

# Fysikk-informerte nevrale nettverk

Prosjekt 1 i TMA4320

## 1. Introduksjon

I 2021 sto bygninger for omtrent en tredjedel av Norges totale energiforbruk, og mesteparten av dette blir brukt til oppvarming (DNV og Industri, 2022). Å minimere energien brukt på oppvarming kan derfor være positivt for både miljøet og lommeboka. Likevel er det mange utfordringer knyttet til dette: Hvor bør varmekildene plasseres? Hvor mye isolasjon er nødvendig? Hvordan kan vi oppnå en jevn temperatur?

Som matematikere kan vi svare på slike spørsmål ved å modellere temperaturen i bygget. Tradisjonelt gjøres dette med numeriske metoder, hvor vi løser en differensiallikning basert på kjente fysiske lover. Dette krever at vi kjenner systemet og alle fysiske parametere. Dessverre er virkeligheten ofte langt mer komplisert. Oppvarmingen kan ikke nødvendigvis modelleres perfekt med enkle likninger, de fysiske parameterne er som regel ukjente, og vi har kun tilgang til upresise målinger fra et fåtall sensorer.

Her kommer fysikk-informerte nevrale nettverk (PINNs) inn i bildet. Introdusert av Raissi mfl. (2019), et slikt nettverk trenes ikke kun mot tilgjengelige måledata, men også for å tilfredsstille fysiske lovene spesifisert gjennom differensiallikninger. En viktig fordel er at metoden ikke forutsetter perfekt kunnskap om det fysiske systemet. Selv om differensiallikningen kun er en tilnærming til den virkelige fysikken, kan PINNs bruke dataene til å korrigere modellen og fange opp effekter som ikke er beskrevet eksplisitt. Dette gjør metoden særlig godt egnet i situasjoner der systemet er komplekst og bare delvis observert.

I dette prosjektet skal vi anvende PINNs for å modellere temperaturfordelingen i et rom. Vi begynner med å implementere en klassisk numerisk løser for å forstå problemet og generere syntetiske data. Deretter bygger vi et nevralt nettverk fra bunnen av, og til slutt utvider vi dette til et PINN som kombinerer kunnskap om varmelikningen med observert sensordata.

**Ved slutten av prosjektet skal du ha oppnådd følgende:**

1. Implementert en endelig differansemetode for den to-dimensjonale varmelikningen med Robin randbetingelser.
2. Implementert et nevralt nettverk ved bruk av JAX (Bradbury mfl., 2018).
3. Implementert et PINN som lærer data og varmelikningen.
4. Lært hvordan strukturere et vitenskapelig beregningsprosjekt i Python, og hvordan verifisere kode gjennom automatisk testing.

## 2. Praktisk informasjon

**Innleveringsfrist** Søndag 8.januar klokka 24:00 (midnatt).

**Innleveringsplattform** Øvsys

**Språk** Dere kan velge om dere vil svare på engelsk eller norsk.

**Vurderingsansvarlig** Elling Svee ([elling.svee@ntnu.no](mailto:elling.svee@ntnu.no))

Oppgavene er delt inn i kode- og refleksjonsoppgaver. Kodeoppgavene er plassert i en **blå** boks, mens refleksjonsoppgavene er plassert i en **grønn** boks.

### 2.1. Kodeoppgaver

#### Oppgave 2.1

Dette er et eksempel på en kodeoppgave.

Her skal dere skrive kode for å implementere funksjonalitet. I hver kodeoppgave vil vi spesifisere hvilken fil som skal redigeres (f.eks. `src/project/model.py`). Inne i fila vil det være tydelig markert hvor dere skal fylle inn kode. F.eks. dersom dere løser Oppgave 2.1 vil det se ut som

```
#####
# Oppgave 2.1: Start
#####

# Skriv deres kode innenfor her
print("Hello!")

#####
# Oppgave 2.1: Slutt
#####

# IKKE skriv koden deres utenfor
```

### 2.2. Refleksjonsoppgaver

#### Oppgave 2.2

Dette er et eksempel på en refleksjonsoppgave.

I refleksjonsoppgavene skal dere svare på spørsmål som krever at dere anvender implementasjonen fra kodeoppgavene, at dere, gjør litt selvstendig forskning, og at dere må tenke kreativt for å svare på spørsmål.

Koden for å svare på disse oppgavene må dere skrive helt fra bunnen av (ikke inne i definerte områder slik som for kodeoppgavene). Det kan f.eks. være nyttig å bruke `.py` eller `.ipynb` filer, og plassere disse i `scripts/` mappa. Husk å inkludere denne koden i innleveringen!

Flere av refleksjonsoppgavene er med vilje ganske åpne. Ikke bekymre dere for mye om å finne «det riktige svaret». Vi er mer interessert i å se at dere har reflektert over problemstillingen, og kommer under retting til å se gjennom fingrene på mindre feil og mangler. Heller kom

med gode og interessante idéer enn AI-genererte svar som dere ikke har skrevet eller funnet på selv!

## 2.3. Innlevering

Innleveringen skal være en komprimert `prosjekt1.zip` fil bestående av to deler.

1. Alle kodefiler (i samme filstruktur som det utdelt kode-prosjektet). Pass på at koden kjører uten feil, og at all kode dere har skrevet for refleksjonsoppgavene er inkludert (i `scripts/` mappa). Ikke inkluder filene og mappene spesifisert i `.gitignore`-filene. Dette inkluderer blant annet `__pycache__/` og `.venv/` mapper, samt bildefiler på formater som `.png` og `.gif`.
2. En `rapport.pdf` fil som presenter svarene på refleksjonsoppgavene. Presenter nødvendig teori, men dere trenger ikke gjenta hva som allerede står i dette dokumentet. Fokuser på å forklare hvordan dere har løst oppgavene, hvilke resultater dere har fått, og deres refleksjon rundt resultatene. Å inkludere figurer som viser relevante plott er viktig for å underbygge svarene deres. For de åpne spørsmålene må dere forklare tankegangen deres og begrunne svarene deres. Gjør rapporten lesbar og ryddig. Det er ikke satt noen øvre grense på antall sider, men vi oppfordrer dere til å være konsise. Rapporten skal være skrevet i LaTeX. Det kan være nytting å bruke [Overleaf](#) for å skrive rapporten.

