Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Trym Dyrkorn
Marius Sæther
Snorre Barthel Sveen
Christian Wiggo Nielsen

# Need For Speed

## Sanntidsplattform for rekkverksdeteksjon og bildevalidering på iSi inSight

Bacheloroppgave i Robotikk og automatisering
Veileder: Christian Fredrik Sætre
Mai 2024

NTNU
Kunnskap for en bedre verden

iSi

Trym Dyrkorn
Marius Sæther
Snorre Barthel Sveen
Christian Wiggo Nielsen

# Need For Speed

## Sanntidsplattform for rekkverksdeteksjon og bildevalidering på iSi inSight

Bacheloroppgave i Robotikk og automatisering
Veileder: Christian Fredrik Sætre
Mai 2024

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU
Kunnskap for en bedre verden

NTNU

**Oppgavetittel:**
Need for Speed
Sanntidsplattform for rekkverksdeteksjon og bildevalidering på iSi inSight
Real-Time platform for guard rail detection and image validation on iSi inSight


**Forfattere:**
Trym Dyrkorn
Marius Sæther
Snorre Barthel Sveen
Christian Wiggo Nielsen



**Prosjektnummer:** E2402
**Innleveringsdato:** 21.05.2024
**Gradering:** [ x ] åpen
[ ] lukket


**Studium:** Bachelor i ingeniørfag, Elektro
**Studieretning:** Robotikk og automatisering


**Veileder internt:** Christian Fredrik Sætre
**Institutt:** Institutt for teknisk kybernetikk


**Oppdragsgiver:** iSi AS
**Kontaktperson:** Christian Fredrik Sætre, christian.f.satre@ntnu.no


**Sammendrag:**

Denne bacheloroppgaven har hatt som hensikt å utforske sanntids objektdeteksjon for iSi AS sitt Insight system, og om dette kan benyttes for å redusere feil i deres datainnsamlingsprosess. Prosjektet har resultert i en sanntidsplattform tiltenkt å implementeres på datainnsamlingsriggen til iSi inSight. Systemarkitekturen er basert rundt en Nvidia Jetson Orin Nano med en Nvidia Triton Inference Server. Serveren er satt opp med to maskinlæringsmodeller for deteksjon av rekkverk og optiske forstyrrelser i innkommende bildedata. Prosjektet konkluderer med at sanntids objektdeteksjon kan gjennomføres lokalt på innsamlingsriggen, men foreslår klassifiseringsmodell fremfor objektdeteksjonsmodell for deteksjon av optiske forstyrrelser i bildene.

**Summary:**

The aim of this bachelor thesis has been to explore real time object detection for iSi AS inSight system. The project has resulted in a platform for real time object detection intended for the data gathering vehicle of iSi Insight. The system architecture is based upon a NVIDIA Triton Inference Server which runs on a NVIDIA Jetson Orin Nano. The server deploys two machine learning models used for detecting guard rails and optical disturbances in the incoming image data. The conclusion of the report is that real time object detection on iSi Insight is possible, but suggests using a classification model rather than an object detection model for detection of optical disturbances.



**Stikkord:**
Sanntid, Objektdeteksjon, Datavalidering, Maskinlæring, Datasyn

**Keywords:**
Real-time, Object detection, Data validation, Machine Learning, Computer vision

# Forord

Hensikten med dette prosjektet er å utforske muligheten for å videreutvikling av det eksisterende iSi inSight systemet. Rapporten vil gjennomgå utviklingen av en plattform for å innføre funksjonalitet for utnyttelse av maskinlæring i sanntid. Plattformen er utviklet for å validere datainnsamlingsprosessen til systemet i sanntid. Hensikten er å avgjøre om slik funksjonalitet er mulig å tilføre, samt om det er hensiktsmessig på systemet. Prosjektet er en bacheloroppgave med oppstart i januar 2024, og ble ferdigstilt i Mai 2024. Gjennomførelsen er utført av 4 studenter ved Norges tekniske og naturvitenskaplige universitet(NTNU) i Trondheim, i samarbeid med iSi AS og institutt for teknisk kybernetikk ved NTNU.

Det uttrykkes en spesiell takk til Christian Fredrik Sætre, Senior dataforsker i iSi AS og veileder for prosjektgruppen. Videre ønsker vi å takke Lars Ole Alstad, Senior systemutvikler i iSi AS, og iSi AS for bidrag med nødvendig utstyr for prosjektgjennomførelsen, nyttig informasjon om iSi inSight systemet, og veiledning i prosjektperioden.

# Innhold

# Figurer

# Tabeller

# Sammendrag

*"iSi AS er en IT-bedrift i vekst som jobber med digitalisering av arbeidsprosesser for å effektivisere og automatisere prosesser for bedrifter"*[1]. iSi AS har utviklet et system for digital rekkverkskontroll, kalt iSi inSight. Systemet gjennomfører datainnsamling ved å ta bilder av rekkverk langs det norske veinettet. Deretter gjennomføres typedeteksjon på bildene for å avdekke eventuelle feil.

iSi har bemerket at enkelte bilder kan være utsatt for optiske forstyrrelser. Dette kan medføre at feil ved rekkverket ikke oppdages av typedeteksjonsmodellene. I slike tilfeller kan det være nødvendig å dokumentere en strekning på nytt. Dette er kostbart, og tidkrevende. Bedriften har derfor ytret ett ønske om å utforske muligheten for å benytte maskinlæringsmodeller lokalt på innsamlingsriggen for å kvalitetssikre datainnsamling i sanntid. Hensikten er å varsle sjåføren om forekomst av forstyrrelser, slik at disse kan utbedres når de oppstår. På denne måten vil potensielt tap av verdifull data reduseres, og systemet vil effektiviseres.

Prosjektet ønsker å svare på følgende problemstillinger:

1. Utforsk muligheten for å gjennomføre objektdeteksjon i sanntid, lokalt på iSi-Insight systemet.
2. Utred i hvilken grad sanntids-objektdetektering kan benyttes for å oppdage feil i datainnsamlingsprosessen.

Prosjektets hensikt har vært å utvikle en plattform for sanntids objektdeteksjon tiltenkt å implementeres på datainnsamlingsbilen i inSight systemet. Prosjektgjennomføringen har innebefattet utvikling av programvare og systemarkitektur, og trening og testing av maskinlæringsmodeller. iSi AS har disponert en laptop med NVIDIA T600 og annotert 1166 bilder for trening av maskinlæringsmodeller. Det har også blitt utdelt en NVIDIA Jetson Orin Nano som utgangspunktet for systemarkitekturen.

For å undersøke problemstilling 1 ble det besluttet å sammenlikne objektdeteksjonsmodellene YOLOv8(n,s,m,l) og RT-DETRl. Hensikten med dette var å kartlegge ulike modellstørrelsers ytelse ved metrikken F1-score, og få innsikt i hvordan modellene fungerer som del av et sanntidssystem. For at en modell skal kunne anvendes på plattformen i sanntid, må systemet som en helhet kunne prosessere 20.5 bilder i sekundet. Dette tilsvarer bildefrekvensen levert fra kamerariggen når datainnsamlingsbilen kjører i 80km/t. Bildefrekvensen plattformen kan håndtere ble derfor sammenliknet med de ulike modellene implementert, i den hensikt å kartlegge hvilke modeller som oppfylte dette kravet.

I henhold til problemstilling 2 ble det besluttet å undersøke om objekteteksjonenes *confidence* kunne indikere om det var optiske forstyrrelser i bildet. Det ble ytterligere implementert en klassifiseringsmodell for å bestemme hvilken type optisk forstyrrelse som var tilstede i bildet. For å trene denne modellen ble det utviklet syntetisk data for å simulere forstyrrelser på bildene, med utgangspunkt i datasettet tildelt av iSi.

Prosjektet har resultert i en modulær og skalerbar plattform som kan implementeres på iSi inSight. Plattformen består av en NVIDIA Triton Inference Server som kjører på en Nvidia Jetson Orin Nano. Serveren er satt opp med en objekt- og en feildeteksjonsmodell som respektivt detekterer rekkverkstype og eventuelle optiske forstyrrelser i bildet. Ytterligere nødvendig bildeprosessering gjennomføres også på serveren. Plattformen består også av en klientapplikasjon tiltenkt å implementere på eksisterende maskinvare på datainnsamlingsbilen. Klienten sender bildedataen som samles inn av kamerariggen videre til serveren over gRPC, og presenterer de returnerte deteksjonene for sjåføren i et enkelt brukergrensesnitt. Ved detekterte optiske forstyrrelser loggføres disse med tilhørende tidsstempel og bildeadresse slik at datainnsamlingen kan ettergås.

Rapporten konkluderer med at objektdeteksjon gjennomført lokalt på innsamlingsriggen er mulig. Våre resultater viser at plattformen kan anvende komplekse modeller som YOLOv8l med half precision, og samtidig opprettholde hastighetskravet på 20.5 bilder per sekund. Det er ikke observert noen direkte sammenheng mellom *confidence score* ved rekkverksdeteksjoner og forekomst av

forstyrrelser i bildet. Det oppfattes derfor som mer egnet å kun bruke klassifiseringsmodellen for deteksjon av bildeforstyrrelse. Klassifiseringsmodellen demonstrerer evne til å korrekt klassifisere forstyrrelser og regnes i nåværende tilstand som et tilfredsstillende konseptbevis.

# Definisjoner

**Dyplæring** - Dyplæring er en type maskinlæring som bruker kunstige nevrale nettverk til å legge til rette for at digitale systemer kan lære og ta beslutninger basert på ustrukturerte, umerkede data [2].

**Sanntidssystem** - Ett sanntidssystem karakteriseres ved at logisk korrekthet ikke er eneste mål på ytelse, men også at beregninger leveres tidsnok.

**Sanntids objektdetektering** - Henviser i denne rapporten til dyplæringsmodeller, som med lav tidsforsinkelse, kan lokalisere og identifisere ulike objekter i et bilde.

**Annotasjon** - Informasjon om plassering og type for objekter i et bilde. Annotering benyttes av dyplæringsmodeller under trening.

**iSi AS** - iSi er en IT-bedrift i vekst som jobber med digitalisering av arbeidsprosesser for å effektivisere og automatisere prosesser for bedrifter.[1]

**iSi Insight** iSi AS sin innovative løsning for digital inspeksjon av rekkverk langs vegnettet. Med moderne teknikker basert på kunstig intelligens og maskinlæring kan type og tilstand kartlegges raskt, effektivt og med høy kvalitet.[3]

**inferens** - I AI feltet er inferens prosessen som en trent maskinlæringsmodell bruker for å dra konklusjoner om ny data.[4]

**Overfitting** betyr at en modell er for spesialisert på treningssettet, og mangler en generalisert *forståelse* av liknende oppgaver.

**Underfitting** skjer når en modell ikke er tilstrekkelig trent. dette kan komme av for lite datasett, for liten varians i datasettet, eller for kort treningsperiode.

**Ground truth** Annoterte objekter eller bilder

**Features** Typiske trekk/kjennetegn ved ett objekt.

**Half Precision/FP16** Flyttall som okkuperer 16 bit

**Tensor** Flerdimensjonell matrise.

**Hyperparametere** Variabler som definerer hvordan treningen av maskinlæringsmodeller gjennomføres.

**Bildeforstyrrelse** henviser i denne rapporten til optiske forstyrrelser eller feil i ett bilde

**Epoke** En fullstendig gjennomgang av alle bildene i et datasett under trening.

**Batch-size** En gruppering av bilder.

**Boundingbox** Refererer til bokskordinater som omslutter ett objekt i ett bilde. Boksene brukes til å bestemme et objekt sin posisjon i bildet

**FLOPs** *Floating point operations* viser til antallet flyttall operasjoner som må gjennomføres i en dyplæringsmodell før ett svar er klart.

**FLOPS** *Floating point operations per second* viser til antallet flyttall operasjoner en GPU kan gjennomføre per sekund.

**Kost funkjson** er en matematisk funksjon som beregner avviket mellom den estimerte utgangen fra ett nevralt nettverk og den ønskede utgangsverdien

**Backpropagation** er en algoritme som brukes i maskinlæring for å justere vektene i nettverket, basert på kost funksjonen[5]

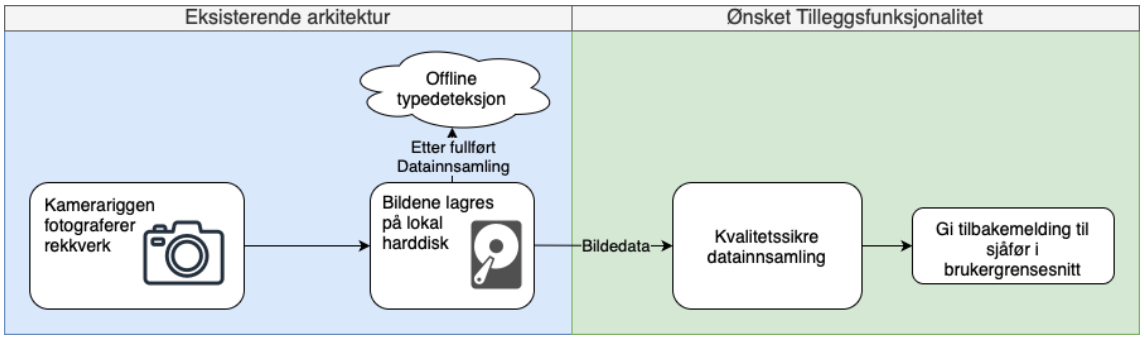
**RT-DETR** Real-Time Detection Transformer

**Semafor** Er en datatype brukt til å kontrollere tilgang til en felles ressurs

# 1 Introduksjon

## 1.1 Bakgrunn

iSi inSight er en løsning for digital inspeksjon av rekkverk langs vegnettet. inSight gjennomfører datainnsamling, og benytter maskinlæring for å gjenkjenne rekkverk- stolpe- og bolttype, samt eventuelle defekter på rekkverket. Slik systemet er satt opp i dag, gjennomføres inspeksjonen offline, etter at dataen er innsamlet. Under datainnsamlingsprosessen kan bilder bli utsatt for optiske forstyrrelser som påvirker bildekvaliteten. Slike bilder kan redusere presisjonen maskinlæringen kan levere, og innføre usikkerhet i resultatene. I slike tilfeller kan det bli nødvendig å gjennomføre datainnsamling ved et veistrekke på nytt. Dette kan være kostbart, og tidkrevende. Bedriften ønsker i denne sammenheng å undersøke muligheten for å gjennomføre objektdeteksjon i sanntid, lokalt på innsamlingsbilen. Videre ønsker bedriften innsikt i om objektdeteksjon kan anvendes for å oppdage forstyrrelser på bildedataen i sanntid. Hensikten er å varsle sjåfør om forekomst av optiske forstyrrelser i datainnsamlingsprosessen. Målet er at sjåfør kan utbedre forstyrrelser når disse oppstår. Dette vil trolig bidra til en sikrere innsamling og begrense fremtidig verditap i innsamlingsprosessen.




Figur 1: Overordnet arkitektur på iSi inSight med ønsket sanntidsfunksjonalitet

Datainnsamlingen gjennomføres av en bil med en påmontert kamerarigg. Bilen har en operativ hastighet på 80 km/t. Kamerariggen består to optiske kamera, og et infrarødt kamera. Optikken er koblet til et triggerkort som sender et signal om at kameraene skal ta bilde hver 1.09 meter. Ved operativ hastighet på 80km/t tilsvarer dette at hvert kamera tar 20.5 bilder per sekund. Bilen er utstyrt med en PC med 3x8TB harddisker til drift, og 1x8TB harddisk til forflytning. Når et bilde blir tilgjengeliggjort i kameraenes buffer, overføres det til drift-harddisken. Ved endt kjøring gjennomføres bildeprosessering, som komprimerer og overfører bildene til forflytning-harddisken. Bildedataen blir deretter fysisk flyttet til iSi sine kontorer, og lastes opp i skyen. Etter at dataen tilgjengeliggjøres benyttes maskinlæringsmodeller for å kartlegge rekkverk-, stolpe- og bolttyper. Modellene kartlegger også eventuelle feil og defekter ved rekkverket.


Figur 2: Fotografi av Matti Bernitz, gitt med tillatelse av iSi AS

## 1.2 Problemstilling

Formålet med oppgaven er å sette opp en plattform for sanntids objektdeteksjon som en del av iSi AS sitt Insight system. Som del av dette skal det settes opp en infrastruktur som prosesserer innkommende data, drifter sanntidsmodeller og gir tilbakemeldinger til sjåføren gjennom et brukergrensesnitt. Det er også tiltenkt at ytelsen til sanntids objektdeteksjonsmodeller skal kartlegges for å se om de kan implementeres på en slik platform. Videre ønskes det at prosjektet utforsker om plattformen kan benyttes til å detektere forstyrrelser i bildedataen. Oppgaven sammenfattes til følgende 2 problemstillinger:

1. Utforsk muligheten for å gjennomføre objektdeteksjon i sanntid, lokalt på iSi inSight systemet.
2. Utred i hvilken grad sanntids-objektdeteksjon kan benyttes for å oppdage feil i datainnsamlingsprosessen.

### 1.2.1 Systemkrav

I den hensikt å konkretisere problemstilling 1, er det formulert 3 krav den ferdige sanntidsplattformen må oppfylle for å tilfredsstille problemstillingen.

1. Plattformen skal være modulær, og kompatibelt med dagens iSi inSight system
2. Maskinlæringsmodellen må være rask nok til at platformen kan prosessere 20,5 bilder i sekundet[1].
3. Maksimere modellnøyaktighet innenfor hastighetskravet basert på F1-score.

Tilsvarende, er det spesifisert ytterligere 2 spesifikasjonskrav som på oppfylles for at problemstilling 2 kan regnes som oppfylt.

4. Plattformen skal kunne detektere bilder med forstyrrelser.
5. Relevant informasjon fra modellen presenteres oversiktlig til sjåføren.

[1] Bildefrekvens ved operativ hastighet

## 1.3 Vår løsning

Prosjektets formål er å etablere en modulær og skalerbar plattform for sanntids verifisering av rekkverksbilder, tiltenkt å implementeres på det eksisterende Insight systemet. Plattformen vil benytte sanntids-objektdeteksjons for å gjenkjenne rekkverk ved skinne-, stolpe- og bolttype. Prosjektet ønsker å kartlegge hvor komplekse modeller som kan anvendes i et slikt sanntidssystem.

Prosjektet ønsker videre å undersøke om confidence ved modellenes prediksjoner kan benyttes for å avgjøre om det er forstyrrelser i bildene. Ved å presentere rekkverks-deteksjoner og tilhørende confidence i sanntid, vil sjåføren kunne vurdere at det er forstyrrelser i bildet hvis confidence er for lav, eller systemet detekterer feil rekkverk. Det er også tiltenkt å parallelt implementere en klassifiseringsmodell som vil benyttes for å bestemme hvilken type forstyrrelse som forårsaket lav confidence.

## 1.4 Begrensninger

Under prosjektgjennomføringen har det blitt tatt hensyn til enkelte begrensinger. Det vil senere i rapporten diskuteres hvordan disse har påvirket resultatet.

1. Det annoterte datasettet som har blitt brukt til trening av maskinlæringsmodellene har bestått av 1166 bilder. Dette har satt en begrensning på hvor god modellnøyaktighet det har vært mulig å oppnå. Det presiseres derfor at resultatmetrikkene presentert i denne rapporten ikke er representative for maskinlæringsmodellenes fulle potensial, men heller vil brukes som sammenlikningsgrunnlag mellom modeller trent på samme premisser.
2. Til trening av maskinlæringsmodellene har prosjektet brukt to bærbare PCer med respektive GPUer, Nvidia T600 Laptop og Nvidia GTX 1660 Ti. Dette har satt en begrensning på den totale treningsmengden som har kunnet bli gjennomført på maskinlæringsmodellene.
3. Under prosjektutførelsen ble det tatt i bruk et *NVIDIA Jetson Orin Nano Developer kit*[6]. Denne maskinvaren vil begrense prosesseringshastigheten til systemet.
4. PCen som bil-riggen er utstyrt med tar i bruk en Intel(R) Xeon(R) E-2278GE CPU. Under Prosjektutførelsen har det ikke vært tilgang på denne PCen. Det har derfor blitt tatt i bruk en ASUS Zenbook 14X OLED (UX3404) med 13th Gen Intel i7-13700H CPU. PCen brukt under prosjektutførelsen vil prestere bedre enn PCen i Insight-systemet. [7]. Klientmaskinen som brukes under dette prosjektet kjører på ubuntu 22.04 LTS operativsystemet.

## 1.5 FNs bærekraftmål

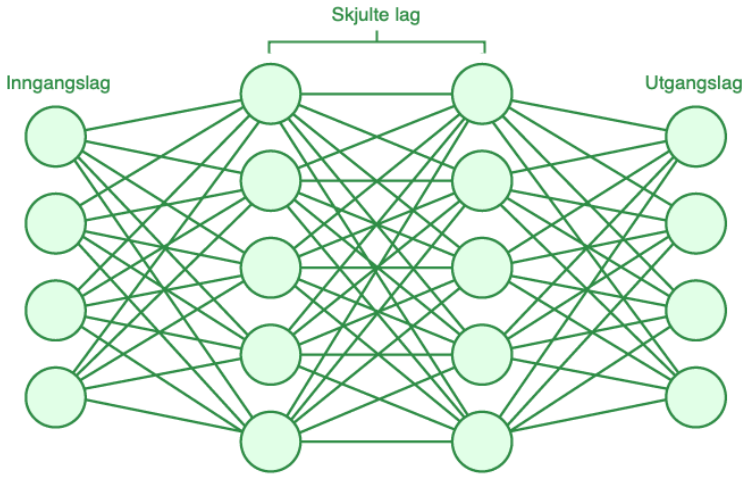
iSi inSight bidrar i dag til arbeidet med å sikre norske veier ved å detekter defekter på rekkverk. Målet er at arbeidet i dette prosjektet vil bidra til å gjøre dette produktet enda bedre i fremtiden. I den hensikt har det blitt vurdert som mest relevant å fokusere på de bærekraftsmålene som rettes mot å gjøre dagens veier tryggere. Fra dette har det blitt valgt 2 delmål fra FNs bærekraftsmål [8].

