Ansiktsgjenkjenning

Bacheloroppgave Marineingeniør Elektronikk og Data

Narum, Pål Andreas
Sjøkrigsskolen Høsten 2015

Linje:	Elektronikk & Data
År:	2014
Veileder:	Kjetil Børkje
Antall sider:	55
Trykk:	Sjøkrigsskolen trykkeri

Oppgaveformulering

«Bacheloroppgaven skal gi offiserene anledning til å anvende kunnskaper og ferdigheter de har tilegnet seg ved bransjeutdanningen ved Sjøkrigsskolen. Oppgaven skal gi erfaring i å arbeide med en problemorientert oppgave. Den skal gi øvelse i å gjennomføre et større arbeid alene eller i gruppe. Den skal gi offiseren tid til fordypning og trening i å løse teoretiske, eksperimentelle eller praktiske problemstillinger. Det skal legges vekt på å bruke tekniske problemstillinger fra Sjøforsvaret. Det kan være en oppgave gitt av Sjøkrigsskolen alene, i samarbeid med Forsvarets organisasjoner eller i samarbeid med sivil bedrift. Kadettene kan og fremme egne problemstillinger til avdeling for Teknologi for godkjenning.»
(Sjøkrigsskolen, 2013)

Forord

Dette dokumentet tar for seg min Bacheloroppgave ved linjen Marineingeniør Elektronikk og Data ved Sjøkrigsskolen. Oppgaven ble påbegynt i slutten av 4. semester (Mai 2015) og avsluttet i Desember 2015.

Mitt navn er Pål Andreas Narum og jeg er Kadett på Sjøkrigsskolen. Jeg har videregående skole (Norges Toppidrettsgymnas) og 2 års tjeneste i Sjøforsvaret som erfaring før jeg startet på Sjøkrigsskolen i januar 2013.

For meg har denne oppgaven vært utrolig spennende. Jeg har lært mye jeg tviler på at jeg kommer til å få bruk for i den videre tjenesten min, men noe jeg håper jeg, i første omgang, kan fortsette å bedrive med på siden.

Under arbeidet mitt har jeg plaget en del med kadetter med å ta bilder av de. Takk til dere. Og avslutningsvis vil jeg på forhånd beklage for at det er så mange «selfier» av meg selv i oppgaven, det er uting jeg vil ta en god pause fra fremover.

Pål Andreas Narum

Innhold

Oversikter

Figuroversikt:

Tabelloversikt:

1. Sammendrag

Hensikten med denne bacheloroppgaven har i stor grad handlet om å tilegne seg forståelse for bildeprosessering, og hva som ligger bak ansikts- deteksjon og gjenkjenning på en datamaskin. Utførelsen min er en blanding mellom teori innenfor bildeprosessering og gjenkjenningsalgoritmer og programmering av et testprogram i C++.

I testprogrammet kan jeg detektere et ansikt i en sanntidsvideo (tatt opp med internkamera på datamaskinen) og gjenkjenne dette ved å sammenligne det med bilder i en database. Dersom man ikke er lagt inn i databasen kan dette gjøres enkelt ved å ta en rekke bilder og lagre de systematisert i filsystemet. Samtidig må programmet oppdateres med den nye databasen.

Oppgaven beskriver en problemstilling, som jeg har valgt å prøve å svare på ved hjelp av min forståelse av teori og algoritmer, hvordan jeg har utviklet programmet og hvordan jeg har testet programmet, alt i den hensikt å skape mest mulig forståelse for temaet ansiktsgjenkjenning.

Målet mitt har hele tiden vært å kunne lage et program som kjenner igjen et ansikt, noe jeg har fått til. Likevel er det store utfordringer knyttet til programmet og ikke minst forståelsen av hvorfor jeg får disse utfordringene. Etter å ha lagd mange forskjellige testversjoner av programmet sitter jeg igjen med noe som fungerer, men som absolutt har potensiale for å videreutvikles til et mer robust gjenkjenningsprogram.

2. Introduksjon

Jeg vil i dette avsnittet gå igjennom grunnlaget for besvarelsen min, herunder presentere problemstillingen og hvorfor jeg har valgt den samt drøfte avgrensningene til oppgaven.

2.1 Bakgrunn

Jeg brukte en del tid på å finne ut hva jeg ønsket å skrive i en bachelor-oppgave. Linjen Elektronikk og data dekker et bredt spekter med mange mulige retninger innenfor oppgavevalg. Etter en del idemyldring bestemte jeg meg for retningen ansiktsdeteksjon og gjenkjenning. Ansiktsdeteksjon og gjenkjenning innenfor programmering omfatter bildeprosesseringsemnet.

Ansiktsdeteksjon og gjenkjenning er to mye brukte funksjoner i dagens teknologiske hverdag. Blant annet har man begynt å benytte seg av ansiktsgjenkjenning i passkontroller og politiet ser på teknologien i forbindelse med identifisering av personer på overvåkningsvideoer. Ansiktsdeteksjon er f.eks. en svært normal funksjon i de fleste digitale kamera og også et viktig steg i gjenkjenningsprosessen. Formålet er å kunne analysere et bilde for så å skille ut hva som er et ansikt og etter det kunne sammenligne dette med ansikt i en database for å finne ut hvem ansiktet tilhører.

Grunnen til at jeg valgte dette temaet var fordi jeg synes det er relativt interessant at en datamaskin kan etterligne menneskelige analytiske evner. Samtidig synes jeg det er interessant at vi også har begynt å bytte ut menneskelige jobber med datamaskiner, som på en måte forteller hvor bra denne teknologien fungerer og det sier i alle fall noe om potensialet for fremtidens utvikling.

Temaet er stort og meget komplekst, og jeg forventet aldri at jeg kom til å kunne utvikle et slikt program fra bunn av. Da jeg startet å undersøke temaet fant jeg fort ut at det var mange forskjellige selskap som hadde mange ingeniører som har jobbet med dette i flere år. Jeg bestemte meg derfor for at oppgaven min skal omhandle teknisk forståelse av ansiktsdeteksjon og gjenkjenning. Og med det kom jeg frem til problemstillingen min.

2.2 Problemstilling

«Hvordan kan en datamaskin kjenne igjen et ansikt?»

2.3 Avgrensning

Etter å ha undersøkt problemstillingen fant jeg ut at bildeprosessering er en kompleks del av programmeringsverden og at jeg mangler kompetanse for å kunne bygge et slikt program fra bunnen av. Derfor valgte jeg å benytte meg av ferdiglagde biblioteker fra OpenCV. Valg av biblioteker blir bedre forklart senere i oppgaven. For å svare på problemstillingen var også

hensikten min å lage et testprogram, med fokus på forståelse fremfor et ferdig produkt da det er utenfor min kompetanse og tidsramme. Videre så er mulighetene uendelig mange når det kommer til hva man kan bruke ansiktsgjenkjenning til. I oppgaveteksten til bacheloroppgaven står det at det vektlegges å bruke tekniske problemstillinger fra Sjøforsvaret, noe denne oppgaven ikke er direkte. Dette er noe som blir litt drøftet under punkt 8.

2.4 Ambisjonsnivå

Ambisjonsnivået jeg satt for oppgaven var at jeg skulle lære hvordan man prosesserte bilder i programmer, og hvordan man bruker OpenCV sine funksjoner i den hensikt å fortsette å lære videre i etterkant av oppgaven og for å muligens fortsette å utvikle et ferdig brukervennlig program som en videre del av utdannelsen.

Ambisjonen for testprogrammet var at det skulle kunne gjenkjenne ansikt i sann tid fra webkameraet på PC-en og at det skulle være enkelt å legge til nye personer i databasen.

Jeg ønsker også å utforske robustheten til ansiktsgjenkjenning ved å bruke forskjellige prosesser og sammenligne de.

3. Teori

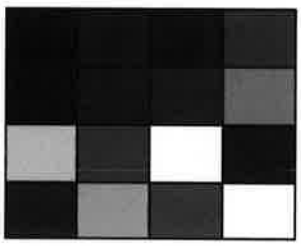
I programmet bruker jeg forskjellige teorier for både ansiktsdeteksjon og gjenkjenning. Disse teoriene/algoritmene vil jeg nå gå detaljert inn på for å gi et helhetlig blikk av hva programmet faktisk gjør når det kjører. Både ansiktsdeteksjon og gjenkjenning baserer seg på noe som kalles Machine Learning. Det vil si at prosessene for gjenkjenning og deteksjon faktisk må lære og trene seg opp får å gi et korrekt resultat. Videre i denne delen av oppgaven skal jeg utdype teorien bak disse prosessene.

3.1 Digitalt bilde

Et digitalt bilde kan i dag være satt opp på mange forskjellige måter. Jeg kommer til å forklare om prinsippene til 8-bits gråtone bilder og 24-bits RGB (Red, Green, Blue) bilder da disse typene gir grunnleggende forklaringer til hva et digitalt bilde er. Hensikten er å kunne enklere forklare prosessene for deteksjon og gjenkjenning senere i oppgaven.

En definisjon på et digitalt bilde er: «En samling punkter/små bildeelementer med kjent posisjon og kjent gråtoneverdi/RGB-verdier. Dvs. at et digitalt bilde egentlig kun består av numeriske verdier for intensitet, innenfor enten gråtoner eller RGB, i et gitt system. 8- og 24-bits bilder forteller noe om dybden i bildene basert på størrelsen til verdien i et enkelt piksel. Dersom bildedybden er på 8-bit betyr det at hver eneste piksel kan ha en verdi mellom 0 og 255 der 0 er mørkt og 255 er lyst. I et gråtone (svart/hvitt) bilde vil verdien 0 vise fargen sort, og verdien 255 vil vise hvitt.

0	45	26	87
18	53	65	145
201	98	254	45
56	176	104	255

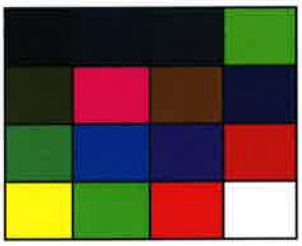


Figur 1 - Eksempel på et digitalt gråtone bilde

I et 24-bits bilde er det litt mer komplisert, men enkelt forklart består det av 3 8-bits bilder «oppå» hverandre. Et 24-bits RGB bilde vil bestå av «3 lag» der hvert lag består av et 8-bits

bilde, henholdsvis et for hver av primærfargene. Hver piksel vil være en kombinasjon av disse fargene som da vil vise fargen de 3 verdiene lager sammen.

Rød				Grønn				Blå			
0	23	34	0	0	35	45	255	0	46	45	0
78	245	129	25	77	21	79	1	14	123	13	78
84	0	47	199	145	0	10	34	75	255	111	0
255	12	255	255	255	255	74	0	0	72	0	255

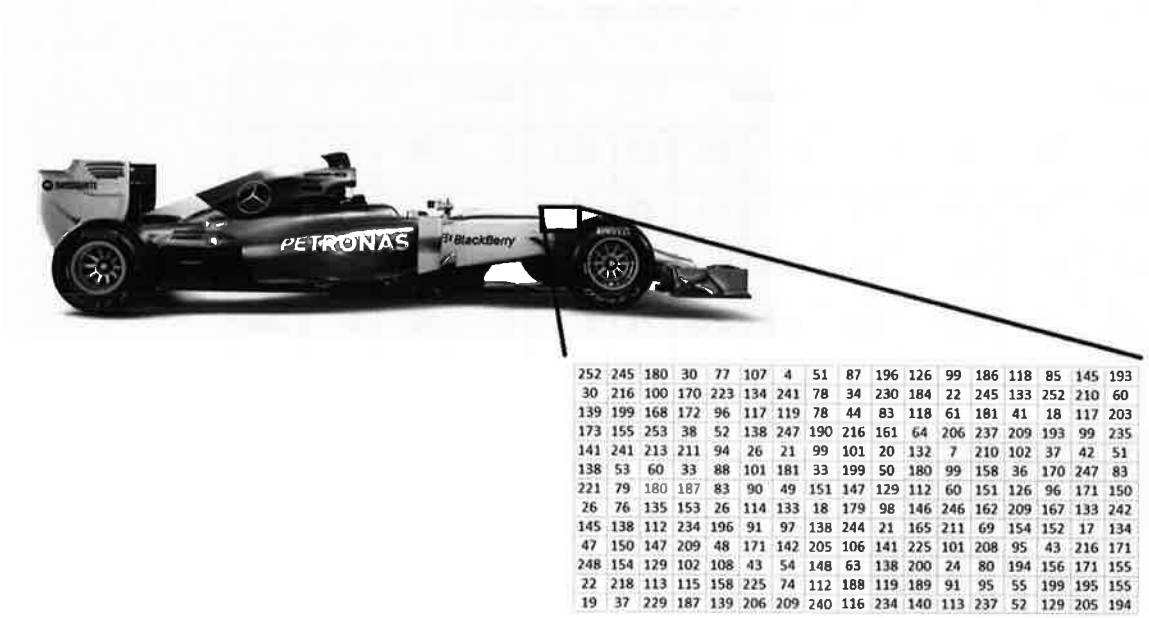


Figur 2 - Eksempel på et 3 Kanals RGB bilde

Posisjonen til hver enkelt piksel er ofte gitt som bildekoordinater i et koordinatsystem der origo er i øverste venstre hjørne. Med x-aksen vertikalt nedover og y-aksen horisontalt mot høyre. Når vi snakker om størrelser på digitale bilder blir disse ofte gitt som f.eks. 512*512 piksler, dvs. 512 horisontale linjer med 512 piksler i hver linje. Eksempelvis er størrelsen på figurene over 4*4 piksler, altså 4 horisontale linjer med 4 piksler i hver og pikselen øverst i venstre hjørne vil ha posisjonen/koordinaten (0,0) og pikselen i nederste høyre hjørne vil ha posisjonen/koordinaten (3,3).