### 3. Oppvarming

Følg beskrivelsen i `README.md` for å sette opp prosjektet lokalt på deres maskin.

#### 3.1. Prosjektstruktur

Bruk litt tid på å få en oversikt over koden dere har fått utdelt. Koden er strukturert som følger:

`src/project/` Her ligger kjerneimplementasjonen. Det meste av stukturen er allerede på plass, men dere skal fylle inn kode i flere av filene. Hver fil har et tydelig ansvarsområde, blant annet `fdm.py` for den numeriske løseren, `model.py` for det nevrale nettverket og `loss.py` for objektivfunksjonene.

`scripts/` Kjørbare skript for å teste og eksperimentere. Disse importerer funksjonalitet fra `src/project/`, og kan brukes til å gjøre simuleringer og generere resultater. For å svare på refleksjonsoppgavene oppfordres dere til å lage egne skript her.

`tests/` Automatiske tester som verifiserer at implementasjonen deres er korrekt. Dere skal ikke endre disse filene.

`config.yaml` Se Kapittel 3.2 for detaljer. Merk at dere kan lage flere ulike konfigurasjonsfiler for å eksperimentere med forskjellige parameterkonfigurasjoner.

#### 3.2. Konfigurasjon

Alle parametere (fysiske konstanter, nettverksarkitektur, antall treningsiterasjoner etc.) er samlet i `config.yaml`. Verdiene fra fila lastes deretter inn som en dataklasse `Config` definert i `src/project/config.py`. Dere oppfordres til å lese denne fila for å se hva dere har tilgjengelig. I Python-koden laster dere inn og bruker konfigurasjonen slik:

```
from project import load_config
cfg = load_config("config.yaml")
# Dere kan nå bruke verdier fra Config dataklassen, som f.eks.
nx, sensor_locations = cfg.nx, cfg.sensor_locations
```

#### 3.3. Kjøre tester

For hver kodeoppgave vil det også være en test som dere kan kjøre for å verifisere implementasjonen deres. Disse testene er implementert med bruk av `pytest`. Navnet på testen vil bli oppgitt i oppgaven. F.eks. kan dere kjøre testene definert under `TestForward` klassen i `test_model.py` i terminalen med kommandoen

```
pytest tests/test_model.py::TestForward
```

Dersom alle testene i klassen passerer, betyr det at dere kan være *ganske* sikre implementasjonen deres fungerer som den skal for den gitte oppgaven. Merk at det kan være noen edge-cases som vi ikke tester for.

## 4. Generering av syntetiske data

### 4.1. Varmelikningen

Vi skal modellere varmeoverføring i et to-dimensjonalt rom. La  $\Omega \subset \mathbb{R}^2$  være et begrenset, sammenhengende område som representerer plantegningen til rommet, og la  $[0, t_{\max}]$  være et endelig tidsintervall. Temperaturfeltet betegnes med

$$T : \Omega \times [0, t_{\max}] \rightarrow \mathbb{R} \quad (1)$$

hvor  $T(x, y, t)$  representerer temperaturen ved den romlige posisjonen  $(x, y) \in \Omega$  og tiden  $t \in [0, t_{\max}]$ . Den initielle temperaturfordelingen i rommet er gitt som

$$T(x, y, 0) = T_{\text{out}}, \quad (2)$$

hvor  $T_{\text{out}} \in \mathbb{R}$  er temperaturen utendørs. For enkelhetens skyld antar vi at  $T_{\text{out}}$  ikke varierer med posisjon eller tid.

Temperaturutviklingen modelleres ved hjelp av varmelikningen

$$\frac{\partial T(x, y, t)}{\partial t} = \alpha \Delta T(x, y, t) + q(x, y), \quad (x, y, t) \in \Omega \times [0, t_{\max}], \quad (3)$$

hvor  $\alpha > 0$  er den termiske diffusiviteten til luft, og  $\Delta = \partial_x^2 + \partial_y^2$  betegner Laplace-operatoren. Vi antar at varmekilden er på formen

$$q(x, y) = P \cdot I_Q(x, y) \quad (4)$$

hvor  $P > 0$  spesifiserer dens styrke.  $I_Q(\cdot)$  er en indikatorfunksjon

$$I_Q(x, y) = \begin{cases} 1 & \text{if } (x, y) \in Q \\ 0 & \text{else} \end{cases},$$

hvor  $Q \subset \Omega$  spesifiserer en mindre område inne i domenet. I Figur 1 ser du et eksempel hvor  $Q$  er spesifisert som en liten kvadrat i midten av rommet. Varmeutveksling gjennom veggene i rommet modelleres ved en [Robin randbetingelse](#). La  $\partial\Omega$  betegne grensen til domenet og  $\mathbf{n}$  den utadrettede enhetsnormalvektoren. Randbetingelsen er gitt ved

$$-k \nabla T(x, y, t) \cdot \mathbf{n} = h(T(x, y, t) - T_{\text{out}}), \quad (x, y, t) \in \partial\Omega \times [0, t_{\max}], \quad (5)$$

hvor  $k > 0$  er veggens termiske ledningsevne og  $h > 0$  er varmeovergangskoeffisienten som beskriver varmetap til omgivelsene.

### 4.2. Sensordata

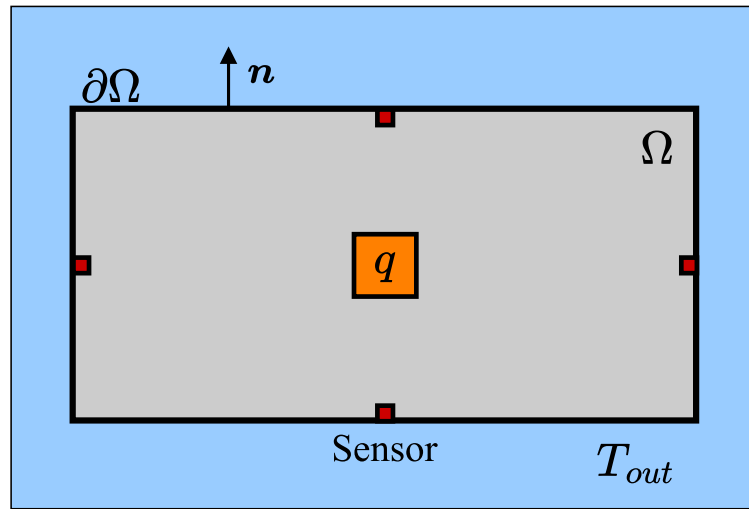
Vi antar at vi har tilgang til temperaturmålinger fra et begrenset antall sensorer plassert i rommet. Disse målingene er potensielt støyete og gir bare delvis informasjon om det fullstendige temperaturfeltet. Vårt mål er å rekonstruere det  $T$  kun ved hjelp av de tilgjengelige målingene.