- **3.6** - Mål nr. 3: God helse og livskvalitet. Delmål nr.6 sikter å halvere antall dødsulykker og skader forårsaket av trafikkulykker globalt innen 2030.
- **9.1** - Mål nr.9: Industri, innovasjon og infrastruktur. Delmål nr.1 omhandler arbeid for utvikling av pålitelig, bærekraftig og solid infrastruktur.

# 2 Teori

## 2.1 Kunstige nevrale nettverk

Et nevralt nettverk er bygd opp av sammenkoblede noder. Sammenkoblingen representerer muligheten for sending av data mellom nodene. Noden i seg selv representerer gjennomføringen av en operasjon. Nodene er organisert i lag. Et vanlig nevralt nettverk består av inngangs-, skjulte- og utgangslag. Hvis et nevralt nettverk har to eller flere skjulte lag kalles det et dypt nevralt nettverk. Hvor mange lag, størrelsen på lagene og hva slags noder de består av, beskriver nettverkets arkitektur. Hvilke arkitektur som benyttes vil bestemme hvilke oppgaver nettverket har evne til å løse.




Figur 3: Eksempel arkitektur dypt nevralt nettverk[9]

Sammenkoblingene mellom nodene i de forskjellige lagene påvirkes av vekter. Når en node mottar informasjon fra det forrige laget kalkulerer den det vektede signalet den skal sende videre. På denne måten kan vektene styre hvilken data som prioriteres mens dataen flyter gjennom nettverket. Når nettverket trenes vil validiteten til en respons bli vurdert ved hjelp av en kost funksjon. Resultatet fra dette vil være et tap som vil bli brukt med *backpropagation* til å justere vektene i nettverket.[10][9]

## 2.2 Objektdeteksjonsmodeller

Objektdeteksjon kan deles opp i to oppgaver: klassifisering og lokalisering. Klassifiseringsdelen går ut på å finne ut hvilken klasse et objekt i et bilde tilhører. Lokalisering handler om å finne ut hvor i bildet et objekt befinner seg. Objektdeteksjon ønsker å utføre begge disse oppgavene simultant.

For å trene en objektdeteketeringsmodell krever det at man har et datasett med forhåndsbestemte *ground truths*. En *ground truth* vil for et objektdeteksjonproblem være en boks rundt et objekt definert med en tilhørende klasse. Treningen foregår overordnet ved at modellen gjennomfører prediksjoner på et bilde ved å lage sine egne bokser, også kalt *bounding bokser*. *Bounding boksene* vil så bli designert en sannsynlighet for at de tilhører en klasse. Etter at dette er gjort vil prediksjonen sammenliknes med *ground truth* og det gis en tilbakemelding på hvor godt disse stemmer overens. Denne tilbakemeldingen kalles et tap og kan bestå av flere deler. For objektdeteketeringsmodeller er de viktigste typene tap: klasse og boks. Klassetap definerer hvor god modellen er til å predikere klasser og bokstap definerer om hvor godt bounding boksene er plassert.

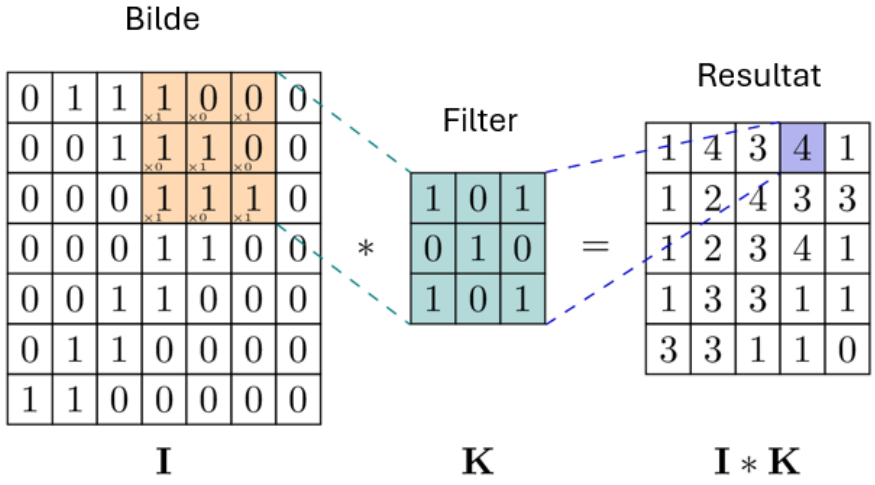
## 2.3 Konvolusjonelle nevrale nettverk(KNN) og feature extraction

For å klassifisere og lokalisere objekter bruker objektdeteketeringsmodeller et konsept kalt feature extraciton. Ekstraksjonen gjennomføres av et KNN og blir ofte kalt ryggrad som en del av den større objektdetekteringsmodellen. Et KNN følger de samme prinsippene som et nevralt nettverk, men ekspanderer på konseptet ved å implementere ekstra funksjonaliteten i de skjulte lagene. De skjulte lagene i et KNN består hovedsakelig av tre typer: konvolusjon(2.3.1), activation og max pool [11]. I en typisk KNN-modell vil bildet først gå gjennom et konvolusjonslag, deretter et activation lag og til slutt et max-pool lag. I større KNN-modeller vil denne prosessen gjentas flere ganger med forskjellige typer av de tre lagene.

Når en KNN-modell prosesserer et bilde er det vanlig å presentere det som en tredimensjonal tensor. Dimensjonene vil respektivt representere: høyde, bredde og RGB/gråtone-verdi. Etter hvert som bildet prosesseres vil de romlige dimensjonene (høyde, bredde) reduseres. Den tredje dimensjonen, kalt kanaldimensjonen vil derimot øke. Denne dimensjonen vil representere mange versjoner av bildet som inneholder forskjellige type features. Bildene kalles derav feature maps. [12][13]

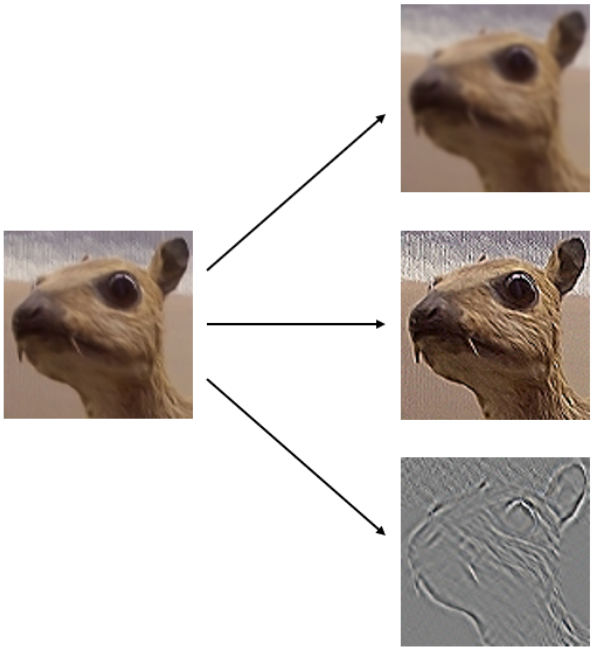
### 2.3.1 Konvolusjonslag

Konvolusjon gjennomføres med to komponenter. For vårt eksempel vil dette være det originale bildet på matriseform I(mxm) og et mindre filter K(nxn). Filteret vil gli over I-matrisen og ta indreproduktet av seg selv med nxn-delen av I-matrisen den overlapper ved hvert steg. Resultatet av hvert steg vil bli en verdi i en resultatmatrise.




Figur 4: Konvolusjon gjennomføres av filtermatrise i midten på bildematrisen til venstre. Til høyre lagres resultatet av operasjonen i en resultatmatrise.[14]

Hver pikselverdi i det originale bildet vil bli påvirket av sine naboer når det gjennomgår konvolusjonsprosessen. Med et filter hvor alle vektene er like vil områder med høye verdier få enda høyre verdier, og motsatt for områder med lave verdier. Visuelt vil dette gjøre raske endringer i farger eller lysstyrker blir mer fremtredende. Det er også mulig å oppnå andre effekter ved å variere filterets steglengde, størrelse og verdier. Mest vanlig er bruken av en steglenge lik 2. Dette vil gjøre at de romlige dimensjonene til resultat tensoren er halvparten av det originale bildet.[15][11]



Figur 5: Originalt bilde til venstre. Eksempelbilder etter konvolusjon til høyre: blur, sharpen, edge detection. Bilder hentet fra[16]

## 2.4 Generelle Datasett

### 2.4.1 ImageNet

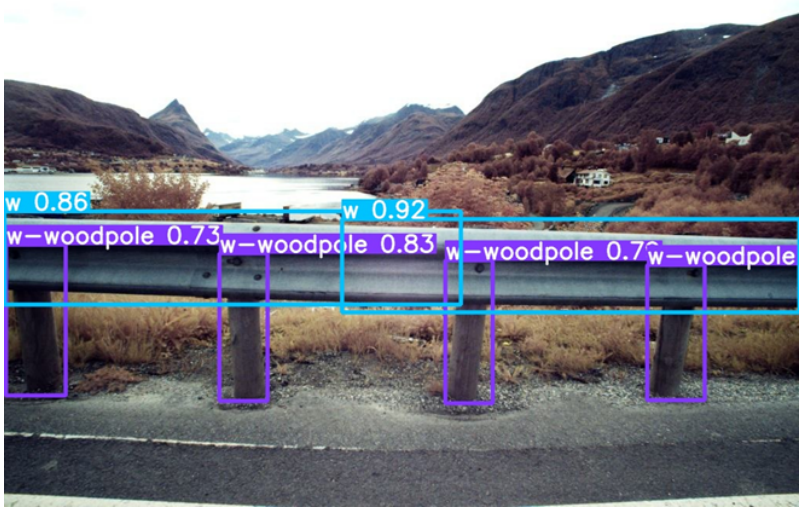
ImageNet dataset[17] er ett datasett med mer enn fjorten millioner manuelt annoterte bilder, fordelt på tusen klasser. Av disse bildene er mer enn en million bilder annotert med *bounding bokser*.

### 2.4.2 COCO

COCO, Micsosoft Common Objects in Context [18] er ett stort datasett på om lag 328 000 bilder, hvorav 200 000 bilder er annoterte med *bounding bokser*. Datasettet er utviklet for trening av objektdeteksjon og segmenteringsmodeller.

## 2.5 YOLO(You only look once)

YOLO er en dyplæringsmodell som tillater for hurtig objektdetektering. Modellen er et ende til ende nevralt nettverk som kun trenger at bildet går gjennom modellen en gang før den kan gjøre en prediksjon. Dette gjør modellen veldig rask og revolusjonerte objektdetekertingsfeltet når den første versjonen ble lansert i 2015.




Figur 6: Eksempel på resultat fra en YOLO objektdeteksjonsmodell

YOLO er en konvolusjonsbasert modell. Dette betyr at modellen baserer seg på konvolusjon for å lage feature maps som den bruker til å utføre prediksjoner. Modellen beskrives ofte som å være delt inn i 3 deler: Ryggrad, nakke og hode. Arkitekturen til de forskjellige delene har variert i hver versjon, men den overordene strukturen har holdt seg lik. Videre vil arkitekturen til til YOLO versjon 8 forklares.

Ryggraden er en modifisert versjon av KNN CSPDarknet53. Nettverket fungerer i grove trekk likt som et vanlig KNN hvor hovedoppgaven er å utføre feature extraction(2.3). Den største forskjellen ligger i implementasjonene av en C2f modul som splitter opp tensoren i biter, gjennomfører konvolusjon på bitene før de settes sammen igjen. Utgangen fra ryggraden vil bestå av tre tensorer med dimensjonene (20x20x512), (40x40x512) og (80x80x256) gitt en inputtensor på formen (640,640,3).

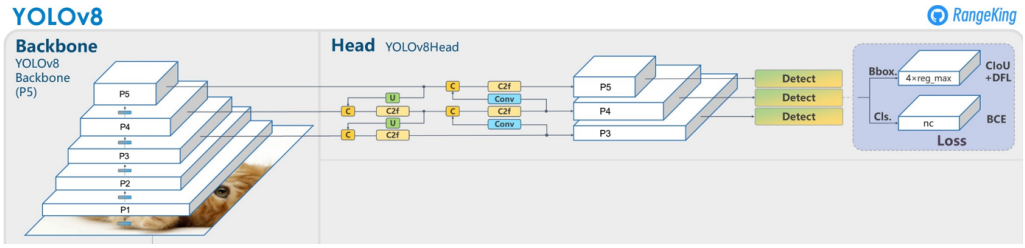
Nakken har som oppgave å kombinere feature maps fra de forskjellige nivåene. Dette gjøres ved hjelp av konsepter fra feature pyramid network(FPN) og Path aggregation network(PAN)[19][20]. Begge disse arkitekturene ønsker å sette sammen feature maps fra forskjellige romlige dimensjoner. Først bruker nettverket FPN-konseptet ved å ta de mindre romlige outputene fra ryggraden, skalere opp disse, og slå de sammen med de større outputene. PAN-konseptet gjør dette, men i motsatt retning.

For å redusere de romlige dimensjonene i denne prosessen brukes konvolusjonslag. De tre forskjellige utgangene fra ryggraden beholder sine rommelige dimensjoner gjennom nakken og sendes videre til hodet. Hodet tar inn de tre utgangene fra nakken og bruker disse til å gjennomføre prediksjoner for klasser og *bounding bokser*. Grunnen til at det gjennomføres prediksjoner på tre forskjellige størrelser er for at modellen skal se små, mellomstore og store objekter. Etter at prediksjonene er gjennomført kalkuleres *loss* ved å benytte en kostfunksjon.

YOLOv8 bruker CIoU[21] og DFL-loss[22] for bounding boxes og BCE-loss for klasser[23]. Etter at modellen har kalkulert *loss* vil klasse og *bounding boks* prediksjonene gå gjennom en postprosesserings algoritme kalt **NMS(non maximum supression)**. NMS itererer gjennom alle prediksjonene, og filtrerer ut overlappende bounding boxes. Hvor mye overlapp som tillates avgjøres av et IoU-terskel. Alle *bounding bokser* med større overlapp enn IoU-terskel vil behandles som samme prediksjon. NMS-algoritmen vil beholde bounding boxen med høyest confidence, og filtrere ut resten.

Det presiseres at dette er en overordnet og forenklet forklaring av modellen. For en dypere innsikt i arkitekturen til YOLO-modellen refereres til rapportene ([24],[25]) og Ultralytics sine sider([26])[1].

[1]Det er ikke i skrivende stund utgitt en offisiell rapport fra Ultralytics for YOLOv8




Figur 7: Forklarende skisse for YOLOv8 arkitektur[27]

## 2.6 YOLO-Objektdeteksjon datasett format

Yolov8 objektdeteksjon benytter ett spesifikt datasettformat som kalles YOLO-format[28]. Formatet benytter en .yaml-fil for å henvise til filplasseringen til dataen. For at .yaml-filen skal kunne hente dataen på riktig måte, må den settes opp i følgende mappestruktur.

```
dataset
├── train
│   ├── images
│   │   ├── bilde1.jpg
│   │   ├── bilde2.jpg
│   │   └── ...
│   └── lables
│       ├── bilde1.txt
│       ├── bilde2.txt
│       └── ...
├── val
│   ├── images
│   │   ├── bilde1.jpg
│   │   ├── bilde2.jpg
│   │   └── ...
│   └── lables
│       ├── bilde1.txt
│       ├── bilde2.txt
│       └── ...
└── test (optional)
```

Mappestrukturen er inndelt i *train* og *val*, som inneholder bildedata og annoteringsfiler til henholdsvis trening og validering. Annoteringen må formateres som tekst-filer, hvor hver fil inneholder annoteringsdata for et enkelt bilde. Annoteringen må videre ha samme filnavn som bildet det annoterer, og ligge på samme plass som det respektive bildet. Annoteringsfilene inneholder klasse og bounding boks for hvert objekt i bildet. Bounding boxer må noteres på *xywh*-format. *xy* henviser til pixelkordinatet i senteret av bounding boxen, og *wh* er bredden og høyden i pixler.

### 2.6.1 YOLO klassifisering

YOLO klassifisering er en *enkeltklasse* bildeklassifiseringsmodell, optimalisert for å klassifisere hele bilder. Modellen identifiserer effektivt innholdet i ett bilde, uten hensyn på innholdets plassering i bildet. Modellen presenterer høy ytelse innen presisjon og hastighet. [29]

| Model | Size (Pixels) | acc top1 | acc top5 | Speed CPU ONNX (ms) | Speed A100 TensorRT (ms) | Params (M) | FLOPs (B) at 640 |
|---|---|---|---|---|---|---|---|
| YOLOv8n-cls | 224 | 69.0 | 88.3 | 12.9 | 0.31 | 2.7 | 4.3 |
| YOLOv8s-cls | 224 | 73.8 | 91.7 | 23.4 | 0.35 | 6.4 | 13.5 |
| YOLOv8m-cls | 224 | 76.8 | 93.5 | 85.4 | 0.62 | 17.0 | 42.7 |
| YOLOv8l-cls | 224 | 76.8 | 93.5 | 163.0 | 0.87 | 37.5 | 99.7 |
| YOLOv8x-cls | 224 | 79.0 | 94.6 | 232.0 | 1.01 | 57.4 | 154.8 |

Tabell 1: Yolov8-cls modellene[30] er trent og validert på ett utdrag av ImageNet Datasettet[31] på ca 1.3 millioner bilder til trening, og 50 tusen bilder til validering.

### 2.6.2 YOLO klassifisering format

YOLO klassifisering datastrukturen følger et spesifikt split-direcotryformat. Formatet består av en mappe for henholdsvis treningsdata, valideringsdata og testdata. hver mappe, har en undermappe for hver klasse. I disse mappene lagres bildene tilhørende de respektive klassene[32].

```
dataset
├── train
│   ├── klasse1
│   │   ├── bilde1_1.png
│   │   ├── bilde1_2.png
│   │   └── ...
│   └── klasse2
│       ├── bilde2_1.png
│       ├── bilde2_2.png
│       └── ...
└── test
    ├── klasse1
    │   ├── bilde3_1.png
    │   ├── bilde3_2.png
    │   └── ...
    └── klasse2
        ├── bilde4_1.png
        ├── bilde4_2.png
        └── ...
```

Som følge av at YOLO klassifisering er en *en-klasse* klassifiseringsmodell, vil mappenavnet fungere som klassenavnet. Dette skiller seg fra objektdeteksjon formatet (2.6), ved at det ikke er behov for annoterte *bounding bokser*.

## 2.7 RT-DETR

RT-DETR er en transformer basert sanntids objektdeteksjonsmodell. RT-DETER har sammenliknet med YOLO en mer strømlinjeformet arkitektur som reduserer bruken av modellspesifikke komponenter. RT-DETR modellen bruker for eksempel ikke NMS(*Non-maximum suppersion*). Dette gjør at modellen sparer tid og slipper å designere parametere til denne spesifikke oppgaven. RT-DETR modellen består overordnet av tre deler: ryggrad, enkoder og dekoder. Ryggraden er er et KNN som er ansvarlig for *feature extraction*. Transformer definisjonen til modellen kommer fra bruken av en enkoder og dekoder. Enkoder delen av nettverket reduserer den tredimensjonale tensoren(høyde,bredde,kanal) den får fra KNN ned til to dimensjoner (Høyde*bredde,kanal). På denne måten tar den features fra den tredimensjonale matrisen og gjør dette til en sekvens med features. Deretter benyttes en seleksjonsalgoritme til å bestemme hvilke features som skal sendes videre til dekoderen som objektprediksjonsforslag. Dekoderen tar disse prediksjonsforslagene og iterativt optimaliserer de før den gjør avsluttende prediksjoner på klasser og bounding bokser. Det presiseres at dette er en overordnet og forenklet forklaring av modellen. For en dypere innsikt i

arkitekturen til RT-DETR modellen vil utfyllende informasjon være oppgitt i rapportene [33] [34].

## 2.8 Half precision

Half precision henviser til et flyttall som okkuperer 16 bit. Navnet refererer til at datatypen benytter halvparten så mange bits som FP32, et flyttal bestående 32 bit.

Maskinlæringsmodeller kan konfigureres til å benytte FP32, eller half precision (FP16). Ved å benytte 16 bit vil inferenshastigheten til modellen øke fordi aritmetiske operasjoner er raskere med mindre datatyper. Nøyaktigheten til maskinlæringsmodellen vil derimot bli lavere fordi hvert datapunkt inneholder mindre data. [35]

## 2.9 Ytelsesmetrikker

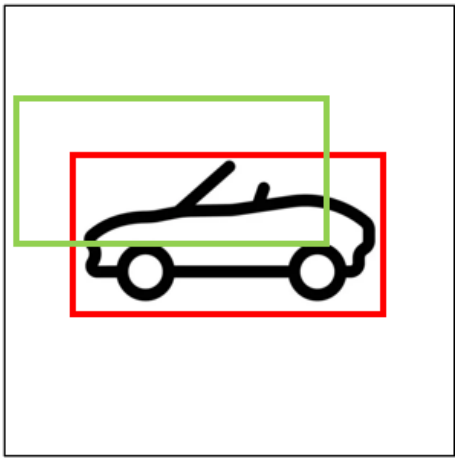
For å kvantifisere ytelsen til en objektdetekteringsmodell finnes det en rekke metrikker. Rapporten vil videre forklare hvordan de metrikkene som er relevante for prosjektet er definert og hvilke nytte de gir. Informasjon om metrikkene som presenteres videre i seksjonen er hentet fra [36] og [37].

### 2.9.1 IOU

**Intersect Over Union** referer til hvor stor del av den predikerte boundig boksen som overlapper med ground truth boksen. Boks loss kan beregnes ved å bruke UIO til å kvantifisere hvor godt en bounding boks er plassert.

$$IOU = \frac{Area\ of\ overlap}{Area\ of\ union}$$

Bildet under viser ground truth boksen i rødt og den predikerte bounding boxen i grønt.