Hvor mange piksler et bilde består av forteller noe om oppløsningen eller skarpheten til bildet. Et moderne mobil-kamera kan ta bilder med en oppløsning på opptil 12 megapiksler altså 12 millioner piksler. Et bildet med så stor oppløsning vil se relativt skarpt ut, men hvis man forstørrer bildet eller zoomer inn vil man til slutt se enkeltpikslene i bildet.

En datamaskin kan ikke faktisk se et bilde, men sitter kun på verdiene og posisjonene som utgjør bildet. Videre kan datamaskinen fortelle en skjerm hvilke piksler som skal ha hvilken farge for å så vise det frem. Dette betyr da at en datamaskin ikke vet hva som er avbildet, igjen så vet maskinen kun hvilke data som utgjør bildet. Figuren under viser et eksempel på hva vi ser og hva en datamaskin ser.



Figur 3 - Hva vi ser vs. Hva en datamaskin "ser"

3.2 Ansiktsdeteksjon

Ansiktsdeteksjon er prosessen som finner et ansikt i et bilde. For et menneske er dette relativt enkelt, men en datamaskin trenger presise instruksjoner og restriksjoner for å klare oppgaven. Prosessen bryr seg ikke om *hvem* som er avbildet, men *om* det finnes et ansikt i bildet. Dersom det er et ansikt i bildet vil prosessen også fortelle hvor ansiktet befinner seg. Det finnes flere forskjellige prosesser som gjør dette, og OpenCV inkluderer to forskjellige måter, Haar-Classifier og Local Binary Patterns.

3.2.1 Haar

Med Haar funksjonen mener jeg metoden for objekt deteksjon som ble utviklet av Paul Viola og Michael Jones i 2001. De utviklet en algoritme for å detektere objekter i bilder, og denne er senere blitt primært brukt til deteksjon av ansikter. Algoritmen var i 2001 unik i den forstand at den kunne detektere objekter eller ansikt i bilder i sann tid, til forskjell fra hva som ellers da fantes innenfor området.

Den grunnleggende ideen til metoden er at om man ser på de fleste ansikt vil område rundt øynene være mørkere enn pannen og kinnene og område rundt munnen vil være mørkere enn kinnene osv. Prosessen sjekker mer enn 20 slike faktorer for å avgjøre om det finnes et ansikt i bildet. Problemet er at denne prosessen må gjennomføres for alle posisjoner på bildet og for

alle mulig størrelser av ansikt innenfor bildet (Dette kan man justere på i programmet mitt). Resultatet blir flere tusen sjekker på et bilde.

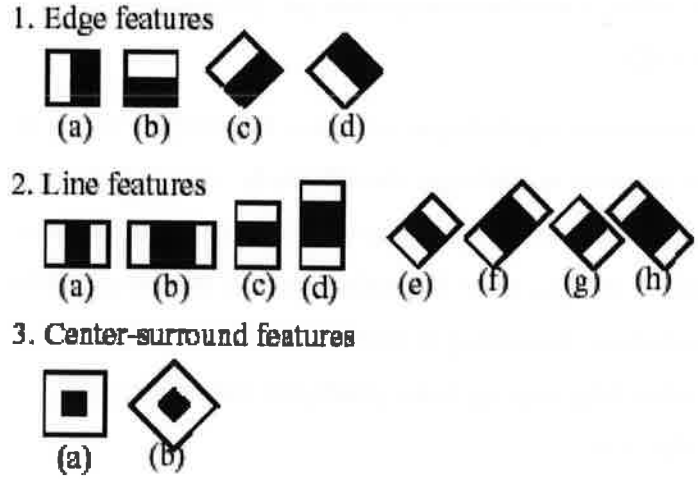
OpenCV kommer med en ferdig trent klassifiseringsfunksjon, som man kan benytte seg av på input bilder. Treningen går ut på å vise tusenvis av bilder og klassifisere de som positive eller negative. Positive bilder er av ansikt, og de negative eksemplene er av vilkårlige ting. Etter at klassifisereren er trent kan den bli brukt på et input bilde. Klassifisereren vil skanne et område av et bilde og gi en verdi ut lik 1 dersom det er sannsynlig at område den skanner viser et ansikt, og verdien 0 ellers. Deretter vil den følge opp og bruke ytterligere funksjoner for å stadfeste om den har funnet et ansikt eller ikke.

For å detektere et ansikt i ett gitt bilde brukes følgende metode med 4 hovedfaktorer.

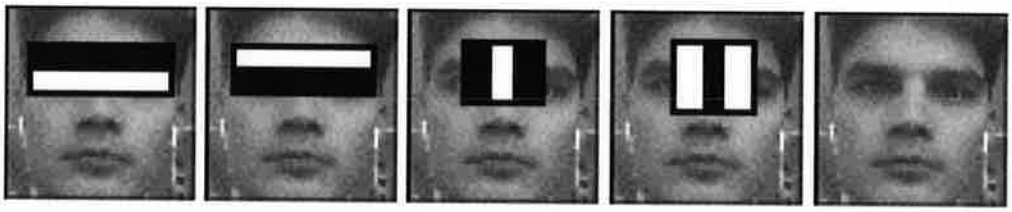
1. Haar funksjoner
2. Integral bilde
3. Adaboost
4. Cascading (Serie)

Haar funksjoner og cascading er viktig for å forstå hvordan selve deteksjonen fungerer, og vil være fokuset i denne teoridelen av oppgaven. Integral bilde og Adaboost er funksjoner som fremmer fart i prosesseringen og er noen av de viktigste faktorene for at man kan detektere ansikt i sann tid.

Bildet under viser Haar funksjonene. Haar funksjonene blir brukt på bildet, og med det menes det at hver firkant blir plassert over piksler der man trekker fra summen av pikslene under det hvite rektangelet fra summen av pikslene under det sorte rektangelet. I teorien kan en firkant dekke så lite som to piksler, men det er urealistisk at et ansikt er avbildet på to piksler, derfor øker størrelsen på Haar funksjonene får å dekket områder av et bilde. Som tidligere nevnt består et bilde av piksler med verdier for gråtonene i bildet. Resultatet vil enten gi en positiv verdi som vil si til "skanneren" at dette kan være et ansikt, eller en negativ verdi som forkaster det skannede området som et ansikt.



Figur 4 – Haar-funksjoner, kilde: OpenCV, www



Figur 5 - Visuelt eksempel av hvordan Haar funksjoner fungerer

Som tidligere nevnt baserer denne metoden seg på at ansikt har like og gjenkjennelige trekk. Bildet under viser hvordan Haar funksjonene kan detektere et ansikt.

Hvis vi ser for oss et bilde med 24*24 piksler vil det være omtrent 160 000+ kalkulasjoner å utføre. Dette er utrolig mange kalkulasjoner som krever både tid og prosessering. Derfor brukes det noe som heter integral bilder. Integral bilder forenkler kalkulasjonene av summene av piksler, uavhengig av hvor mange piksler som er dekket, til å utføre en matematisk funksjon med kun 4 piksler. Dette gjør prosessen mye raskere.

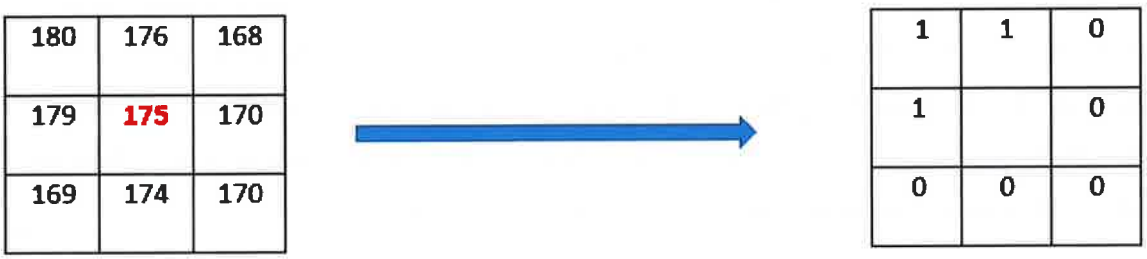
For å gjøre prosessen enklere, og mindre krevende ble det også lagd serier med funksjoner som måtte bestås for at de neste funksjonene skulle bli testet ut på bildet. Enkelt fortalt vil de første funksjonene enten gi ja, dette kan være et ansikt som peker til at man skal prøve ut neste funksjon og deretter neste og neste hvis svarene hele tiden er positive. Dersom man får et negativt svar går man videre til neste område på bilde.

Samtidig så er ofte store deler av et bilde irrelevant i forhold til om det er et ansikt (les: alt på et bilde som ikke er et ansikt). OG det er ikke alle delene av et ansikt som bruker de samme Haar funksjonene. For eksempel vil 1b kanskje passe inn i område ved øyne og panne, men 1c

vil ikke ha noe der å gjøre. For å velge de beste funksjonene av 160000 funksjoner som blir testet på et bilde bruker man Adaboost.

3.2.2 Local binary patterns (LBP)

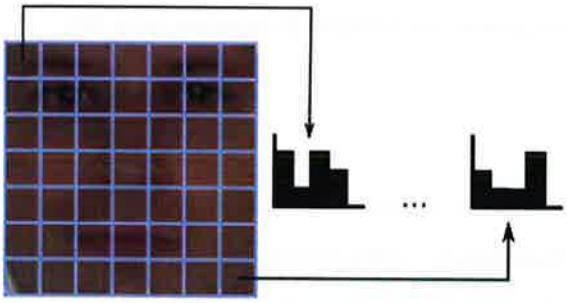
Local Binary pattern er muligens en enda mer effektiv måte å detektere ansikter på enn Haar-funksjonene og ble utviklet av Ahonen, Hadid og Pietikäinen i 2006 (Jo Chang-Yeaon, 2008). LBP operatoren merker pikslene i et bildene som binærverdier ved å se på 3x3 firkanter og sammenligne de ytterste elementene med det midterste.



Figur 6 - LBP

Fra figuren over henter vi ut et binærtall, og i dette tilfellet er det 10000011 = 131. Dette gjøres for hvert eneste område i bildet og man lager så et histogram som viser hvor mange ganger hver verdi dukker opp i bildet. På denne måten har man skapt et histogram til en viss tekstur som vises i bildet. For å finne et ansikt må, forenklet forklart, disse histogrammene som blir funnet ha likheter med histogram som allerede er klassifisert som et ansikt.

Et bilde av et ansikt kan man se på som en sammensetning av mikro-mønstre (micro-patterns) som også kan beskrives som LBP histogrammer. Men hvis man regner ut et LBP histogram for et helt bilde vil man miste informasjon om hva som befinner seg hvor. Derfor delte man ansikt opp (som vist i figuren under) for deretter å sammenligne detaljer i forskjellige områder med hverandre. På denne måten kan man også vektlegge hvilke deler av ansiktet man ser på som viktigst og som da prioriteres i deteksjons algoritmen. (På samme måte som Haar-metoden)



Figur 7 - Histogram fra et bilde

3.3 Ansiktsgjenkjenning

Ansiktsgjenkjenning er å identifisere et allerede detektert ansikt som enten et kjent ansikt, eller et ukjent ansikt. Og i mer avanserte tilfeller fortelle hvem sitt ansikt det er. Forskjellen mellom deteksjon og gjenkjenning er at deteksjon skal identifisere et objekt som et ansikt, og gjenkjenning skal avgjøre om ansiktet er kjent eller ukjent. Dette gjør at inngangsverdiene til ansiktsgjenkjenning baserer seg på utgangsverdiene til deteksjon.

Det finnes mange team verden over som har jobbet med ansiktsgjenkjenning i mange år. Og mange forskjellige fremgangsmåter er prøvd ut. De to dominerende metodene er:

1. Geometrisk – Ser på særtrekk i ansiktet

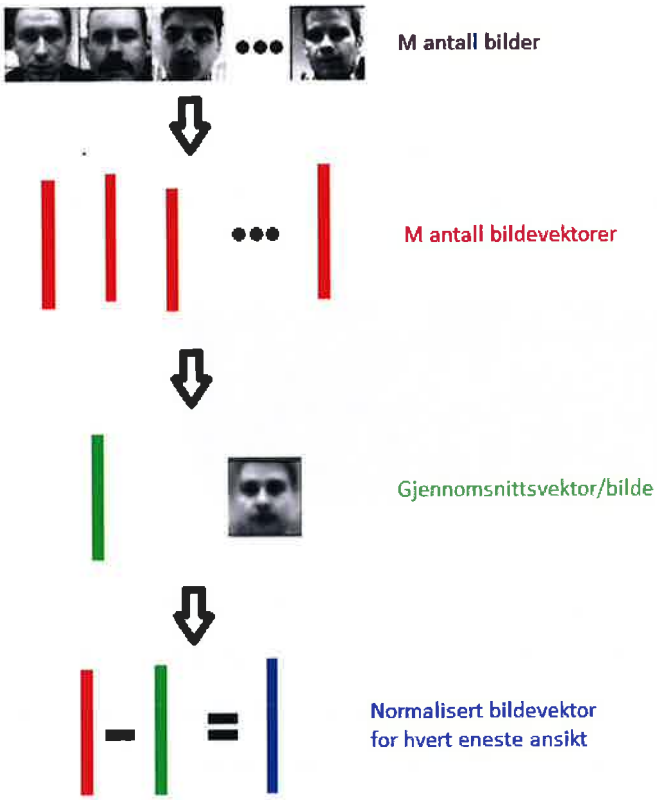
2. Fotometrisk – Er en statistisk tilnærming som omgjør et bilder til verdier for senere å sammenligne verdiene med maler for å eliminere avvik.

Det finnes også mange forskjellige gjenkjenningsalgortimer. I denne oppgaven bruker jeg to av de mest kjente, Eigenfaces og Fisherfaces.

3.3.1 Eigenfaces

Eigenfaces er en teknikk funnet opp av Kirby og Sirovich i 1988. Denne metoden bruker et sett med bilder av ansikt for å lage et treningssett, og et bilde av et ukjent ansikt, altså input bilde for å avgjøre om det er et kjent eller ukjent ansikt. For å kunne bruke denne metoden krever det at bildene er i samme størrelse.