La  $\{\mathbf{s}_i\}_{i=1}^{N_s} \subset \Omega$  betegne posisjonene  $\mathbf{s}_i = (x_i, y_i)$  til  $N_s$  sensorer. Sensorene tar målinger ved diskrete tidspunkter  $t_k \in [0, t_{\max}]$  for  $k = 1, \dots, N_m$ . Vi antar at sensorene er upresise,

modellert som Gaussisk støy med et kjent standardavvik  $\sigma_e$ . De observerte målingene fra sensor  $i$  ved tidspunkt  $t_k$  er dermed gitt ved

$$T_{\text{obs}}^{i,k} = T(s_i, t_k) + \varepsilon_{i,k} \quad \varepsilon_{i,k} \stackrel{\text{iid}}{\sim} \mathcal{N}(0, \sigma_e^2) \quad (6)$$

Slik de er definert gir sensorene bare delvis og upresis informasjon om det sanne temperaturfeltet  $T$ . Vårt mål er å rekonstruere  $T$  ved hjelp av de tilgjengelige målingene. Figur 1 illustrerer et scenario med en varmekilde i midten av rommet og fire sensorer plassert langs veggene. I `config.yaml` har vi også plassert en sensor i midten av rommet for å måle temperaturen til varmekilden.



Figur 1: Illustrasjon av situasjonen vi ønsker å modellere. Varmekilden er plassert i midten av rommet, og sensorer er plassert langs veggene.

### 4.3. Diskretisering og numerisk løser

Vi implementerer først en numerisk løser som gir oss den «sanne» løsningen. Denne lar oss undersøke hvordan ulike faktorer (antall sensorer, støynivå, nettverksarkitektur) påvirker resultatene, noe som ville vært vanskelig med ekte eksperimentelle data der vi ikke kjenner den sanne løsningen. Vi kommer også til å bruke denne løseren til å generere syntetiske sensordata som vi senere kan bruke til å trene de nevrale nettverkene.

Diskretiser det romlige domenet  $\Omega$  ved hjelp av et rutenett med  $N_x$  punkter i  $x$ -retningen og  $N_y$  punkter i  $y$ -retningen. La  $\Delta x$  og  $\Delta y$  betegne rutenettsavstanden i hver retning, henholdsvis. Tidsintervallet  $[0, t_{\text{max}}]$  diskretiseres i  $N_t$  tidssteg med steglengde  $\Delta t$ . La de diskrete temperaturverdiene ved rutenettpunktet  $(i, j)$  og tidssteget  $k$  betegnes med

$$T_{i,j}^k \approx T(x_i, y_j, t_k), \quad i = 0, \dots, N_x - 1, \quad j = 0, \dots, N_y - 1, \quad k = 0, \dots, N_t - 1, \quad (7)$$

for  $x_i = i\Delta x$ ,  $y_j = j\Delta y$ , og  $t_k = k\Delta t$ .

For å løse Likning Ligning 3 numerisk, bruker vi en differansemetode kalt [implicit Euler](#). Her erstatter vi de første- og andreordens deriverte med approksimasjoner. Setter vi approksimasjonene inn i Ligning 3 får vi

$$\frac{T_{i,j}^{k+1} - T_{i,j}^k}{\Delta t} = \alpha \left( \frac{T_{i+1,j}^{k+1} - 2T_{i,j}^{k+1} + T_{i-1,j}^{k+1}}{(\Delta x)^2} + \frac{T_{i,j+1}^{k+1} - 2T_{i,j}^{k+1} + T_{i,j-1}^{k+1}}{(\Delta y)^2} \right) + q(x_i, y_j).$$

Robin-randbetingelsen fra Ligning 5 må også diskretiseres. På venstre rand ( $x = 0$ ) har vi  $\mathbf{n} = (-1, 0)$ . For å approksimere den deriverte på randen innfører vi en  $T_{-1,j}^{k+1}$  utenfor domenet. Dette kalles gjerne en *fictitious node* (se f.eks. [her på side 24](#)). Den diskretiserte randbetingelsen på venstre rand blir dermed

$$k \frac{T_{0,j}^{k+1} - T_{-1,j}^{k+1}}{\Delta x} = h(T_{0,j}^{k+1} - T_{\text{out}}),$$

og løser vi for  $T_{-1,j}^{k+1}$  får vi

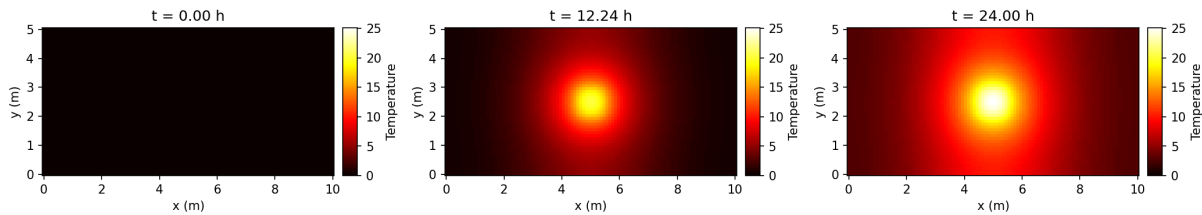
$$T_{-1,j}^{k+1} = T_{0,j}^{k+1} - \frac{h\Delta x}{k}(T_{0,j}^{k+1} - T_{\text{out}}).$$

Ved å sette dette inn i den diskretiserte varmelikningen ved  $i = 0$  får vi en modifisert likning som kun involverer punkter innenfor domenet. Tilsvarende uttrykk kan utledes for de andre randene.