Figur 8: Visuelt eksempel på 0.5 IOU

### 2.9.2 Prediksjonstyper

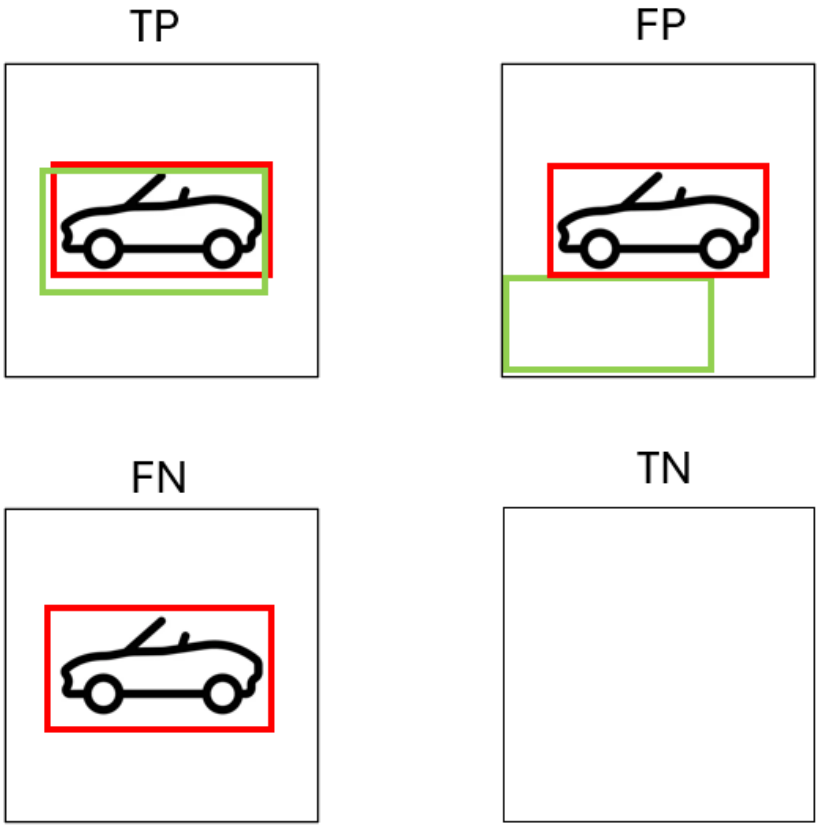
En prediksjon kan klassifiseres på fire forskjellige måter. Disse er forklart under.

**True positive (TP)** - Modellen sin bounding boks prediksjon har et overlapp med ground truth boksen som overstiger UIO-terskelen

**False positive (FP)** - Modellen gjør en bounding boks prediksjon som ikke når UIO-terskelen for overlapp med ground truth boksen.

**False negative (FN)** - Modellen ikke gjøre en prediksjon på en tilgjengelig ground truth bounding boks.

**True negative (TN)** - Modellen fraviker fra å gjøre en prediksjon der det ikke er noen ground truth boks.



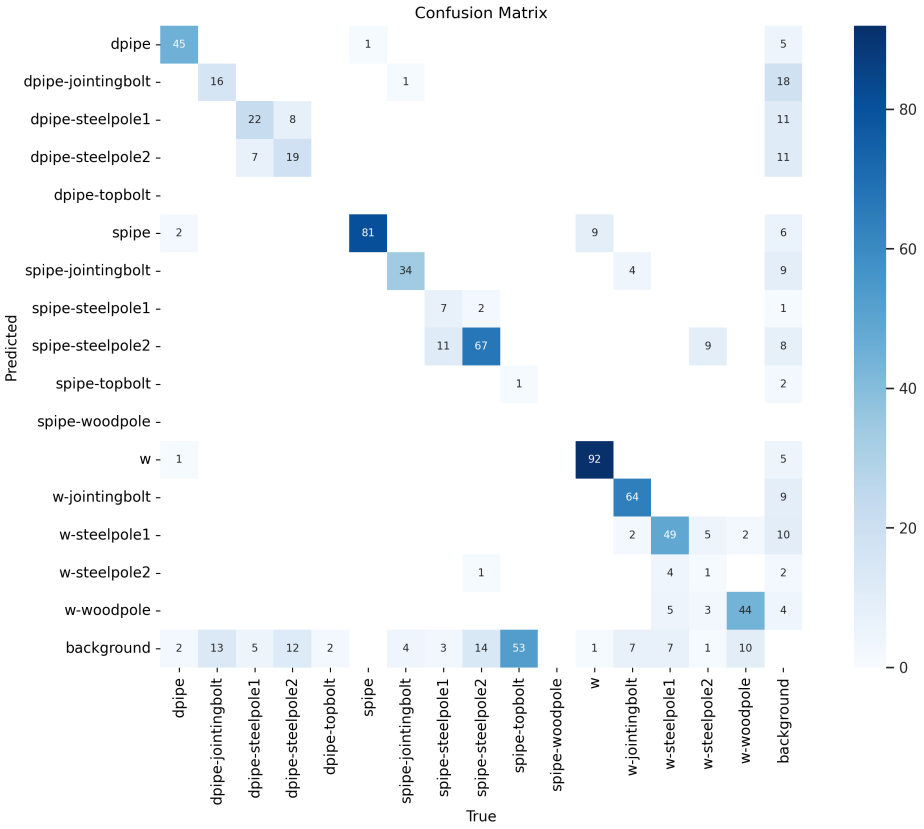

Figur 9: Eksempel på alle de fire prediksjons klassifiseringene

### 2.9.3 Condfidence (C)

Når en modell definerer en bounding boks vil denne ha en confidence verdi tilknyttet. Denne verdien sier noe om hvor sikker modellen er på at objektet i boksen tilhører en spesifikk klasse. Det er vanlig å sette en confidence terskel for modellen. Terskelen bestemmer at alle bounding bokser med mindre confidence skal forkastes.

### 2.9.4 Feilmatrise

Feilmatrisen presenterer alle prediksjonene til modellen slik at de predikerte klassene blir satt opp modt *ground truth* klassene. På denne måten visualiseres TP,FP og FN for de forskjellige klassene. Videre analyse kan gi dypere innsikt i hvilke klasser modellen er god og dårlig til å predikere.




Figur 10: Feilmatrise til en RT-DETR-modell trent på iSi-datasettet

### 2.9.5 Precision (P)

Presisjonen til en modell defineres ved å dividere antall korrekte prediksjoner på det totale antall prediksjoner. Dette gir da ett mål på treffsikkerheten til en modell. Høy presisjon indikerer få **False Positive (FP)** og betyr at en prediksjon gjort av modellen mest sannsynlig stemmer.

$$P = \frac{Korrekte\ prediksjoner}{Totale\ prediskjoner} = \frac{True\ Positives(TP)}{True\ Positives(TP) + False\ Positives(FP)} \tag{1}$$

### 2.9.6 Recall (R)

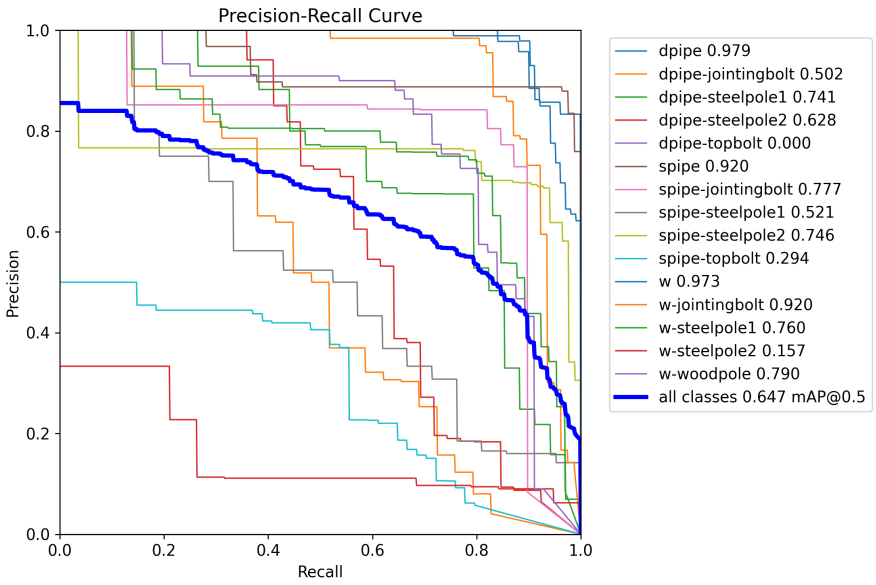
(Tilbakekall) defineres ved å dividere antallet korrekte prediksjoner på totalt antall instanser av *ground truth* bokser. Høy *recall* indikerer at modellen detekterer en høy andel av instansene til en klasse i datasettet.

$$R = \frac{Korrekte\ prediksjoner}{Totale\ instanser} = \frac{True\ Positives(TP)}{True\ Positives(TP) + False\ Negatives(FN)} \tag{2}$$

I et ideelt scenario ville precision og recall vært lik en. Da ville modellen predikert alle instansene uten noen *False positive (FP)* eller *False negatives (FN)*. Dette er sjeldent tilfelle, isteden oppstår det ofte en balanse mellom presisjon og recall. For at en modell skal oppnå høy recall krever det at den gir ut mange prediksjoner. Dette vil også øke sannsynligheten for at den bommer og dermed reduserer presisjonen. Confidence terskelen vil også ha en innvirkning på precision og recall. En høy confidence terskel vil føre til færre prediksjoner og dermed lav recall, men høy precision. En lav confidence terskel vil gi det motsatte.

### 2.9.7 PR-curve

PR-kurven plotter verdiene for precision og recall opp mot hverandre for confidence terskeler mellom 0-1. PR-kurven er et nyttig verktøy for å finne en god balanse mellom precision og recall. Et eksempel er vedlagt under.




Figur 11: PR-curve til en YOLO-modell trent på iSi-datasettet

### 2.9.8 FN-score

FN-score er en metode for å vekte precision og recall mot hverandre. For å regne ut FN-score tas det i brukt en variabel, $\beta$. $\beta > 1$ prioriterer recall, og $\beta < 1$ prioriterer precision. Vanlig notasjon for metrikken er F$\beta$-score (Eks. F1-score).

$$F\beta = (1 + \beta^2) \cdot \frac{P \cdot R}{\beta^2 P + R} \tag{3}$$

Ettersom F-score gir mulighet til å vekte forholdet mellom precision og recall, egner den seg godt for å vurdere modellytelse basert på oppgavespesifikke behov.

### 2.9.9 Average precision(AP)

Average precision er definert som arealet under PR-kurven. Denne metrikken gir et godt innblikk i hvordan modellen fungerer for hele spekteret av confidence terskler. Average precision er som regel kalkulert for hvert enkelt klasse.

$$Average\ Precision\ (AP) = \int_{r=0}^{1} p(r)dr \tag{4}$$

### 2.9.10 mAP(Mean Average Precision)

Mean average precision tar gjennomsnittet av AP for alle klassene. mAP defineres også ofte for spesifikke IOU eller confidence terskel, men kan også defineres for et verdispenn. De mest brukte er mAP50 og mAP50-95.

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{5}$$

- $N$ er antall klasser
- $AP_i$ er gjennomsnittlig presisjon for klasse $i$

AP og mAP er industriledende metrikker for objektdeteksjon. AP brukes som hovedmetrikk for evaluering av COCO-datasettet[38]. mAP brukes av Ultralytics til å sammenlikne sine modeller[26].

### 2.9.11 Top(N) accuracy

Når en klassifiseringsmodell klassifiserer ett bilde, returnerer den en sortert liste av alle mulige klasser modellen mener bildet kan høre til i. Listen er sortert med hensyn på *confidence score*, slik at høyeste *confidence* er sortert først i listen. Top(N) gir ett mål på hvor ofte en klassifiseringsmodell inkluderer den korrekte klassen i de N første elementene i den sorterte listen. Metrikken er en mye brukt metrikk i evaluering av klassifiseringsmodeller. [39]

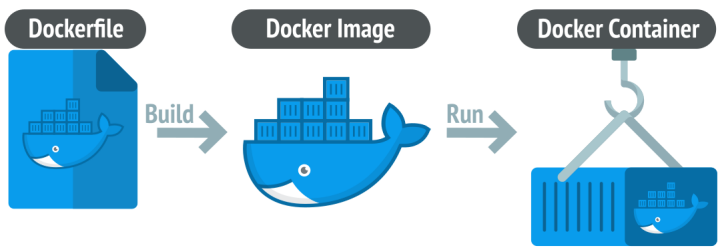
## 2.10 Grafikkprosessor

Grafikkprosessorer muliggjør utførelsen av mange operasjoner parallelt. I motsetning til en CPU som ofte er begrenset til mellom 2 og 8 kjerner kan en grafikkprosessor ha flere tusen. Ved å ta i bruk en grafikkprosessor kan operasjonene som bygger opp en maskinlæringsmodell parallelliseres, som drastisk vil øke hastigheten til modellen[40].

## 2.11 Docker

Docker er en programvareplattform som tilrettelegger for rask og enkel testing, utvikling og utplassering av applikasjoner[41]. Docker platformen består i hovedsak av tre elementer: *image*, *container* og *dockerfile*.

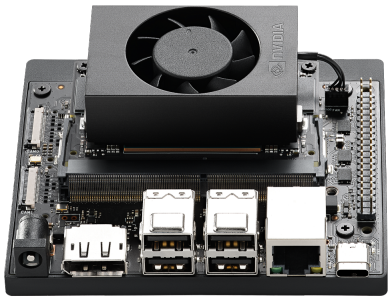
En Container er et isolert miljø med all nødvendig programvare tilgjengelig for å kjøre en applikasjon. Dette muliggjør for enkel kjøring av applikasjonen på tvers av maskinvare. Et Docker Image er en fil som inneholder all informasjon om bibliotekene og avhengighetene som en container trenger for å kjøre et program. En dockerfil er et tekstdokument som inneholder instruksjonene for å lage et image. filen støtter en rekke funksjoner som gjør det mulig for utvikleren og spesifisere applikasjonsmiljøet. Mange programmer krever versjonspesifikke biblioteker og avhengigheter for å kjøre. Docker løser dette problemet ved at applikasjonene utplasseres i ferdig konfigurerte miljø slik at brukeren slipper konfigurasjonsprosessen[42].




Figur 12: Docker utviklingsprosess [43]

## 2.12 Nvidia jetson

NVIDIA Jetson er den ledende plattformen for autonome maskiner og embeddede aplikasjoner (NVIDIA", n.d., avsnitt 1)[44]. Jetson produktene kan settes opp med ferdigutviklede *software development kit* kalt Jetpack SDK, som er spesiallaget for kjøring av KI applikasjoner på jetson platformen. Dette inkluderer bland annet Docker [2.11] for enkel installasjon og oppsett av applikasjoner, samt cuda[45] og tensorrt[46] for GPU prosessering og modell optimalisering.



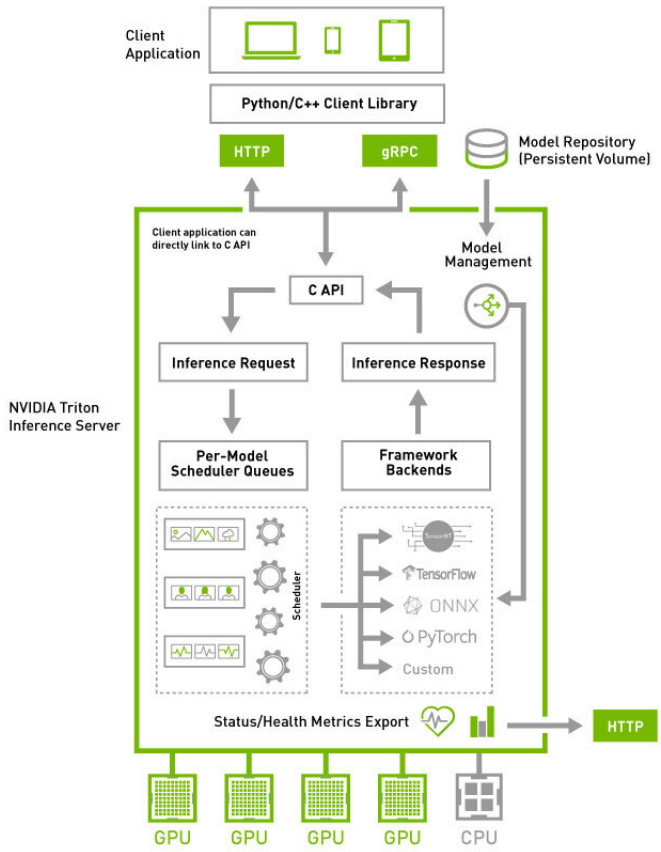
Figur 13: Bilde av NVIDIA Jetson Orin Nano ([47])

## 2.13 Nvidia Triton Inference Server

NVIDIA Triton Inference Server er en åpen serverplattform som effektiviserer og forenkler utplassering av KI-modeller[48]. Serveren sørger for et standardisert, fleksibelt og enkelt format for å utplassere maskinlæringsmodeller. Videre følger en oppsummering av tritonserverens funksjonalitet og arkitektur. For mer utfyllende informasjon, vennligst se NVIDIA Tritons egne sider[48].

### 2.13.1 Arkitektur

Serverarkitekturen er bygget opp av flere deler. Triton har et filorientert modellager med modellene tritonserveren skal gjøre tilgjengelig for inferens. Triton støtter også flere forskjellig kommunikasjonsprotokoller. En klient kan kommunisere via http[49], gRPC[50] eller direkte med C programmeringsgrensesnittet. Forespørsler sendes til serveren fra en klientapplikasjon, og organiseres så i serverens *scheduler*. Modellene prosesserer så data fra forespørslene og resultatene returneres til klienten.




Figur 14: Visualisering av arkitektur på NVIDIA Triton Inference Server[51]

.

### 2.13.2 Modellager

Modellageret har en spesifisert filbasert struktur som må følges for at den skal tolkes av NVIDIA Triton Inference Server. Øverst er hovedmappen der de ulike modellene skal konfigureres. Under denne mappen spesifiseres så en mappe for hver modell som skal gjøres tilgjengelig for inferens på serveren. I disse mappene er det nødvendig med en versjonsmappe der den faktiske modellen ligger. Denne må ha navnet *model.type* hvor *type* er formatet til modellen. Dette kan blant annet være *.onnx* for open neural netowrk exchange[52] modeller eller *.plan* for tensorrt[46] modeller. På samme nivå som versjonsmappen kan det også lages en konfigurasjonsfil kalt *config.pbtxt* Her spesifiseres parametere som navn, *backend* og input- og outputformat. Under vises en illustrasjon av modellagerets mappestruktur.

```
model_repository
├── yolov8
│   ├── 1
│   │   └── model.onnx
│   └── config.pbtxt
└── <other_model>
    ├── 1
    │   └── model.<type>
    └── config.pbtxt
```

### 2.13.3 Modell optimalisering

NVIDIA Triton Inference Server er utviklet med hensyn på å kunne minimere systemforsinkelse og maksimere gjennomstrømningen til modellene [53]. Videre følger noen av mulighetene Triton legger frem for å øke inferenshastigheten til maskinlæringsmodellene.

**Dynamisk batcher**
Den dynamiske batcheren finner den beste batch størrelsen for å maksimere gjennomstrømning for modellen. Den kombinerer så enkle inferensforespørsler til en større batch før dataen sendes gjennom modellen.

**Instanse grupper**
Ved å spesifisere instansegrupper for en modell, kan man definere hvor mange kopier av hver modell man ønsker å lage. Dette vil kunne øke modellhastigheten fordi det tillater overlapp av minnetransferoperasjoner med inferensberegninger, som betyr at en modell kan starte en inferens mens den andre fortsatt overfører dataen til grafikkprosessoren.

### 2.13.4 Python Backend

Triton tilater å prosessere informasjon i Python ved bruk av *Triton backend for Python*[54]. For at Triton skal kunne tolke en Python modell, må den følge spesifikke strukturkrav. I programmet må det defineres en klasse som heter *TritonPythonModel*. Denne klassen må ha en funksjon kalt *execute*, som itererer gjennom forespørsler og publiserer responser til modellens utgang.[54]

### 2.13.5 InferInput

*InferInput* er en klasse fra Pythonbiblioteket *tritonclient*. Klassen brukes for å beskrive en tensor som skal sendes som en forespørsel til en Triton-modell.

### 2.13.6 Custom Execution Enviroment

Triton tillater brukere å spesifisere egne Python miljø som skal tolke Python modellen under kjøring. Miljøet kan konfigureres i miniconda[55] og så komprimeres med *conda-pack* funksjonen. For å spesifisere hvilket miljø modellen skal kjøre i, defineres en parameter i config.pbtxt filen til Python modellen. *EXECUTION_ENV_PATH* spesifiserer at det er kjøremiljøet til modellene man ønsker å definere.

```
    parameters: {
        key: "EXECUTION_ENV_PATH",
        value: {string_value: "Path/to/python_model"}
    }
```

## 2.14 Trådprogramering i python

Trådprogranmering består av å bygge opp en prosess med mindre 'del'-prosesser kalt tråder. Hver tråd er en separat operasjonsflyt, som vil si at prosessen gjør flere ting samtidig. I Python kjører ikke trådene faktisk samtidig, men vil ha den effekten ved at de tildeles prosesseringstid av CPU-kjernen etter behov. Dette betyr at dersom man har en tråd som for eksempel har behov på å vente på svar fra en ekstern prosess. Kan en annen tråd tildeles CPU-tid, og gjøre andre operasjoner i mellomtiden.

### 2.14.1 Race conditions

En race condition er uønsket funksjonalitet som oppstår på bakgrunn av at et program prøver å utføre flere operasjoner samtidig[56]. Når flere tråder prøver å gjennomføre operasjoner på minneadresser avhenger utfallet av timingen til trådene, som styres av CPU-enheten og derfor ikke kan forutsies. For å unngå race conditions kan man blant annet ta i bruk *semaforer*[57] fra *threading* biblioteket eller *Lock* fra *multiprocessing* biblioteket. Dette er flagg som kun kan holdes av en prosess eller tråd av gangen. Ved å kreve at en tråd må holde semaforen eller locken for å utføre operasjoner på en minneadresse som andre tråder eller prosesser har tilgang til minsker man sannsynligheten for slike race conditions.

