Deep learning assignment 3

Anders Schultz (201909248) & Marc Jensen (201907421)

17/12-2021

Opgave: Semantisk segmentering af GTA billeder

Ideen i semantisk segmentering er, at markere hvilke pixels i et billede, som tilhører en bestemt klasse. I vores projekt laver vi semantisk segmentering på billeder, som er taget i computerspillet GTA V. I projektet har vi haft fokus på specielt at segmentere genstande observeret i trafikken.

Datasæt

De billeder vi fokuserer på, er taget fra kølerhjelmen på forskellige biler. Billederne er taget forskellige steder i spillet og i forskellige sekvenser. Dvs. at man har taget billeder mens man har kørt rundt i spillet. Til hvert screenshot er der også givet en segmenteret version. I denne version er der en specifik farve til hver klasse af objekter (se evt. et eksempel på input og target i bilag 1). Vi har hentet datasættet fra [7].

Data Behandling

Vi har vurderet screenshottene med henblik på kun, at anvende dem af god kvalitet. Vi har fjernet nogle af input/target sættene, da der er opstået fejl-klassifikationer på nogle af target billederne. Derudover har vi også fjernet input og target-billeder med regn/tåge. Dette skyldes, at vi ønsker, at netværket skal prædiktere godt på billeder med tydelige trafik-elementer. Dog ikke med den hensigt, at modellen skal overfitte til dette data.

Vi har besluttet at slå nogle klasser sammen til én: "everything else" klasse. De resterende klasser kan ses i figuren nedenfor.

```
invcol = {0:"everything else", 1:"curb", 2:"road", 3:"cars", 4:"trucks",
          5:"motorcycle", 6:"humans", 7:"busses", 8:"tram/train"}

colors = {(0, 0, 0):0, (244, 35, 232):1, (128, 64, 128):2, (0, 0, 142):3,
          (0, 0, 70):4, (0, 0, 230):5, (220, 20, 60):6, (255, 0, 0):6,
          (0, 60, 100):7, (0, 80, 100):8}
```

Figure 1: Vores valgte klasser og tilsvarende farve representationer

I vores target billeder er der 2 forskellige representationer af "humans" alt efter om de sidder på et køretøj eller er fodgængere. Vi syntes ikke det er en relevant forskel at pointere, så derfor har vi sat dem i samme klasse. Derfor er der 10 farver og 9 klasser.

Herefter laver vi onehot encoding med disse klasser. Her bruges filen "GTA_onehot", som kan transformere vores target data med 3 channels til en onehot encoded version med 9 channels. De 9 channels svarer til hver af de 9 klassers representation i billedet. Grundideen i funktionen er, at vi kører igennem hver pixel i target billedet. For hver pixel tjekkes det, om dens farve er i "colors" dictionaryet i såfald sættes et 1-tal ind på key-indgangen i vores onehot encodede map. Hvis ikke pixelens farve findes i colors eller farven er sort, så sættes et 1-tal på det 0-lag.

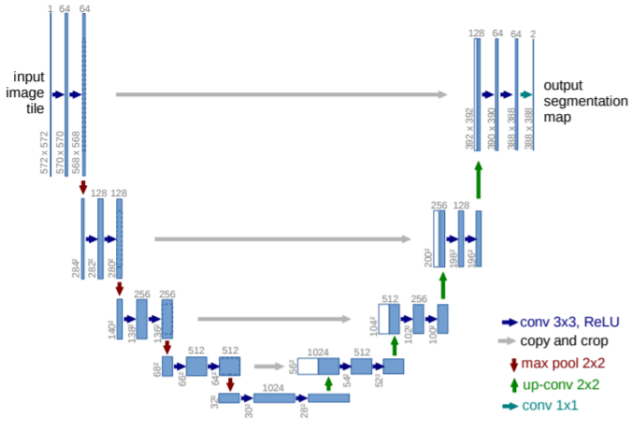
Udover det så har vi lavet to data-loaders (disse ligger i GTApack mappen): "GTA_loader" bruges til at hente vores input og target billeder på en standard form med 3 channels. "GTA_hotloader" bruges til at hente input billeder samt tilhørende onehot-encoded targets. Begge Data-loadere ændrer også størrelsen af billederne til 400 x 300. Bemærk, at dataloaderne er udarbejdet med inspiration fra part 2 i assignment 1. I data-loaderne anvnder vi cv2 til at indlæse billeder. Herefter sikrer vi os, at farvekoderne passer med matplotlibs ved at anvnede [:, :, ::-1]. For at få tensorens format på (channels x rows x columns), som er standard i pytorch, anvender vi permute(2, 0, 1)/255. Vi deler med 255 for at få værdierne til at være mellem 0 og 1. For at indlæse de onehot encodede targets anvendes torch.load.

For at træne, test og validere modellen har vi valgt at have 4961 trænings-, 2255 test- og 857 pseudo-test-billeder. Det

skal bemærkes, at alle splits indholder forksellige billede-sekvenser. Dette skyldes, at vi ikke vil teste netværket på billeder fra samme sekvens. Vi anvender "pseudo-test" splittet til at teste netværkerne i vores hyper-parametre tuning.

Netværk arkitektur

Vores netværk er inspireret af U-nets arkitettur [1]. Det er et meget kendt FCN (fully convolutional network), som er brugt i semantisk segmentering. Figuren nedenfor er fra [1], denne illusterer den gennerelle U-net struktur. Vi har anvendt en del af dens features. Vi har dog ændret lidt på up-samplingen, så input højde/bredde er lig output højde/bredde.




```
def forward(self, x):
    x1 = self.inc(x)
    x2 = self.down1(x1)
    x3 = self.down2(x2)
    x4 = self.down3(x3)
    x5 = self.down4(x4)
    x = self.up1(x5, x4)
    x = self.up2(x, x3)
    x = self.up3(x, x2)
    x = self.up4(x, x1)
    output = self.outc(x)
    return output
```

Figure 2: Illustration af U-net og vores kode til netværket

Vores netværk tager et input billede af 400 x 300 med 3 channels, og giver et output som består af 9 channels med samme dimension. (Dog kan netværket også tage mindre/større billeder, da det er en FCN).