Dersom vi nå isolerer temperaturene fra tidssteg  $k + 1$  på venstre side og  $k$  på høyre, kan dette skrives som systemet

$$\mathbf{A}\mathbf{T}^{k+1} = \mathbf{b}^k, \quad (8)$$

hvor  $\mathbf{T}^{k+1} \in \mathbb{R}^{N_x \cdot N_y}$  er en *vektorisering* av temperaturmatrisen ved tidspunkt  $t_{k+1}$ . Vi bruker radvis rekkefølge, slik at indeks  $(i, j)$  i matrisen tilsvarer posisjon  $i \cdot N_y + j$  i vektoren. Løser vi dette systemet for hvert tidssteg vil vi få en numerisk løsning av hvordan temperaturen utvikler seg over tid. Figur 2 viser et eksempel på hvordan dette kan se ut.



Figur 2: Eksempel på hvordan varmedistribusjonen i rommet fordeler seg.

## 4.4. Oppgaver

### Oppgave 3.1

Finn formel for elementene i den diskretiserte matrisen  $\mathbf{A}$  og vektoren  $\mathbf{b}^k$  i Ligning 8. Sammenlign disse med de ferdigskrevne funksjonene `_build_matrix()` og `_build_rhs()` i `src/project/fdm.py` for å forstå hvordan de er implementert.

**Merk:** Du trenger bare å beskrive elementenes plassering i matrisen og vektoren, samt hvordan de beregnes. Du trenger ikke skrive ut hele matrisen eksplisitt.

### Oppgave 3.2

Fullfør funksjonen `solve_heat_equation()` i `src/project/fdm.py`. Denne konstruerer matrisen  $\mathbf{A}$  og høyresiden  $\mathbf{b}^k$ , og løser det lineære likningssystemet ved hvert tidssteg.

Bruk `scripts/run_fdm.py` for å generere og visualisere løsningen. Her har vi i `scripts/viz.py` allerede skrevet kode for å plote og animere løsningen over tid. Vi oppfordrer dere likevel til å endre på plotten etter ønske og behov.

#### Tips:

- Den fullstendige løsningen  $\mathbf{T}$  et 3D-array med `shape=(cfg.nt, cfg.nx, cfg.ny)`.
- Benytt funksjonene `_build_matrix()`, `_build_rhs()`
- Bruk verdien `cfg.T_outside` for initialbetingelsen i Ligning 2.
- Funksjonen `np.linalg.solve()` kan også være nyttig.

Test løseren med `pytest tests/test_fdm.py`.

### Oppgave 3.3

I `src/project/data.py` fullfør funksjonen `generate_training_data()` for å generere syntetiske sensordata ved bruk av den numeriske løseren. Plott sensordataene for å verifisere at de ser rimelige ut.

#### Tips:

- Benytt først `solve_heat_equation()` for å generere den numeriske løsningen.
- Hjelpfunksjonen `_generate_sensor_data()` kan benyttes til å generere sensordataene ut ifra output `T_fdm` fra `solve_heat_equation()`.

Test med `pytest tests/test_data.py`.

### Oppgave 3.4

Hvordan kan vi bruke en modell som dette for å redusere energiforbruket under oppvarmingen av hus?

#### Ideer:

- Se effekten av isolasjon.
- Smart varmekilde som automatisk justerer effekten basert på utetemperatur og ønsket innendørstemperatur.
- Optimere plassering av varmekilder og sensorer.

### Oppgave 3.5

I vår løser bruker vi en implisitt metode. Hva er forskjellen på eksplisitte og en implisitte metoder? Hvorfor kan implisitte metoder være tregere enn eksplisitte? Kan du tenke deg noen metoder for å håndtere dette?

#### Ideer:

- Iterative metoder (f.eks. [Conjugate Gradient](#)).
- Utnytte at  $\mathbf{A}$  inneholder få ikke-null elementer (en såkalt *sparse matrix*).
- Parallellisering av løseren.

### Oppgave 3.6

Anta at vi ikke kjenner de fysiske parameterne  $\alpha$ ,  $k$ , og  $h$ . Hvordan kan vi estimere disse parameterne ved å bruke sensordata og den numeriske løseren vi har implementert?

#### Tips:

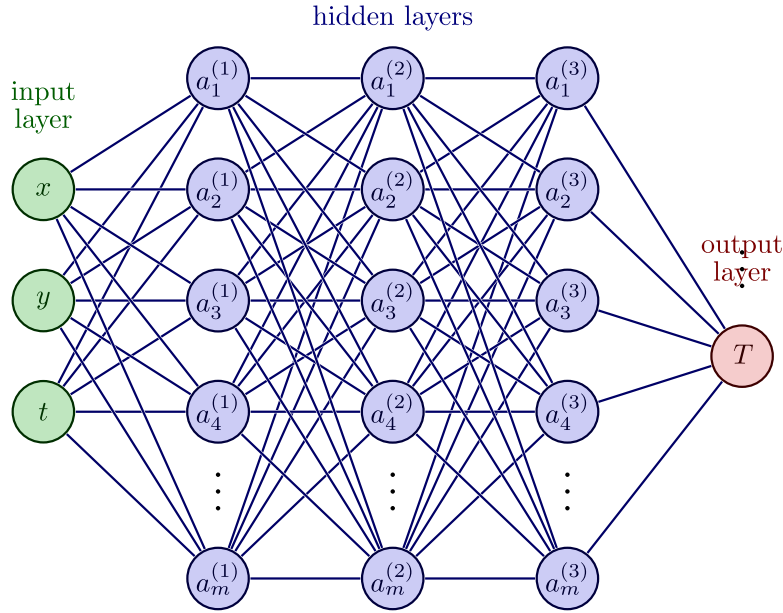
- Tenk på dette som et optimeringsproblem. Denne ideen blir sentral når vi senere ser på NNs.