## 2.15 GNU General Public License

GNU lisensen er en åpen kilde lisens som gir frihet til å anvende, modifisere og utgi programvare som er underlagt lisensen. GNU er en 'copyleft' lisens som betyr at hvis du publiserer programvare som benytter kildekode underlagt GNU, må programvaren som en helhet underlegges GNU. Ved å for eksempel utvikle et program som benytter en maskinlæringsmodell underlagt GNU, må hele programmet publiseres med GNU lisensen.[58]

## 2.16 Trening av objektdetekterings- og klassifiseringsmodeller

Som tidligere forklart gjennomføres treningen av objektdetekterings- og klassifiseringsmodeller ved at modellene forsøker å predikere gitte klasser eller objekter i et bilde. Modellen blir deretter vurdert på hvor godt den presterer ved hjelp av en kostfunksjon. Videre benyttes *backpropagation* for å justere vektene i modellen basert på kostfunksjonen. Det overordnede treningsoppsettet styres av *hyperparametere*. De viktigste hyperparameterne ([59]) for treningen av en modell er beskrevet under.

- **Batch size** - Indikerer antallet bilder som sendes gjennom modellen før interne vekter oppdateres.
- **Epochs** - En epoke representerer en fullstendig gjennomgang av ett datasett.
- **Optimizer** - Er en algoritme som jobber for å minimere tapet gjennom endring av vektene til modellen.
- **Learning rate** - Bestemmer hvor mye vektene til en modell oppdateres.
- **Scheduler** - Bestemmer hvordan learning rate skal endre seg i løpet av en trening.
- **Tap** - For noen modeller er det mulig å bestemme i hvilke grad de ulike typene av tap skal vektes.
- **Data augmentation** - Data augmentation går ut på å endre aspekter ved et bilde som: farger, saturering, lysstyrke eller geometri. Augmenatation benyttes ofte på treningsdataen for å skape et mer mangfoldig datasett. Dette vil gjøre en trente modellen bedre til å predikere bilder som er veldig ulike fra det originale datasettet.

Et datasett bruk til trening og testing av en modell splittes ofte opp i tre deler. Trening-, validering og testdatasett. Valdideringdatasettet brukes etter hver epoke til å teste hvor god modellen er, uten å oppdatere vektene til modellen. På denne måten når treningen er ferdig vil modellen som gjorde det best på valdideringsdatasettet etter en gitt epoke bli valgt ut som den beste modellen. Testsettet er et helt separat datasett som ikke brukes under noen del av treningen. Hensikten er å gi et fullstendig upartisk sammenligningsgrunnlag mellom modellene.

## 2.17 Optiske forstyrrelser

Hvordan oppstår diverse feil i optikken.

### 2.17.1 Eksponeringsfeil

Eksponering refererer til mengden lys som treffer den fotosensitive sensoren i ett optisk kamera. Eksponeringen påvirker lysstyrken, og klarheten i ett resulterende bilde.

**Overeksponering** skjer når for mye lys treffer den fotosensitive sensoren. Dette kan skje ved at lukkeren er åpen for lenge, eller blenderåpningen er for bred. Overeksponering resulterer i ett *utvasket* bilde med tap av detalj i lysere deler av bildet.

**Undereksponering** skjer, motsatt av overeksponering, ved for rask lukkerhastighet eller for liten blenderåpning. Undereksponerte bilder oppleves som mørke, med tap av detalj i skyggeområdene.

### 2.17.2 Solskinn

Refleksflekker på linse kan oppstå når en sterk lyskilde, som solen, treffer direkte på linsen [60]. En optisk linse består av flere glass eller plastelementer. Når lyset passerer en en linse, vil det brytes av hvert element i linsen. når lyset brytes av elementene, vil noe lys reflekteres. Det eksisterer antirefleksbelegg [61] for å minimere dette, men ingen eksisterende belegg fungerer optimalt mot alle bølgelengder. videre vil det reflekterte og brutte lyset spre seg innenfor linsen, og påvirke det resulterende bildet. Denne effekten kan forsterkes ved mikroskopiske ujevnheter eller partikler på linsen. Moderne optiske kameraer måler gjennomsnittlig lysstyrke i ett bilde og justerer lukkerhastigheten basert på dette. Dette kalles eksponeringskompensasjon[62]. I tilfeller der sollys treffer rett på linsen, vil dette resultere i svært kort eksponering, som kan resultere i at resten av bildet vil oppleves undereksponert.

### 2.17.3 Astigmatisme

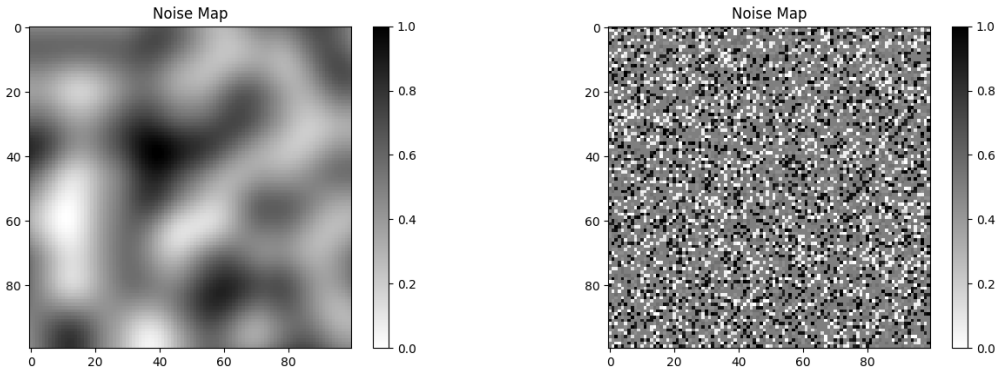
Astigmatisme viser til en linsefeil der vertikale og horisontale linjer ikke opptrer like skarpt[63]. feilen oppstår ved at linsen ikke klarer å fokusere alt lyset på ett enkelt punkt. Fenomenet oppstår typisk ved vibrasjoner og bevegelse i høy hastighet. Det er mulig å korrigere for slike linsefeil ved bruk av asymmetrisk konvolusjon 2.3.1, likevell kan slike korreksjoner introdusere feilaktig data i bildet.

### 2.17.4 Partikler på linse

Når partikler fester seg på linsen, vil partiklene absorbere og reflektere enkelte bølgelengder. Dette vil påvirke lyset som treffer den optiske sensoren, og resultere ett bilde mørke og uskarpe områder.

## 2.18 Perlin-støy

Perlin-støy er en form for gradientstøy[64], utviklet av Ken Perlin. Metoden er mye brukt innen datagrafikk for å skape naturlige, pseudo-tilfeldige teksturer. Perlin-støy fungerer ved å beregne tilfeldige gradienter langs en graf, og interpolere mellom disse for å skape ett støylandskap". Som illustrert i figur 15 resulterer dette i naturlige teksturer som kan skaleres, og flyttes på.




(a) Perlin støy (b) Tilfeldig støy

Figur 15: Sammenlikning mellom Perlin og tilfeldig støy

## 2.19 Ultralytics

Ultralytics[65] er et teknologiselskap som er kjent for å utvikle og bidra til forskjellige åpen kilde prosjekter innenfor feltene datavitenskap, kunstig intelligens og maskinlæring. De er spesielt kjent for å utvikle og vedlikeholde PyTorch-baserte verktøy og biblioteker. YOLOv8 (2.5) og RTDETR (2.7) Modellene som tas i bruk under prosjektet, og python APIen(programeringsgrensesnitt) som blir brukt for trening av modellene, leveres av Ultralytics.

# 3 Metodikk

Gjennomføringen av prosjektet kan deles opp i to hoveddeler. Trening og testing av maskinlæringsmodeller, og utvikling av programvare. Delene ble gjennomført parallelt gjennom prosjektetutførelsen. Trening og testing startet med at det ble undersøkt hvilke modeller som var aktuelle for oppgaven. Når dette var kartlagt ble det satt opp infrastruktur for trening og testing av maskinlæringsmodellene. Programvare-utviklingen startet med oppsett av en Nvidia Triton Inference Server(2.13) på en NVIDIA Jetson(2.12). Deretter ble det lagt til rette for at maskinlæringsmodellene kunne anvendes på serveren. Videre ble det satt opp programvare på klientmaskinen for å kommunisere med Triton-serveren. Avslutningsvis ble det utviklet et enkelt brukergrensesnitt. Når hele plattformen var satt opp ble det gjennomført testing av hastighet og ytelse.

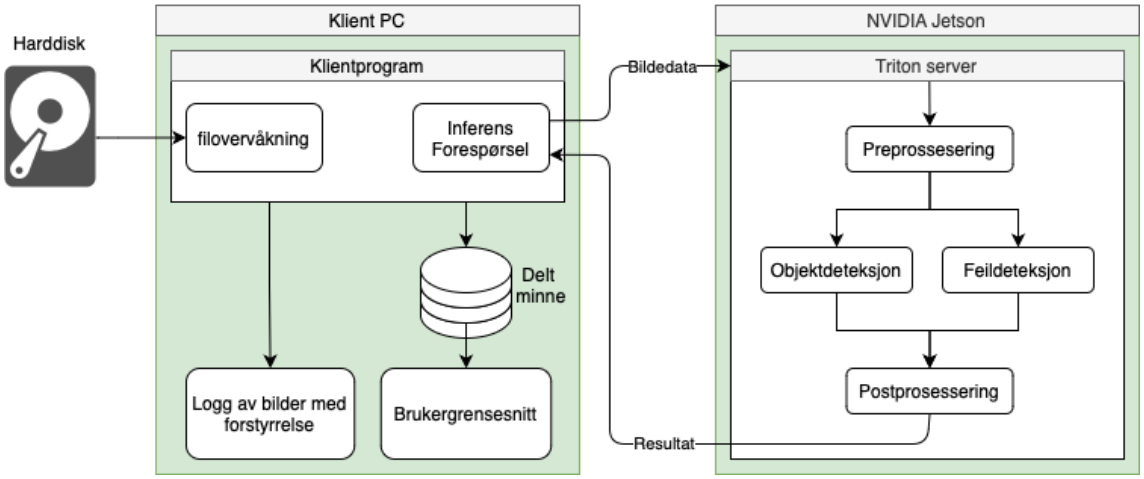
## 3.1 Systemarkitektur

Systemarkitekturen ble utviklet med utgangspunkt i systemspesifikasjonene (1.2.1). Det ble kartlagt at sanntidsplattformen måtte disponere en GPU for å nå spesifikasjon 2 vedrørende modellhastighet. Årsaken til dette er at en GPU benytter parallell prosessering som tillater modellene å kjøre betraktelig raskere inferens (2.10). En Nvidia Jetson prosesseringsenhet ble derfor implementert for objektdeteksjonen.

NVIDIA Triton Inference Server ble valgt som plattform for tjenesteprogramvare for inferens på NVIDIA Jetson(2.12). Serveren støtter GPU-utplassering av dyplæringsmodeller på NVIDIA Jetson, og tilrettelegger simultan utplassering av flere modeller. Triton tilrettelegger videre for enkel utbytting av modeller, som støtter oppunder systemspesifikasjon (1), vedrørende modularitet. Triton benytter også godt etablerte kommunikasjonsprotokoller.

Som følge av at bildene fra kamerariggen tilgjengeliggjøres på bilriggens PC, videre kalt klientmaskinen, måtte systemarkitekturen inneholde en kommunikasjonslinje mellom klientmaskinen og NVIDIA Jetson. Det ble derfor besluttet å utvikle programvare til klientmaskinen, som sammen med Triton danner grunnlaget for en klient/server kommunikasjonsarkitektur.

Sanntidsplattformen brukes til å kjøre inferense med to dyplæringsmodeller. En objektdeteksjonsmodell, og en klassifiseringsmodell. For å etterkomme spesifikasjonskrav (4), var det tiltenkt at objektdeteksjonens *confidence* skulle benyttes. Hypotesen var at ved bilder preget av optiske forstyrrelser, ville *confidence* synke. I tilfellet der deteksjonsmodellens confidence sank, ville klassifiseringsmodellen benyttes for å fastslå hvilken forstyrrelse som var årsaken.




Figur 16: Systemarkitektur for sanntidsplattformen

## 3.2 Valg av modeller

I den hensikt å kartlegge potensialet for å gjennomføre sanntidsobjektdeteketring på iSi-insight ble det fastslått å trene flere forskjellige maskinlæringsmodeller. Det ble besluttet å fokusere på de industriledende sanntidsobjektdetekteringsmodellene YOLO og RT-DETR(2.5, 2.7). Det finnes mange forskjellige firmaer som disponerer disse modellene. Det ble valgt å benytte Ultralytics (2.19) som følge av at de tilbyr en infrastruktur for trening og testing som er enkel og godt dokumentert. Tabell (2) og (3) lister de tilgjengelige modellversjonene Ultralytics disponerer, med relevant statistikk.

| Model | mAPval (50 - 95) (COCO) | Speed A100 TensorRT (ms) | params (M) | FLOPs (B) |
|---|---|---|---|---|
| YOLOv8n | 37.3 | 0.99 | 3.2 | 8.7 |
| YOLOv8s | 44.9 | 1.20 | 11.2 | 28.6 |
| YOLOv8m | 50.2 | 1.83 | 25.9 | 78.9 |
| YOLOv8l | 52.9 | 2.39 | 43.7 | 165.2 |
| YOLOv8x | 53.9 | 3.53 | 68.2 | 257.8 |

Tabell 2: Størrelsene av YOLOv8 Ultraltyics tilbyr med statistikk for mAP(50-95), størrelse og hastighet[26]

| Model | mAPval (50-95) (COCO) | T4 TensorRT FP16(FPS) | params (M) | FLOPs (G) |
|---|---|---|---|---|
| RT-DETRl | 53,0 | 114 | 33 | 110 |
| RT-DETRx | 54,8 | 74 | 67 | 234 |

Tabell 3: Størrelsene av RT-DETR Ultraltyics tilbyr med statistikk for mAP(50-95), størrelse og hastighet[66]

Det ble besluttet å ikke trene de største modellene YOLOv8x og RT-DETRx[1]. Dette er fordi større modeller krever flere *FLOPs* og antas derav ikke som raske nok til å etterkomme systemkrav 2. På samme grunnlag ble det besluttet ikke å trene den nyeste YOLO-modellen, v9[67] fra Ultralytics. De gjenværende modellene som ble valgt å trene var derfor YOLOv8(n,s,m,l) og RT-DETR-l.

Modellen som benyttes videre i prosjektet er forhåndstrent på *COCO(Common Objects in Context)*datasettet[68]. COCO datasettet består av 330 000 bilder hvorav 200 000 er annoterte med 80 ulike objektklasser (2.4.2). Forhåndstreningen gjør at modellen allerede trent på *feature exctraction* som gjør at det tar kortere tid å trene dem på andre datasett.

## 3.3 Metrikker for testing av løsning

I den hensikt å kartlegge hvilke modeller som oppnår spesifikasjonskravene satt i (1.2.1), vil to metrikker bli definert, for respektivt krav nr.(2) og (3).

Kamerariggen tar et bilde hver 1,09 meter. Dette betyr at bildefrekvensen inn til sanntidsplattformen vil variere basert på bilens hastighet. Det ble derfor vurdert som gunstig å kartlegge maksimal bildefrekvens sanntidsplattformen håndterer med objektdeteksjonsmodeller av ulik størrelse implementert. Dette tallet vil gi et mål på hvor raskt objektdeteksjonsmodellene kan operere når de implementeres på plattformen. Det er definert en operativ hastighet for bilen på 80km/t, tilsvarende 20.5 bilder per sekund. At objektdeteksjonsmodellene kan operere i en høyere hastighet enn dette anses som irrelevant. Med dette menes det som mer gunstig å implementere en større, mer

[1]Størrelse er gitt i params(M) i tabellene.

nøyaktig objekt-deteksjonsmodell som fører til at systemet kan prosessere 20.5 bilder per sekund, enn å benytte en mindre nøyaktig modell hvor systemet håndterer 30 bilder per sekund.

Valget av metrikk for modellpresisjon stod hovedsaklig mellom F1-score og mAP(50-95) (2.9.8, 2.9.10). F1-score er oftest brukt i binære klassifiserings oppgaver. Dette er fordi metrikken ikke naturlig vurderer IOU eller confidence, men heller ser på om en prediskjon er binært riktig eller feil. mAP(50-95) på den andre siden bruker enten IoU- eller confidenceterskel (2.9.1, 2.9.3) til å vurdere verdien til hver prediksjon[2]. Den viktigste oppgaven til modellen i kontekst av spesifikasjon nr. 3 vil være å detektere objektene som finnes i et bilde, irrelevant av et en perfekt IOU. Videre med tanke på begrensninger med lite datasett vil en vurdering basert på IOU ta fokus vekk fra det hovedoppgaven til modellen. Basert på dette vil F1-score bli benyttet som metrikk for spesifikasjonskrav nr.3. Det presiseres fortsatt at valget av F1-score er utradisjonelt ettersom mAP(50-95) er en industristandard. Samtidig vurderes det som mer gunstig for å kartlegge modellens ytelse innenfor rammene til dette prosjektet.

## 3.4 Trening av objektdeteksjonsmodeller

Trening og testing av objektdeteksjonsmodellene ble gjennomført ved hjelp av *train-* og *val-*funksjonene i Ultralytics sitt Python bibliotek[70][71]. Maskinvaren som ble benyttet var to bærebare PCer med respektivt Nvidia T600 Laptop og Nvidia GTX 1660 Ti . All trening og testing ble gjort på et annotert datasett levert av iSi AS.

### 3.4.1 Datasett

Datasettet består av 1166 bilder med totalt 16 ulike klasser. Blant klassene finnes rekkverkstypene *w*, *spipe* og *dpipe*, tilsvarende w-skinne, enkeltrør og dobbeltrør. De resterende klassene består av stolper og bolter tilhørende de forskjellige rekkverkstypene. Datasettet ble levert med en parquet fil, som inneholder annoteringsdata. Bildene er annotert med boundingbokser og tilhørende klassetype.

| **Klasse** | **Annoteringer** |
|---|---|
| w | 483 |
| w-jointingbolt | 358 |
| w-steelpole1 | 295 |
| w-steelpole2 | 110 |
| w-woodpole | 248 |
| spipe | 383 |
| spipe-jointingbolt | 191 |
| spipe-steelpole1 | 111 |
| spipe-steelpole2 | 359 |
| spipe-topbolt | 251 |
| spipe-woodpole | 1 |
| dpipe | 303 |
| dpipe-joitingbolt | 161 |
| dpipe-steelpole1 | 192 |
| dpipe-steelpole2 | 203 |
| dpipe-topbolt | 4 |

Tabell 4: Klassene i datasettet med antall annoteringer

For å anvende dataen ble den formatert som et datasett på YOLO-format. Dette ble gjort ved hjelp av et egenutviklet konverteringsprogram i Python. Programmet splitter dataen 80/20 til henholdsvis trening og validering. Programmet fordeler videre dataen i mapper, tilsvarende strukturen fremlagt i (2.6). Datasettet tilgjengeliggjøres med en *.yaml*-fil, som inneholder filplassering og

[2]Ultralytics benytter IOU-terskel under beregning av mAP[69]

(a) W-skinne (b) Enkeltrør (c) Dobbeltrør

Figur 17

klassenavn for dataen. Det ble med bakgrunn i begrensningen av lite datasett valgt å ikke benytte et testsett. Alle objektdeteksjonsmodellene ble trent på det samme datasettet.

På bakgrunn av størrelsen på datasettet, ble hyperparameteroptimalisering vurdert som overflødig. Med ett større datasett og økt tilgang på datakraft, ville optimaliseringsrutiner vært mer relevant.

### 3.4.2 Manuell trening

For å trene modellene ble Ultralytics sin train-funksjon[70] brukt. Funksjonen tillater å definere en rekke *hyperparametere* som bestemmer hvordan treningen skal gjennomføres. I den hensikt å redusere antall variabler som påvirket treningen av objektdeteksjonsmodellen ble det valgt å fokusere på utvalgte *hyperparamterere*. Hyperparameterne brukt under hver trening ble notert i et tabelloppsett ført i Microsoft Excel. De parametrene som det ble bestemt å fokusere på var:

- Bildestørrelse
- Epoker
- Batch størrelse
- CLS-loss
- Box-loss
- DFL-loss

Alle andre hyperparametere og data augmentation ble holdt til standarden satt av ultralytics. LambdaLR og AdamW ble brukt som respektivt optimizer og scheduler[72][73]. Disse er også satt som standard av Ultralytics og har en medfølgende *learning rate* på $1 * 10^{-4}$. Det ble besluttet å gjennomføre all den manuelle optimaliseringen med denne standarden. Fremgangsmåten for endringen av hyperparametere ble gjennomført med bakgrunn i empiri, for å oppnå høyes F1-score. Som følge av størrelsen på datasettet ble det vurdert som lite hensiktsmessig å benytte verktøy for hyperparameteroptimalisering som Optuna[74].

### 3.4.3 Validering

Etter modellen var trent ble den validert ved bruk av Ultralytics sin *validation* funksjon[71]. Denne funksjonen produserer: feilmatrise, kurve for (P, R, PR og F1) og verdier for (P, R, mAP50 og mAP50-95)(2.9). Verdiene fra testingen ble oppført i det samme tabelloppsettet som treningsparameterne. Fra verdiene for P og R ble også F1-score beregnet (2.9.8).

## 3.5 Trening av feildeteksjonsmodell

I den hensikt å etterkomme systemkrav nr.4 ble det utviklet og implementert en feildeteksjonsmodell parallelt med objektdeteksjonsmodellen. Initialt i prosjektet var hensikten bak feildetek-

sjonsmodellen å klassifisere alle forekomster av forstyrrelse i et bilde. Tanken bak dette var at når objektdeteksjonsmodellens *confidence* synker, vil feildeteksjonsmodellen kunne gi informasjon om hvilken type forstyrrelse som er tilstede i bildet. Senere ble formålet endret til å kun klassifisere forstyrrelser som fører til at bildet ikke kan benyttes i rekkverksverifiseringen iSi gjennomfører i dag. For å kartlegge hvilken *forstyrrelse* som har forekommet på et bilde ble det ansett som mest hensiktsmessig å benytte en enkeltklasse-klassifiseringsmodell . Det ble videre besluttet å bruke en YOLO klassifiseringsmodell, levert av Ultralytics (2.6.1). På denne måten vil systemet effektivt kunne klassifisere optiske forstyrrelser, uten hensyn til hvor i bildet forstyrelsen er tilstede. For å evaluere ytelsen til feildeteksjonsmodellen ble det bestemt å bruke *top 1* treffsikkerhet som metrikk (2.9.11).