Det skal bemærkes, at koden til netværket står i "GTA_Unet" filen under GTApack mappen på github. Vi har implementeret netværket i mindre dele, som tilsidst samles i et stort netværk.

Den først klasse som defineres hedder "Double_Convolution" og består af Conv2d() med kernel på 3x3, batchnorm() og ReLU() gennemløbet 2 gange (se linje 37-44). På figuren svarer dette til to af de mørkeblå pile. Denne klasse anvendes i "Down_Scale", som først anvender en maxpool() med 2x2 og derefter "Double_Convolution" (se linje 59-63). Dette markerer en rød pil samt to blå.

I koden, på figure 2 kan det ses, hvordan vi først avender en "Double_Convolution" (self.inc) derefter anvendes 4 x "Down_Scale". Hver udregning gemmes under vejs. På dette tidspunkt har vi 1024 channels i netværket.

Herefter kan upsamplingen begynde. Dette gøres med funktionen "Up_Scale" som består af en Upsample() med kernel 2x2, ConvTranspose2d() med kernel 2x2, en concatenation fra tidligere output og en "Double_Convolution". Det skal bemærkes, at netværket indsætter 0'ere på de kanter hvor det er nødvendigt. Dette får dimentionerne til at passe (dette skyldes, at resultatet skal konkateneres med tidligere output) (se linje 81-97). Da kan "Up_Scale" kaldes fire gange: "up1 - 4". Bemærk at i sidste kald ("up4") erstattes Conv2d() med ConvTranspose2d() for at output får de rigtige dimensioner.

Tilsidst anvendes "OutConv" som udfører en 1x1 convolution og en softmax (se linje 113-117). Dette sørger for at outputtets channels bliver lig n_classes, som i vores tilfælde er 9. Derudover får vi pseudo-sandsynligheder langs channels.

Træning og test af netværket

For at teste netværket har vi implementeret en "GTA_tester" funktion i vores GTApack pakke. Denne funktion tager en dataloader ind, samt modellen. Da vil funktionen køre modellen på dataet og sammenligne prædiktionen med target. Target er onehot encoded og prædiktionen er givet med samme dimention dog med pseudo-sandsynligheder. Derfor køres Pytroch's argmax langs channels på både target og prædiktion. Da fåes to maps med en channel, herefter kan man sammenligne target og prædiktion pixel-by-pixel. Til sidst tages gennemsnittet, og på denne måde opnåes den procentdel af pixels, som modellen har klassifiseret korrekt (se linje 31-33).

Den måde vi træner de forskellige netværker på, kan ses i "final-network" filen (under 3. part...). Først udvælges hvilke billeder der skal læses ind i netværket som train, validation og test. Dette gøres med Pytorch's random_split og numpy's

randint. Dette data kan gives til loader funktionerne, som herefter laves til en generator, med Pytorch's DataLoader funktion. Da vi anvender Neptune til at følge udviklingen, så indstilles denne med run. Herefter indstilles antallet af epoker. Parametrene kan oplyses til Neptune. I vores træninger har vi altid brugt loss funtionen MSELoss. Herefter kan netværket defineres og sendes til device. I forhold til hvilket netværk der køres, så defineres optimizor og scheduler (se næste sektion for information om schedulers). I selve netværket holder vi øje med den gennemsnittelige loss for både train og validation, derfor laves der lister til dette. Herefter kører vi træningen i det antal epoker, der er defineret. For hver epoke løber vi igennem trainloaderen. Ved hver mini-batch finder vi netværkets loss. Herefter backpropagerer vi fejlen. Nu kan average loss for træningen opdateres og oploades til Neptune. Herefter kan validation loss findes, ved at vi kører igennem validation sættet. Vi finder også den gennemsnittelige pixel-by-pixel accuarcy med funktionen "GTA_tester" på vores validations sæt. Tilsidst kan disse informationer oploades til Neptune, og vi kan tage et step med vores scheduler. Da kan netværket gemmes både lokalt og i neptune, og vi udregner accuarcy på test-sættet med "GTA_tester". Denne information oploades også til Neptune, og til sidst stoppes informations-strømmen til neptune.

I vores træninger har vi været nødsaget til at anvende en batch_size på 1, da vi ellers løber tør for hukommelse på vores GPU.

De anvendte learning-rate schedulers

I vores test for at finde det perfekte setup til vores netværk har vi taget udgangspunkt i de to optimizers: Adam og SGD. Derudover har vi anvendt forskellige learning rate schedulers. Herunder har vi undersøgt CosineAnnealingLR, CyclicLR og ReduceLROnPlateau. Ideen i schedulerne:

CosineAnnealingLR: Ideen i denne scheduler er, at man anver en cosinus kurve til at cycle learning rate op og ned. Dette gøres ved formlen: $\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \cdot (1 + cos(\frac{T_{cur}}{T_{max}} \cdot \pi))$. Her sættes η_{max} lig learning-rate indstillingen i optimizeren (også kaldet initial lr) og η_{min} sættes som en hyper-parameter. Det skal bemærkes, at disse parametre bestemmer hvilke grænser som cosinus kurven løber mellem. Værdierne T_{cur} står for den nuværende epoke og T_{max} står for det antal epoker, som det tager at lave en halv udsvingning (dvs. at hvis man starter med den højeste learning rate defineret, så vil man efter T_{max} epoker have den laveste learning rate defineret). [2]

CyclicLR: Ideen i denne scheduler er, at man lineært "cykler" learning raten mellem et minimum og et maximum. Dvs. at man skal indstille en minimum og en maximum learning rate. Derudover skal man indstille step_size_up, som bestemmer hvor mange epoker CyclicLR bruger på at lave en halv udsvingning. [6] anbefaler at step_size_up vælges så man opnår 3 eller flere hele udsvingninger. [3]