## 5. Nevrale nettverk

### 5.1. Bakgrunn

Nå som vi har en numerisk løser som kan generere syntetiske sensordata, er vi klare til å implementere maskinlæringsmetodene. Vi begynner med et vanlig nevral nettverk (NN) som lærer fra sensordataene. Et NN er i bunn og grunn en ikke-lineær funksjon med mange parametre som kan trenes til å tilpasse seg data. NNs er *universelle funksjonsapproximatorer*, noe som betyr at de under milde antakelser kan tilnærme enhver funksjon vilkårlig godt (Hornik mfl., 1989). Dette er en bemerkelsesverdig egenskap som gjør NNs til et kraftig verktøy for å modellere komplekse og ikke-lineære fenomener.

Et *dypt* NN har som regel flere *skjulte* lag. Figur 3 viser et eksempel på et NN med tre skjulte lag. I figuren ser vi at hvert av lagene består av flere noder kalt *nevroner*, hvor hvert nevron utfører en lineær transformasjon av sin input etterfulgt av en ikke-lineær *aktiveringsfunksjon*. Legg også merke til at hvert nevron i et lag er koblet til alle nevronene i det påfølgende laget, noe som kjennetegner et *fullt koblet* NN.



Figur 3: Eksempel på et fullt koblet NN med tre skjulte lag.

I vårt tilfelle noterer vi et NN med parametre  $\theta$  som  $f_\theta : \mathbb{R}^3 \rightarrow \mathbb{R}$ . Funksjonen tar inn en vektor  $(x, y, t)$  som beskriver posisjonen i domenet og tidspunktet, og returnerer skalaren  $T = f_\theta(x, y, t)$  som representerer temperaturen ved den gitte posisjonen og tiden. Vi deler parametrene for lag  $i$  inn i vektor  $\mathbf{W}^{(i)} \in \mathbb{R}^{n_i \times m_i}$  og bias  $\mathbf{b}^{(i)} \in \mathbb{R}^{m_i}$ , hvor dimensjonene  $n_i$  og  $m_i$  avhenger av antall nevroner i lagene. Dersom vi noterer  $\mathbf{x} = \mathbf{a}^{(0)} \in \mathbb{R}^3$  vil transformasjonen for hver lag  $i$  i vårt NN være gitt som

$$\mathbf{a}^{(i+1)} = \sigma(\mathbf{W}^{(i)} \mathbf{a}^{(i)} + \mathbf{b}^{(i)}).$$

Her anvendes aktiveringsfunksjonen  $\sigma(\cdot)$  komponentvis på alle elementene i vektoren. For dette prosjektet velger vi aktiveringsfunksjonen til å være den hyperbolske tangens

$$\sigma(\mathbf{z}) = \tanh(\mathbf{z}).$$

Ettersom denne funksjonen er glatt, kan vi senere beregne de deriverte som trengs for PINNs.

## 5.2. Objektivfunksjon

For å estimere parametrene  $\theta$  i NN, må vi definere en *objektivfunksjon*  $\mathcal{L}(\theta)$  som måler hvor godt nettverket tilpasser seg de observerte sensordataene fra Ligning 6. Når vi trener et NN, ønsker vi å minimere forskjellen mellom de predikerte verdiene fra nettverket og de observerte målingene. Dette tilsvarer å minimere  $\mathcal{L}(\theta)$  med respekt til parametrene  $\theta$  i nettverket, ofte kalt å minimere *tapet*.

Objektivfunksjonen kan bestå av flere kompotenter som hver måler ulike aspekter av modellens ytelse. Vi kaller funksjonene som utgjør objektivfunksjonen for *tapsfunksjoner*. For vårt NN definerer vi to tapsfunksjoner: en for sensordataene og en for initialbetingelsen (som vi antar er kjent).

En vanlig brukt tapsfunksjon for regresjonsproblemer er den *gjennomsnittlige kvadratiske feilen* (MSE). Gitt de totalt  $N_s \cdot N_m$  sensordataene er denne definert som

$$\mathcal{L}_{\text{data}}(\boldsymbol{\theta}) = \frac{1}{N_s \cdot N_m} \sum_{i=1}^{N_s} \sum_{k=1}^{N_m} (f_{\boldsymbol{\theta}}(s_i, t_k) - T_{\text{obs}}^{i,k})^2.$$

Vi kjenner den initelle temperaturen ved alle posisjoner i rutenettet. Likevel kan å bruke alle disse punktene i objektivfunksjonen være beregningstungt. I praksis håndterer vi dette ved å bruke *stokastisk sampling*. Vi kaller hver treningsiterasjon for en *epoke*, men merk at i maskinlæringsverdenen brukes dette ordet for ulike ting basert på treningsstrategien. For hver epoke trekker vi et utvalg av  $N_{\text{ic}}$  posisjoner  $\{s_i^{\text{ic}}\}_{i=1}^{N_{\text{ic}}} \subset \Omega$  fra domenet, og evaluerer initialbetingelsen kun ved disse posisjonene. Tapsfunksjonen for initialbetingelsen blir

$$\mathcal{L}_{\text{ic}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{ic}}} \sum_{i=1}^{N_{\text{ic}}} (f_{\boldsymbol{\theta}}(s_i^{\text{ic}}, 0) - T_{\text{out}})^2.$$

Objektivfunksjonen kan derfor uttrykkes som summen av de to komponentene

$$\mathcal{L}(\boldsymbol{\theta}) = \lambda_{\text{data}} \mathcal{L}_{\text{data}}(\boldsymbol{\theta}) + \lambda_{\text{ic}} \mathcal{L}_{\text{ic}}(\boldsymbol{\theta}), \quad (9)$$

hvor  $\lambda_{\text{data}}, \lambda_{\text{ic}} > 0$  er vekter som balanserer inflytelsen til hver av de to komponentene.