Man starter med M antall bilder i databasen som man gjør om til M vektorer. En bildevektor består av en kolonne med like mange rader som det er piksler i bildet. Dvs. at et 50x50 bilde danner en vektor som er en kolonne bred med 2500 rader. Komponentene til vektorene er verdiene til pikslene i bildet. Deretter finner man gjennomsnittsvektoren og trekker den ifra bildevektoren. Da står man igjen med M normaliserte bildevektorer.



Figur 8 - Fra bildesett til normaliserte vektorer

Deretter finner man kovariansmatrisen K som består av en matrise av de normaliserte vektorene ganger den transponerte til den samme matrisen.

$$K = NN^T \quad Hvor\ N = [NV_1, NV_2, \dots, NV_M]$$

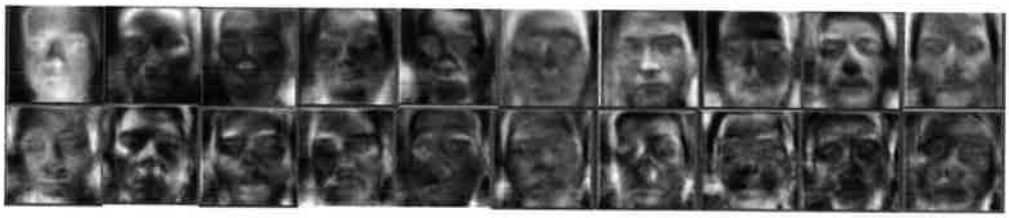
Kovariansmatrisen vil nå være av størrelsen 2500 x 2500 dersom bildene er av størrelsen 50x50. Det betyr at vi har 2500 egenvektorer, som alle har dimensjonen 2500x1. Dette krever utrolig mye prosessering, og kan i noen tilfeller bruke opp minnet til PC. Derfor bruker man heller en annen kovariansmatrise K, som reverserer formelen over.

$$K = N^T N \quad Hvor\ N = [NV_1, NV_2, \dots, NV_M]$$

Denne reduserte kovariansmatrisen vil være av størrelsen MxM. Denne formelen vil gi M egenvektorer som har dimensjon på Mx1. Deretter må det velges ut X antall egenvektorer der X<M men kravet er at disse vektorene er representative for alle bildene i treningssettet. Disse egenvektorene må så transformeres tilbake til deres opprinnelige størrelse som i dette tilfellet

var på 2500x1. Dette er for å ikke miste noe informasjon fra treningssettet. Dette gjøres ved å bruke lineær algebra.

Nå har man «lagd» X antall eigenvektorer som inneholder blandede funksjoner fra alle bildene i databasen. Figur 9 viser hvordan disse eigenvektorene ser ut når de blir gjort om til bilder.



Figur 9 – De 20 første eigenansiktene fra min database

Hvert bilde i databasen består av en vektet sum av eigenansiktene + gjennomsnittsnansiktet. Det vil si at det første bilde i Figur 8 kan gjenskapes om man f.eks. tar [10% av Eigenansikt 1 + 14% av Eigenansikt 2 + … + Z prosent av Eigenansikt X] + Gjennomsnittsansiktet. Dette kaller vi en egenverdiene, eller den vektede vektoren. Det kalkuleres eigenverdier for hvert eneste bilde i databasen.

For å kjenne igjen et inputansikt må dette ansiktet først omgjøres til en ansiktsvektor og deretter normaliseres. Inputbildet normaliseres på ved å trekke fra gjennomsnittsansiktet som ble brukt tidligere. Deretter skal man representere det nye normaliserte ansiktet som en kombinasjon av eigenansiktene, altså på samme måte som for databasen. Dette gjør at man får en eigenverdi til det nye ansiktet. Så sammenligner man den nye eigenverdien med de som allerede er lagd for å finne hvilken som ligner mest.

3.3.2 Fisherfaces

Fisherfaces (også kjent som Linear Discriminant Analysis (LDA)) er en annen metode for å gjenkjenne ansikt. Denne metoden ligner på Eigenfaces metoden, men er forskjellig da den anerkjenner at bilder av samme ansikt kan være ulike.

Hver person er hver sin klasse C og databasen inneholder N bilder i hver klasse. Fisherface metoden bruker både forskjellen mellom klassene og forskjellen i klassene som variabler for videre kalkulering. Det lages matriser fra forskjellene på nesten samme måte som for Eigenfaces metoden.

Kovarians matrisen KM = (gjennomsnittsbilde i klassen – gjennomsnittsbilde for alle klassene) x (gjennomsnitssbilde i klassen – gjenomsnittsbilde for alle klassene)^T.

Kovarians matrisen KI = (bilde n i klassen – gjennomsnittsbilde i klassen) x (bilde n i klassen – gjennomsnittsbilde i klassen)^T.

Deretter maksimerer man forskjellen mellom klassene, samtidig som man minimerer forskjellene i klassene. Og det resulterer i en eigenverdi til hver klasse som da kan sammenlignes med eigenverdien til et input bilde.

Figur 10 - Fisherfaces, fra min database

4. Programvare

4.1 Microsoft Visual Studio Express 2012

Microsoft Visual Studio (VS) er et integrert utviklingsmiljø lagd for å utvikle applikasjoner for Windows. VS støtter programmeringsspråkene Visual Basic, Visual C++, Visual C# og Visual J#. Å utvikle og arbeide i VS var et enkelt valg, fordi det var det utviklingsmiljøet jeg hadde programmert i fra tidligere kurs på SKSK, og fordi det fungerer nesten sømløst sammen med OpenCV bibliotekene.

4.2 Open CV

OpenCV står for Open Source Computer Vision og er et open source bibliotek for visuell databehandling. Biblioteket inneholder over 500 funksjoner assosiert med bildebehandling og inneholder et generelt Machine Learning bibliotek. OpenCV ble valgt som bibliotek som fordi det er open-source og dermed gratis for akademisk bruk og siden det er mye brukt finnes det mye dokumentasjon på bibliotekene.

OpenCV blir stadig oppdatert og kommer ut med nye versjoner. Den nyeste versjonen per i dag er OpenCV 3.0 og den ble sluppet i juni 2015. Jeg bruker OpenCV 2.4.11 fordi det var den mest nedlastede versjonen fra OpenCV da jeg startet på arbeidet med denne oppgaven i slutten av Mai 2015.

4.3 Valg av programvare

Da jeg startet å undersøke metoder å løse oppgaven på hadde jeg noen krav til programvare. Jeg ønsket et utviklingsmiljø som var kjent, for at det ikke skulle være en ekstra påkjenning å lære seg å programmere i et nytt miljø. Jeg ønsket også å bruke biblioteker som hadde god dokumentasjon, og som var mye brukt i den hensikt å kunne lære av andre sine utfordringer (ved å lese diverse forum ol.). C++ var kjent fra tidligere av, og OpenCV viste seg å være det mest vanlige, derfor falt valget på de.

5. Utvikling

I denne delen av oppgaven vil jeg gå igjennom oppbygningen av testprogrammet og selve programmeringen. Jeg skal forklare hvordan jeg bruker funksjoner og hvordan ansiktsgjenkjenningen faktisk fungerer ved å trekke inn teorien bak funksjonene jeg bruker i programmet. Samtidig vil jeg fortelle om andre muligheter jeg har hatt under utviklingen og hvorfor jeg har valgt noen metoder fremfor andre.

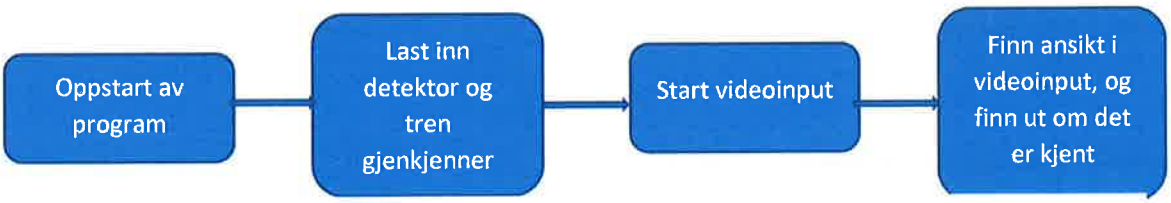
For å gjenkjenne et ansikt må man altså ha noen tidligere bilder av ansiktet for å sammenligne med. For å løse dette på en måte som var lærerik for meg lagde jeg et program som detekterte et ansikt og som lagret de på rett sted, med rett navn, størrelse, fargetone osv. Dette er fordi selve gjenkjenningsprogrammet trenger informasjonen gitt på en spesifikk måte.

I denne delen av oppgaven vil jeg bruke programmeringsterminologi og forklare ferdiglagde funksjoner fra OpenCV bibliotekene. En beskrivelse av hver enkelt funksjon eller klasse vil bli vist på følgende måte:

Tabell 1 - Eksempeltabell

Klasse / funksjon	Beskrivelse
Hva funksjonen klassen heter	Hva funksjonen/klassen gjør, og hvor den ligger (generelt)

5.1 Programstruktur



Figur 11 - Programstruktur

5.2 Grunnleggende prosesser i programmet

1. Ansiktsdeteksjon
2. Prosessering/forberedelser av ansiktsbilder
3. Innsamling av ansikt og trening
4. Gjenkjenning

5.2.1 Ansiktsdeteksjon

Ansiktsdeteksjon er prosessen som finner et ansikt innenfor et område av et bilde. Denne prosessen bryr seg ikke om hvem sitt ansikt som er avbildet, men om det finnes et ansikt der. For å finne et ansikt måtte jeg først kunne hente bilder fra webkamera som videre skulle brukes.

Tabell 2 - Funksjoner i programmet 1

Klasse / funksjon	Beskrivelse
Mat	Mat er en forkortelse for Matrise eller Matrix. Mat er altså en type variabel. En Mat variabel kan være et bilde. Defineres på samme måte som de andre typene f.eks. Mat x; sier at variabelen x er et bilde.
VideoCapture	OpenCV sin måte å få tilgang til webkameraet på en PC. VideoCapture A(0); Man kan hente ut et bilde fra videostrømmen på samme måte som man henter inputverdier fra tastaturet. For eksempel A>>Bilde; vil legge det bildet du ser inn i variabelen Bilde og kan deretter prosesseres.

I programmet har jeg lagt inn en test for å sjekke at man faktisk får åpnet kameraet. Hele prosessen ser slik ut:

```
VideoCapture A(0);

if(!A.isOpened()){
    return (-1);
}
```

Figur 12 – Kode: Åpne webkamera

Deretter sender jeg bildet fra kameraet til videre prosessering ved å legge det inn som en variabel av typen Mat. Nå har jeg et bilde som jeg ønsker å finne ut om viser et ansikt eller ikke, og da må jeg bruke en av detektorene. OpenCV kommer med ferdig trente XML detektorer som kun trengs å lastes inn for å fungere i et program.

Tabell 3 - Funksjoner i programmet 2

Klasse / funksjon	Beskrivelse
CascadeClassifier	CascadeClassifier er en klasse som laster inn XML detektorer.
CascadeClassifier::load(filnavn)	Laster inn en klassifiserer fra en fil.

CascadeClassifier::detectMultiscale(bilde, rektangel, scaleFaktor, minimum neighbors, minimum size, maximumesize)	Detekterer objekt i et bilde, basert på hvilken fil som er lastet inn i klassen. Returnerer et rektangel som vil være rundt det detekterte objektet. Rect er en egen klasse. Første parameter er hvilket bilde man ønsker å sjekke, deretter må man vise til et tidligere definert rektangel (egen funksjon i openCV). Minimum neighbors er en parameter som forteller funksjonen hvor sikker den skal være på at den har faktisk har funnet et ansikt. scaleFactor er en parameter som forteller om detektoren skal lete etter ansikt i forskjellige størrelser.

Dersom jeg skal bruke Haar funksjonen eller LBP funksjonen må jeg laste inn hhv. riktig fil som ligger i mappestrukturen til OpenCV. Deretter bruker jeg detectMultiScale funksjonen til for å detektere et ansikt. Denne funksjonen returnerer posisjonen til et rektangel som kan lage et omriss av ansikt, altså visualisere hvor et ansikt befinner seg i et bilde ved å legge et rektangel over det.

Tabell 4 - Funksjoner i programmet 3

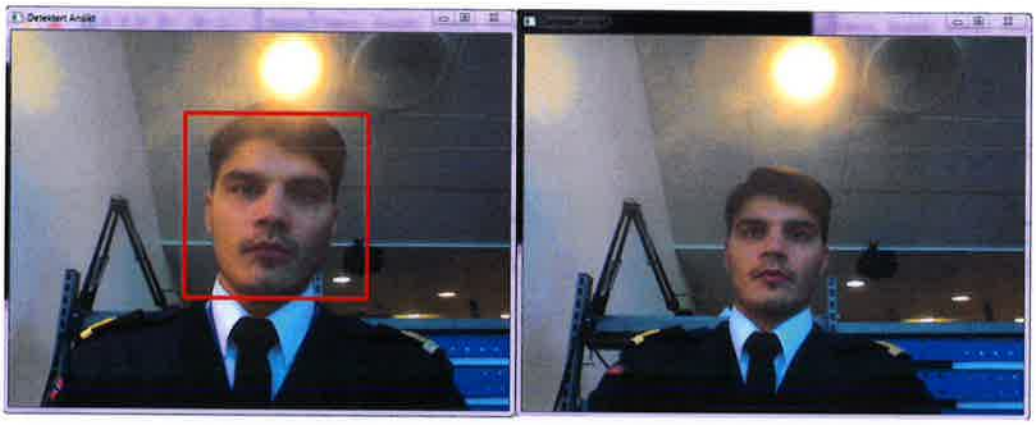
Klasse / funksjon	Beskrivelse
Rectangle()	Denne funksjonen tegner et rektangel oppå et bilde. Funksjonen har input verdiene: Orginalt bilde, Rektangel, Farge, tykkelse på tegnet strek)
Imshow()	Denne funksjonen oppretter et vindu og har to parametere: Vindusnavn, og inputbilde. Vindusnavn velger hva det skal stå på bildet, og inputbildet er hvilket bilde du ønsker at skal vises i vinduet.