Både i CosineAnnealingLR og CyclicLR hæver og sænker man learning rate enten en eller flere gange. En høj learning-rate kan påvirke netværket negativt - da det kan medføre en lavere accuarcy. Forhåbningen er dog, at den lave accuarcy kun er midlertidig, og at man ved at træne netværket senere med en lavere learning rate opnår en endnu bedre performance. Dette kan fx. ske, hvis den forhøjede learning rate hjælper med at bryde ud af et lokalt minimum. Med disse metoder er det anbefalingsværdigt, at man slutter med en lav learning rate. [6]

ReduceLROnPlateau: Denne scheduler ændrer learning rate i forhold til en såkaldt "metric". En metric kan b.la. være validation loss/accuarcy. Ideen er, at scheduleren sænker learning rate med en factor (her vil man oftest vælge mellem 2-10), når metric'en stagnere. Dette indstilles med to parametre: "threshold" og "patience". "threshold"-værdien er grænsen for hvor lidt metricen må ændre sig, før scheduleren anser metric'en for stagneret. "patience" fortæller scheduleren hvor mange epoker, som metric'en skal være stagneret i. [4]

Resultater af hyperparameter tuning

Med disse 3 schedulers og 2 optimizors stætter vi syv forskellige netværker op. Disse netværker trænes i 50 epoker med 440 billeder, valideres på 60 billeder og testes på 100 billeder. Dvs. at vi har anvendt 500 tilfældige billeder fra train-sættet og 100 tilfældige billeder fra pseudo-test-sættet. Det skal bemærkes, at vi anvender SGD med momentum på 0.9. De syv netværker samt deres accuarcy på test-sættet kan aflæses nedenfor (se koden vi anvendte i mappen "1. part the seven networks"):

Netværk	optimzior + scheduler	accuarcy	Netværk	optimzior + scheduler	accuarcy
Netværk 1	SGD	93.5	Netværk 5	Adam + CyclicLR	93.2
Netværk 2	SGD + ReduceLROnPlateau	94.7	Netværk 6	SGD + CyclicLR	94.5
Netværk 3	Adam + ReduceLROnPlateau	91.7	Netværk 7	SGD + CosineAnnealingLR	94.2
Netværk 4	SGD + CyclicLR	94.5			

Med disse netværker eksperimentede vi med forskellige indstillinger. Se hvordan learning rate er indstillet i bilag 2. Det skal bemærkes, at vi anvendte noget nær den samme learning rate for de 7 netværker (undtagen netværk 7, hvor

vi startede med en scheduler med højere learning rate, og sluttede med en anden scheduler med lavere learning rate). I bilag 3 ses træning og validation-loss for de syv netværker. Her ses ikke nogen tydelige tegn på overfitting (loss for både train og validation aftager i samme takt). Derudover ses også en rimelig høj test-accuarcy på pseudo-test sættet (som ses i tabellen ovenfor).

Det skal også bemærkes, at vi testede netværkerne ved at afslutte træningen med både høj og lav learning-rate værdi. Dette gjorde dog ikke den store forskel. Ud fra disse data kunne vi se, at et setup med SGD og ReduceLROnPlateau eller CyclicLR kan have potentiale til at producere gode resultater. Derfor tager vi udgangspunkt i disse ved næste step af processen.

For at få det optimale setup skal vi b.la. finde en god learning rate. I henhold til [5] (side 6 - afsnit 4.1) og [6] så beskrives det, at man skal lave et præ-run af sit netværk, hvor learning rate øges lineært. På dette run skal man indsamle learning rate samt validation accuarcy. Så kan man lave et plot med learning rate og accuarcy. Her skulle man helst se, at accuarcy stiger hurtigt. Den learning rate det sker ved, vil være en god base rate (minimum). Når accuarcy derefter begynder at flade ud, blive ujævn eller falde, så har man fundet sin maximale learning rate. Dette kørte vi på et netværk (se filen: "max_learning-finder" i mappen 2. part...), hvor vi anvendte SGD, med learning rate fra 0.001 og til 2. Vi brugte 41 epoker og plottede resultaterne i venstre side af bilag 4. Det vi fandt er, at maximum learning rate ligger omkring 1.25 og måske ved en lidt mindre værdi. Dog undrede det os, at accuarcy blev ved med at stige. Derfor kørte vi eksperimentet igen men med learning rate fra 1 til 10. Her kan man se resultatet på højre plot i bilag 4. Det lader til, at netværket er rimelig resistent overfor høje learning rates.

For at undersøge disse høje learning rates, så lavede vi nogle test med et par mindre netværker. Disse klarede sig dog ikke godt. I vores eksperimentation blev vi derfor bekræftet i, at maximum learning rate ligger omkring 1.25.

På side 6 i [5] beskrives det, at man skal vælge en lavere learning rate end den maksimale learning rate, hvis man anvender en konstant learning rate. Ellers vil netværket ikke begynde at konveregere. Dog har vi ikke en konstant learning rate, men vi vil helst have, at netværket begynder at konvergere når vi anvender ReduceLROnPlateau. Derfor har vi i vores test valgt en learning rate på 1, som er tæt på den maximale learning rate. Vi valgte reduce faktoren til 10, patience til 3 og threshold til 0.01.