### 5.3. Optimering

For å minimere objektivfunksjonen  $\mathcal{L}(\boldsymbol{\theta})$  bruker vi en optimeringsalgoritme. Denne oppdaterer parametrene  $\boldsymbol{\theta}$  iterativt for å redusere tapet. La  $\boldsymbol{\theta}^{(i)}$  notere de estimerte parametrene etter  $i$  epoker, og la  $\nabla_{\boldsymbol{\theta}} \mathcal{L}$  notere gradienten av objektivfunksjonen med respekt til parametrene. Siden  $\mathcal{L}$  lokalt avtar raskest i retningen av den negative gradienten, vil en enkel måte å oppdatere parametrene være å bevege seg litt i denne retningen. La

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \tau \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(i)}), \quad (10)$$

hvor parameteren  $\tau$  forteller hvor langt vi skal gå langsmed gradienten. Denne metoden blir noen ganger omtalt som *Gradient Descent*. Den har mange variasjoner, hvor noen er beskrevet i [denne artikkelen](#).

En svært populær variant av oppdateringsregelen fra Ligning 10 er den stokastiske optimeringsalgoritmen *Adam*. Mellom 2015 og 2020 var Adam den mest siterte vitenskapelige artikkelen på tvers av alle felt (Crew, 2020). Forfatteren [Diederik Kingma](#) var også med å starte OpenAI i 2015, og utviklet Variational Autoencoders som bidro til å popularisere dype NN for bildegenerering. Vi har allerede implementert Adam i `src/project/optim.py`, og du kan se på Algorithm 1 i Kingma og Ba (2017) for detaljene.

### 5.4. Automatisk derivasjon ved bruk av JAX

Observer fra Ligning 10 at vi trenger å beregne gradienten  $\nabla_{\boldsymbol{\theta}} \mathcal{L}$  for hver iterasjon i optimeringsalgoritmen. Merk at  $\mathcal{L}$  kan ha tusenvis av parametre, og består av ikke lineære funksjoner. Numerisk derivasjon ved hjelp av differansemetoder kan være unøyaktig og ineffektiv, mens å utlede gradienten analytisk betyr at vi må gjøre dette på nytt hver gang vi endrer objektivfunksjonen eller nettverksarkitekturen. Kan du tenke deg en alternativ måte å håndtere dette problemet på?

Teknikken kalt *automatisk derivasjon* (AD) gir en elegant løsning på dette problemet. Ikke bare er den svært effektiv, men den gir også eksakte deriverte (opp til maskinpresisjon). AD har ikke bare revolusjonert treningen av NNs, men også blitt et sentralt verktøy i mange andre områder av vitenskapelig beregning. Kort fortalt bryter AD ned hele sekvensen av operasjoner som utgjør vårt NN, og bruker kjedenregelen for derivasjon til å kombinere de lokale deriverte av hver enkelt operasjon. For å lære mer om AD kan [denne videoen](#) være en fin introduksjon.

Å skrive en grunnleggende AD implementasjon fra bunnen av er gjennomførbart. Se blant annet [dette eksemplet](#) for hvordan det kan gjøres på under 100 linjer i Python. For enkelhetens skyld vil vi i dette prosjektet i stedet bruke JAX (Bradbury mfl., 2018). Dette biblioteket er utviklet av Google, og tilbyr en Numpy-liknende API med innebygd støtte for AD. JAX har også mange andre spennende funksjoner, som blant annet GPU-akselerasjon og just-in-time kompilering. Som en kort introduksjon til biblioteket har vi lagt ved `jax_intro.ipynb`. Vi anbefaler at du går gjennom denne før du begynner på oppgavene.

## 5.5. Oppgaver

### Oppgave 4.1

I `src/project/model.py` bruk JAX til å implementere et NN i funksjonen `forward()`.

#### Tips:

- Merk at `x`, `y`, og `t` kan være arrays, og men at vi allerede har implementert kode for å forsikre oss om at de har samme størrelse og datatype.
- Funksjonen `a = jnp.stack([x, y, t], axis=-1)` kan brukes for å stable inputvariablene til en enkelt array med shape `(N, 3)`, hvor `N` er antall datapunkter.
- Se fra `init_nn_params()` funksjonen at `nn_params` er en liste av tuples som inneholder  $\mathbf{W}^{(i)}$  og  $\mathbf{b}^{(i)}$  for hvert lag i nettverket.

Test med `pytest tests/test_model.py::TestForward`.

### Oppgave 4.2

Implementer funksjonen `data_loss()` i `src/project/loss.py`. Denne bruker sensordataen til å beregne tapskomponenten  $\mathcal{L}_{\text{data}}$ .

Implementer deretter `ic_loss()` for å beregne tapskomponenten  $\mathcal{L}_{\text{ic}}$ . Denne bruker de tilfeldig valgte  $N_{\text{ic}}$  punktene.

#### Tips:

- Bruk `forward()` for å få predikere med NN.
- `jnp.mean()` kan brukes til å beregne gjennomsnittet over et array.

Test `data_loss()` med `pytest tests/test_loss.py::TestDataLoss` og `ic_loss()` med `pytest tests/test_loss.py::TestICLoss`.

### Oppgave 4.3

I `src/project/train.py`, implementer treningen av NN i funksjonen `train_nn()`.

#### Tips:

- Iterer over epokene spesifisert i `cfg.num_epochs`. For hver epoke:
  1. Sample posisjoner hvor initialbetingelsene skal evalueres.
  2. Evaluer objektivfunksjonen og dens gradient.
  3. Oppdater parametrene ved å ta et steg i Adam-algoritmen.
  4. Lagre tapene (total, data og ic) fra hver epoke for logging.
- Bruk `ic_epoch`, `key = sample_ic(key, cfg)` for å sample posisjoner hvor initialbetingelsene skal evalueres.
- Vektene  $\lambda_{\text{data}}$  og  $\lambda_{\text{ic}}$  er lagret i `cfg.lambda_data` og `cfg.lambda_ic`.
- For en funksjon `objective_fn(nn_params)`, kan du bruke JAX til å beregne både verdien av objektivfunksjonen og gradienten samtidig ved å bruke `jax.value_and_grad()`. Merk at dersom `objective_fn` returnerer ekstra verdier (f.eks. for logging) må du sette `has_aux=True`.
- Ta et steg i den forhåndsimplementerte Adam-algoritmen i ved

```
# adam_state: vektor som oppdateres for hvert steg
# grads: gradienten av loss_fn med respekt til nn_params
nn_params, adam_state = adam_step(
    nn_params, grads, adam_state, lr=cfg.learning_rate
)
```

- Funksjonen som evalueres i hver epoke gjøres raskere ved å utnytte [JAX just-in-time \(JIT\) kompilering](#).
- Du kan bruke [tqdm](#) for å lage en pen progress-bar. For eksempel

```
# Vanlig for-løkke
for _ in range(cfg.num_epochs):
    ... # Din kode her

# for-løkke med pen progress-bar
from tqdm import tqdm
for _ in tqdm(range(cfg.num_epochs), desc="Training NN"):
    ... # Din kode her
```

Test med `pytest tests/test_train.py::TestTrainNN`.

