracy * 100:.2f}%")
```

Her er det viktig å vite at vi ser kun på *accuracy*, og for å få et mer konkret bilde av hva som foregår i dette beslutningstreet må man visualisere treet. Kanskje må man gjennomføre pruning, som er nevnt tidligere i kompendiet.

Her er ikke nødvendigvis accuracy det beste målet på modellens ytelse, **hvorfor det**?

## 7.9 Random Forests

## 7.10 Boosting

## 7.11 Nevrale nettverk

### 7.11.1 Epochs

### 7.11.2 Regulariseringstekikker

## 7.12 Whisper

## 7.13 CNN

## 7.14 LSTM

# 8 Oppgaver

## 8.1 Oppgaver

1. Beskriv hva som menes med vektorer og matriser i maskinlæring. Gi eksempler på hvordan de brukes i praktiske anvendelser.

2. Forklar betydningen av normer i lineær algebra og hvordan de anvendes i maskinlæring.

3. Hva er transponering av en matrise? Illustrer med et eksempel og forklar hvordan denne operasjonen kan være nyttig.

4. Beskriv skalar multiplikasjon og gi et eksempel på hvordan det brukes i sammenheng med maskinlæring.

5. Forklar hva tensorer er, og hvordan de skiller seg fra matriser. Gi eksempler på anvendelser av tensorer i maskinlæring.

6. Forklar hva som menes med gradienten av en funksjon, og hvorfor den er viktig i optimeringsproblemer i maskinlæring.

7. Beskriv hvordan gradient descent fungerer, og forklar hvordan læringsraten påvirker prosessen.

8. Gradient av flervariabel funksjon.

   a) Beregn gradienten av funksjonen $f(x, y) = x^2 + y^2 - 4x - 6y + 10$.

   b) Start fra punktet $(x_0, y_0) = (0, 0)$ og bruk gradient descent med læringsrate $\eta = 0.1$ for å finne et lokalt minimum av funksjonen. Utfør tre iterasjoner og vis utregningene.

9. Funksjonen $f(x, y) = x^2 + y^4 - 3xy + 7$ skal minimeres.

   a) Beregn gradienten av $f(x, y)$.

   b) Bruk gradient descent fra startpunkt $(x_0, y_0) = (1, 1)$ og læringsrate $\eta = 0.01$ til å finne et lokalt minimum etter fire iterasjoner.

10. Forklar forskjellen mellom L1- og L2-normer.

    a) Beregn L1- og L2-normene til vektoren $\mathbf{v} = \begin{bmatrix} 3 \\ -4 \\ 1 \end{bmatrix}$.

    b) Forklar hvordan regularisering ved bruk av disse normene påvirker maskinlæringsmodeller.

## 8.2 Løsningsforslag

1. Vektorer og matriser er sentrale i representasjonen av data. Vektorer kan representere datapunkter, mens matriser kan representere datasett eller transformasjoner. For eksempel brukes matriser i nevrale nettverk for å lagre vektene.

2. Normer er viktige for å måle lengden av vektorer, noe som er nyttig i optimalisering og for å evaluere tap. L2-norm og L1-norm er to vanlige normer brukt i maskinlæring.

3. Transponering bytter plass på radene og kolonnene i en matrise. For eksempel, hvis vi har matrisen

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

så er den transponerte matrisen $A^T$:

$$A^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

Transponering kan være nyttig i beregninger som involverer dot-produktet mellom vektorer.

4. Skalar multiplikasjon er når en matrise eller vektor multipliseres med et tall. For eksempel, hvis vi har $k = 2$ og vektoren

$$x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

så er $k \cdot x = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$. Dette brukes for å justere vekter i modeller.

5. Tensorer er en utvidelse av matriser og vektorer til høyere dimensjoner. For eksempel, et gråskalabilde representeres som en rank-2 tensor (2D-matrise), mens et fargebilde (RGB) representeres som en rank-3 tensor (høyde $\times$ bredde $\times$ kanaler).

# References


[Aase, 2022] Aase, M. J. (2022). Predicting persistent weak layers in maritime regions in norway using meteorological parameters. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway. Supervisor: Jo Eidsvik.

[Curry, 2021] Curry, R. (2021). The complete guide to random forests: Part 2. Accessed: 2024-10-31.

[Deng et al., 2021] Deng, H., Zhou, Y., Wang, L., and Zhang, C. (2021). Ensemble learning for the early prediction of neonatal jaundice with genetic features. *BMC Medical Informatics and Decision Making*, 21.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, New York, 2nd edition. Access provided by UiT The Arctic University of Norway.

[Hull, 2021] Hull, I. (2021). *Machine Learning for Economics and Finance in TensorFlow 2: Deep Learning Models for Research and Industry*. Apress, Nacka, Sweden.

[James et al., 2023] James, G., Witten, D., Hastie, T., Tibshirani, R., and Taylor, J. (2023). *An Introduction to Statistical Learning with Applications in Python*. Springer Nature Switzerland AG.

[Lang, 2022] Lang, N. (2022). From vectors to tensors: Exploring the mathematics of tensor algebra. Accessed: 2024-11-01.

[Mishra, 2023] Mishra, M. (2023). Stochastic gradient descent: A basic explanation. *Medium*. Accessed: 2024-10-17.

[Murray, 2024] Murray, C. (2024). Whisper audio to text: Transcribing long audio files with python. Accessed: 2024-10-30.

[Ogbemi, 2021] Ogbemi, M.-E. (2021). What is overfitting in machine learning? *freeCodeCamp*. Accessed: 2024-10-17.

[Radford et al., 2022] Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., and Sutskever, I. (2022). Robust speech recognition

via large-scale weak supervision. *OpenAI*. https://cdn.openai.com/papers/whisper.pdf.

[Zammataro, 2022] Zammataro, L. (2022). One-vs-all logistic regression for image recognition: How to implement a machine learning python code for classifying images. *Towards Data Science*. https://towardsdatascience.com.