Figur 13 er et kodeeksempel på hvordan man kan finne et ansikt i et bilde fra webkamera:

```
string baneTilHaar = "C:/haarcascade_frontalface_alt.xml";
string baneTilLbp = "C:/lbpcascade_frontalface.xml";

CascadeClassifier ansiktDeteksjon;

int main(){

    vector<Rect> FunnetAnsikt;          //En vektor som holder på posisjoner til detekterete ansikt
    Mat orginal;                        //Variablen til bilde vi ønsker å detektere et ansikt fra

    VideoCapture A(0);                  //Initialiserer kamera

    if(!A.isOpened())                   //Sjekke om kameraet ble åpnet
        return (-1);

    ansiktDeteksjon.load(baneTilHaar);  //Laste inn detektor (I dette tilfellet bruker jeg Haar metoden)

    A>>orginal;                         //Hente inn bilde fra kamera for å arbeide med videre

    //Sjekker om det finnes et ansikt i bildet, og kaller fnnet for "FunnetAnsikt". Sier at ansiktet skal være mins 100*100 piksler)
    ansiktDeteksjon.detectMultiScale(orginal, FunnetAnsikt, 1.1, 2, 0 | CASCADE_SCALE_IMAGE, Size(100, 100));

    //Legger et rektangel rundt det funnede ansiktet
    rectangle(orginal, FunnetAnsikt, CV_RGB(255, 0, 0), 4, 8, 0);

    //Viser bildet
    imshow("Detektert Ansikt", orginal);
}
```

Figur 13 - Kode: Deteksjon

Resultatet når jeg selv sitter i bildet blir slik:



Figur 14 - Resultat av kode

I kodeeksempelet kan man se at jeg har lagt inn at minimumsstørrelse på et ansikt skal være 100*100Piksler. Når jeg da trekker hodet mitt litt lengre vekk fra kameraet vil ansiktet mitt ikke lenger oppdages. Dette skjer fordi jeg forteller at Haar metoden ikke skal lete etter ansikt som er mindre enn 100*100Piksler. Altså kan hele algoritmen hoppe over ekstremt mange kalkuleringer. Dersom jeg hadde ønsket at den skulle detektere ansikt av alle størrelser for et bilde kunne jeg f.eks. bedt om at minimumsstørrelsen på et ansikt skal være 1 x 1 piksler (teoretisk, da 1 piksel kun kan vise en fargetone, og et ansikt er sammensatt av flere piksler).

Dette krever selvfølgelig mer av algoritmen da det kan bli flere hundre tusener kalkulasjoner per bilde, og dette vil igjen gå på bekostning av tid.

For å få dette programmet til å kjøre i sann tid, og ikke kun analysere et bilde, men analysere en live video trenger jeg bare å gjøre en kjapp endring. Dersom jeg setter hele prosessen inn i en while-løkke vil den kjøre om og om igjen, til jeg stopper den. Altså, når en loop er ferdig vil den hente inn neste bilde fra kameraet og sjekke det for et ansikt osv. Hvis man da ber algoritmen om å detektere selv de minste ansikt vil programmet oppleves som veldig hakkete men hvis man setter minimumsstørrelsen på et ansikt til en grei størrelse vil programmet detektere et ansikt med en relativt høy rate.

5.2.2 Prosessering/forberedelser av ansiktsbilder

Som beskrevet i teoridelen av oppgaven finnes det noen krav for å lage en robust ansikts gjenkjenner. Noen av kravene har med utformingen på bildene et nytt ansikt skal sammenlignes med, altså hvordan bildene i databasen ser ut. Ansiktsgjenkjenning kan påvirkes av flere faktorer som f.eks. lysforhold, ansiktsuttrykk, om ansiktet er rettet rett mot kamera eller litt vridd osv. Gjenkjenningsalgortimen kan tro at det er mer likhet mellom to bilder tatt med like lysforhold, men av forskjellige personer enn to like ansikt tatt i ulike situasjoner. Hensikten med å prosessere bildene er da å normalisere hvert eneste ansikt slik at vi skaper minst mulig rom for misforståelser.

Først og fremst bryr gjenkjenningsalgoritmen seg kun om et ansikt, ikke et helt bilde med et ansikt i et område. Derfor begynner jeg med å «klippe ut» et utsnitt av alt som er innenfor rektangelet. Posisjon og størrelse til rektangelet som omrisser ansiktet ligger i FunnetAnsikt. Derifra kan jeg hente høyden og bredden på rektangelet, og også posisjonen til det øverste venstre hjørnet. Ettersom hvilket område et ansikt befinner seg på bildet, og avstanden ansiktet faktisk er fra kamera vil størrelsen på rektangelet som omrisser det detekterte ansiktet variere. Derfor vil jeg også endre størrelsen på hvert av bildene som skal lagres av de forskjellige ansiktene, for å gjøre disse like. I deteksjonspunktet nevnte jeg at man kan velge en minimumsstørrelse på et ansikt som skal detekteres, noe som for prosesseringens del betyr at jeg allerede har en størrelse å forholde meg til. Dette er en fornuftig ting å forholde seg til fordi at hvor mye man ønsker å endre størrelsen på bilde vil påvirke utfallet. Hvis jeg ønsker å forstørre bilde er det en stor sjanse for at det blir veldig uskarpt, og hvis jeg ønsker å minimere bilde er det en sjanse for at det blir for «pikslete». Samtidig så ønsker jeg at størrelsen på bildene i databasen (bildene jeg nå prosesserer) ikke skal være for store, da det resulterer i mange flere kalkulasjoner for gjenkjenningsalgoritmen.

Tabell 5 - Funksjoner i programmet 4

Klasse / funksjon	Beskrivelse
Rect	Klassen Rect består av 4 int-variabler. Bredde, høyde, og x,y-kooridnater til venstre øverste del av rektangelet.
Resize()	Funksjon som endrer størrelsen på et bilde. Funksjonen returnerer ett nytt bilde av typen Mat med størrelse man selv velger. Denne funksjonen bruker interpolasjon for å komme frem til verdiene i hver piksel i det nye bildet. Dvs. at funksjonen bruker en matematisk prosess for å tilnærme verdien av en piksel i et punkt ved hjelp av kjente verdier for piksler i nærliggende punkt.
cvtColor()	Denne funksjonen endrer fargetonene i et bilde. Brukes til å lage gråtonebilder fra kameraet.

Kodeeksempelet under viser hvordan jeg videre kun bruker ett bilde av ansiktet, før jeg endrer størrelse på det og til slutt endrer det til et svarthvitt-bilde.

```
//Variablene
Mat KunAnsikt, KunAnsikt_nyStr, KunAnsikt_graa;

//Lager et nytt bilde, som kun er et utsnitt av det detekterte ansiktet
KunAnsikt = orginal(FunnetAnsikt);

//Endrer størrelsen til å være 100*100piksler
resize(KunAnsikt, KunAnsikt_nyStr, Size(100, 100), 1.0, INTER_CUBIC);

//Gjør bildet om til et gråtone bilde
cvtColor(KunAnsikt_nyStr, KunAnsikt_graa, CV_BGR2GRAY);
```

Figur 15 - Kode: prosessering

Under er noen eksempler på hvordan bildene vil se ut i forskjellige størrelse:

Figur 16 - 25x25 50x50 100x100 400x400

Som man ser på bildene over varierer kvaliteten veldig når man forstørrer eller minimerer bildene. For oss ser bildet som er på 100x100 piksler best ut, og det er fordi det er det som er nærmest originalen. Bildene som er forminsket blir kornete og bildet som er forstørret er mer

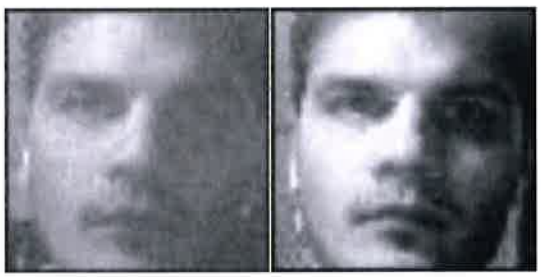
uskarpt. Dersom det var et menneske som skulle gjenkjenne ansikt på et bilde er det klart at man vil ha det skarpeste bildet, men en datamaskin bryr seg ikke nødvendigvis om det. En datamaskin ønsker bilder som er enkle å sammenligne med, og som nevnt tidligere har størrelsen på bildene et ansikt skal sammenlignes med stor betydning for hvor raskt programmet skal være basert på kalkulasjoner som skal utføres. Bildet som er forminsket til 50x50 piksler viser ansiktets hovedtrekk, og er foreløpig en god nok størrelse å bruke.

Et meget vanlig trekk på bilder av ansikt er at den ene siden av ansiktet er lysere enn den andre. Dette kommer selvfølgelig av lysforholdene der bildet blir tatt. For en gjenkjenningsalgoritme har dette store utslag, da høyre og venstre side av et ansikt kan se ut som to ulike personer nettopp på grunn av lyset. For å gjøre noe med dette kan jeg standardisere lysstyrken og kontrasten på hver side av ansiktet. For å gjøre dette bruker jeg utjevning ved hjelp av histogram.

Tabell 6 - Funksjoner i programmet 5

Klasse / funksjon	Beskrivelse
equalizeHist()	Utjevner histogrammet til et gråtone bilde. Funksjonen har parameterne inputbilde, og ut-bilde. Algoritmen normaliserer lysstyrken og øker kontrasten i bildet.

Under er bildet fra forrige eksempel, kjørt gjennom utjevningsfunksjonen i OpenCV. Bildet har nå blitt skarpere, og det generelle lysforholdet er bedre i det nye bildet.



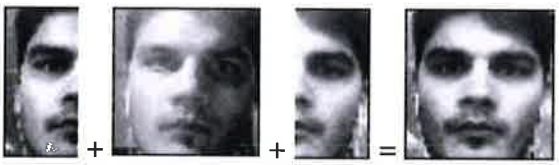
Figur 17 - Gammelt og nytt bilde

Men, et stort problem er fortsatt tilstede. Venstre del av bildet er mye lysere enn høyre. Derfor må høyre og venstre side standardiseres i forhold til hverandre, ikke hele bildet i ett. For å gjøre det deler jeg bildet i 3, hhv. Høyre, venstre og hele bildet. Deretter utjevner jeg hver del for seg og setter sammen bildet fra de 3 delene som er redigert hver for seg. På grunn av detektoren har jeg allerede ansiktet sentrert i bildet, og jeg vet at bildet er på 50x50piksler. Derfor vet jeg også at midten av bildet er langs den 25. kolonnen av piksler. Jeg har også

tidligere vist at vi kan definere et bilde innenfor et bilde ved å hente ut et utsnitt. Koden under viser hvordan jeg deler bilde i 3, for så å utjevne hver side, hver for seg og deretter sette det sammen.

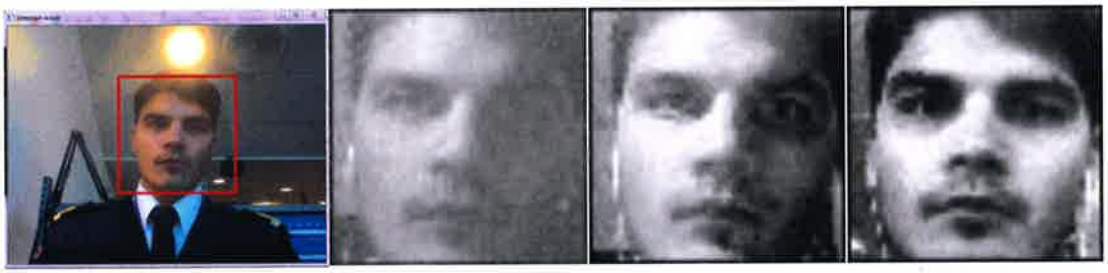
```
//Utjevner hele bildet
equalizeHist(graa, utjevnet);
int midtenX = b/2;

Mat VenstreSide = graa(Rect(0,0,midtenX,h));
Mat HoyreSide = graa(Rect(midtenX, 0, b-midtenX, h));

equalizeHist(VenstreSide, VenstreSide);
equalizeHist(HoyreSide, HoyreSide);

for(int y=0;y<h;y++){
    for(int x=0;x<b;x++){
        int A;
        if(x<b/4){
            A = VenstreSide.at<uchar>(y,x);
        }
        else if(x<b*2/4){
            int VA = VenstreSide.at<uchar>(y,x);
            int HeleA = utjevnet.at<uchar>(y,x);
            float f = (x-b*1/4)/(float)(b/4);
            A = cvRound((1.0f-f)*VA+(f)*HeleA);
        }
        else if(x<b*3/4){
            int HA = HoyreSide.at<uchar>(y,x-midtenX);
            int HeleA = utjevnet.at<uchar>(y,x);
            float f = (x-b*2/4)/(float)(b/4);
            A = cvRound((1.0f-f)*HeleA+(f)*HA);
        }
        else{
            A = HoyreSide.at<uchar>(y,x-midtenX);
        }
        utjevnet.at<uchar>(y,x) = A;
    }
}
```

Figur 18 - Kode for å utjevne lysforhold i et bilde, koden er hentet fra et eksempel. Kilde: OpenCV



Figur 19 - Visualisering av koden over

Bildet under viser nå hvordan det opprinnelige bildet har blitt prosessert og resultatet er et distinkt ansikt som er klar for å lagres i en database for deretter å brukes til å gjenkjenne det samme ansiktet igjen.