#### Oppgave 4.4

I `scripts/run_nn.py`, skriv et liknende script som `scripts/run_fdm.py` for å trene og evaluere NN. Plott hvordan tapene utvikler seg i løpet av treningen, og visualiser prediksjonene fra det ferdig trente nettverket.

##### Tips:

- Benytt `generate_training_data()`, `train_nn()`, og den ferdigskrevne hjelpefunksjonen `predict_grid()` for å generere data, trene nettverket, og gjøre prediksjoner på hele det diskretiserte domenet.

#### Oppgave 4.5

Evaluer resultatene ved å sammenligne med den numeriske løseren. Er NN i stand til å lære temperaturfeltet fra sensordataene alene?

##### Andre ting som kan være relevante å diskutere:

- Hvordan påvirkes resultatene av antall sensorer og mengden støy i målingene?
- Hvordan påvirkes resultatene av nettverksarkitekturen (antall lag og nevroner per lag)?
- Hvordan påvirkes resultatene av antall epoker og parametrene til Adam-algoritmen?

## 6. Fysikk-informerte nevrale nettverk

### 6.1. Oppdatert objektivfunksjon

Vårt NN fra Kapittel 5 kan fungere bra når vi har rikelig med data, men i vårt tilfelle har vi bare et fåtall sensorer med støyete målinger. Som dere kanskje ser når dere evaluerer resultatene, er ikke nettverket i stand til å rekonstruere temperaturen særlig godt. Heldigvis kjenner vi de underliggende fysiske prinsippene som styrer systemets oppførsel, ettersom varmelikningen forteller oss hvordan temperaturen *bør* utvikle seg i tid og rom. Dersom vi er i stand til å inkorporere denne kunnskapen i treningsprosessen, kan vi forbedre vårt NN.

PINNs oppnår nettopp dette ved å inkludere en straff for brudd på de fysiske lovene beskrevet i Ligning 3. La vår PINN bestå av et nevral nettverk  $f_{\theta_{\text{NN}}}$  med parametre  $\theta_{\text{NN}}$  som tar inn  $(x, y, t)$  og returnerer temperaturen  $T$ . I tillegg, ettersom de i vikeligheten er ukjente, må vi inkludere de fysiske parameterne  $\alpha$ ,  $k$ ,  $h$  og  $P$  i treningen. For å forsikre oss om at parameterestimaterne forblir positive optimerer vi med hensyn på de log-skalere

$$\theta_{\log \alpha}, \theta_{\log k}, \theta_{\log h} \text{ og } \theta_{\log P}. \quad (11)$$

For å forenkle notasjonen definerer vi den estimerte  $\alpha$  som

$$\theta_{\alpha} = \exp(\theta_{\log \alpha}), \quad (12)$$

og tilsvarende for  $\theta_k$ ,  $\theta_h$  og  $\theta_P$ . Vi samler parametrene i vektoren

$$\theta = (\theta_{\text{NN}}^T, \theta_{\log \alpha}, \theta_{\log k}, \theta_{\log h}, \theta_{\log P})^T \quad (13)$$

På tilsvarende måte som for  $\mathcal{L}_{ic}$  benytter vi stokastisk sampling under treningen av PINNs. Gjør et tvalg av  $N_{ph}$  punkter  $\{s_i^{ph}\}_{i=1}^{N_{ph}} \subset \Omega$  og tider  $\{t_i^{ph}\}_{i=1}^{N_{ph}} \subset [0, t_{max}]$  hvor vi evaluerer brudd på varmelikningen. Vi kan da definere tapsfunksjonen

$$\mathcal{L}_{ph}(\theta) = \frac{1}{N_{ph}} \sum_{i=1}^{N_{ph}} \left( \frac{\partial f_{\theta_{NN}}(s_i^{ph}, t_i^{ph})}{\partial t} - \theta_{\alpha} \Delta f_{\theta_{NN}}(s_i^{ph}, t_i^{ph}) - q(s_i^{ph}, \theta_P) \right)^2 \quad (14)$$

hvor  $q(\cdot, \theta_P) = \theta_P \cdot I_Q(\cdot)$  spesifiserer varmekilden med den estimerte styrken  $\theta_P$ . Desto mindre  $\mathcal{L}_{ph}$  er, desto bedre overholder nettverket varmelikningen. På tilsvarende kan vi definere straff for brudd på randbetingelsene

$$\mathcal{L}_{bc}(\theta) = \frac{1}{N_{bc}} \sum_{i=1}^{N_{bc}} \left( -\theta_k \nabla f_{\theta_{NN}}(s_i^{bc}, t_i^{bc}) \cdot \mathbf{n} - \theta_h (f_{\theta_{NN}}(s_i^{bc}, t_i^{bc}) - T_{out}) \right)^2, \quad (15)$$

hvor vi velger  $N_{bc}$  tider  $\{t_i^{bc}\}_{i=1}^{N_{bc}}$  og punkter  $\{s_i^{bc}\}_{i=1}^{N_{bc}} \subset \partial\Omega$  langs randen av domenet.

Ved å kombinere Ligning 14 med tapsfunksjonene for sensordata og initialbetingelsene, får vi en ny objektivfunksjon som både tilpasser seg dataene og respekterer fysikken

$$\mathcal{L}(\theta) = \lambda_{data} \mathcal{L}_{data}(\theta_{NN}) + \lambda_{ic} \mathcal{L}_{ic}(\theta_{NN}) + \lambda_{ph} \mathcal{L}_{ph}(\theta) + \lambda_{bc} \mathcal{L}_{bc}(\theta). \quad (16)$$

Her bestemmer vektene  $\lambda_{ph}, \lambda_{bc} > 0$  hvor mye vi vektlegger at nettverket skal overholde den fysiske modellen og randbetingelsene.