Det er beskrevet i [6] (side 4), at de fundene base (minimum) - og maximum learning rate er gode bounds for ens cyclic scheduler. Dog valgte vi, at antage en lidt lavere værdi for learning rates maximum. Vi valgte grænserne for cyclical fra 0.001 til 1.15. Vi valgte også step_size_up til 10. Da vi træner beggge netværker på 61 epoker, vil det resultere i, at Netværk_cyclic får hele tre udsvingninger, og ender med den laveste learning rate. I denne test vælger vi at anvende 720 trænings-, 80 validations- og 200 pseudo-test-billeder. Da opnåede vi resultaterne (se koden i mappen 2. part ...):

Netværk	optimzior + scheduler	accuarcy	Netværk	optimzior + scheduler	accuarcy
Netværk_reduce	SGD + ReduceLROnPlateau	95.59	Netværk_cyclic	SGD + CyclicLR	96.30

Det bemærkes, at der i begge tilfælde, er en stigning i test-accuarcy. Dog lader det til, at netværket: Netværk_cyclic er den der har opnået den højeste accuarcy på pseudo-test dataet. Ud fra disse resultater lader det ikke til, at hverken netværk_cyclic eller netværk_reduce har overfittet til træningsdataet. Det ses på plottene i bilag 5 + 6 at validation- og trænings loss aftager i sammen takt, samt test accuarcy er høj.

Resultater for det endelige netværk (se koden i 3. part final network)

I vores hyper-parameter tuning har vi fundet, at netværk_cyclic har de bedste evner til, at løse opgaven: At segmentere GTA data. Derfor vil vi prøve at køre dette netværk igennem 61 epoker og med 2100 trænings-, 300 validations- og 600 test-billeder. Så vi anvender et (70 %, 10 %, 20 %) split. Bemærk, at vi nu tester netværket på det rigtige test-data. I bilag 7 ses det, at loss for træning og validation aftager i en fin takt. Det ses også, at validation når en rigtig høj accuarcy (omkring 97 %). Dog viser det sig, at test-accuarcy ender på 88 %. Det lader altså til, at vores model har overfittet lidt til træningsdataet. På figur 17 (i bilag 7) ses det, at netværket generalt får en høj accuarcy på test-billederne. Dog trækkes den gennemsnittelige accuarcy lidt ned af, at netværket klarer sig skidt på nogle billeder. Vi tjekkede også hvordan netværket klarede sig på pseudo-test dataet. Her fik netværket en accuarcy på 97 %. Med "plotting_results_of_final" filen laves en figur med input, target og prædiktion fra netværket (denne figur ses som figure 18 i bilag 7). Her ses det, at netværket korrekt finder vejen, fortovet, to biler og et menneske. Netværket finder dog ikke det sidste menneske, og har også lidt svært ved at klassifisere køretøjet, som billedet er taget fra.

Diskussion af resultater

Det ses, at validation-accuarcy er 9 procentpoint større end test-accuarcy for det endelige netværk. Dette vurderer vi til, at netværket har overfittet til træningsdataet. Accuarcy på pseudo-test dataet er 97 %. Derfor tror vi, at vores test-data

indeholder billeder fra andre omgivelser / trafik som ikke rigtigt er blevet repræsenteret i trænings/pseudo-test dataet. Fx. kan dette ske hvis trænings- og pseudo-test dataet har mange billeder fra byen, og test-dataet har billeder taget på landet. Dette kunne forklare hvorfor netværket har lav accuarcy på specielt nogle billeder (se evt. bilag 7 - figur 17).

Derudover har vi også den mistanke, at det validation data samt pseudo-test data vi har anvendt, har mindet for meget om trænings-dataet. Dette skyldes, at vi ikke ser nogen betydelige tegn på overfitting i vores test-netværker.

I forhold til hvordan dataet er splittet, valgte vi at anvende "leave one out"-metoden (se slide 6 i powerpoint 11), hvor man gemmer hele sekvenser af billeder i sit test data, og træner/validerer netværket på det resterende data. Dette gjorde vi for, at netværket ikke skulle trænes på data der minder meget om det, der er i test-dataet.

Det kan diskuteres om pseudo-test-dataet er nødvendigt. På den ene side kan man argumentere for, at mere trænings-data kan gøre modellen bedere. Man kan også argumentere for, at et uheldigt valg af pseudo-data kan føre til, at man indstiller hyper-parametrene forkert. På den anden side kan man argumentere for, at det er bedst at have et data-sæt, som man kan teste sine netværker på under hyper-parametre tuning. Her kan man også argumentere for, at man ikke må evaluere sine modeller på det "sande" test-data. Dette skyldes, at man så kan komme til at overfitte sin model til test-dataet. Hvis vi fx. havde brugt vores pseudo-test data, som var det, det rigtige test-data, så vil vi have konkluderet, at præcisionen af vores netværk er 97 %, selvom den i virkeligheden er lavere.

Det kan vurderes hvordan man får den bedste performance, når hukommelse på GPU'en er et problem. En af løsningerne er at gå på kompromis med data-kvaliten, og så gøre billederne mindre - så man mister pixels. Derved vil input og target dataet blive "lettere". Man kan også spare plads, ved at vælge et andet netværk, som ikke anvender skip-connections. I vores setup har vi 4 skip-connections, og for hver connection gemmer vi outputtet på GPU'ens memory. Man kunne også reducere input billedet ved fx. at bruge gray-scale. Dog er vores vurdering, at farven på de forskellige objekter i vores data har betydning (fx. er vejen oftest grå).

Dog kan en løsning også være, at anvende en lille batch_size. På side 8 i [5] anbefales det, at køre den højest mulige batch_size (som hardware kan håndtere) men med en højere learning-rate. Hvis batch_size er lille nok, så kan dette tillade en højere learning rate [5]. Dette kan være grunden til, at vores model kan opnå en god accuarcy selv med en relativ høj learning rate (se evt. bilag 4).

I vores udvikling af det mest optimale netværk, testede vi først syv mindre netværker med forskellige optimizors og schedulers. Herefter udvalgte vi de bedste netværk og arbejdede videre med disse (SGD med CyclicLR eller ReduceLROnPlateau). For disse netværk fandt vi den maximale learning rate og indstillede learning rate optimalt. Det kan diskuteres, om dette finder det optimale netværk + hyper-parametre. På den ene side, kan man argumentere for, at udvælgelses-fasen tillader, at man tester flere netværks-kombinationer, da man ikke skal tilpasse hyper-parametre for alle netværk. På den anden side kan man argumentere for, at man i udvælgelses-fasen kan komme til, at fravælge et optimalt netværk, fordi dens hyper-parametre ikke er indstillet optimalt. En anden tilgang til at finde det optimale netværk kunne være, at indstille hyper-parameterne først, og så derefter kigge på performance af de forskellige test netværker. Dog kan det tage tid, hvis man afprøver forskellige schedulers. Her kan man risikere, at skulle tune flere forskellige hyper-parametre, som passer til de schedulers, man vil afprøve.