### 3.5.1 Datasett

For å produsere en presis og relevant feildeteksjonsmodell, var det aktuelt å prioritere forstyrrelser som påvirker bildekvaliteten betraktelig. Forstyrrelser som ikke påvirker synligheten av bildemotivet, ble derfor dømt uinteressante å detektere. På bakgrunn av dette, ble det kartlagt 6 typiske forstyrrelser som kan oppstå i ett optisk kamera i bruk utendørs, og i bevegelse, slik som iSi inSight kamerariggen.

- **Normal** - Bilder uten nevneverdige feil.
- **Astigmatisme**
- **Overeksponerte bilder**
- **Undereksponerte bilder**
- **Solskinn**
- **Skyggelagt motiv**
- **Partikler på linse**

I det tildelte datasettet fra iSi AS, eksisterer tilfeller av samtlige forstyrrelser, med høy påvirkningsgrad. Dette demonstrerte også behovet for en slik modell, og motiverte videre utvikling av feildeteksjonsmodellen. Enkelte feil var likevel mindre vanlig enn andre.

I henhold til en studie dokumentert av MIT News, kan syntetisk datasett produsere bedre trente og generaliserte modeller. *The researchers were surprised to see that all three synthetic models outperformed models trained with real video clips on four of the six datasets"*[75]. Dette demonstrerer potensialet i å kunstig simulere forstyrrelser. Noe som videre støttes av *IBM* som i en artikkel beskriver at syntetisk data kan øke kvaliteten på dataen, og redusere bias, ved å generere balanserte datasett [76].

På bakgrunn av lite datasett, og tidligere studier som antydet at syntetisk data potensielt kunne produsere gode resultater, ble det besluttet å generere syntetisk data. For å unngå feilaktig data, ble kun *enklere* feil simulert. Ettersom f.eks. solskinn er en komplisert og linsespesifikk feil, ble denne feilen ikke generert syntetisk.

**Astigmatisme**
Ved høye hastigheter og/eller *dårlige* veier, vil det oppstå vibrasjoner. Dette kan påvirke presise linser, og forårsake astigmatisme (2.17.3). Astigmatisme kan replikeres ved å benytte konvulsjon (2.3.1) med asynkront/rektangulært filter. Ved å øke bredden på filteret vil en oppleve større grad av uskarphet langs bildets bredde. Som illustrert på figur 18 replikerer dette effekten av astigmatisme. I praksis ble dette gjennomført med cv2 metoden *GaussianBlur*()

```
1    import cv2
2
3    # Tilforer blurr ved a benytte GaussianBlur metoden
4    astimatisme_bilde = cv2.GaussianBlur(src=bilde, ksize=filterstorrelse, 0)
```

Figur 18: Astigmatisme

**Eksponeringsfeil**
For å replikere effekten av over- og undereksponerte bilder, benyttes bildeprosseseringsbiblioteket cv2, levert av openCV [77]. cv2 har funksjonalitet for å justere pikselverdiene i et bilde, ved å bruke metoden *convertScaleAbs*().

```
import cv2

# Justerer pikselverdiene i bildet med metoden convertScaleAbs
justert_bilde = cv2.convertScaleAbs(orig_bilde, alpha=alpha, beta=beta)
```

Alpha er en skaleringsfaktor som multipliseres med pikselverdiene for å øke eller redusere lysintensiteten relativt til den originale pikselverdien. Beta er en forskyvningsfaktor, som forskyver alle pikselverdier uniformt frem til saturering. Ved å benytte høye alfa(>1) og beta(>0) verdier, vil effekten av overeksponerte bilder replikeres. Ved å benytte lave alfa(<1) og beta(<0) verdier, vil effekten av undereksponerte bilder replikeres.

(a) Overeksponert bilde (b) Undereksponert bilde

Figur 19: Justert lysstyrke

**Solskinn**
I iSi AS datasettet var det 147 tilfeller av solskinn i varierende grad, noe som utgjør $\approx$ 12.6% av det tildelte datasettet fra iSi. Å replikere denne feilen på en virkelighetsnær måte, ble regnet som utfordrende. På bakgrunn av dette, ble det vurdert som ikke aktuelt å kunstig utvide denne klassen.

(a) Solskinn som blokkerer motivets synlighet

(b) mindre solskinn i veien for bildemotiv

Figur 20: Solskinn på linse

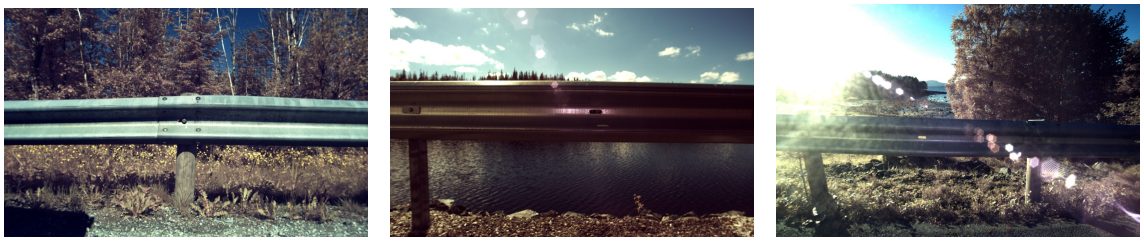
**Skyggelagt motiv**
I likhet med solskinnsbilder, eksisterer det 70 tilfeller av skyggelagte motiv i datasettet. Dette utgjør $\approx 6\%$ av det tildelte datasettet. forstyrrelsen oppstår ved at kamerariggen skjermer for solen på enkelte områder. Skyggelagte bilder ansees som ikke aktuelle å replikere, grunnet utfordringer ved å simulere realistiske tilfeller.

Figur 21: Skyggelagt motiv

**Partikler på linse**
Ved å benytte perlin-støy, som introdusert i seksjon 2.18, kan en produsere et normalisert gradientstøykart. Et slikt støykart vil ha en *organisk* struktur, og kan brukes til å bestemme alpha-kanalen(opasiteten) i en overliggende tekstur. På denne måten kan en replikere effekten av partikler av varierende størrelse og tetthet på linsen.

Figur 22: Gjørme på bildet

**Fremgangsmåte for oppsett av datasett**
For å opprette treningsdatasettet, ble iSi datasettet sortert i korrekte klasser i henhold til formatet diskutert i seksjon 2.6.2. Dette ble gjort manuelt for å sikre korrekt sortering. I tillegg til en klasse for hver relevante bildeforstyrrelse, ble det opprettet en klasse kalt *Normal*. For at et bilde skulle klassifiseres som normalt, måtte det oppfylle ett krav. Hele motivet skal være synlig, upåvirket av forstyrrelser. Tilfeller av forstyrrelser, som ikke påvirket synligheten av ønsket bildemotiv, ble dermed klassifisert som *Normal*. For å gi en visuell inndeling, henvises det til figur 23.



(a) Bilde med tilfredstillende synlig motiv

(b) Bilde med marginalt synlig motiv

(c) Bilde preget av forstyrrelser

Figur 23: Visuelt eksempel på inndeling

Videre ble det utviklet et Python program for å iterere gjennom alle *Normale* bilder. For hvert bilde, ble bildet kopiert og augmentert en gang for hver simulerte feil. Hvor omfattende en simulert feil var, ble bestemt av en pseudo-tilfeldig fordeling. Videre ble de augmenterte bildene lagret i korrekte klasser. Dette resulterte i ett treningssett på 3139 bilder og ett valideringssett på 822 bilder. fordelt på de 7 klassene. Noe som utgjør en fordeling på 79% trening, 21% validering. Fordelingen av det nye datasettet er illustrert i figur 5.

| Klasse | Naturlige instanser | Simulerte feil | Totalt datasett |
|---|---|---|---|
| Normal | 725 | 0 | 725 |
| Astigmatisme | 18 | 713 | 731 |
| Overeksponert | 18 | 709 | 727 |
| Undereksponert | 145 | 710 | 855 |
| Solskinn | 123 | 0 | 123 |
| Skygge | 65 | 0 | 65 |
| Partikler på linse | 22 | 713 | 735 |
| Totalt | 1116 | 2845 | 3961 |

Tabell 5: klassefordeling i datasett for klassifisering av bildeforstyrrelser

### 3.5.2 fremgangsmåte for trening

Feildeteksjonsmodellen ble videre trent ved å bruke Python biblioteket til Ultralytics.

```
    from ultralytics import YOLO # Importerer YOLO fra yltralytics

    model = YOLO('yolov8n-cls.pt') # Klassifiseringsmodell forhandstrent pa COCO
    model.train(data=data, params=params) # Fintrener modellen pa eget datasett
```

Feildeteksjonmodellen ble trent på 20 epoker, bildestørrelse 416, og en batch size på 8.

## 3.6 Triton server

Utviklet programvare for triton server ligger vedlagt i *server_software_bachelor_e2402.zip*

### 3.6.1 Oppsett på NVIDIA Jetson

For å implementere NVIDIA Triton Inference Server (2.13) på NVIDIA Jetson ble besluttet å ta i bruk Triton sitt Docker image. I forbindelse med spesifikasjon 5 tillater dette enkel implementering av programvaren på ønsket maskinvare. Det muliggjør også oppskalering ved at flere konteinere kan deployeres ved behov.

Docker Image som ble tatt i bruk i dette prosjektet var *nvcr.io/nvidia/tritonserver/24.03-py3-igpu*[78]. Dette Image kommer ikke ferdiginstallert med Miniconda, Pytorch og Torchvision. Dette var nødvendig programvare for dataprosessering som skulle implementeres på Tritonserveren. For å løse dette ble det laget en Dockerfil (2.11) for å ekspandere funksjonaliteten til Triton Image.

Dockerfilen er utviklet sammen med et innstallasjonscript. Innstallasjonscriptet bygger Docker Image, konfigurerer et modellager og gjør nødvendige systemendringer for å kunne kjøre Tritonserveren.

### 3.6.2 Implementasjon av maskinlæringsmodeller

Treningen fra seksjon 3.4 og 3.5 resulterte i modeller på *Pytorch-format*. For å optimalisere hastigheten til de trente modellene ble det besluttet å konvertere de til *TensorRT-format*. TensorRT er et dyplæringsrammeverk som optimaliserer hastighet og ytelse på NVIDIA GPUer [46]. Modellene ble også konvertert til *TensorRT-format* med *half-precision*. En standard TensorRT-modell vil bestå av 32bits flyttal. Å konvertere modellen med half-precision endrer da datastrukturen til 16bits flyttal. Dette vil føre til lavere nøyaktighet, men øke modellens hastighet [79].

For å implementere objektdeteksjonsmodellene på *TensorRT-format* ble de først konvertert til onnx-format med Ultralytics *YOLO.export* metode. Modellene ble så konvertert til TensorRT modeller ved å bruke den ferdiginstallerte *trtexec*[80] binæren på NVIDIA Jetson. *–onnx* spesifiserer hvilken onnx modell som skal konverteres. *–fp16* spesifiserer at modellen skal kompileres med *half-precision*, og *–inputIOFormat=fp16:chw* og *inputIOFormat=fp16:chw* spesifiserer at in- og utgangen til modellene skal bestå av 16bits flyttall. *–saveEngine* definerer navnet på tensorrt modellen og *–buildOnly* spesifiserer at trtexec kun skal bygge modellen uten å kjøre den.

```
trtexec --onnx=<modelname.onnx> --fp16 --inputIOFormat=fp16:chw --outputIOFormat=
    fp16:chw --saveEngine=model.plan --buildOnly
```

For å gjøre modellene tilgjengelig på Tritonserver ble de flyttet til modellageret i henhold til spesifikasjonene i 2.13.2.

### 3.6.3 Implementasjon av Pre- og Postprosesserings modeller

Pre og postprosessering relatert til objektdeteksjonsmodellene ble implementert som Python-modeller på Tritonserveren. Bakgrunnen for dette var å kunne utnytte GPU-prosessering på NVIDIA Jetson, og derav øke systemhastigheten.

**python backend**
Ettersom biblioteker som Pytorch og Torchvision ble tatt i bruk under prosesseringsstegene var det nødvendig å konfigurere et Python miljø som Triton kunne tolke Python modellene med. Installasjonscriptet sørger for dette ved å bygge og komprimere et tilpasset Conda miljø og flytte det til respektive plasser i modellageret. I tilhørende config.pbtxt filer ble så en parameter satt for å linke Conda-miljøet til kjøretiden av modellene.

**Preprosessering**
Preprosseseringsmodellen ble spesifisert med en inngang og en utgang i modellens tilhørende config.pbtxt fil (2.13).

Preprosesseringsmodellen mottar bildedataen som en 1-dimensjonal Array med uint8 verdier. Bildedataen konverteres så til en tensor med Torchvision-funksjon, *io.decode_jpeg*, og overføres til GPUen. Dette gir en 3 dimensjonal tensor med bildedataen på GPUen som tillater for parallellprosessering

av tensoren ved de neste preprosesseringsstegene. Maskinlæringsodellene har spesifiserte innganger med dimensjon (3,640,640). Ved å bruke transform klassen fra Torchvision endres dimensjonene på tensoren, og verdiene normaliseres. Tensoren sendes så til utgangen til preprosesseringsmodellen.

**Postprossesering**
Postprosesseringsmodellen ble konfigurert med to innganger og en utgang ettersom den skulle motta data fra både objektdeteksjons og klassifiseringsmodellen. Dette utdypes videre i seksjon (3.6.4).

Modellen ble utviklet med *nms* (2.5) for å filtrere ut overlappende prediksjoner fra objektdeteksjonsmodellen. Koden i *nms*-funksjonen ble i stor grad hentet fra kildekoden til Ultralytics, underlagt GNU-lisens (2.15). Årsaken til at *nms* funksjonen ikke ble importert direkte fra Ultralytics-biblioteket var fordi det ble oppdaget konflikter mellom Cuda-versjonene til Torch og Ultralytics, som ble ungått ved å skrive *nms*-funksjonen selv.

Postprosseseringen består også av en funksjon kalt *format_output*. Hensikten med funksjonen var at dataen skulle formateres på en måte som var enkel å behandle i Klientprogrammet. Funskjonen mottar prediksjonene fra objektdetekteringsmodellen etter de har blitt prossesert med *nms*, og feildeteksjonene fra klassifiseringsmodellen. Dataen restruktureres, og returneres som en [1x6] tensor, indeksert med følgende verdier.

**Detektert rekkverk:** Hvis det gjøres flere deteksjoner av samme eller ulike rekkverk vil denne verdien tilsvare rekkverksklassen med høyest confidence.

**Detektert stolpe:** Hvis det gjøres flere deteksjoner av samme eller ulike stolper vil denne verdien tilsvare rekkverksklassen med høyest confidence.

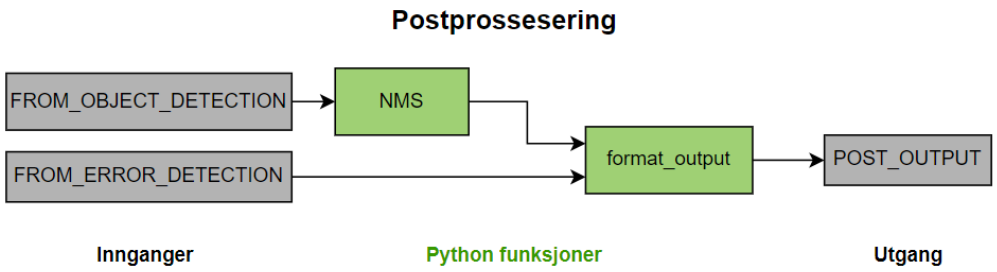
**Antall stolper** Defineres som antall stolpe prediksjoner gjort av modellen.

**Confidence:** Gjennomsnittlig confidence fra detektert rekkverk og stolpe.

**Feiltype:** Klasse fra feildeteksjonsmodellen

**Feil confidence:** Confidence fra feildetesjonsmodellen.

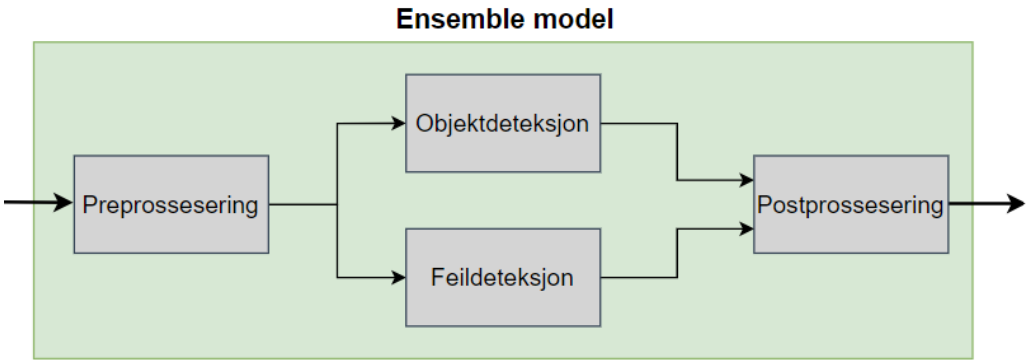
*pb_utils.InferenceResponse.*




Figur 24: Dataflyt i postprosessering på Tritonserver

### 3.6.4 Konfigurasjon av Ensemblemodell

For å bygge en arkitektur for dataflyt innad i Triton ble det benyttet en *Ensemble model*. Denne modellen består kun av en konfigurasjonsfil som forteller Triton hvor den skal sende data. Ved å bruke konfigureringsparameteren *ensemble scheduler* ble deteksjons- og prosesseringsmodellene koblet sammen. Dette resulterte i følgene dataflyt.




Figur 25: Dataflyt i i Ensemble model på Tritonserver

## 3.7 Klient

Klientprogrammet vil sende inferensforespørsler til serveren, og videreføre resultatdataen til et brukergrensesnitt. Programmet vil også logge detekterte bildefeil. På denne måten vil sjåføren motta tilbakemelding om hvilke bilder som må utbedres. Utviklet programvare for klientprogram ligger vedlagt i *client_software_bachelor_e2402.zip*. Klientprogrammet består av to hoveddeler: filmonitorering og sending av inferensforespørsler.

### 3.7.1 Filmonitorering

Bilder fra kamerariggen lagres på lokale disker på klientmaskinen(1.1). For at klientprogrammet skulle monitorere harddiskene ble Pythonbiblioteket *Watchdog* benyttet. Fra *Watchdog*-biblioteket ble klassene *Observer* og *Patternmatchingeventhandler* importert. Klassen, *Observer*, tillater klientprogrammet å starte en filmonitoreringsprossess på harddiskene. Det ble deretter utviklet en klasse, *Handler*, som bestemmer hva som skal gjøres når *Observer*-objektet oppdager en ny fil på harddisken. *Handler* arver fra *Patternmatchingeventhandler*. Dette tillot å spesifisere at *Observer*-objektet kun skal reagere på lagrede bilder. Videre ble det i *Handler* definert en funksjon, *on_created*, som oppretter en tråd (2.14) for å sende en inferensforespørsel hver gang et nytt bilde oppdages på harddisken. Dette tillot systemet å sende nye forespørsler uten å vente på at tidligere forespørsler ble sendt, som økte den maksimale bildefrekvensen til systemet. *on_created* lagrer også tidstempelet for når bildet ble registrert.

### 3.7.2 tråder

Tråden har som oppgave å sende én inferensforespørsel til Tritonserveren for et enkelt bilde. Tråden holdes levende helt til svaret på sin forespørselen er tilgjengelig. Som følge av dette vil det kunne være flere tråder som opererer samtidig. Hver tråd kjører funksjonen *infer_request*, med bildeadressen og tidsstempelet til deteksjonen som argumenter. I *infer_request* åpnes bildet som et uint8 array med numpy funksjonen, *from_file* ([81]). Bildedataen blir så formatert til et *InferInput* (2.13.5). Videre opprettes et GRPC-klient-objekt med IP-adressen til Nvidia Jetson. GRPC-objektet sender bildedataen som en asynkron inferensforespørsel til *Ensemble modellen*. Tråden venter til inferensresponsen er tilgjengelig, og legger det til i listen *shared_memory*.

### 3.7.3 Delt minne

Listen, *shared_memory* er et *ShareableList*-objekt ([82]), importert fra Python-biblioteket *multiprocessing*. Hensikten med dette var at andre program på klientmaskinen skulle ha tilgang til inferensresponsene. *shared_memory*-listen ble definert med lengde 20. Årsaken til dette har med hvordan brukergrensesnittet presenterer data, og vil forklares nærmere i 3.8. Det ble også definert

en variabel, *counter*, som brukes for å holde styr på indekseringen til listen. Denne ble initialisert til 0. Når en *Tråd* skal legge til en inferensrespons i listen benyttes en *Lock* 2.14.1 for å unngå *race-conditions*(2.14). Inferensresponsen legges til i listen på indeks=counter. Tråden øker deretter *counter* med 1, slik at neste *Tråd* vil legge inn sin respons på neste plass i listen. *Counter* nullstilles på 20. Dette resulterer i at *shared_memory*-listen vil inneholde inferensresponsene fra de siste 20 bildene som er tatt.

### 3.7.4 Logging

Hver *Tråd* sjekker også fra inferensresponsen om det er detektert feil i bildet. Hvis det er detektert en feil skriver tråden tidsstempel, feilmelding og bildenavn til en csv-fil. Her benyttes det også en semafor for å unngå at to Tråder prøver å skrive til filen samtidig.

## 3.8 Brukergrensesnitt

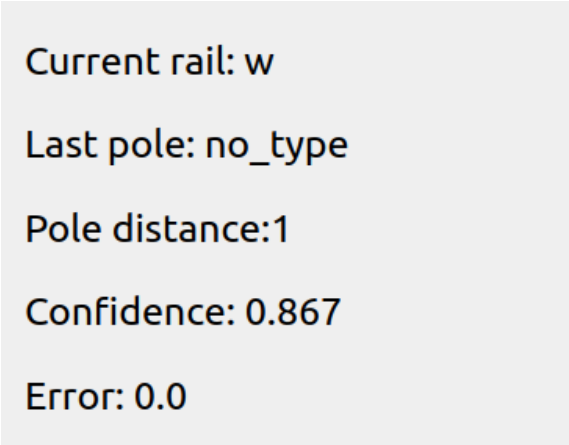
Den initielle planen for brukergrensesnittet var å gi sjåføren et rask sammendrag av de siste prediksjonen gjort av modellen. Dette ville gitt mulighet til å sjekke om disse stemmer overens med det rekkverket bilen kjører forbi. Senere ble det også besluttet, etter hvert som funksjonalitet for dette ble implementert, at typen feil på bildene også skulle fremvises. Informasjonen som fremvises i brukergrensesnittet ble derfor bestemt til å være:

- Nåværende rekkverk
- Siste stolpetype
- Stolpeavstand for rekkverket
- Sikkerheten til prediksjonene gjennomført av modellen
- Visuell varsling når et rekkverk starter og slutter

Med tanke på at sjåføren rask skal kunne observere og fordøye informasjonen samtidig som at brukergrensesnittet ikke fremstår forstyrrende. Ble det holdt enkelt med svart skrift på grå bakgrunn(26).