Figur 20 - Prosessen frem til nå

5.2.3 Innsamling av ansikt og trening

For å kjenne igjen ansikt trenger man en database å sammenligne ansiktet med. Etter det må gjenkjenningsalgoritmen trenes, og deretter er den klar for et inputbilde. Å samle inn bilder og lagre de slik at de enkelt kan brukes kan sannsynligvis gjøres på mange måter. Under arbeidet mitt har jeg prøvd meg frem med to forskjellige metoder, enten å lagre bildene fast i en mappe på datamaskinen og lage en tekst-fil med stiene til hvert enkelt bilde og navn. Eller å lagre bildene i en matrise med en korresponderende matrise med navn. Valget mitt falt på å bruke filsystemet i Windows 7 som databasen, da det var enklere å jobbe med under utviklingen fordi jeg ikke mistet informasjon hver gang jeg gjorde endringer eller kompilerte testprogrammet mitt.

Når ansiktet fra forrige steg er ferdig prosessert lagres det i mappen Ansikt underfilnavnet som består av etternavn, hvilket bilde det er av det ansiktet og hvilken filtype det er. F.eks. vil et bilde av meg selv lagres som Narum01.jpg, i mappen c:\ansikt\. Dersom det er snakk om mange bilder per ansikt man ønsker å legge inn i databasen er det kanskje bedre å lage en mappe per person for å ha en overordnet struktur, men det har egentlig lite å si. Grunnen til at bildene lagres slik er fordi det skal være enkelt å samle alle bildene for å trene gjenkjenningsalgoritmen, enten man bruker Eigenfaces eller Fisherface metoden.

Samtidig så skriver jeg plasseringen på bildet og et nummer for hvilken person det er inn i en csv (comma seperated value) fil som ser noe sånn ut som bildet under. Denne filen er det gjenkjenningsklassen som leser, og henter ut informasjon om hvor bildene ligger, samt hvem som er på bildet.

```
c:\ansikt\andersen01;0
c:\ansikt\andersen02;0
...
c:\ansikt\andersen0x;0
c:\ansikt\pedersen01;1
c:\ansikt\pedersen02;1
...
c:\ansikt\pedersen0x;1
...
c:\ansikt\maartens01;X
c:\ansikt\maartens02;X
...
```

Figur 21 - eksempel på csv fil

Tabell 7 - Funksjoner i programmet 6

Klasse / funksjon	Beskrivelse
Ptr<T>	OpenCV sin egne peker-funksjon. Med denne funksjonen peker man mot et objekt av typen T. Forskjellen fra «den vanlige» pekeren er at objektet vil bli ryddet opp når alle forekomster av pekeren er borte. Altså, en peker som er tryggere å bruke.
FaceRecognizer	Klassen FaceRecognizer sitter på alle algoritmene som hører til ansiktsgjenkjenning. Klassen har funksjoner som prediksjon, trening, lasting/lagring og mer.
createXXXXFaceRecognizer()	En funksjon der XXXX er lik gjenkjenningsalgoritmen man ønsker å bruke. For denne oppgaven står valget mellom Eigenfaces eller Fisherfaces.
FaceRecognizer::Train	En funksjon som trener en gjenkjenner. Funksjonen tar inn bilder og de tilhørende merkinger (merking 0 på ansikt 0 osv.) og «husker» treningen så lenge programmet kjører. Man kan ikke trene gjenkjenneren mer enn en gang per prosess man kjører, og gjenkjenneren har ingen hukommelse.

Etter at man har lagd seg en database kan man begynne å trene gjenkjenningssystemet basert på de innsamlede ansiktene. Ordet trene kommer av at det som faktisk skjer er at datamaskinen lærer seg opp på hvilket ansikt som tilhører hvem ved hjelp av en maskin

lærings algoritme. Kodeeksempelet under viser hvordan jeg leser informasjon fra en csv-fil og med det innhenter informasjon, altså bilder og respektive merkinger, til som vektorer som skal senere brukes til å trene gjenkjenneren.

```
static void les_csv(const string& filnavn, vector<Mat>& bilder, vector<int>& merkinger, char skille = ';'){
    ifstream fil(filnavn.c_str(), ifstream::in);
    if(!fil){
        string error_message = "Feil! Sjekk filnavnet";
        CV_Error(CV_StsBadArg, error_message);
    }

    string linje, hvor, klassetype;
    while(getline(fil, linje)) {
        stringstream linjenummer(linje);
        getline(linjenummer, hvor, skille);
        getline(linjenummer, klassetype);
        if(!hvor.empty() && !klassetype.empty()){
            bilder.push_back(imread(hvor, 0));
            merkinger.push_back(atoi(klassetype.c_str()));
        }
    }
}
```

Figur 22 - Kode: for å lese csv-fil kilde: OpenCV, www

Deretter trener jeg gjenkjenneren, slik at den vet hvilke ansikt et input ansikt skal sammenlignes med ved å bruke koden fra figur 22.

```
string csvfil = "csv.ext";

vector<Mat> bilder;
vector<int> merkinger;

try {
    les_csv(csvfil, bilder, merkinger);
} catch (Exception& e) {
    cerr<<"Feil med innlasting av filen"<<endl;
    exit(1);
}

Ptr<FaceRecognizer> modell = createEigenFaceRecognizer();
modell->train(bilder, merkinger);
```

Figur 23 - Kode: Trene gjenkjenneren

Nå vil programmet være klart til å kjenne igjen et ansikt dersom det finnes i databasen fra før av. For å få forståelse for hva som faktisk har skjedd under treningen så er det mulig å hente ut data strukturene som ble generert i treningen. For Eigenfaces og Fisherfaces er dette samme type data, så det gjør det enkelt å sammenligne de to metodene og samtidig forstå de på en bedre måte. Hva som faktisk skjer i denne prosessen er beskrevet i teoridelen av oppgaven under 4.3.1 Eigenfaces og 4.3.2 Fisherfaces.

5.2.4 Gjenkjenning

Nå er gjenkjenneren trent, og klar for å kjenne igjen et ansikt. Selve programmeringen er veldig enkel da klassen FaceRecognizer i OpenCV kommer med en funksjon som predikerer hvem som er avbildet, basert på treningen. For å presisere så trenger funksjonen et inputbilde, som er i samme størrelse som bildene i databasen. Altså kan man kjøre samme prosess som tidligere for å lage et utsnitt fra ett live bilde av et ansikt, og bruke det som inputbilde. Koden under viser hvordan jeg predikerer hvem som er avbildet. Datamaskinen tenker ikke på ansiktene som navn, men som tall beskrevet i csv-filen.

```
//Predikerer hvem som er avbildet
int prediksjon = modell->predict(NyttAnsikt);
```

Figur 24 - Kode: prediksjon

Nå sitter jeg med ett tall mellom 0 og X, som avgjør hvem som er avbildet. I vedlegg A så finner man hele koden for testprogrammet, og der ser man hvordan jeg viser frem bildet i sann tid, sammen med et navn på det detekterte og identifiserte ansiktet. Problemet nå er at algoritmen vil predikere at et ansikt er kjent uavhengig av om det faktisk er kjent eller ikke. Dersom et ukjent ansikt blir avbildet, vil algoritmen predikere hvem som er likest. Dette gjør at programmet foreløpig er vanskelig å stole på.

For å avgjøre om et ansikt er kjent, og i så fall hvem det er med sikkerhet trenger vi en konfidens variabel som kan gjøre det lettere å stole på programmet. FaceRecognizer klassen kan faktisk returnere en slik variabel, men OpenCV dokumentasjonen sier selv at dette ikke er en pålitelig variabel. Derfor gjør jeg det på den tungvinte måten, men som gir et bedre resultat.

Metoden jeg bruker for å avgjøre konfidensvariabelen baserer seg på å rekonstruerer ansiktet ved hjelp av eigenvektorene og eigenverdiene, og deretter sammenligne det rekonstruerte ansiktet med input bildet. Hvis jeg bruker eigenvektorene fra inputbildet og kombinerer de med eigenverdiene til et av eigenansiktene bør dette konstruere et bilde som ligner på input bildet. Deretter sammenligner jeg det rekonstruerte bildet med input bildet, og får i retur en konfidensvariabel, eller en variabel som forteller hvor ulike bildene er. Denne kan jeg bruke til å sette min egen terskel for hvor likt et ansikt skal være. Dersom ansiktet som skal gjenkjennes har mange bilder i databasen bør dette fungere godt siden det er mye som kan sammenlignes. Men, dersom ansiktet ikke finnes fra før av, eller er ulikt ansiktene i databasen pga. andre faktorer som lys, tilt osv. vil det rekonstruerte ansiktet se veldig annerledes ut enn

inputbildet, og dermed signalisere at det kanskje ikke er et kjent ansikt. Bildene under viser hvordan et rekonstruert ansikt ser ut dersom det er av en person som ligger inne i database, og dersom det er av et ukjent ansikt.

Figur 25 - Inputansikt vs. rekonstruert ansikt når ansiktet er kjent og ukjent

Måten jeg viser om ansiktet er kjent eller ukjent på kan selvfølgelig gjøres på mange måter. Jeg har valgt å vise resultatet ved å vise navnet på personen rundt ansiktet i sann tid, eventuelt teksten «ukjent person» dersom programmet ikke klarer å identifisere hvem som er avbildet. Bildet under viser det ferdige resultatet.

Figur 26 - Bilde av programmet

6. Testing og feilsøking

I denne delen av oppgaven vil jeg gå igjennom testing av programmet. Testingen består av å sjekke hvordan de forskjellige funksjonene i programmet takler forskjellige utfordringer i den hensikt å stå igjen med en robust ansikts gjenkjenner. Under arbeidet har jeg hele tiden testet programmet og gjort endringer for at programmet skal gjøre det jeg ønsker. Totalt har jeg vel vært gjennom minst 15 versjoner, alle med små forbedringer. Testene viser hvordan jeg har kommet frem til de viktigste forbedringene.

6.1 Ansiktsdeteksjon

Formålet med å teste ansiktsdeteksjon var å finne ut hvilken algoritme som fungerte best samtidig for å se om det var store forskjeller mellom de, til mitt bruk. Samtidig ønsket jeg å finne ut hvilke faktorer som påvirket detektoren og dens ytelse.

6.1.1 Test av ansiktsdetektor med Haar metoden

Testen går ut på: Å bruke Haar-metoden for å detektere ansikt i sann tid fra webkameraet i den hensikt å finne ut om hvilke størrelser som er lure å bruke, med tanke på et effektivt program, hvilke utfordringer detektoren har med tanke på hele ansikt, om de er tildekket, bruker de briller/solbriller og hvor mye tilt/vridning kan et ansikt ha.

Ønsket resultat: Etter testingen ønsker jeg å sitte igjen med svar på parameterne jeg har satt opp over, for å sammenligne det med den andre deteksjonsmetoden jeg kan bruke. Dette er parametere jeg ønsker å bruke videre i oppgaven.

Gjennomføring: For å teste dette bruker jeg programmet mitt som jeg ellers bruker for å legge til ansikt i databasen. Hvordan detektoren fungerer er beskrevet i teorien og i utviklingen av programmet. En liten endring er at man kun skal se et bilde dersom det er detektert et ansikt, noe som gjør at jeg enkelt kan teste hvor langt unna et ansikt kan være i et bilde fordi bildet fryser når et ansiktet ikke lenger detekteres. Jeg har bedt testpersonen starte med ansiktet helt inntil kameraet, for deretter å bevege seg bakover til programmet fryser.

1. Alle størrelser innenfor gitt bilde:

Figur 27 - Haar (Alle størrelser)

Testpersonen er ca. 6,5 meter fra kameraet. Programmet har detektert ansiktet hele veien bort til der personen står på bildet, men det har vært utrolig hakkete med en maks hastighet på 1 FPS (uavhengig av avstand). Selv om jeg har lagt inn at programmet skal detektere et ansikt (teoretisk) på 1x1 piksel, var resultatet på denne testen at det minste detekterte ansiktet var på 21x21piksler.

2. Minimum 25x25 piksler:

Figur 28 - Haar (Min. 25x25 piksler)

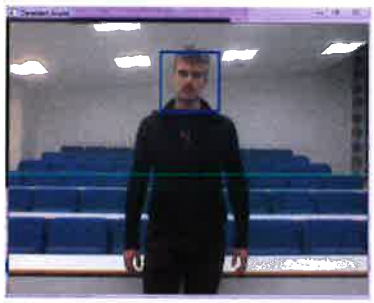
Testpersonen er ca. 5,5 meter unna kameraet. Programmet er like hakkete nå som under første gjennomkjøring, altså med en maks hastighet på 1 FPS. Størrelsen på ansiktet er 26x26 piksler. Tolker dette som at størrelsen fungerer, men at timingen på hvor testpersonen var plassert i det sekundet bildet ble tatt var irrelevant nærmere enn maks.

3. Minimum 50x50 piksler

Figur 29 - Haar (Min. 50x50)

Testpersonen er nå ca. 3 meter fra kameraet. Bildene blir tatt med mellom 3 og 5 FPS, programmet er altså fortsatt hakkete. Rektangelet som dekker ansiktet er på 64x64 piksler, og man ser at ansiktet er litt mindre.

4. Minimum 100x100 piksler



Figur 30 - Haar (Min. 100x100)

Testpersonen står nå ca. 1.5 meter fra kameraet. Programmet har en betydelig bedre bilderate, og det virker å gå relativt jevnt på mellom 15-20 FPS. Denne bilderaten er god, og det er kun veldig krappe bevegelser som gjør at det ser hakkete ut.