Da der kan være mange hyper-parametre, som kan indstilles, kan det være svært, at vælge hvilke der skal korrigeres. Det beskrives i [5] at learning rate anses for at være den vigtigste hyper-parameter at indstille. Det der kan, gøre det svært at indstille hyper-parametrene er, at de oftest afhænger af hinanden. Fx. vil learning raten afhænge af batch_size og momentum (hvis momentum anvendes i optimizoren) og vice versa. Derudover er der også hyper-parametre som fx. antal (og nogle gange størrelsen af) lag i netværket. Da [5] påpeger, at learning rate er den vigtigste hyper-parameter at tune, så valgte vi at have fokus på at tune denne hyper-parameter.

Konklusion

Vi fandt, at den bedste indstilling af vores netværk er med SGD som optimizor (med momentum på 0.9) og CyclicLR som scheduler. Til vores netværk tunede vi hyper-parameteren "learning rate". De optimale værdier vi fandt, var en base learning rate på 0.001 og en maximum learning rate omkring 1.25 (vi valgte 1.15). Til vores optimerede netværk valgte vi step_size_up til 10, da dette vil generere tre udsvingninger, og ende med at give os den laveste learning rate tilsidst, når 61 epoker anvendes. Med denne opsætning af netværket fik vi en test-accuarcy på 88%. Dette er rimelig godt og løser opgaven fint (se evt. figure 18 i bilag 7). Dog tyder det på, at netværket kan være overfittet, da validation-accuarcy var omkring 97 %. Det kan diskuteres, om opgaven kan løses bedre, og her vil man kunne argumentere for, at mere og højere diversitet i træning- samt pseudo-test-dataet muligvis kunne give et bedre og mere generaliseret netværk.

References

[1] (Brox, Fischer & Ronneberger,
U-Net: Convolutional Networks for Biomedical Image Segmentation,
https://arxiv.org/pdf/1505.04597.pdf)

[2] (Pytorch CosineAnnealingLR dokumentation,
https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR.html#torch.optim.lr_scheduler.CosineAnnealingLR)

[3] (Pytorch CyclicLR dokumentation,
https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CyclicLR.html)

[4] (Pytorch ReduceLROnPlateau dokumentation,
https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html#torch.optim.lr_scheduler.ReduceLROnPlateau)

[5] (Smith, Leslie N.,
A DISCIPLINED APPROACH TO NEURAL NETWORK HYPER-PARAMETERS: PART 1 – LEARNING RATE, BATCH SIZE, MOMENTUM, AND WEIGHT DECAY,
https://arxiv.org/pdf/1803.09820.pdf)

[6] (Smith, Leslie N.,
Cyclical Learning Rates for Training Neural Networks,
https://arxiv.org/pdf/1506.01186.pdf)

[7] (Stefan Roth, Stephan R. Richter, Vibhav Vineet and Vladlen Koltun,
Playing for Data: Ground Truth from Computer Games,
https://download.visinf.tu-darmstadt.de/data/from_games/)

Link til Github repository: **https://github.com/marctimjen/Deep_final**

Bilags oversigt:

Bilag 1: Input og target billede

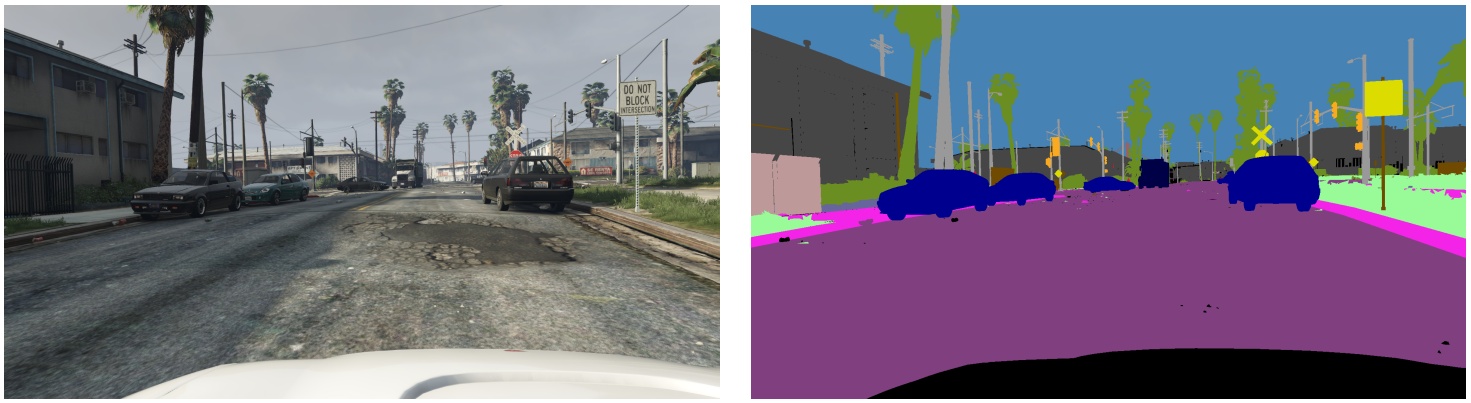


Figure 3: Input billede (venstre) og segmenteret target (højre)