Brukergrensesnittet ble programmert med pythonpakken *PyQt5*[83]. Programmet er satt opp med en python klasse. Innit-funksjonen definerer oppsettet og utseendet til grensesnittet. Funksjonen definerer også Qlables for hvert av informasjonspunktene som skal fremvises. Grensesnittet driftes med en *update_data* funksjon som oppdaterer Qlables basert på sharedmemory list til client.py(3.7).

Som forklart i seksjon 3.7.3 lagres prediksjonene på de 20 siste bildene i shared memory list. For å definere type rekkverk, stolpe og bildefeil finner programmet den mest vanlige av disse typene over de siste 20 prediksjonene. Stolpeavstand defineres ved at programmet dividerer antall stolpe deteksjoner på 22 og avrunder dette til enten 1,2 eller 4. Dette er gjort med bakgrunn i at bilen beveger seg 22 meter ved tagning av 20 bilder og stolpeavstander i Norge alltid er 1,2 eller 4 meter. Confidence-metrikken er basert på et gjennomsnitt av den høyeste rekkverk og stolpe confidencen for hvert av de 20 siste bildene.




Figur 26: Brukergrensesnittet

## 3.9 Systemsikkerhet

*ShareableList* akseseres av inferenstrådene i client.py (3.7.2), og brukerkgrensesnittet i new_inference.py (3.8). For å unngå *race-conditions* under kjøring ble det implementert en *Lock* fra *multiprocessing* biblioteket. *Lock* sørger for at kun en prosess eller tråd kan ta i bruk *ShareableList* om gangen. For å unngå at serveren overmates med inferensforespørsler ved store hastigheter ble det også implementert en tidsfrist for alle tråder.

## 3.10 Simulator

Som fremlagt i 1.1 tar kamerariggen bilder hver $\approx 1.09$ meter, og lagrer det på harddisker. For å gjenspeile hvordan systemet reellt ville mottatt data ble det utviklet en simulator. Simulatoren flytter bilder av rekkverk til mappestrukturen som overvåkes av klientprogrammet (3.7). Bildene simulatoren henter er fra et isolert datasett som ikke har blitt benyttet i treningen av modellene. Bildefrekvensen kan justeres, og settes som bilder pr sekund.

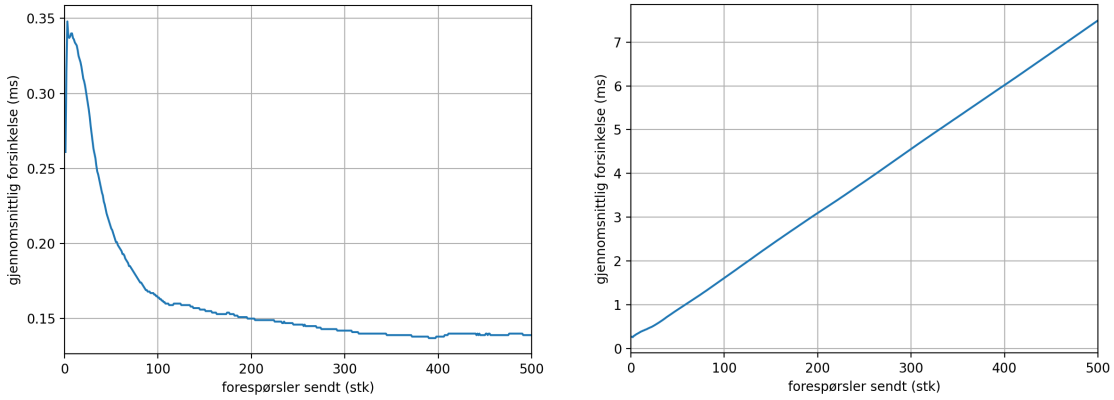
## 3.11 Systemtesting

### 3.11.1 Operativ hastighet

Det ble kartlagt hvor høy bildefrekvens systemet kunne operere ved med de ulike modellene implementert. Hensikten var å finne ut hvilke modeller som kunne håndtere bilens operative hastighet på 80 km/h. Testene ble gjennomført for yolov8n, yolov8s, yolov8m, yolov8l og RT-DETRl. Samtlige modeller ble også testet med *half precission*. Ved alle testene kjørte objektdeteksjonsmodellene parallelt med feildeteksjonsmodell, yolov8n-cls.

For å avgjøre om en modell kunne operere ved en gitt bildefrekvens ble det undersøkt om frekvensen førte til kumulativ forsinkelse i systemet (27b). Med kumulativ forsinkelse menes en gradvis økning i tiden fra klientprogrammer oppdager et bilde, til serveren returnerer responsen. Dette vil oppstå når systemet mottar bilder raskere enn Tritonserveren klarer å behandle dem.

For å måle forsinkelsen ble det implementert en tidtaker i klientprogrammet som måler levetiden til hver *Tråd*. Simulatoren ble deretter satt til 10 bilder/sekund med en kjøretid på 5 minutter. Trådenes gjennomsnittlige levetid ble monitorert, og modellen ble regnet som operativ hvis levetiden stabiliserte seg. Dette ble gjentatt helt til maks frekvens var funnet for samtlige modeller.




(a) Bildefrekvensen er ikke for høy, og gjennomsnittlig forsinkelse stabiliserer seg

(b) Kumulativ forsinkelse ved for høy bildefrekvens

Figur 27: systemforsinkelse med yolov8n

Det ble også besluttet å monitorere for minneutmatelse på NVIDIA Jetson og Klientmaskinen under denne testen. Minneutmattelse vil oppstå når en prosess eller system ikke har mer tilgjengelig minne for å utføre operasjoner[84]. Dette kan føre til at programmet kjører veldig sakte, eller slutter å fungere helt.

For å måle minnebruk på NVIDIA Jetson ble programvaren jetson_stats[85] installert på plattformen. Dette systemet muliggjorde sanntidsobservering av *GPU shared RAM* som er minnet Tritonserveren allokerer. Minnebruk på klientmaskinen ble målt på liknende måte ved å ta i bruk programvaren *system monitor* som kommer ferdiginstallert på Ubuntu 22.04 LTS operativsystemer.

### 3.11.2 Stabilitetstest

For å teste plattformens langvarige stabilitet ble det kjørt en test over syv timer med simulatoren satt til 80km/t, tilsvarene en frekvens på 20.5 bilder i sekundet. For hver inferensrespons klienten mottok, ble forespørselens forsinkelse og den totale gjennomsnittlige forsinkelsen loggført. På 7 timer med 20.5 bilder per sekund tilsvarte dette 516000 datapunkter.

# 4 Resultater

## 4.1 Objektdeteksjonsmodeller

### 4.1.1 F1-verdier

Høyest oppnådd F1-score for FP32- og FP16[1]-konfigurasjon av modellene er listet i tabell 6 og tabell 7. RT-DETRl-FP32 oppnår høyest F1-score med 0.713. Resultatene viser videre at F1-score ikke nødvendigvis øker med modellstørrelsen. Dette illustreres fra tabell 6 ved at YOLOv8s-FP32 har høyere F1-score enn YOLOv8l-FP32. Figur 7 viser at samtlige modeller får lavere F1-verdier når de benyttes i FP16-format. Det vises også at den høyeste F1-verdien ved FP16, oppnådd av YOLOv8s, er lavere enn den laveste F1-verdien ved FP32, fra YOLOv8n.

| **FP32** | | | |
|---|---|---|---|
| **Model** | **P** | **R** | **F1-score** |
| YOLOv8n | 0.672 | 0.615 | 0.642 |
| YOLOv8s | 0.743 | 0.635 | 0.685 |
| YOLOv8m | 0.772 | 0.614 | 0.684 |
| YOLOv8l | 0.714 | 0.634 | 0.673 |
| RT-DETRl | 0.754 | 0.677 | 0.713 |

Tabell 6: Høyest oppnåd F1-score for hver av modellene på FP32-format, med tilhørenede precision og recall.

| **Half precision/FP16** | | | |
|---|---|---|---|
| **Model** | **P** | **R** | **F1-score** |
| YOLOv8n | 0.479 | 0.624 | 0.542 |
| YOLOv8s | 0.596 | 0.599 | 0.597 |
| YOLOv8m | 0.504 | 0.615 | 0.554 |
| YOLOv8l | 0.561 | 0.629 | 0.593 |

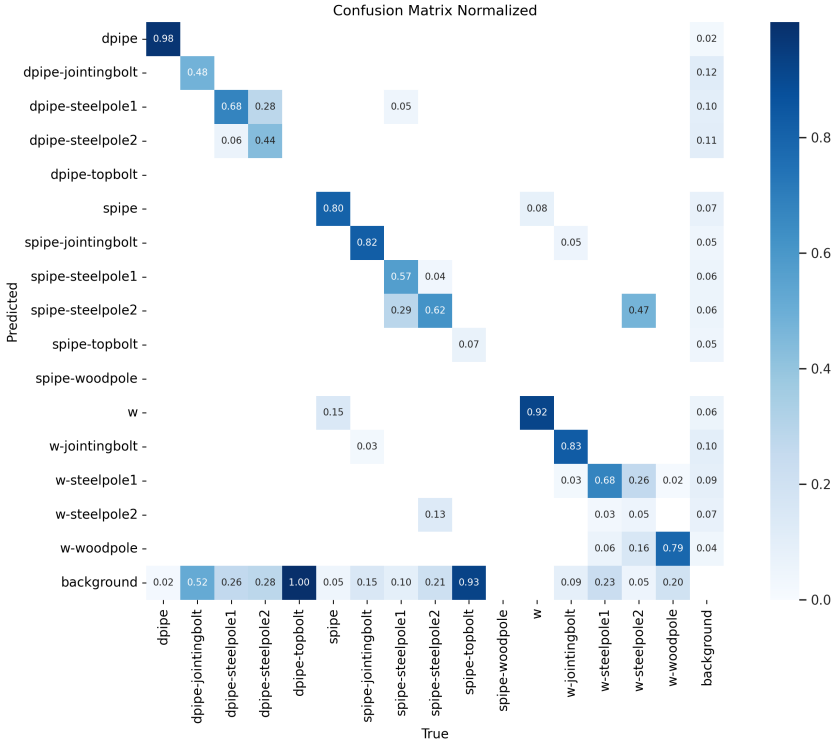
Tabell 7: Høyest oppnåd F1-score for hver av modellene på FP16-format, med tilhørenede precision og recall.

### 4.1.2 Feilmatrise

Den normaliserte feilmatrisen presentert i Figur 11 viser prediksjonssammendraget[2] til yolov8n. Modellen har over 80% treffsikkerhet når den predikerer rekkverkstype. Ved dobbeltrør (dpipe) og w-skinne (w) har yolov8n en treffsikkerhet på henholdsvis 98% og 92%. Ved enkeltrør (spipe) er treffsikkerheten lavere, på 80%. Modellen har også høy treffsikkerhet på utvalgte stolper og bolter: *w-woodpole* (79%), *w-jointingbolt* (83%) og *spipe-jointingbolt* (80%). For resterende stolper og bolter har modellen treffsikkerhet fra 44% til 68%. Unntakene er *spipe-topbolt* og *w-steelpole2* som kun predikeres riktig 7% og 5% av forekomstene.

Feilmatrisene for de øvrige modellene yolov8(s,m,l) og RT-DETRl viser samme trend som yolov8n.

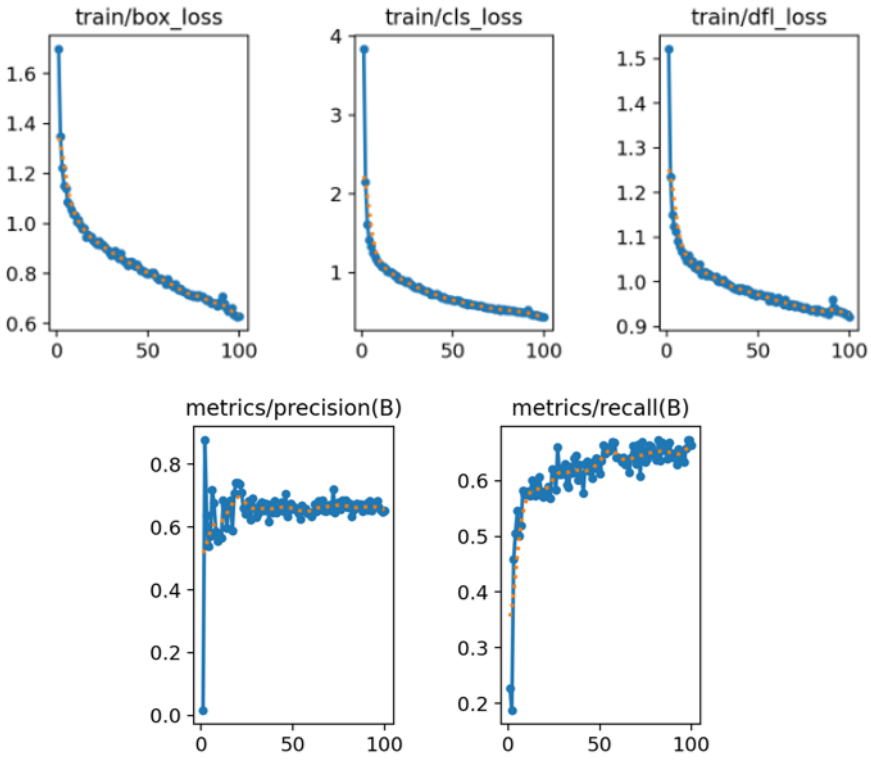
[1]Det ble ikke gjennomført FP16 testing for RT-DETRl fordi dette ikke var støttet av Ultralytics sin val-funksjon
[2]Klassene *dpipe-topbolt* og *spipe-woodpole* hadde ingen instanser i valideringsdatasettet.




Figur 28: Normalisert feilmatrise fra test av YOLOv8n

### 4.1.3 Evalueringsmetrikker ved trening

Tapverdier fra trening og precision- og recall-verdiene ved validering etter hver epoke for trening av YOLOv8n er presentert i figur 29. Precision stagnerer etter 25 epoker og det samme skjer for recall fra epoke nummer 50. Tapverdiene synker jevnt under hele treningen. Ved trening av de øvrige modellene yolov8(s,m,l) og RT-DETRl ble det observert samme tendenser.



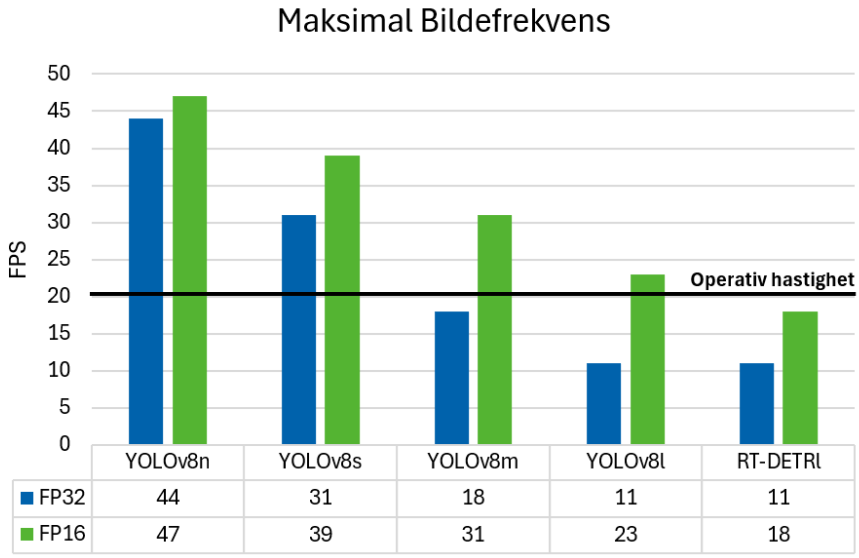

Figur 29: Tapverdier fra trening og precision- og recallverdier fra hver validering under en trening på 100 epoker for YOLOv8n.

## 4.2 Systemhastighet

### 4.2.1 Maksimal bildefrekvens

Figur 30 viser maksimal bildefrekvens systemet klarer å håndtere med de ulike objektedetksjons-modellene implementert. Grafen viser modellene på både FP32 og FP16 format. Modellene kjører parallelt med yolov8n-cls feildeteksjon.

Når modellene implementeres på FP32-format, er det kun yolov8(n,s) som er raske nok til at systemet kan prosessere 20.5 bilder i sekundet. Ved FP-32 format er det derfor bare yolov8n og yolov8s som etterkommer systemspesifikasjon 2. Hvis modellene implementeres med half precision (FP16) øker maks bildefrekvens for modellene. Ved FP-16 format vil systemet etterkommer systemspesifikasjon 2 med alle av de 4 yolo-modellene. RT-DETRl er den eneste modellen som ikke etterkommer systemspesifikasjon 2 ved FP32 eller FP16.



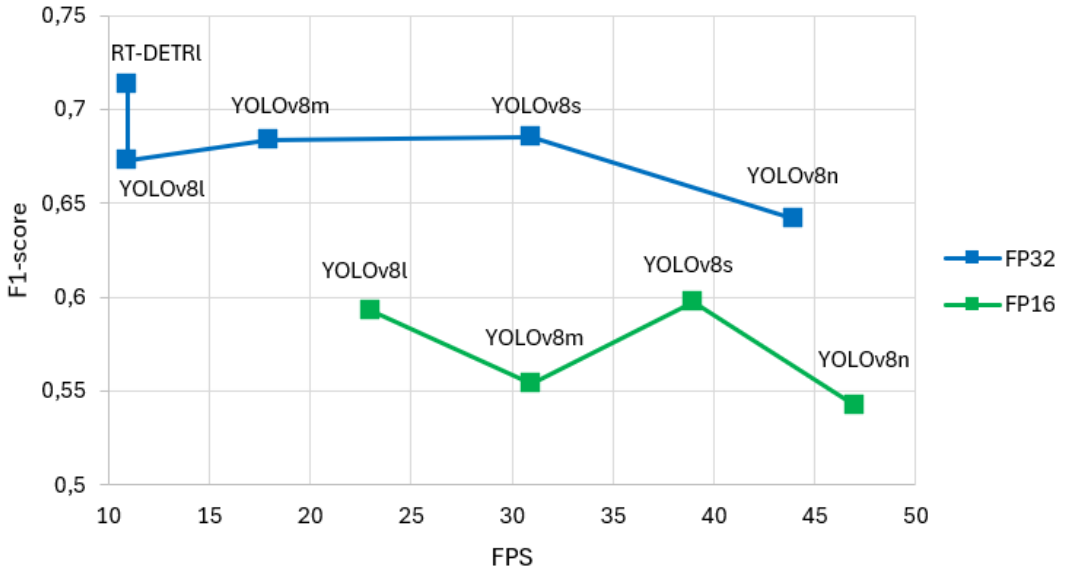

Figur 30: Maks bildefrekvens systemet håndterer med modeller på TensorRT-format, bygget med og uten half precision. Operativ hastighet er markert ved 20.5fps, tilsvarende 80km/h.

Ved å overstige maksimal bildefrekvens observeres det at tritonserveren allokerer økende mengder minne til det oppnås minneutmattelse på NVIDIA Jetson. Dette fører til at operativsystemet krasjer. Minneutmattelse ble ikke observert på klientmaskinen.

## 4.3 Modellytelse

Figur 31 plotter F1-score mot maksimal bildefrekvens. Av modellene som oppfyller systemspesifikasjon 2, maks bildefrekvens over 20.5, er det YOLOv8s-FP32 som har høyest F1-score.

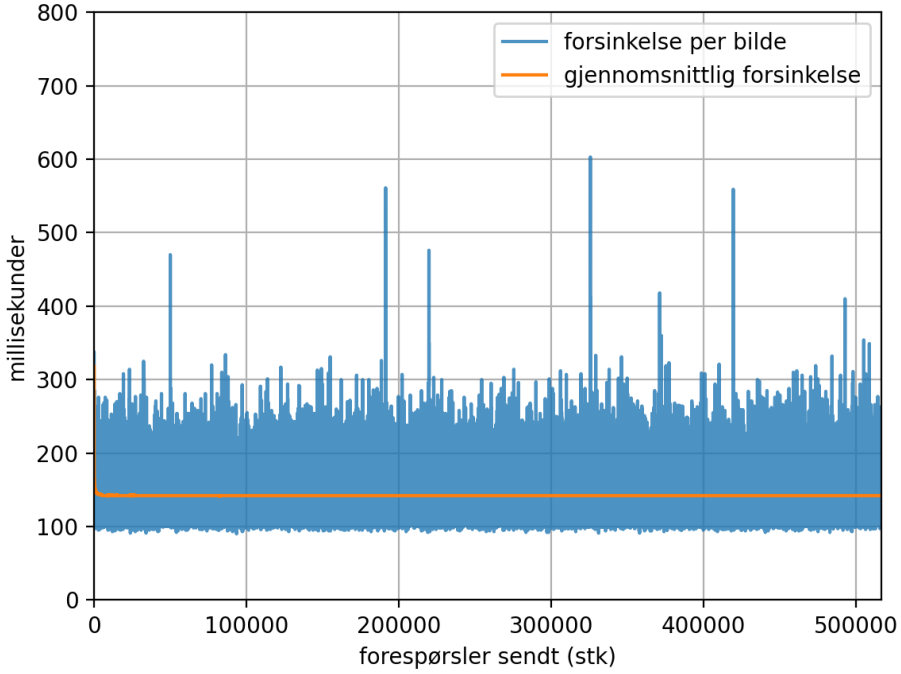
Figur 31 viser alle de testede modellene plottet med hensyn på F1-score mot bildefrekvens systemet kan operere ved når tilsvarende modell er implementert. RT-DETRl har høyest F1-score med en verdi på 0.713. F1-score stagnerer ved bruk av større modeller i YOLOv8-familien. Samtlige modeller med half-precision har lavere F1-score en YOLOv8n-FP32




Figur 31: F1 score plottet mot maks bildefrekvens for objektdeteksjonsmodeller på FP32 og FP16-format

### 4.3.1 Stabilitetstest

Figur 32 viser at systemet er stabilt med implementert yolov8l fp16 modell, og kjørt over 7 timer. Gjennomsnittlig forsinkelse stabiliserer seg på 142ms. Forsinkelse per bilde varierer fra 100 til 300ms med noen utliggere.