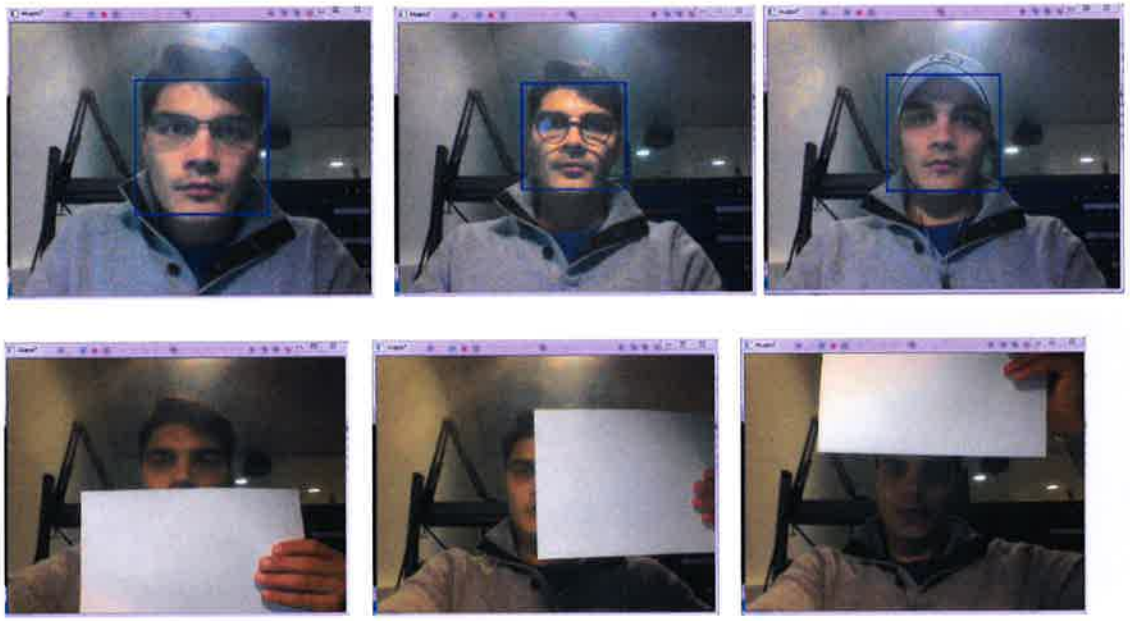
5. Minimum 200x200 piksler

Figur 31 - Haar (Min. 200x200)

Testpersonen sitt ansikt er nå ca. 50cm fra kameraet. Dette er normal størrelse på et ansikt som sitter foran Webkameraet som jobbes med. Bilderaten er ca. 30 FPS, og programmet

kjører sømløst. Detektoren er rask, og man kan bevege hodet mye frem og tilbake, opp og ned, uten at detektoren mister ansiktet.

6. Dekket til ansikt



Figur 32 - Haar, med ulike tildekninger

Bildene over er eksempler fra tester jeg har utført for å finne ut hvor «terskelen» til ansiktsdetektoren går med tanke på tildekninger, både naturlige og unaturlige. De øverste bildene viser at jeg bruker briller, og resultatet er at dette har relativt liten innvirkning på detektoren. Heller ikke en caps betyr noe, med mindre man bøyer hodet unaturlig langt nedover. Bildene i nederste rekke viser hvordan jeg tildekker ansiktet med et papir fra ulike hold. Bildene er tatt rett etter at ansiktet ikke lenger er detektert.

Resultat: Resultatet fra testene over forteller meg at denne algoritmen fungerer godt dersom jeg setter minimumsstørrelsen større eller lik 100x100 piksler. Den fungerer også godt med forskjellige naturlige tildekninger og videre resultat må drøftes opp mot LBP metoden.

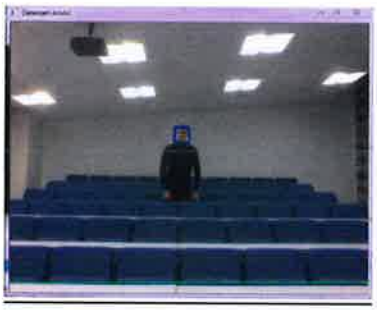
6.1.2 Test av ansiktsdetektor med LBP metoden

Testen går ut på: Å bruke LBP-metoden for å detektere ansikt i sann tid fra webkameraet i den hensikt å finne ut om hvilke størrelser som er lure å bruke, med tanke på et effektivt program, hvilke utfordringer detektoren har med tanke på hele ansikt, om de er tildekket, bruker de briller/solbriller og hvor mye tilt/vridning kan et ansikt ha.

Ønsket resultat: Etter testingen ønsker jeg å sitte igjen med svar på parameterne jeg har satt opp over, for å sammenligne det med den andre deteksjonsmetoden jeg kan bruke. Dette er parametere jeg ønsker å bruke videre i oppgaven.

Gjennomføring: For å teste dette bruker jeg programmet mitt som jeg ellers bruker for å legge til ansikt i databasen. Hvordan detektoren fungerer er beskrevet i teorien og i utviklingen av programmet. En liten endring er at man kun skal se et bilde dersom det er detektert et ansikt, noe som gjør at jeg enkelt kan teste hvor langt unna et ansikt kan være i et bilde fordi bildet fryser når et ansiktet ikke lenger detekteres. Jeg har bedt testpersonen starte med ansiktet helt inntil kameraet, for deretter å bevege seg bakover til programmet fryser.

1. Alle størrelser innenfor gitt bilde:



Figur 33 - LBP (Alle størrelser)

Med LBP metoden er hastigheten i programmet relativt mye raskere enn med Haar-metoden. Når jeg leter etter ansikt av alle størrelser viser den bilder med ca. 10 FPS. Det vil si at det er ganske hakkete, men en stor forskjell fra da jeg testet Haar-metoden med samme størrelse. Ulempen her (i forhold til Haar-metoden) er at det minste detekterte ansiktet er på 25x25 piksler, og dersom testpersonen beveget seg lengre unna ble ikke lengre ansiktet detektert. Likevel er ikke dette en kritisk stor forskjell.

2. Minimum 25x25 piksler:

Figur 34 - LBP (Min 25x25)

Også med denne størrelsen står testpersonen nærmere kameraet enn ved samme test med den andre metoden. Likevel er bilderaten bedre, på mellom 12-14 FPS. Bildet til høyre viser at det er detektert et ansikt der det ikke faktisk finnes noe ansikt. Dette er en ting som skjer relativt ofte når jeg har testet deteksjon ved bruk av LBP. Ustabiliteten er liten, men det er mye i forhold til Haar-metoden som ikke har detektert noe feil.

3. Minimum 50x50 piksler:

Figur 35 - LBP (Min 50x50)

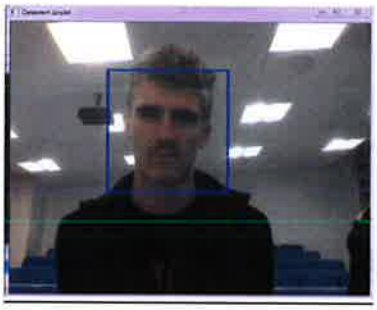
Også her er testpersonen nærmere kameraet enn ved samme test tidligere. Raten er nå på over 20FPS, som er en stor forbedring fra det Haar-metoden kunne levere på samme minimumsstørrelse. I dette bildet begynner man å se antydninger til at rektangelet rundt ansiktet dekker ett større området enn det rektangelet fra Haar-metoden gjorde.

4. Minimum 100x100 piksler:

Figur 36 - LBP (Min. 100x100)

Bilde raten er her på ca. 30 FPS, nok en gang raskere enn før. Testpersonen står også i dette tilfellet nærmere kameraet enn ved samme test med den andre metoden. Man ser også at rektangelet har et større areal i forhold til. Ansiktet enn man hadde ved Haar-metoden.

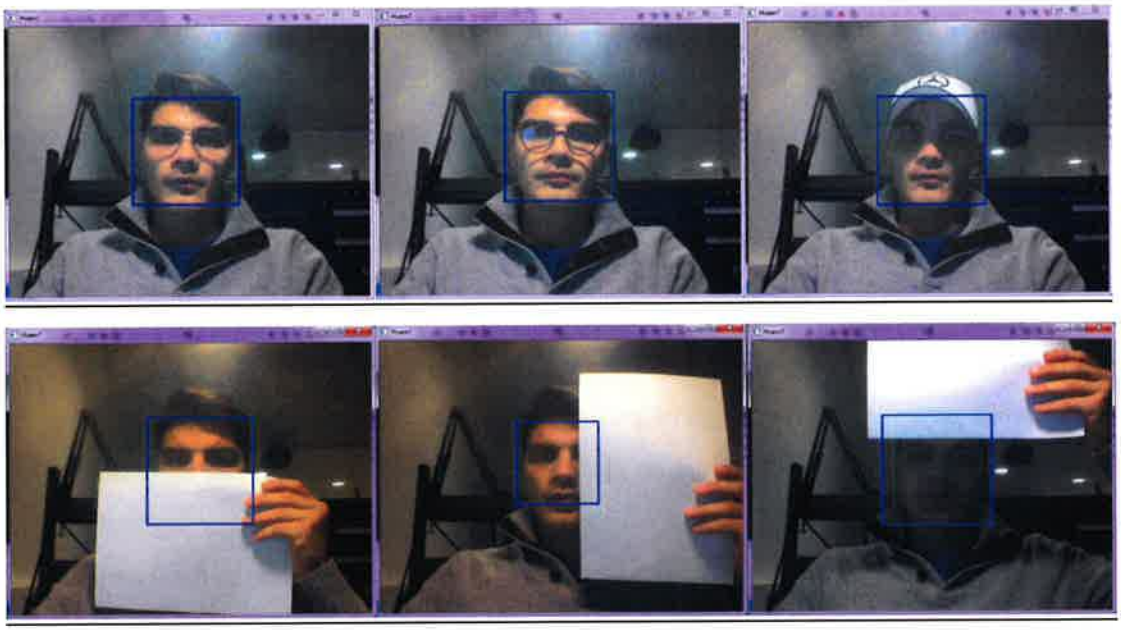
5. Minimum 200x200 piksler:



Figur 37 - LBP (Min 200x200)

Bilderaten med denne størrelsen er like rask som det kameraet har kapasitet til. Et problem, som er nevnt tidligere i denne testen er at LBP-metoden detekterer mange andre ting på bildet som ikke er ansikt, men som algoritmen tolker som et ansikt. Dette er vanskelig å få vist et bilde av det, men man kan forestille seg at det «flimrer» rektangler på ulike steder i bildet på tilfeldige tidspunkt.

6. Dekket til ansikt:



Figur 38 – LBP, med ulike tildekninger

LBP detektoren viser her at den er like god som Haar-detektoren når det kommer til å detektere ansikt som er tildekket av naturlige ting (Briller, caps). Detektoren er faktisk bedre til å detektere et ansikt som er tildekket som man ser på bildene i 2. rekke. Dette er en faktor å vurdere dersom man f.eks. skal bruke detektoren til å detektere ansikt fra f.eks. et overvåkningskamera eller noe lignende.

Resultat: LBP er åpenbart raskt, og detekterer også ansikt godt. Likevel virker det som om metoden er mer usikker, og viser derfor en del feildeteksjoner.

6.1.3 Konklusjon, ansiktsdetektor

I testene over har jeg kjørt programmet med forskjellige deteksjonsalgoritmer for å teste ut hvilken som passer best til mitt bruk. Begge algoritmene har fordeler og ulemper. F.eks. så er Haar-metoden mer hakkete og treig enn LBP-metoden. Men, samtidig er den også mer stabil og fungerer fint når man endrer minimumsstørrelsen. LBP detektoren returnerer et rektangel som er større enn det rektangelet Haar-detektoren returnerer. Det betyr også at dersom jeg tar ut et ansiktsbilde fra LBP-detektoren vil det bestå av mer bakgrunnsstøy enn et bilde tatt med Haar-metoden. Videre så skal jeg bruke denne metoden til en gjenkjenningsalgoritme, og derfor velger jeg å bruke primært Haar algoritmen for deteksjonsdelen av programmet på grunn av følgende punkter:

1. Haar-metoden er rask nok dersom jeg setter minimumsstørrelsen til 100x100 piksler.
2. LBP-metoden har en del feil-detekteringer som vil lage kluss for gjenkjenningsalgoritmene. Haar metoden er, etter det testene har vist meg, mer robust.
3. Haar-metoden lager et mindre rektangel rundt ansiktet. Siden jeg henter ut bildet som er innenfor rektangelet synes jeg det er bedre å minske «støyet» utenfor ansiktet i bildene i databasen.

6.2 Gjenkjenning

Som nevnt mange ganger tidligere vil gjenkjenningsalgoritmene bli påvirket av mange forskjellige faktorer i et bilde. For å skape en gjenkjenner som er relativt robust, og takler forskjellige scenarioer med et godt resultat valgte jeg å teste forskjellige faktorer jeg tenkte kunne påvirke resultatet. For alle testene som er utført ville jeg også måle de to forskjellige algoritmene jeg bruker opp mot hverandre, for å kanskje kunne avgjøre hvilken av de som fungerte best for meg.

6.2.1 Test med antall bilder i databasen

Testen går ut på: Å prøve programmet med ulike antall bilder i databasen i den hensikt å finne et «minimum» antall ansikt jeg trenger for å lage en robust gjenkjenner.

Ønsket resultat: Finne et minimumsantall bilder man trenger per person i databasen. Samtidig se hvilke forskjeller algoritmene har innenfor dette kriteriet.

Gjennomføring: Som jeg har nevnt i utviklingen av programmet så finner jeg en variabel som forteller meg hvor ulikt et inputansikt er ansiktene i databasen, jo lavere tallet er, jo sikrere er gjenkjenningen. Videre vil jeg nå teste programmet med forskjellig antall bilder i databasen for å se hvor mye påvirkning det har. For å gjøre testen så reell som mulig ser jeg for meg at bildene i databasen trenger variasjon. Derfor vil jeg legge inn bilder fra ulike steder (ulike lysforhold) og med ulike vinkler og grimaser på ansiktet. Bildene under viser resultater med ulike antall bilder i databasen. Blått rektangel viser at jeg bruker Eigenfaces algoritmen. Denne testen kan jeg ikke utføre med Fisherface algoritmen, da den krever minst to ulike ansikt for å gjenkjenne.

Figur 39 - Gjenkjenning med Eigenfaces

Bildene over viser et detektert ansikt, og en ulikhetsvariabel fra ansiktene i databasen. Bildene viser verdier i forhold til hvor mange bilder som er i databasen, henholdsvis 2 bilder for verdien øverst til venstre, deretter 5, 10, 20, 50 og 100 bilder.