Bilag 2: Learning rate for de syv netværk

Figure 4: De forskellige learning rate for de syv netværker

Bilag 3: Træning og validation loss for de syv netværk

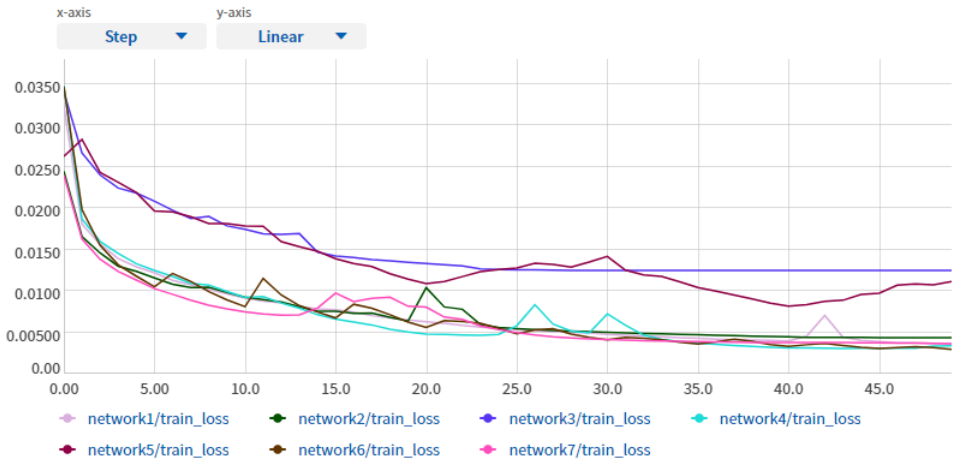


Figure 5: Train loss for de syv test netværker

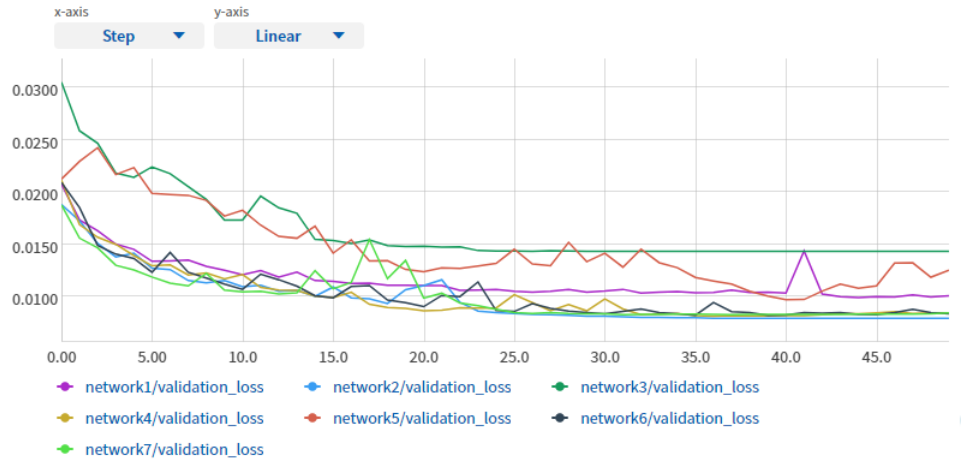


Figure 6: Validation loss for de syv test netværker

Bilag 4: Find maximum og base learning rate