Figur 32: Forsinkelse i systemet ved 20.5fps, tilsvarende 80km/h. Kjørt over 7 timer.

## 4.4 Feildeteksjon

Figur 33 viser resultatet av inferens, gjort med yolov8n, på bilder med og uten feil. I begge tilfellene detekterer modellen riktig rekkverkstype, med en confidence på henholdsvis 0.97 og 0.95. Objektdeteksjonsmodellen viser her ingen markant endring i confidence på bilder med og uten feil.


(a) Bilde uten forstyrrelse (b) Bilde med forstyrrelse av solkinn

Figur 33: Objektdeteksjon gjort på bilde med og uten forstyrrlese, gjennomført med YOLOv8n

I figur 34 vises resultatet av klassifisering gjennomført av klassifiseringsmodellen. Bilde 34a er klassifisert som normal, med 99% sikkerhet. Bilde 34b er klassifisert med solskinn, med 100% sikkerhet. Figur 32 viser korrekt klassifikasjon av et undereksponert bilde. Objektedetksjonsmodellen predikerer korrekte klasser på tilsvarende bilde, med relativt høy sikkerhet.


(a) Bilde uten forstyrrelse (b) Bilde med forstyrrelse av solskinn

Figur 34: bildeklassifisering gjort på bilde med og uten forstyrrelse, gjennomført med YOLOv8n-cls


(a) Undereksponert bilde, tydet av deteksjonsmodell (b) Undereksponert bilde klassifisert av feildeteksjonsmodell

Figur 35: Resultat fra objektdeteksjon og klassifisering av undereksponert bilde

### 4.4.1 Feildeteksjon prediksjoner

Bildene i figur 36 viser bildeklassifiseringer gjort av feildeteksjonsmodellen. Bildene representerer tilfeller som er marginalt *feilfrie* i henhold til retningslinjene satt for *Normale* bilde. Bildene viser at feildeteksjonsmodellen klassifiserer bildene som normalt med henholdsvis 92% og 94% sikkerhet på bilde 36a og 36b.


(a) Bilde uten forstyrrelse (b) bilde uten forstyrrelse

Figur 36: Eksempler på klassifisering av bilder uten optisk forstyrrelse bilder med YOLOv8n-cls

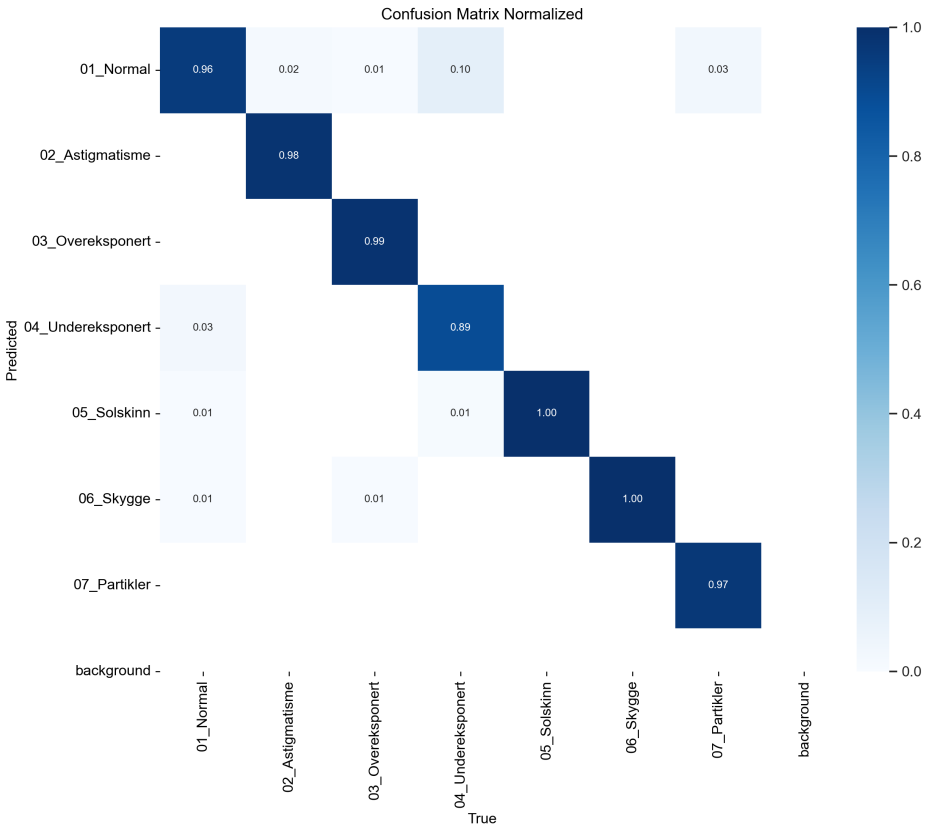
Bildene i figur 37 inneholder to instanser av bilder med forstyrrelser. Bilde 37a viser ett bilde forstyrret av solskinn. feildeteksjonsmodellen klassifiserer bilde i korrekt klasse med 100% sikkerhet. Bilde 37b viser ett undereksponert bilde. Feildeteksjonen klassifiserer bilde korrekt, men med en lav treffsikkerhet på 44%. Feildeteksjonsmodellen viser videre 43% sannsynlighet for skyggelagt motiv.


(a) Bilde med solskinn (b) Undereksponert bilde

Figur 37: Klassifisering gjort på bilde med og uten feil, gjennomført med YOLOv8n-cls

### 4.4.2 Feilmatrise

Figur 38 viser feilmatrisen (2.9.4) til klassifiseringsmodellen, hvis ansvar er å detektere optiske feil i bildedataen. Matrisen viser 96% *true positive* på klassifisering gjort på *normale bilder*. Det noteres at modellen presterer betraktelig lavere på undereksponerte bilder, med 89% *true positive* undereksponerte bilder.




Figur 38: Normalisert feilsmatrise, YOLOv8n-cls

### 4.4.3 Modelltreffsikkerhet

Tabell 8 viser treffsikkerheten til feildeteksjonsmodellen. Det er benyttet en yolov8n-cls modell, som er den minste klassifiseringsmodellen levert av Ultralytics [29]. Tabellen viser en top1 treffsikkerhet

(2.9.11) på $\approx 0.953$, samt top5 treffsikkerhet på $\approx 0.999$. Top1 treffsikkerheten indikerer at modellen korrekt klassifiserte 95.337% av bildene i testsett.

| Model | top1 acc | top5 acc |
|---|---|---|
| yolov8n-cls | 0.95337 | 0.99878 |

Tabell 8: top1 og top5 treffsikkerhet for YOLOv8n-cls

# 5 Diskusjon

## 5.1 Objektdeteksjonsmodell

Resultatene fra objektdeteksjonsmodellen viser ikke en direkte sammenheng mellom F1-score og modellstørrelse. Fra tabell 6 for FP32 resultater kan man se at YOLOv8(m,l) oppnår lavere F1-score enn YOLOv8s. Fra Ultralytics sine resultater fra testing(2) ser man en økning i mAP(50-95) for større modeller. Selv om testingen gjennomført i dette prosjektet er basert på en annen metrikk ville vi forventet bedre resultater fra de større modellene i våre tester. Det antas at denne irregulariteten i resultatene våre stammer fra størrelsen på treningsdatasettet. Datasettet som har blitt brukt til trening under dette prosjektet er på totalt 1166 bilder. Sammenliknet med COCO-datasettet(3.2) som Ultralytics gjennomfører sin trening og testing på er dette et veldig lite datasett. Vår teori er at trening på et lite datasett ikke tillater de større modellene å utnytte de ekstra vektene sine til å effektivt ekstrapolere nyttig informasjon fra datasettet. Tilfellet blir heller at disse modellene etter få epoker når sitt fulle potensial for trening på det gitte datasettet. Dette støttes ved å se på stagneringen som forekommer i precision og recall grafen til YOLOv8n under validering(29), komparativt til tapet under trening som fortsetter å synke. Fra dette kan vi se at modellen ikke klarer å trekke ut ytterligere informasjon som vil hjelpe den å predikere på usette bilder. Det ansees som sannsynlig at trening på større datasett hadde ført til bedre F1-score for de større modellene. Fra spesifikasjonskrav nr.3 var det ønsket å sammenlikne modeller basert på F1-score. Med bakgrunn i problematikken rundt de resultatene presentert og diskutert i dette avsnittet ansees dette ikke som mulig.

Til tross for trening på et lite datasett oppnår modellene relativt gode resultater. I feilmatrisen til YOLOv8n (figur 28), er det synlig at modellen har høy presisjon og recall for de tre skinnetypene. Skinner er de største objektene i datasettet, og har har flest instanser. Dette oppfattes derfor som et naturlig resultat. For stolper og bolter ser vi mer varierende resultater. Boltene og stolpene er mindre objekter og forventes derfor å være vanskeligere å detektere. Dette er fordi det er færre piksler i bildet som representerer disse objektene. Fra dette vil det være vanskeligere å hente ut relevante generaliserende features for disse objektene. Samtidig er det også noen av disse klassene som har få instanser. Som også vil gjøre det vanskeligere for modellen å lære seg disse objektene. Gruppen anser det derfor som sannsynlig at ved trening på et større datasett ville treffsikkerheten for stolper og bolter økt.

## 5.2 Objektdeteksjon for feildeteksjon

Resultatene i 4.4 viser at YOLOv8n klarer å detektere riktig rekkverkstype, med høy *confidence*, selv om det er forstyrrelse på bildet. Til tross for at dette viser en styrke ved ytelsen til objektdeteksjonsmodellen, belyser det en utfordring i forbindelse med problemstilling 2. Prosjektets opprinnelige plan var å benytte fall i *confidence* som en indikasjon på forstyrrelser i bildet, men resultatene viste liten sammenheng mellom modellenes *confidence* og forstyrrelser i bildet. En årsak til dette kan være at modellene har blitt eksponert for bilder med forstyrrelse under trening. Det finnes 123 instanser av bilder forstyrret av solskinn (5) i det opprinnelige datasettet. Modellene er derfor trent på å gjøre deteksjoner på bilder med denne typen forstyrrelse, men det er usikkert om dette faktisk har en innvirkning.

En mulighet for å videre utforske denne løsningen hadde vært å trene modellene på data uten tilfeller av bildeforstyrrelse. Det er mulig at dette hadde ført til større sammenheng mellom confidence på rekkverksdeteksjoner, og forstyrrelser i bildet. En alternativ fremgangsmåte kunne vært å annotere bildeforstyrrelser i datasettet som ble brukt for å trene objektdeteksjonsmodellene. På denne måten ville modellen klassifisere og lokalisere forstyrrelser i bildet. Resulterende bounding bokser ville videre kunne benyttes til å beregne overlapp mellom forstyrrelse og rekkverk. Denne informasjonen kunne senere bli benyttet av iSi til å bestemme om forstyrrelsen var for grov til å benytte bildet videre.

## 5.3 Klassifiseringsmodell for feildeteksjon

Som følge av at prosjektet ikke fant en sammenheng mellom confidence ved rekkverksdeteksjoner og optiske forstyrrelser i bildet, ble det bestemt å benytte feilklassifiseringsmodellen for å se etter forstyrrelser i alle bildene. Denne løsningen medfører at hvert bilde må prosesseres av både objektdeteksjonsmodellen og feil-klassifiseringsmodellen, som gjør at hvert bilde vil kreve mer prosesseringskraft.

Fra feilmatrisen til klassifiseringsmodellen som vist i figur 38, er det mulig å få en innsikt i modellens ytelse. Som introdusert i seksjon 2.9.4, viser feilmatrisen korrekte og inkorrekte prediksjoner for hver klasse. Matrisen demonstrerer en generelt høy treffsikkerhet for samtlige klasser, med treffsikkerhet $\geq 96\%$. Med unntak av en utstikker, undereksponerte bilder. 10% av alle undereksponerte bilder ble klassifisert som normale. Dette kan være en konsekvens av hvilke bilder som ble brukt i treningsdatasettet. Datasettet inneholdt kun bilder med kraftige forstyrrelser. Dette ble gjort med intensjon om å trene modellen til å kun klassifisere feil dersom bildet var av for lav kvalitet til at iSi kunne bruke det videre. Bilder med mindre forstyrrelser, men som fortsatt er brukbare vil dermed ikke bli klassifisert med feil. En modell trent på disse prinsippene vil være mer praktisk relevant for iSi, men det vil samtidig også stille høyre krav til modellen.

Som presentert i tabell 8, har feildeteksjonsmodellen en top1 treffsikkerhet (2.9.11) på 95.337%. Dette ansees som ett godt resultat. Den samme modellen oppnår en top1 treffsikkerhet på 69.0% på ImageNet datasettet. Dette er presentert i tabell 1. Likevell er ikke disse metrikkene direkte sammenlignbare. ImageNet datasettet inneholder 1331167 bilder, fordelt på 1000 forskjellige klasser. For at den implementerte feildeteksjonsmodellen skal være sammenliknbar, er det behov for ett betraktelig større og mer variert datasett. Gruppen anerkjenner også at ettersom klassifiseringsmodellen er trent og validert på delvis syntetisk data, vil resultater trolig være noe misvisende og ikke direkte overførbare til reelle applikasjoner.

Klassifiseringsmodellen som benyttes er en yolov8n-cls. Denne nano-modellen er den minste klassifiseringsmodellen levert av Ultralytics. For å sikre korrekt klassifisering, kan det være aktuelt å benytte en større klassifiseringsmodell i det endelige systemet. Dette vil påvirke den maksimale bildefrekvensen systemet som helhet kan håndtere, og det vil være aktuelt å prioritere vekting mellom deteksjon og klassifiseringsmodell med hensyn på oppdragsspesifikke behov.

## 5.4 Systemarkitektur

Med bakgrunn i spesifikasjon nr.1 har det i løpet av prosjektutviklingen blitt tatt hensyn til at produktet skal være kompatibelt med iSi sitt InSight-system. Et tiltak som er implementert er bruken av en watchdog for filmonitorering. Dette tillater plattformen å kjøre inferens på bilder som en separat prosess, uten å påvirke dagens systemarkitektur.

Systemarkitekturen er også utviklet med hensyn på enkel implementasjon i dagens system. Det er fortsatt lagt opp til at iSi må gjennomføre noen mindre installasjonstiltak. De vil være nødt til å tilføre strøm til NVIDIA Jetson, og installere nødvendig programvare. For å forsøke å simplifisere denne prosessen, ble det utviklet en dockerfil med tilhørende installasjonscript som konfigurerer serverprogramvaren.

Det hadde vært mulig å integrere ROS2 i systemet. Dette ville bidratt til å øke skalerbarheten og modulariteten til produktet. Dette er fordi ROS2 tillater for nodebasert kommunikasjon der serveren kunne publisert til en node, og brukergrensesnittet og eventuelle andre prosesser kunne lyttet på denne noden. Denne løsningen hadde fungert som en erstatning for *Shareablelist*. For å unngå *race conditions* ved bruken av *shareablelist* krever det at man bruker en *Lock*. Hvis den nåværende plattformen skal skaleres vil det dermed oppstå ekstra tidsforsinkelse fordi kun en prosess kan skrive eller lese fra listen samtidig. ROS2 hadde dermed vært en mer skalerbar løsning, men er ikke implementert da det hadde krevd at iSi gjennomført større endringer på dagens system.

### 5.4.1 Skalerbarhet

Systemarkitekturen er utviklet med hensyn på å skape en modulær og skalerbar løsning. I denne sammenheng trekkes Triton frem som et sentralt element i arkitekturen som bidrar til modularitet. Tritonserveren er ikke bundet til modellene dette prosjektet har utforsket, og kan modifiseres til å benytte andre maskinlæringsmodeller etter ønske. Implementeringen av Ensemble modell tillater også for senere utvidelse av dataflyten fra Tabell (25) med flere parallelle modeller, eller egenutviklet dataprosessering. Sistnevnte vil være aktuelt hvis det implementeres nye objektdeteksjonsmodeller som ikke er avhengige av postprosessering med *nms*.

### 5.4.2 Brukergrensesnitt

Brukergrensesnittet er designet med intensjon om å presentere relevant informasjon, uten å virke forstyrrende for sjåfør. Ett brukergrensesnitt som krever mye fokus av sjåfør, er ikke forenlig med trafikkregler eller generell trafikksikkerhet. Det er derfor tatt beviste valg om å benytte løpende gjennomsnitt for presentasjon av tallverdier, samt benytte gråtone farger som ikke fanger unødvendig oppmerksomhet. Prosjektet bemerker at brukergrensesnittet har blitt behandlet som et konseptbevis, og at dets visuelle fremtoning bærer preg av dette.

### 5.4.3 Lisenser

Dyplæringsmodellene, YOLO og RT-DETR, som har blitt fremlagt i dette prosjektet har blitt hentet fra Ultralytics. Slik de anvendes i dette prosjektet er de underlagt *GNU General Public License*(2.15). Dette betyr at hele sanntidsplatformen som fremlegges også underlegges samme lisens hvis den skal benyttes offentlig. For kommersielle aktører som iSi vil det trolig ikke være aktuelt å benytte programvare underlagt denne lisensen. Ved å implementere egne dyplæringsmodeller på sanntidsplattformen vil kravet om GNU-lisens opphøre. Det finnes også muligheter for å betale for bedriftslisenser, som gjør at nevnte modeller kan benyttes uten å være underlagt GNU.

## 5.5 Systemsikkerhet

### 5.5.1 Stabilitetstest

Som beskrevet i seksjon(3.11.2) ble systemet testet for langtidsstabilitet. Testen ble utført på YOLOv8l FP16, som var den største modellen som nådde hastighetskravet satt i systemspesifikasjon nr.2. Resultatene gir ingen indikasjon på at systemet blir ustabilt ved kjøring over lengre perioder. Stabilitetstesten ble gjennomført med forbehold om at operativ hastighet blir fulgt under hele perioden. I et praktisk scenario kan det hende at sjåføren overstiger den operative hastigheten som da potensielt kunne ført til at systemet ble ustabilt. En mulighet for å teste dette scenario er å benytte en variabel bildefrekvens som økes over operativ hastighet i gitte perioder. På denne måten ville det vært mulig å kartlegge om systemet hadde klart å stabilisere seg igjen etter hastighetsøkninger.

### 5.5.2 Race-conditions

Som presentert i seksjon 3.9 ble det implementert en *Lock* for å unngå at trådene på klientprogrammet og brukergrensesnittprosessen modifiserer *Shareblelist* samtidig. Det ble også implementert en *Semafor* for å unngå simultan modifisering ved logging av feilmeldinger fra forskjellige tråder. Begge disse tiltakene ble implementert for å unngå *race conditions*. Det har ikke blitt observert noe problematikk rundt dette etter at disse løsningene ble implementert. Multiprosesseringsprogrammer er vanskelige å feilsøke, og er ikke deterministiske iforhold til når eventuelle feil kan oppstå. På bakgrunn av dette ansees det ikke som fullstendig kartlagt at systemet er fritt for *race conditions*. For å bestemme dette definitivt kunne blitt gjennomført en mer omfattende testprosess av programvaren. Dette ville ført til økt innsikt i robustheten til programvaren.

### 5.5.3 minneutmattelse

Minneutmattelse observeres ikke på klientmaskinen under våre tester. Fra begrensning 3 er det spesifisert at klientmaskinen som brukes under prosjektgjennomførelsen vil prestere bedre enn den faktisk PCen på bil-riggen. Dette betyr at det kan oppstå minneutmattelse på klientsiden selv om det ikke observeres fra våre tester. For å unngå at dette oppstår ble det implementert en *timeout* som avslutter tråden dersom levetiden overstiger 10 sekunder.

Et annet usikkerhetsmoment er i hvilken grad minneutmattelse på klient og serversiden er unngått. Den operative hastighetstesten antyder en flaskehals på Nvidia Jetson. Dette betyr ikke nødvendigvis at dette ville vært tilfelle med implementasjon på bilriggen til iSi. Som presentert i begrensning 4 har klientmaskinen som brukes under prosjektutførelsen bedre CPU og prosesseringskraft en PCen bil-riggen er utstyrt med. Dette kan bety at man hadde oppnådd et annet resultat dersom den faktiske klientmaskinen ble tatt i bruk.

## 5.6 FNs bærekraftsmål

Bærekraftsmålene som prosjektet har valgt å fokusere er FNs mål nr:3.6 og 9.1. Målene omhandler arbeidet mot en tryggere og mer pålitelig infrastruktur. Målene anses spesielt som relevante opp mot arbeidet for å videreutvikle iSi inSight systemet. iSi inSight bidrar i dag indirekte, gjennom kartlegging av defekter på rekkverk langs norske veier, til en tryggere infrastruktur. Dette er et viktig arbeid, som hvis etterfulgt på en god måte potensielt kan redde liv. Vår forhåpning er at løsningen presentert i denne rapporten vil kunne bidra positivt til videreutviklingen av dette produktet og ved dette gjøre norske veier tryggere.

# 6 Konklusjon

Prosjektet har utforsket ulike objektdeteksjonsmodellers hastighet og ytelse som en del av den fremlagte sanntidsplattformen. I henhold til spesifikasjonskrav 2 har det blitt bevist at plattformen kan anvende komplekse modeller som YOLOv8l med half precision, og samtidig opprettholde hastighetskravet på 20.5 bilder/s. Vi mener at dette belyser potensialer ved systemarkitekturen som rammeverk for sanntids objektdeteksjon på iSi InSight.

I spesifikasjonskrav nr.1 defineres det at plattformen skal være modulær og kompatibel med dagens iSi Insight system. Med bakgrunn i momenter diskutert i seksjon 5.4 mener vi at dette kravet er tilstrekkelig oppfylt. Hovedmomentene vi ønsker å trekke frem i denne sammenheng er implementasjonen av en Triton server på NVIDIA Jetson og oppsettet av Docker filer. Tritonserveren er utviklet for å gjøre det enkelt for iSi å implementere egne dyplæring -og prosesseringsmodeller og bygger derfor oppunder modulariteten til plattformen. Docker filene utviklet gir en enkel installasjonsprosess og legger til rette for videreutvikling av systemet i fremtiden.