Resultat: Testen gir et godt svar på hvor mye en god database har å si for gjenkjenning. Selvfølgelig vil det lønne seg å ha så mange bilder i databasen som mulig, men dette kan også gå utover hastigheten til programmet, da det er mye som skal prosesseres. For å teste videre velger jeg å ha 50 bilder per person i databasen. Dette er relativt enkelt å samle inn, samtidig som jeg kan sette en terskel på ca. 1 for å predikere om det er et kjent eller ukjent ansikt.

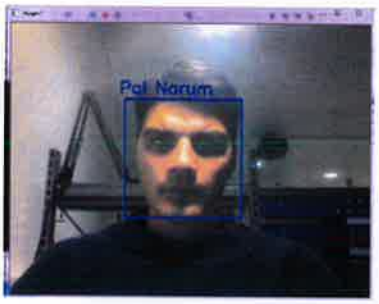
6.2.2 Test med antall ulike personer i databasen

Testen går ut på: Å finne ut om det har noen betydning å a en stor database. Hvordan vil det påvirke resultatet fra gjenkjenneren dersom databasen består av veldig mange forskjellige ansikt? Vil den begynne å tro at noen ligner mer på andre enn tidligere eller vil algoritmene fortsatt skille enkeltpersoner.

Ønsket resultat: Finne ut om det finnes en grense med tanke på antall mennesker man kan gjenkjenne i samme program. Finne ut om den ene algoritmen takler en stor database bedre enn den andre.

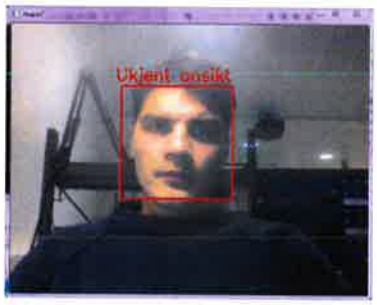
Gjennomføring: Denne testen baserer jeg på en tidligere test, der jeg fant ut at gjenkjenneren gir et godt resultat med 50 bilder per person. Å skaffe 50 bilder av 100 ulike mennesker er relativt vanskelig, og ekstremt tidkrevende. Derfor har jeg benyttet meg av å skaffe så mange bilder som mulig av 100 enkeltpersoner. For å gjøre dette har jeg kombinert det å selv ta

bilder av ansikt og legge de inn i databasen med å bruke diverse ansikts databaser fra internett. Antallet bilder per person varierer, men det er minst 10 av hver eneste person. Jeg har testet hvordan programmet reagerer med henholdsvis 2, 5, 10, 20, 50 og 100 forskjellige personer i databasen.



Figur 40 - Eigenfaces, med 2 personer i databasen

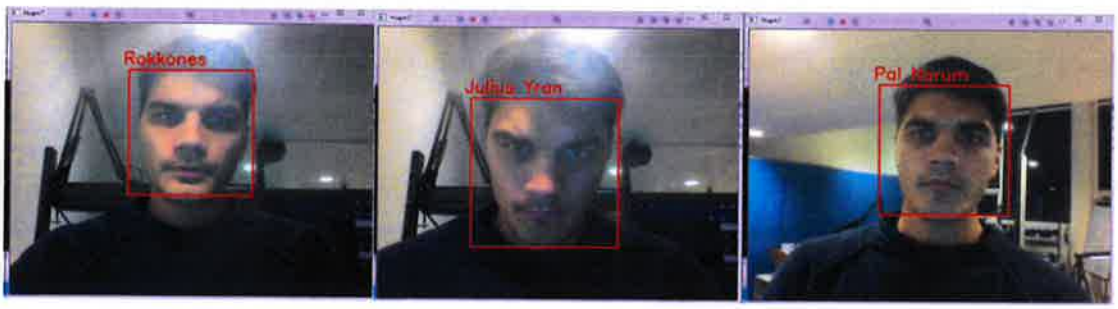
Nå har jeg også lagt inn at terskelen på «godkjent gjenkjenning» skal være mindre eller lik 1. Med Eigenfaces algoritmen og to bilder i databasen fungerer dette helt utmerket, med få antydninger til feile resultat.



Figur 41 - Fisherfaces, med 2 personer i databasen

Med Fisherface algoritmen ble jeg ikke gjenkjent. Verdien for ulikhet var på 1.7, noe som var mye høyere enn forventet. Uten å egentlig ha noe svar på grunnen til dette fortsatte jeg å øke antall personer i databasen, for å finne ut om det ville påvirke utfallet.

Med et høyere antall forskjellige ansikt i databasen begynte verdien på ulikhet å gå ned med Fishermetoden, men likevel så presterte programmet relativt dårlig med denne algoritmen.



Figur 42 - Fisherfaces, 50 ansikt i databasen

Figur 38 visert skjermbilder fra en test med 50 forskjellige ansikt i databasen. Det er vanskelig å illustrere hvor ustabil gjenkjenneren er, men den sliter med å fastslå ett ansikt, og hopper mye mellom alle som ligger i databasen. Når jeg benytter meg av eigenansikt metoden med et større antall i databasen gir den et mer konstant svar, selv om det er noe ustabilitet ved noen bevegelser.

Resultat: Resultatet er godt når jeg bruker Eigenfaces, og mindre godt når jeg benytter meg av Fisherfaces. Det er foreløpig vanskelig å gi en god forklaring på dette uten å teste mer. Det som er interessant er at det kun dukker opp forslag fra de jeg selv har tatt bilder av til databasen, og aldri av de som er hentet fra internett. Det kan ha mye å gjøre med at internett bildene er mer konstante med tanke på hvordan de er forhåndsprosessert. Uansett så tolker jeg resultatene fra denne testen som følgende: Antall ansikt i databasen har en viss påvirkning, da 50 bilder per person vil føre til likheter mellom ulike ansikt. Likevel er programmet relativt robust når jeg bruker Eigenansikt metoden. For Fisherface metoden er det mer ukorrekte svar enn korrekte. Ulikhet verdiene er også meget høye når jeg bruker denne metoden.

6.2.3 Test i forskjellige lysforhold

Testen går ut på: Å prøve å skaffe seg forståelse for hvordan lysforhold i både bildene i databasen, og inputansiktet påvirker gjenkjenneren. Gjennom alle testene jeg har gjennomført, både de som er beskrevet her i oppgaven, men også hver gang jeg har testet programmet har jeg opplevd noen avvik og forskjeller når det kommer til hvor jeg befinner meg. F.eks. så satt jeg sammen det jeg trodde var en god database der jeg la inn 50 bilder av meg selv hvor alle var tatt i samme rom, men med en lampe jeg forflyttet rundt for å prøve å simulere ulike forhold. Da jeg testet programmet, i samme rom, virket gjenkjenningen feilfritt, men da jeg flyttet meg begynte ulikhetsverdien å øke. Konklusjonen min var da at selv om jeg prosesserer bildene slik at de kun består av ansikt, så har forskjellige bakgrunner en viss påvirkning da det ofte er bakgrunnene i et bilde som danner det spesifikke lysforholdet.

Ønsket resultat: er å kunne vite hva slags forskjeller i lysforhold som påvirker et godt resultat, og hvordan jeg videre kan lage en robust database som tar høyde for lysforskjeller.

Gjennomføring: For å prøve å skape en reell test, så brukte jeg kun de bildene i databasen som jeg hadde tatt selv. Dette er fordi det er de som har mest ulike lysforhold, da internettdatabasen er relativt normalisert. Videre gikk kjørte jeg programmet samtidig som jeg gikk rundt på skolebygget. Nye lysforhold overalt.

Når jeg satt i klasserommet der jeg lagde primærdelen av databasen for mitt ansikt fungerte gjenkjenneren relativt godt. Men, i det jeg begynte å bevege meg rundt i bygget ble programmet mer ustabilt. Dersom jeg var i rommet der jeg tok database-bildene av noen andre så begynte programmet å bli veldig usikkert og gjenkjenne mitt ansikt som andre oftere enn meg selv.

Derfor oppdaterte jeg databasen av meg selv ved å ta minst et bilde for alle rom jeg var inne i. Etter det begynte gjenkjenneren å bli mer stabil, men fortsatt med en del unntak.

Resultat: Resultatet viser at jeg foreløpig ikke har et robust program. Det fungerer under det jeg kan kalle optimale forhold, men sliter i ukjente miljø. Programmet er foreløpig utrolig sensitivt for ulike lysforhold, uten at jeg har noen konkrete måter å rette opp i det foreløpig. Fortsatt så viser det seg at Eigenfaces algoritmen fungerer bedre for meg, enn Fisherfaces.

6.2.4 Konklusjon gjenkjenning

Etter å ha gjennomført 3 tester, som har tatt relativt mye tid står jeg egentlig igjen med flere nye spørsmål enn håndfaste svar. Jeg ser at testene mine på svært få måter er omfattende nok til å kunne skaffe sikre resultater, og det får bli en jobb for videre utvikling av et gjenkjenningsprogram. Samtidig har testene også lært meg mye om hva som bør utforskes videre. F.eks. så kan bedre for-prosessering av bildene være noe som kan ha stor effekt da det virker som om algoritmene blir ekstremt påvirket av lysforhold i et bilde. Det som har fungert best for meg er å bruke Eigenfaces metoden, men hvorfor det er sånn akkurat i mitt tilfelle er da neste spørsmål.

7. Konklusjon

Problemstillingen jeg stilte i forbindelse med bacheloroppgaven var: *«Hvordan kan en datamaskin kjenne igjen et ansikt?»*.

I løpet av arbeidet har jeg lært om forskjellige ansikts- deteksjon og gjenkjennings algoritmer. Jeg har laget et testprogram som fungerer, men med mange suksesskriterier for at det skal fungere helt optimalt. Oppgaven har gitt meg mye innsikt i bildeprosessering, og da spesielt bruken av bibliotekene til OpenCV.

Hvordan en datamaskin kan kjenne igjen et ansikt er det mange svar på, og jeg har ikke vært i nærheten av å utforske alle. Oppgaven min svarer på hvordan en datamaskin kan kjenne igjen ansikt ved bruk av de algoritmene og metodene jeg har fordypet meg i. Jeg har lagt vekt på å virkelig ha en god forståelse av disse metodene, og samtidig lage et program som visualiserer metodene. For å oppsummere så har jeg laget et program som detekterer et ansikt, før det tar et utsnitt av ansiktet og prosesserer bildet slik at det kan lagres i en database. Videre så har jeg utviklet selve gjenkjenningsprogrammet som sammenligner et bilde med bilder i databasen for å avgjøre om et ansikt er kjent eller ikke.

Prosessen har vært tung og jeg har sittet mange timer i frustrasjon over småfeil i programkode eller mangel på forståelse. Jeg skulle gjerne ha brukt mer tid på å utforske flere aspekter ved bildeprosessering, som sannsynligvis ville hjulpet meg med å forbedre testingen og forståelsen min. Likevel så mener jeg at jeg har laget et program som svarer på problemstilling min da det, under optimale forhold, kjenner igjen et ansikt.

8. Videre arbeid

Jeg mener at denne kunnskapen er relevant dersom man ønsker å bli bedre kjent med bildeprosesseringsemnet innenfor programmeringsverden. Samtidig så bygger man videre på programmeringsfaget vi hadde i 4. semester på Sjøkrigsskolen. Mulighetene videre er mange, og med en grunnleggende forståelse kan man begynne å implementere ansiktsgjenkjenning i en rekke ulike programmer.

Først og fremst vil jeg påstå at videre arbeid dreier seg om mer testing og utvikling av testprogrammet jeg har laget til denne oppgaven. Hovedmålet, før denne type program kan implementeres i andre systemer, er å lage programmet så robust som overhode mulig. Det at man bruker ansiktsgjenkjenning i passkontrollen er et bevis på at det er mulig.

For å jobbe videre med emnet denne oppgaven omhandler vil jeg anbefale å sette noen krav til et ferdig produkt. F.eks. 70% gjenkjenningsrate under alle forhold. Eventuelt spesialisere programmet til å fungere helt optimalt under et spesifikt forhold. Jeg vil, dersom det er aktuelt, råde folk til å faktisk forstå algoritmene fremfor å kun bruke funksjoner fra OpenCV. Det har gjort programmeringen og feilsøkingen veldig mye enklere.

Jeg håper at denne oppgaven kan gi et utgangspunkt til videre arbeid innenfor temaet.