## 6.2. Oppgaver

### Oppgave 5.1

I `src/project/model.py`, implementer funksjonen `init_pinn_params()` for å initialisere parametrene til PINN. Merk at  $\theta$  inneholder både parametrene til det nevrale nettverket og de fysiske parameterne. Håndter dette i din implementasjon ved å la `pinn_params` være et dictionary.

#### Tips:

- Bruk `init_nn_params()` for å initialisere nevrale nettverksparametrene, og manuelt initialiser de fysiske parameterne til rimelige startverdier.
- Ikke glem at vi skal optimere mhp. de log-skalere fysiske parameterne!

Test med `pytest tests/test_model.py::TestInitPinnParams`.

### Oppgave 5.2

I `src/project/loss.py`, implementer tapet fra Ligning 14 i `physics_loss()`. Merk at tapet fra Ligning 15 er allerede implementert i `bc_loss()`, så dere slipper å gjøre dette selv.

#### Tips:

- Du kan derivere og dobbellderivere i JAX ved å bruke funksjonen `grad()` (importert på toppen av fila). For eksempel, for en funksjon `f(x, y, t)` kan du derivere mhp. `t` og dobbellderivere mhp. `x` ved

```
f_t = grad(f, 2)(x, y, t) # Velger tredje argument t med indeks 2
# Velger første argument x med indeks 0 for begge derivasjonene
f_xx = grad(grad(f, 0), 0)(x, y, t)
```

- Bruk `cfg.is_source(x, y)` funksjonen for å bestemme posisjonen til varmekilden.
- Det vil være nyttig å benytte `jax.vmap()` for å vektorisere over de  $N_{ph}$  posisjonene. F.eks. kan dere som en hjelpefunksjon `_pde_residual_scalar()` som tar til skalarer (arrays med kun ett element) vektorisere med å skrive

```
residuals = vmap(
    lambda xi, yi, ti: _pde_residual_scalar(pinn_params, xi, yi, ti, cfg)
)(x, y, t)
```

Test med `pytest tests/test_loss.py::TestPhysicsLoss`.

### Oppgave 5.3

På tilsvarende måte som for NN, i `src/project/train.py` implementer treningen av PINN i funksjonen `train_pinn()`. Test med `pytest tests/test_train_pinn.py`.

#### Tips:

- For hver epoke, bruk hjelpefunksjoner for å sample kollokasjonspunkter, initialbetingelser og randpunkter

```
interior_epoch, key = sample_interior(key, cfg)
ic_epoch, key = sample_ic(key, cfg)
bc_epoch, key = sample_bc(key, cfg)
```

- Vektene  $\lambda_{ph}$  og  $\lambda_{bc}$  er lagret i `cfg.lambda_physics` og `cfg.lambda_bc`.

Test med `pytest tests/test_train.py::TestTrainPINN`.

### Oppgave 5.4

På tilsvarende måte som for NN, skriv et script i `scripts/run_pinn.py` for å trene og evaluere PINN. Visualiser treningsprosessen og prediksjoner. Print de lærte fysiske parameterne etter trening.

### Oppgave 5.5

Evaluer resultatene ved å sammenligne med den numeriske løseren og NN. Hvordan fungerer PINN sammenlignet med NN i ulike scenarier?

#### Ideer:

- Hva skjer når vi varierer antall sensorer og deres plassering?
- Hvordan påvirker støynivået i målingene de to metodene ulikt?
- Kan PINN gi rimelige prediksjoner i områder langt fra sensorene?

### Oppgave 5.6

Hvor godt klarer PINN å lære de fysiske parameterne  $\alpha$ ,  $k$ ,  $h$  og  $P$ ? Sammenlign med de sanne verdiene brukt i den numeriske løseren. Hvorfor tror du at resultatene er som de er?

### Oppgave 5.7

Hvordan påvirkes resultatene av ulike valg av vektene  $\lambda_{\text{data}}$ ,  $\lambda_{\text{ic}}$ ,  $\lambda_{\text{physics}}$  og  $\lambda_{\text{bc}}$ ? Har du noen anbefalinger for hvordan disse bør velges i praksis? Er det mulig å unngå å måtte velge dem manuelt?

### Oppgave 5.8

Et naturlig spørsmål er: *Hvorfor bruke PINNs når vi allerede har en numerisk løser som kan beregne temperaturfeltet nøyaktig?* Nå som du har satt deg inn i metodikken, disukter hva du ser på som fordelene og ulempene ved PINNs sammenlignet med tradisjonelle numeriske metoder og NNs.

### Oppgave 5.9

Hva ville du utforsket videre dersom du skulle fortsette arbeidet med PINNs? Hvilke forbedringer eller utvidelser ville du implementert? Hvordan måtte du endre den nåværende implementasjonen for å håndtere disse nye utfordringene?

#### Ideer:

- Forbedre treningsstrategien, for eksempel ved å dele treningen inn i epoker, batches og iterasjoner.
- Representere varmekilden som et nevralt nettverk.
- Tidsavhengige eller romavhengige fysiske parametere.
- [Hamiltonian Neural Networks](#) for å sikre energikonservering.
- Amortisere treningen av PINN for å generalisere til flere ulike scenarier

## Bibliografi

- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., Zhang, Q., 2018. JAX: composable transformations of Python+NumPy programs [WWW Document].. URL <http://github.com/jax-ml/jax>
- Crew, B., 2020. Google Scholar reveals its most influential papers for 2020. Nature Index (News).
- DNV, Industri, N., 2022. Energy Transition Norway 2022: A National Forecast to 2050.
- Hornik, K., Stinchcombe, M., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural Networks 2, 359–366.. [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8)
- Kingma, D.P., Ba, J., 2017. Adam: A Method for Stochastic Optimization [WWW Document].. <https://doi.org/10.48550/arXiv.1412.6980>
- Raissi, M., Perdikaris, P., Karniadakis, G.E., 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics 378, 686–707.. <https://doi.org/10.1016/j.jcp.2018.10.045>