Vi mener det ikke kan konkluderes med hvilke av objektdeteksjonsmodellene som egner seg best på plattformen basert på våre resultater for modellnøyaktighet. Vi ser at modellene på generell basis oppnår relativt god F1-score, selv ved trening med lite datasett, men anerkjenner at disse resultatene ikke representerer et troverdig sammenlikningsgrunnlag for modellene. Til tross for dette opplever vi at anvendelsen av modellene har gitt innsikt i hvordan sanntids objektdeteksjon kan implementeres på inSight systemet.

Vi mener derved at sanntidsplattformen oppfyller systemkrav 1-3, og videre tilfredsstiller problemstilling 1. Gjennom dette mener vi at er mulig å gjennomføre sanntids objektdeteksjon på iSi InSight systemet.

Prosjektet har videre tatt stilling til om sanntidsplattformen kan benyttes for å oppdage feil i datainnsamlingsprosessen. I henhold til systemkrav 4 ble det ble initielt undersøkt om confidence score ved deteksjon av rekkverk kunne brukes som en pålitelig indikator på tilfeller av optiske forstyrrelser. Denne rapporten finner ingen direkte sammenheng mellom objektdeteksjonsmodellenes confidence og forstyrrelser i bildet.

For å videre etterkomme spesifikasjonskrav 4, ble det utviklet og implementert en klassifiseringsmodell på systemet. Denne modellen ble trent på delvis syntetisk data, for å replikere effekten av optiske feil som ikke var godt representert i datasettet. Som diskutert i seksjon 5.3, kan denne modellen klassifisere optiske forstyrrelser i bilder, med høy presisjon. Som videre diskutert, innfører størrelse på datasett, og tilstedeværelsen av syntetisk data usikkerhet rundt modellens ytelse i en reell applikasjon. Feildeteksjonen kan ansees som ett konseptbevis, men regnes ikke som tilfredsstillende for det endelige systemet uten ytterligere trening.

Prosjektet mener at sanntids-objektdeteksjon kan benyttes for å oppdage feil i datainnsamlingsprosessen. Fra våre resultater anses det som mest hensiktsmessig å benytte en egen klassifiseringsmodell til dette formål.

## 6.1 Videre arbeid

Vi tror at arbeidet gjennomført under dette prosjektet kan fungere som et fundament for implementering av sanntids objektdeteksjon på iSi inSight. For å videreutvikle plattformen som har blitt presentert foreslås følgende.

### 6.1.1 Optimalisering av Tritonserver

For å gi ytterligere frihet ved implementering av objektdeteksjonsmodell på plattformen nevnes dynamic-batching (2.13.3) og instancegroups 2.13.3 som videre optimaliseringstiltak å utforske på Tritonserveren. Prosjektets oppfatning er at dette kunne ført til høyere gjennomstrømning og lavere systemforsinkelse for modellene, og dermed økt den totale hastigheten til systemet.

### 6.1.2 Valg av hardware

NVIDIA Jetson Orin Nano som tas i bruk under prosjektgjennomførelsen begrenser sanntidsplattformens av prosesseringskraft. Resultatene fra hastighetstesten vil derfor ikke gi et totalt overblikk over hvor store modeller iSi AS kunne ha implementert i inSight systemet sitt. Ved å benytte kraftigere GPUer, ville større modeller vært mulig å benytte. Det vil være aktuelt å undersøke om det er økonomisk og praktisk bærekraftig å benytte større GPUer. Dette for å implementere de større modellene som i dag brukes i offline inspeksjon, på sanntidsplattformen.

### 6.1.3 Robusthetsoptimalisering

For at plattformen skal virke som tiltenkt, er det viktig at plattformen reagerer forutsigbart i alle situasjoner. Omfattende robusthetstesting og optimalisering, vil derfor være nødvendig. Det vil være behov for å etablere protokoller for håndtering av svakheter i systemets robusthet.

# Referanser

[1] iSi AS. «Om oss». (2024), adresse: https://isi.no/team (sjekket 26.01.2024).

[2] Microsoft. «Hva er dyplæring?» (2024), adresse: https://azure.microsoft.com/nb-no/resources/cloud-computing-dictionary/what-is-deep-learning (sjekket 26.01.2024).

[3] iSi AS. «Produkter og tjenester». (2024), adresse: https://isi.no/products (sjekket 26.01.2024).

[4] Cloudfare. «AI inferencing vs. Training: What is the difference?» (2024), adresse: https://isi.no/team (sjekket 15.05.2024).

[5] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman og G. Hinton, «Backpropagation and the brain», *Nature Reviews Neuroscience*, årg. 21, nr. 6, s. 335–346, 2020.

[6] NVIDIA. «jetson-orin». (2024), adresse: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/ (sjekket 15.05.2024).

[7] cpu-monkey. «Intel Xeon E-2278GE vs Intel Core i7-13700H». (2024), adresse: https://www.cpu-monkey.com/en/compare_cpu-intel_xeon_e_2278ge-vs-intel_core_i7_13700h (sjekket 15.05.2024).

[8] FN. «Industri, innovasjon og infrastruktur». (), adresse: https://fn.no/om-fn/fns-baerekraftsmaal/industri-innovasjon-og-infrastruktur (sjekket 16.05.2024).

[9] V. Ghorakavi. «What is a neural network?» (), adresse: https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/ (sjekket 19.05.2024).

[10] J. Schmidhuber, «Deep learning in neural networks: An overview», *Neural networks*, årg. 61, s. 85–117, 2015.

[11] GeeksforGeeks. «Introduction to Convolution Neural Network». (), adresse: https://www.geeksforgeeks.org/introduction-convolution-neural-network/?ref=header_search (sjekket 20.05.2024).

[12] K. O'shea og R. Nash, «An introduction to convolutional neural networks», *arXiv preprint arXiv:1511.08458*, 2015.

[13] Wikipedia. «Convolutional neural network». (), adresse: https://en.wikipedia.org/wiki/Convolutional_neural_network (sjekket 19.05.2024).

[14] J. Riebesell, «Convolution Operator», 2021. adresse: https://tikz.net/conv2d/.

[15] Ø. B. D. (NMBU). «konvolusjon». (2023), adresse: https://snl.no/konvolusjon (sjekket 20.05.2024).

[16] Wikipedia. «Kernel (image processing)». (), adresse: https://en.wikipedia.org/wiki/Kernel_(image_processing) (sjekket 19.05.2024).

[17] ImageNet. «ImageNet». (2024), adresse: https://paperswithcode.com/dataset/imagenet (sjekket 03.05.2024).

[18] Paperswithcode. «MS COCO». (2024), adresse: https://paperswithcode.com/dataset/coco (sjekket 15.05.2024).

[19] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan og S. Belongie, «Feature pyramid networks for object detection», i *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, s. 2117–2125.

[20] S. Liu, L. Qi, H. Qin, J. Shi og J. Jia, «Path aggregation network for instance segmentation», i *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, s. 8759–8768.

[21] Z. Zheng, P. Wang, D. Ren mfl., «Enhancing geometric factors in model learning and inference for object detection and instance segmentation», *IEEE transactions on cybernetics*, årg. 52, nr. 8, s. 8574–8586, 2021.

[22] X. Li, W. Wang, L. Wu mfl., «Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection», *Advances in Neural Information Processing Systems*, årg. 33, s. 21 002–21 012, 2020.

[23] U. Ruby og V. Yendapalli, «Binary cross entropy with deep learning technique for image classification», *Int. J. Adv. Trends Comput. Sci. Eng*, årg. 9, nr. 10, 2020.

[24] J. Terven og D. Cordova-Esparza, «A comprehensive review of YOLO: From YOLOv1 to YOLOv8 and beyond», *arXiv preprint arXiv:2304.00501*, 2023.

[25] D. Reis, J. Kupec, J. Hong og A. Daoudi, «Real-time flying object detection with YOLOv8», *arXiv preprint arXiv:2305.09972*, 2023.

[26] Burhan-Q, glenn-jocher, Laughing-q, AyushExel og fcakyon. «YOLOv8». (2023), adresse: https://docs.ultralytics.com/models/yolov8/ (sjekket 15.05.2024).

[27] RangeKing. «Brief summary of YOLOv8 model structure». (2023), adresse: https://github.com/ultralytics/ultralytics/issues/189 (sjekket 16.05.2024).

[28] glenn-jocher. «Object Detection Datasets Overview». (2024), adresse: https://docs.ultralytics.com/datasets/detect/.

[29] G. Jocher, A. Chaurasia og J. Qiu, *Ultralytics YOLOv8*, versjon 8.0.0, 2023. adresse: https://github.com/ultralytics/ultralytics.

[30] Ultralytics. «Image Classification». (2024), adresse: https://docs.ultralytics.com/tasks/classify/#train (sjekket 20.05.2024).

[31] ultralytics. «ImageNet Dataset». (2024), adresse: https://github.com/ultralytics/ultralytics/blob/main/ultralytics/cfg/datasets/ImageNet.yaml (sjekket 16.04.2024).

[32] Ultralytics. «Image Classification Datasets Overview». (2024), adresse: https://docs.ultralytics.com/datasets/classify/#dataset-structure-for-yolo-classification-tasks (sjekket 20.05.2024).

[33] Y. Zhao, W. Lv, S. Xu mfl., «Detrs beat yolos on real-time object detection», *arXiv preprint arXiv:2304.08069*, 2023.

[34] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov og S. Zagoruyko, «End-to-end object detection with transformers», i *European conference on computer vision*, Springer, 2020, s. 213–229.

[35] Nvidia. «Train With Mixed Precision». (), adresse: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html (sjekket 16.05.2024).

[36] R. Padilla, W. L. Passos, T. L. Dias, S. L. Netto og E. A. Da Silva, «A comparative analysis of object detection metrics with a companion open-source toolkit», *Electronics*, årg. 10, nr. 3, s. 279, 2021.

[37] A. Anwar. «What is Average Precision in Object Detection  Localization Algorithms and how to calculate it?» (), adresse: https://towardsdatascience.com/what-is-average-precision-in-object-detection-localization-algorithms-and-how-to-calculate-it-3f330efe697b (sjekket 20.05.2024).

[38] C. objects in Context). «Detection Evaluation». (2024), adresse: https://cocodataset.org/#detection-eval (sjekket 20.05.2024).

[39] R. Nagda og Medium. «Evaluating models using the Top N accuracy metrics». (2019), adresse: https://medium.com/nanonets/evaluating-models-using-the-top-n-accuracy-metrics-c0355b36f91b (sjekket 20.05.2024).

[40] R. Merritt. «Why GPUs Are Great for AI». (2023), adresse: https://blogs.nvidia.com/blog/why-gpus-are-great-for-ai/ (sjekket 20.05.2024).

[41] Docker. «Docker overview». (2024), adresse: https://docs.docker.com/get-started/overview/ (sjekket 15.05.2024).

[42] AWS. «What's the Difference Between Docker Images and Containers?» (2024), adresse: https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/ (sjekket 15.05.2024).

[43] V. Lobo. «What is Docker? How to create a Docker image and execute an application within a container ?» (2023), adresse: https://www.linkedin.com/pulse/what-docker-how-create-image-execute-application-within-varun-lobo/ (sjekket 20.05.2024).

[44] NVIDIA. «embedded systems». (2024), adresse: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/ (sjekket 15.05.2024).

[45] Nvidia. «CUDA Toolkit». (), adresse: https://developer.nvidia.com/cuda-toolkit (sjekket 04.05.2024).

[46] N. Developer. «NVIDIA TensorRT». (2024), adresse: https://developer.nvidia.com/tensorrt (sjekket 15.05.2024).

[47] NVIDIA. «Jetson Orin Nano Developer Kit User Guide». (2022), adresse: https://developer.nvidia.com/embedded/learn/jetson-orin-nano-devkit-user-guide/index.html (sjekket 20.05.2024).

[48] NVIDIA. «NVIDIA Triton Inference Server». (2024), adresse: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html (sjekket 26.04.2024).

[49] SNL. «HTTP». (2024), adresse: https://snl.no/HTTP (sjekket 15.05.2024).

[50] gRPC. «gRPC». (2024), adresse: https://grpc.io/ (sjekket 15.05.2024).

[51] NVIDIA. «Triton Architecture». (2024), adresse: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/architecture.html (sjekket 20.05.2024).

[52] ONNX. «Open Neural Network Exchange». (2019), adresse: https://onnx.ai/ (sjekket 26.04.2024).

[53] NVIDIA. «Optimization». (2024), adresse: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/optimization.html (sjekket 26.04.2024).

[54] Nvidia. «Python Backend». (2024), adresse: https://github.com/triton-inference-server/python_backend?tab=readme-ov-file (sjekket 26.04.2024).

[55] Anaconda. «Miniconda docs». (), adresse: https://docs.anaconda.com/free/miniconda/index.html (sjekket 12.05.2024).

[56] B. Lutkevich og B. Posey. «race condition». (2024), adresse: https://www.techtarget.com/searchstorage/definition/race-condition (sjekket 15.05.2024).

[57] Y. Technologies. «Understanding Semaphores: Synchronization and Resource Management in Concurrent Programming». (2024), adresse: https://www.linkedin.com/pulse/understanding-semaphores-synchronization-resource/ (sjekket 15.05.2024).

[58] «GNU General Public License, version 3». (2007), adresse: https://www.gnu.org/licenses/gpl-3.0.html (sjekket 18.05.2024).

[59] AWS. «Amazin SageMaker Developer Guide, Object Detection Hyperparameters». (), adresse: https://docs.aws.amazon.com/sagemaker/latest/dg/object-detection-api-config.html (sjekket 21.05.2024).

[60] K. Skarsfjord. «Optikk og Linser». (2002), adresse: https://snl.no/astigmatisme_-_fysikk (sjekket 08.05.2024).

[61] S. Johannes Skaar. «Antirefleksbelegg». (2024), adresse: https://snl.no/antirefleksbelegg (sjekket 06.05.2024).

[62] H. Photo. «Eksponeringskompensasjon». (2024), adresse: https://www.scandinavianphoto.no/kunnskap/eksponeringskompensasjon (sjekket 06.05.2024).

[63] T. Holtsmark. «astigmatisme (fysikk)». (2020), adresse: https://snl.no/astigmatisme_-_fysikk (sjekket 06.05.2024).

[64] D. Mount og R. Eastman. «Procedural Generation: 2D Perlin Noise». (2018), adresse: https://www.cs.umd.edu/class/spring2018/cmsc425/Lects/lect13-2d-perlin.pdf (sjekket 20.05.2024).

[65] Ultralytics. «hjemmeside». (2024), adresse: https://www.ultralytics.com/ (sjekket 15.05.2024).

[66] glenn-jocher og RizwanMunawar. «Baidu's RT-DETR: A Vision Transformer-Based Real-Time Object Detector». (2024), adresse: https://docs.ultralytics.com/models/rtdetr/ (sjekket 15.05.2024).

[67] C.-Y. Wang og H.-Y. M. Liao, «YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information», 2024.

[68] glenn-jocher, RizwanMunawar og Laughing-q. «COCO Dataset». (2024), adresse: https://docs.ultralytics.com/datasets/detect/coco/ (sjekket 15.05.2024).

[69] glenn-jocher, RizwanMunawar og abirami-vina. «Performance Metrics Deep Dive». (2024), adresse: https://docs.ultralytics.com/guides/yolo-performance-metrics/ (sjekket 15.05.2024).

[70] dependabot[bot], glenn-jocher, fcakyon, Laughing-q og Burhan-Q. «Model Training with Ultralytics YOLO». (2024), adresse: https://docs.ultralytics.com/modes/train/ (sjekket 15.05.2024).

[71] glenn-jocher, Burhan-Q og RizwanMunawar. «Model Validation with Ultralytics YOLO». (2024), adresse: https://docs.ultralytics.com/modes/val/ (sjekket 15.05.2024).

[72] Pytorch. «Adamw». (), adresse: https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html (sjekket 16.05.2024).

[73] Pytorch. «LambdaLR». (), adresse: https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.LambdaLR.html (sjekket 16.05.2024).

[74] Optuna. «optuna docs». (2024), adresse: https://optuna.readthedocs.io/en/stable/index.html (sjekket 17.04.2024).

[75] M. N. O. Adam Zewe. «In machine learning, synthetic data can offer real performance improvements». (2022), adresse: https://news.mit.edu/2022/synthetic-data-ai-improvements-1103 (sjekket 02.05.2024).

[76] IBM. «What is synthetic data?» (2024), adresse: https://www.ibm.com/topics/synthetic-data (sjekket 03.05.2024).

[77] opencv. «opencv-python 4.9.0.80». (2023), adresse: https://pypi.org/project/opencv-python/ (sjekket 03.05.2024).

[78] Nvidia. «Triton Inference Server». (), adresse: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/tritonserver (sjekket 20.05.2024).

[79] M. Dukhan og S. E. Frank Barchard. «Half-precision Inference Doubles On-Device Inference Performance». (2023), adresse: https://blog.tensorflow.org/2023/11/half-precision-inference-doubles-on-device-inference-performance.html (sjekket 15.05.2024).

[80] r. asfiyad-nvidia rajeevsrao, *TensorRT/samples/trtexec*, https://github.com/NVIDIA/TensorRT/tree/main/samples/trtexec, 2024.

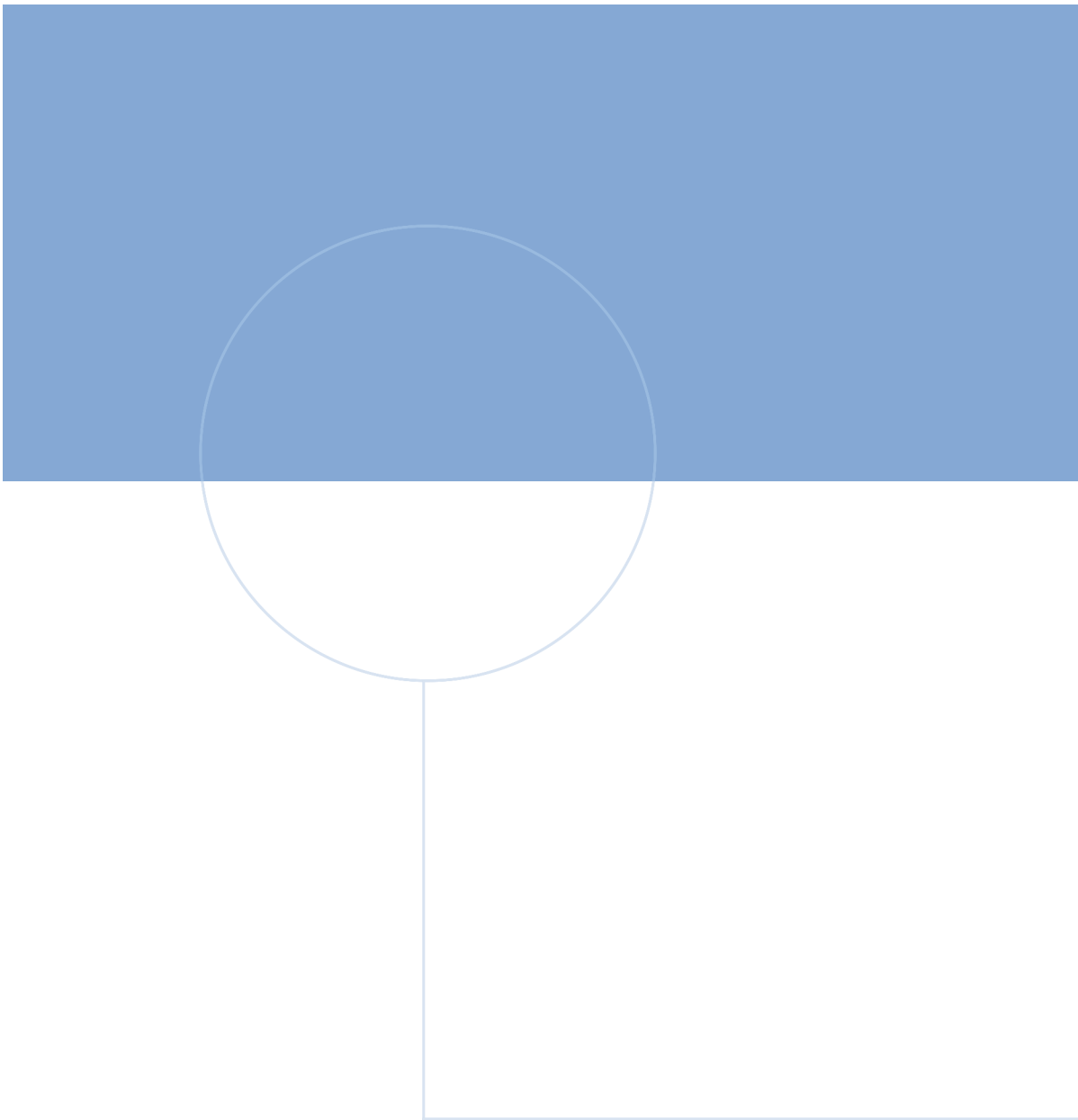
[81] Numpy. «numpy.fromfile». (2022), adresse: https://numpy.org/doc/stable/reference/generated/numpy.fromfile.html (sjekket 20.05.2024).

[82] Python. «multiprocessing.shared$_memory|Sharedmemoryfordirectaccessacrossprocesses$». (2024), adresse: https://docs.python.org/3/library/multiprocessing.shared_memory.html (sjekket 20.05.2024).

[83] P. Thompson. «PyQt5 5.15.10». (2023), adresse: https://pypi.org/project/PyQt5/ (sjekket 15.05.2024).

[84] Woobewoo. «Understanding memory exhaustion error: causes, diagnostics, and prevention tips». (2024), adresse: https://woobewoo.com/glossary/understanding-memory-exhaustion-errors-causes-diagnosis-and-prevention-tips/ (sjekket 15.05.2024).

[85] b. r. Raffaello Bonghi, *jetson_stats*, https://github.com/rbonghi/jetson_stats, 2024.



NTNU
Kunnskap for en bedre verden

iSi