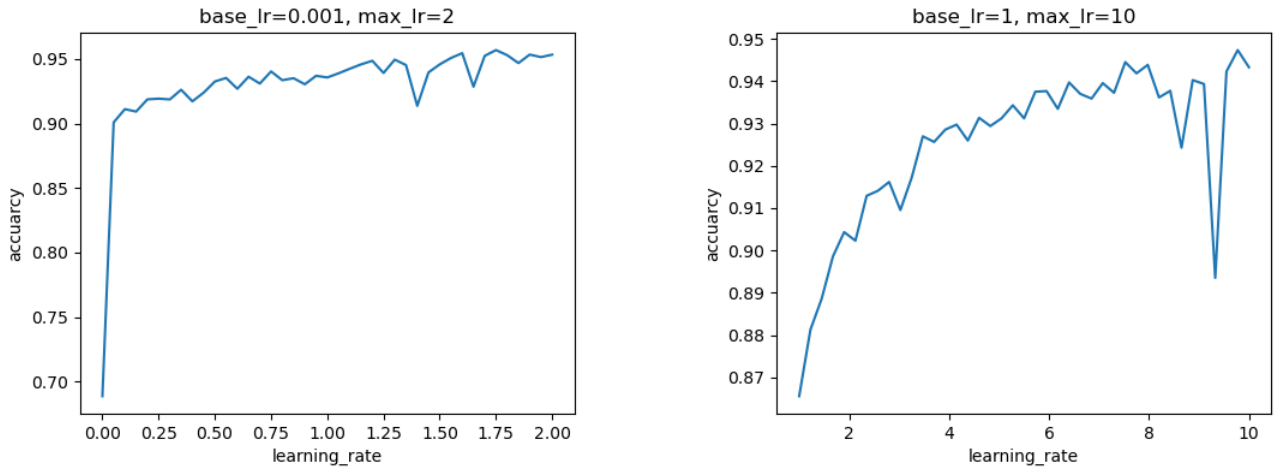


Figure 7: Figur over resultat af vores tuning af learning rate

Bilag 5: Netværk_cyclic

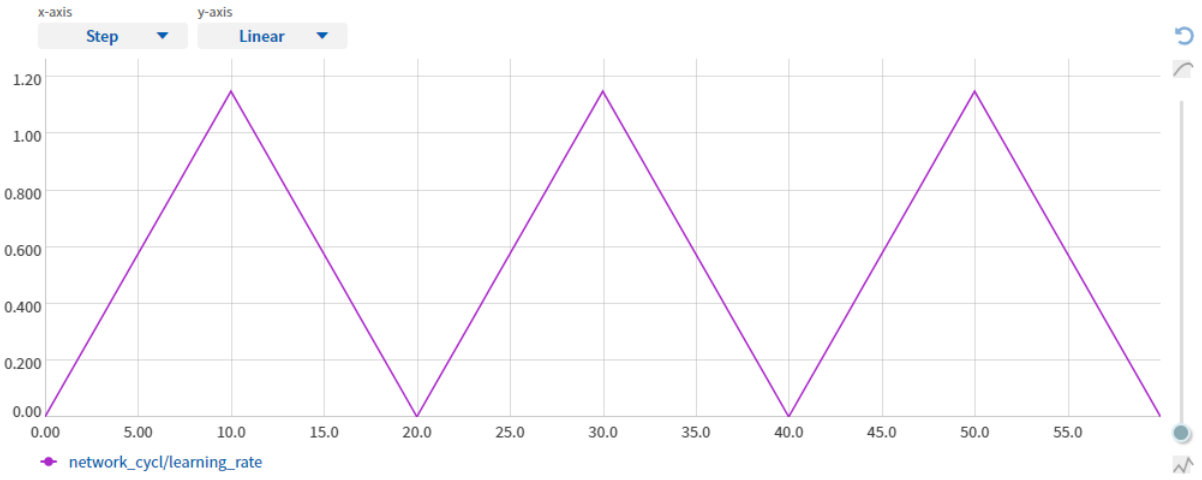


Figure 8: Learning rate gennem epokerne for netværk_cyclic

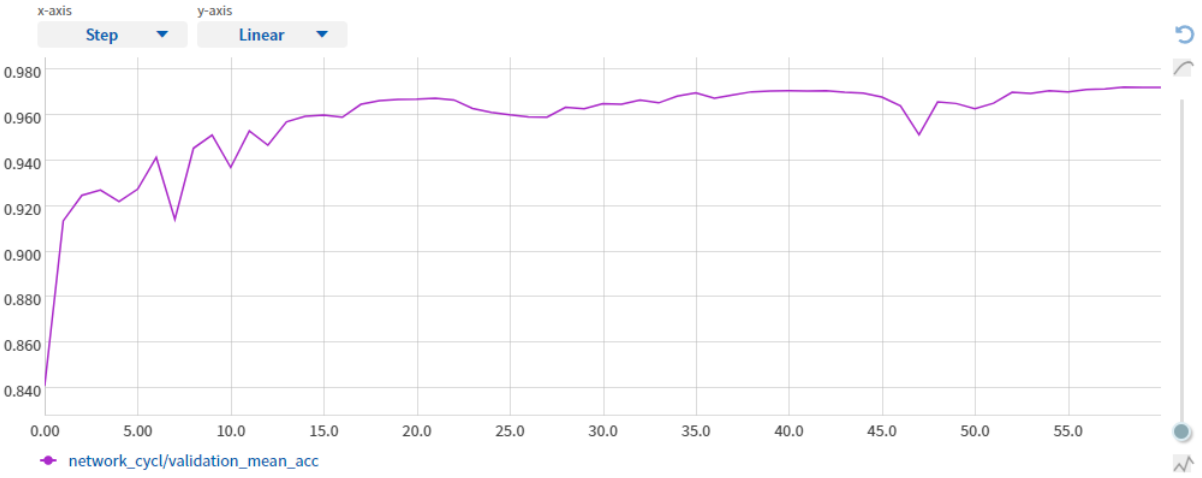


Figure 9: Den gennemsnittelige pixel accuarcy for netværk_cyclic (på validation dataet)