9. Bibliografi

Monografier

Bradski Gary, Kaehler Adrian

2008, Learning OpenCV // Computer Vision with the OpenCV Library: O`Reilly Media Inc, USA

Lervik Else, Ljosland Mildrid

2009, Programmering i C++ // En innføring i strukturert og objektorientert programmering: Gyldendal, Norge

Baggio Daniel Lélis, Emami Shervin, Escriva David, Levgen Khvedchenia, Mahmood Naureen, Saragih Jason, Shilkrot Roy

2012, Mastering OpenCV with Practical Computer Vision Projects, Packt Publishing, UK

Dokumenter

Stanford

2008, *Face Detection using LBP Features // Project Report*

Reinius, Staffan

2013, Object recognition using the OpenCV Haar cascade-classifier on the iOS platform, Uppsala Universitet, Sverige

Websider

Wikipedia Homepage

2015, Microsoft Visual Studio, www

(https://no.wikipedia.org/wiki/Microsoft_Visual_Studio)

2015, Cascading Classifiers, www

(https://en.wikipedia.org/wiki/Cascading_classifiers)

OpenCV Homepage

2015, Haar Cascades, www

(http://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0)

2015, Basic Structures, www

(http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html)

2015, Basic Image Container, www

(http://docs.opencv.org/2.4/doc/tutorials/core/mat_the_basic_image_container/mat_the_basic_image_container.html)

2015, Face Recognizer, www

(http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_api.html)

OpenCV Tutorial C++

2013, What is OpenCV? www

(http://opencv-srf.blogspot.no/2010/09/what-is-opencv.html)

2013, Installing & Configuring with Visual Studio, www

(http://opencv-srf.blogspot.no/2013/05/installing-configuring-opencv-with-vs.html)

Shervin Emami

2012, Introduction to Face Detection and Face Recognition, www

(http://www.shervinemami.info/faceRecognition.html)

10. Vedlegg

VEDLEGG A – Programkode Ansiktsgjenkjenning

```
//Program for gjenkjenning av ansikt
//
//Pål A. Narum, 2015
//
//For at programmet skal fungere må det finnes en database med ansiktsbilder
//av samme størrelse. Hvert bilde må knyttes opp mot et navn i en txt-fil

//Include filene fra openCV
#include <opencv2\core\core.hpp>
#include <opencv2\objdetect\objdetect.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\contrib\contrib.hpp>
#include <opencv2\imgproc\imgproc.hpp>

#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;
using namespace cv;


//Funksjonen les_csv er hentet fra OpenCV dokumentasjon
//Funksjonen leser inn informasjon til trening av gjenkjenningsalgoritme
static void les_csv(const string& filnavn, vector<Mat>& bilder, vector<int>& merkinger,
char skille = ';'){
	ifstream fil(filnavn.c_str(), ifstream::in);
	if(!fil){
		string error_message = "Feil! Sjekk filnavnet";
		CV_Error(CV_StsBadArg, error_message);
	}

	string linje, hvor, klassetype;
	while(getline(fil, linje)) {
		stringstream linjenummer(linje);
		getline(linjenummer, hvor, skille);
		getline(linjenummer, klassetype);
		if(!hvor.empty() && !klassetype.empty()){
			bilder.push_back(imread(hvor, 0));
			merkinger.push_back(atoi(klassetype.c_str()));
		}
	}
}

//Funksjon som brukes til sammenligning av to bilder
double getSimilarity(const Mat A, const Mat B);

//Navneliste for å vise predikert person som navn
//istedet for å vise det som et tall
string navneliste[] =
{
	"Navn 1",
	"Navn 2",
	"Navn 3",
	"Navn 4",
	"Navn 5",
	"Navn 6",
	"Navn X"

};
```

```
//Main
int main(int argc, const char *argv[]){

//Definisjoner av variabler
string Haar = "c:/haarcascade_frontalface_alt.xml";
string LBP = "c:/lbpcascade_frontalface.xml";
string csvfil = "csv.ext";

vector<Mat> bilder;
vector<int> merkinger;

Mat bildeFraKamera;

//Gjør klar informasjon om databasen til trening av gjenkjenner
try {
	les_csv(csvfil, bilder, merkinger);
}
catch (Exception& e) {
	cerr<<"Feil med innlasting av filen"<<endl;
	exit(1);
}

//Innlasting og trening av gjenkjenneren
Ptr<FaceRecognizer> modell = createFisherFaceRecognizer();
//Ptr<FaceRecognizer> modell = createEigenFaceRecognizer();
modell->train(bilder, merkinger);

//Innlasting av deteksjonsalgoritme
CascadeClassifier Cascade;
Cascade.load(Haar);

//Henter ut informasjon om størrelsen på bildene i databasen
int bredde = bilder[0].cols;
int hoyde = bilder[0].rows;

//Initialiserer webKameraet
VideoCapture A(0);

if(!A.isOpened()){
	return (-1);
}


//Her starter loopen som prosesserer hvert bilde som vises fra webkameraet.

for(;;){

//Lagrer bildet fra kameraet inn i en Mat(rise)
A>>bildeFraKamera;
//Endrer bildet til et gråtonebilde
Mat graa;
cvtColor(bildeFraKamera, graa, CV_BGR2GRAY);
//ansikt er variabelen til rektangelet som blir detektert
//Detekterer et ansikt i gråtonebildet.
vector<Rect_<int>> ansikt;
Cascade.detectMultiScale(graa, ansikt, 1.1, 2, 0 | CASCADE_SCALE_IMAGE, Size (130, 130));

//Her starten prosessen for å kjenne igjen et ansikt
for(int i=0;i<ansikt.size();i++){
	Rect ansikt_i=ansikt[i];
	//alfa er det beskjærte bildet, kun av ansiktet
	Mat alfa = graa(ansikt_i);

	//Endrer størrelsen, slik at det blir likt som bildene i databasen
	Mat ansiktNyStr;
```

```
	resize(alfa, ansiktNyStr, Size(bredde, hoyde), 1.0, INTER_CUBIC);

	//Jevner ut lysforholdene
	//Koden under er hentet fra OpenCV
	//Den deler ansiktsbildet i 2 og endrer kontrasten i hvert bilde
	//før bildet blir satt sammen igjen
	equalizeHist(ansiktNyStr, ansiktNyStr);
	int midtenX = bredde/2;

	Mat VenstreSide = ansiktNyStr(Rect(0,0,midtenX,hoyde));
	Mat HoyreSide = ansiktNyStr(Rect(midtenX, 0, bredde-midtenX, hoyde));

	equalizeHist(VenstreSide, VenstreSide);
	equalizeHist(HoyreSide, HoyreSide);

	for(int y=0;y<hoyde;y++){
		for(int x=0;x<bredde;x++){
			int A;
			if(x<bredde/4){
				A = VenstreSide.at<uchar>(y,x);
			}
			else if(x<bredde*2/4){
				int VA = VenstreSide.at<uchar>(y,x);
				int HeleA = ansiktNyStr.at<uchar>(y,x);
				float f = (x-bredde*1/4)/(float)(bredde/4);
				A = cvRound((1.0f-f)*VA+(f)*HeleA);
			}
			else if(x<bredde*3/4){
				int HA = HoyreSide.at<uchar>(y,x-midtenX);
				int HeleA = ansiktNyStr.at<uchar>(y,x);
				float f = (x-bredde*2/4)/(float)(bredde/4);
				A = cvRound((1.0f-f)*HeleA+(f)*HA);
			}
			else{
				A = HoyreSide.at<uchar>(y,x-midtenX);
			}
			ansiktNyStr.at<uchar>(y,x) = A;
		}
	}

//Henter ut egenvektorene fra databasen og gjennomsnittsansiktet
Mat eigenvectors = modell->get<Mat>("eigenvectors");
Mat averageFaceRow = modell->get<Mat>("mean");
//Lager egne egenverdier av bildet for å kunne finne en ulikhetsvariabel
Mat projection = subspaceProject(eigenvectors, averageFaceRow, ansiktNyStr.reshape(1,1));
Mat reconstructionRow = subspaceReconstruct(eigenvectors, averageFaceRow, projection);

Mat reconstructionMat = reconstructionRow.reshape(1, hoyde);

Mat reconstructedFace = Mat(reconstructionMat.size(), CV_8U);
reconstructionMat.convertTo(reconstructedFace, CV_8U, 1, 0);

//Predikerer hvem som er avbildet
int prediksjon = modell->predict(ansiktNyStr);

//Sammenligner inputbildet med database, setter terskel til 1.5
double similarity = getSimilarity(ansiktNyStr, reconstructedFace);
cout<<similarity<<endl;
if(similarity > 1.5){
	prediksjon = 8;
}

//Lager et rektangel rundt ansiktet
rectangle(bildeFraKamera, ansikt_i, CV_RGB(0, 0, 255),2, 4);
//rectangle(bildeFraKamera, ansikt_i, CV_RGB(255, 0, 0),2, 4);
```

```
//Skriver ut hvem som er avbildet
string prediksjonTekst;

if(prediksjon>=0 && prediksjon <=6){
	prediksjonTekst.append(navneliste[prediksjon]);
}
else prediksjonTekst.append("Ukjent ansikt");

int posX = max(ansikt_i.tl().x -10, 0);
int posY = max(ansikt_i.tl().y - 10, 0);
auto test = to_string(similarity);


putText(bildeFraKamera, prediksjonTekst, Point(posX, posY), FONT_HERSHEY_SIMPLEX, 1.0,
CV_RGB(0, 0, 255), 2.0);
//putText(bildeFraKamera, prediksjonTekst, Point(posX, posY), FONT_HERSHEY_SIMPLEX, 1.0,
CV_RGB(255, 0, 0), 2.0);
//putText(bildeFraKamera, test, Point(posX, posY), FONT_HERSHEY_SIMPLEX, 1.0, CV_RGB(10, 0,
255), 1.0);
}

//Viser bildet
imshow("Hvem?", bildeFraKamera);

if(waitKey(30)==27)
	break;
}
	return 0;
}

//Funksjon for å sammenligne bilder
double getSimilarity(const Mat A, const Mat B){
	double errorL2 = norm(A, B, CV_L2);
	double similarity = errorL2 / (double) (A.rows*A.cols);
	return similarity;
}
```

VEDLEGG B – Programkode Legg til ansikt i databasen

```
//Program for å legge til ansikt i en database
//
//Pål A. Narum
//
//Dette programmet detekterer et ansikt og kan på kommando
//lagre ansiktet i en database for gjenkjenning

//Headerfiler fra OpenCV
#include "opencv2\core\core.hpp"
#include "opencv2\objdetect\objdetect.hpp"
#include "opencv2\highgui\highgui.hpp"
#include "opencv2\imgproc\imgproc.hpp"

#include <iostream>
#include <fstream>
#include <sstream>
#include <time.h>

using namespace std;
using namespace cv;

//Variabler
string Haar = "c:/haarcascade_frontalface_alt.xml";
string LBP = "c:/lbpcascade_frontalface.xml";
string navn;
string filnavn;
int filnummer;
int persNummer;
Mat bilde, beskjaert, nyttBilde, graa, utjevnet;

CascadeClassifier Cascade;

int main(void){

vector<Rect> ansikt;

//Initialiserer webcamera for å hente bilder.
VideoCapture A(0);

if(!A.isOpened())
	return (-1);

//Innlasting av deteksjonsalgoritme
if(!Cascade.load(Haar))
	return (-1);

//Her starter loopen som gjør at bildene vises i sann-tid
for(;;){

//Lagrer bildet fra kamera
A>>bilde;
//Detekter et ansikt
Cascade.detectMultiScale(bilde, ansikt, 1.1, 2, 0 | CASCADE_SCALE_IMAGE, Size(150,
150));
Rect FunnetAnsikt_i;

//For-løkke for å vise et rektangel, og også finne
//punkter for beskjæring av bildet
for(int i=0;i<ansikt.size();i++){
```

```
        FunnetAnsikt_i = ansikt[i];
        Mat KunAnsikt = bilde(FunnetAnsikt_i);
        Point hjorne1(ansikt[i].x, ansikt[i].y);
        Point hjorne2((ansikt[i].x + ansikt[i].height), (ansikt[i].y +
        ansikt[i].width));
        rectangle(bilde, hjorne1, hjorne2, Scalar(255, 0, 0), 2, 4, 0);
}

//Viser bildet. Hvis jeg setter linjen under inn i for-løkken over
//vil programmet "fryse" når det ikke er et ansikt i bildet
//denne funksjonen bruker jeg i noen av testene
imshow("Detektert Ansikt", bilde);

//Venter på et tastetrykk (mellomromstasten) for å starte prosessen som er å lagre et
bilde
if(waitKey(30) == 32){
//Beskjærer hovedbildet til å kun bestå av det som er inne i rektangelet
beskjaert = bilde(FunnetAnsikt_i);
//Endrer størrelse, fordi alle bildene må lagres med samme størrelse
resize(beskjaert, nyttBilde, Size(50, 50), INTER_LINEAR);
//Konverterer bildet til sort/hvitt
cvtColor(nyttBilde, graa, CV_BGR2GRAY);

//Henter ut informasjon om størrelsen på bildet, bredde og høyde
int b = graa.cols;
int h = graa.rows;
//Jevner ut lysforholdene
//Koden under er hentet fra OpenCV
//Den deler ansiktsbildet i 2 og endrer kontrasten i hvert bilde
//før bildet blir satt sammen igjen
equalizeHist(graa, utjevnet);
int midtenX = b/2;
Mat VenstreSide = graa(Rect(0,0,midtenX,h));
Mat HoyreSide = graa(Rect(midtenX, 0, b-midtenX, h));
equalizeHist(VenstreSide, VenstreSide);
equalizeHist(HoyreSide, HoyreSide);

      for(int y=0;y<h;y++){
            for(int x=0;x<b;x++){
                  int A;
                  if(x<b/4){
                        A = VenstreSide.at<uchar>(y,x);
                  }
                  else if(x<b*2/4){
                        int VA = VenstreSide.at<uchar>(y,x);
                        int HeleA = utjevnet.at<uchar>(y,x);
                        float f = (x-b*1/4)/(float)(b/4);
                        A = cvRound((1.0f-f)*VA+(f)*HeleA);
                  }
                  else if(x<b*3/4){
                        int HA = HoyreSide.at<uchar>(y,x-midtenX);
                        int HeleA = utjevnet.at<uchar>(y,x);
                        float f = (x-b*2/4)/(float)(b/4);
                        A = cvRound((1.0f-f)*HeleA+(f)*HA);
                  }
                  else{
                        A = HoyreSide.at<uchar>(y,x-midtenX);
                  }
                  utjevnet.at<uchar>(y,x) = A;
            }
      }
```

```
//Skaffer informasjon om bildet
cout<<"Skriv inn navn og nummer på person som skal algres i databasen"<<endl;
cin>>navn>>persNummer;

//Setter sammen innhentet informasjon, og knytter det opp
//mot plasseringen av databasen
stringstream z;
z << "c:/ansikt/" << navn << filnummer << ".jpg";
filnavn = z.str();
filnummer ++;

//Lagrer bildet i databasen
imwrite(filnavn, utjevnet);

//Lagrer navn og plassering i en txt-fil
ofstream utfil;
utfil.open("csv.txt", ios::app);
utfil<<filnavn<<";"<<persNummer<<endl;
utfil.close();
}

else if(waitKey(3)==27)
        break;
}
}
```