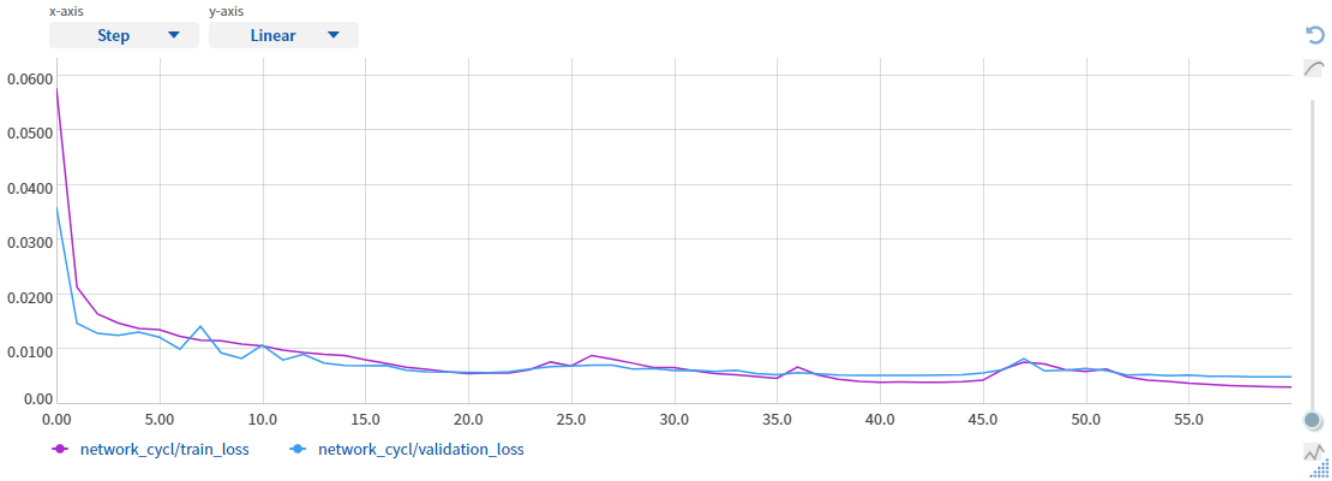


Figure 10: Validation og træning loss for netværk_cyclic

Bilag 6: Netværk_reduce

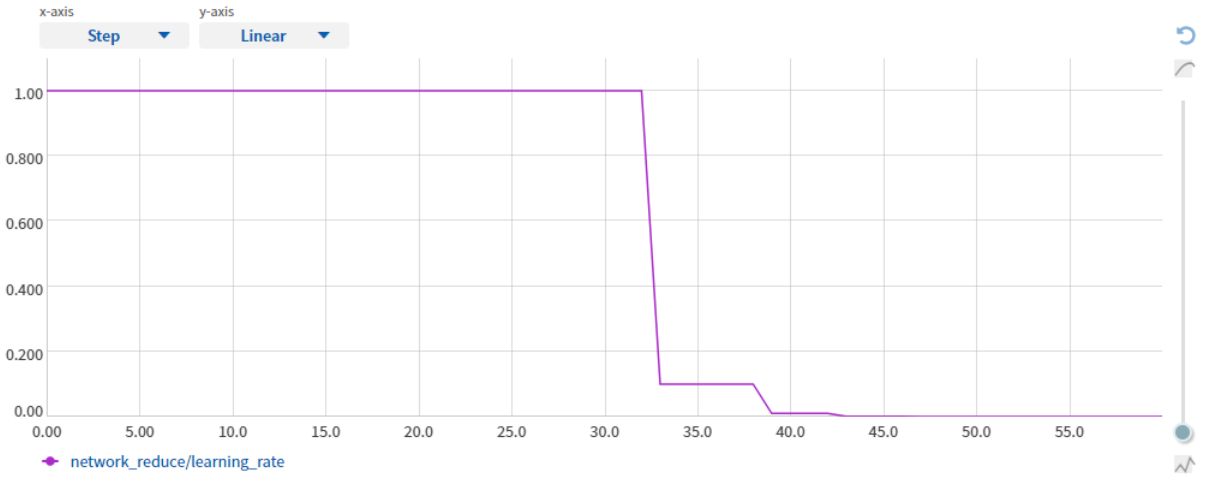


Figure 11: Learning rate gennem epokerne for netværk_reduce

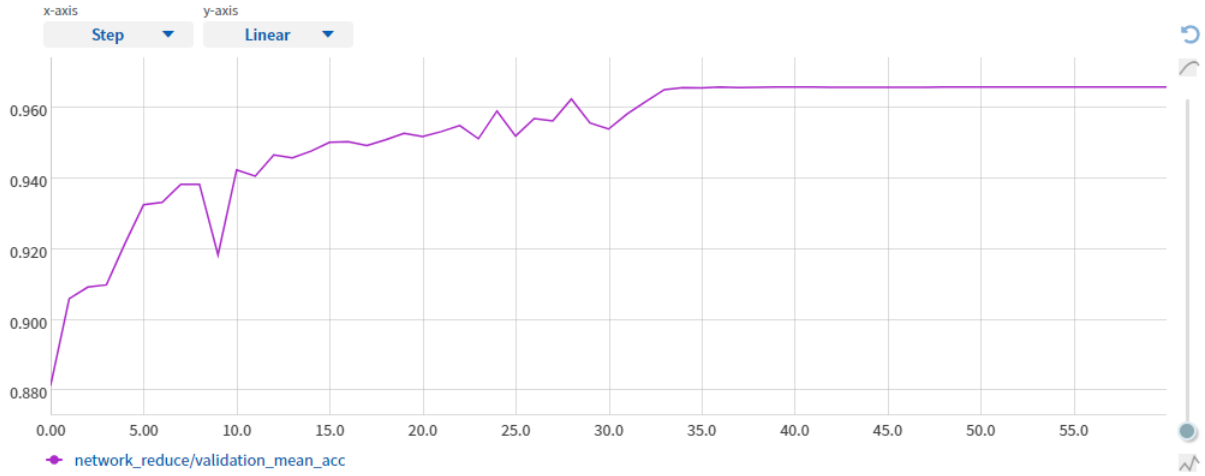


Figure 12: Den gennemsnittelige pixel accuarcy for netværk_reduce (på validation dataet)

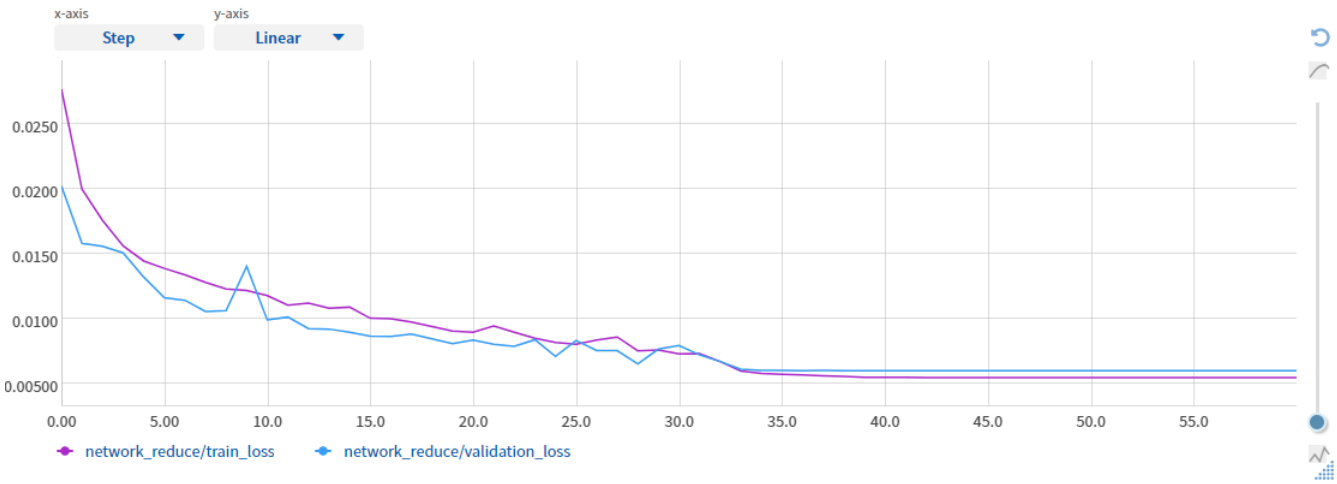


Figure 13: Validation og træning loss for netværk_reduce

Bilag 7: Netværk_final

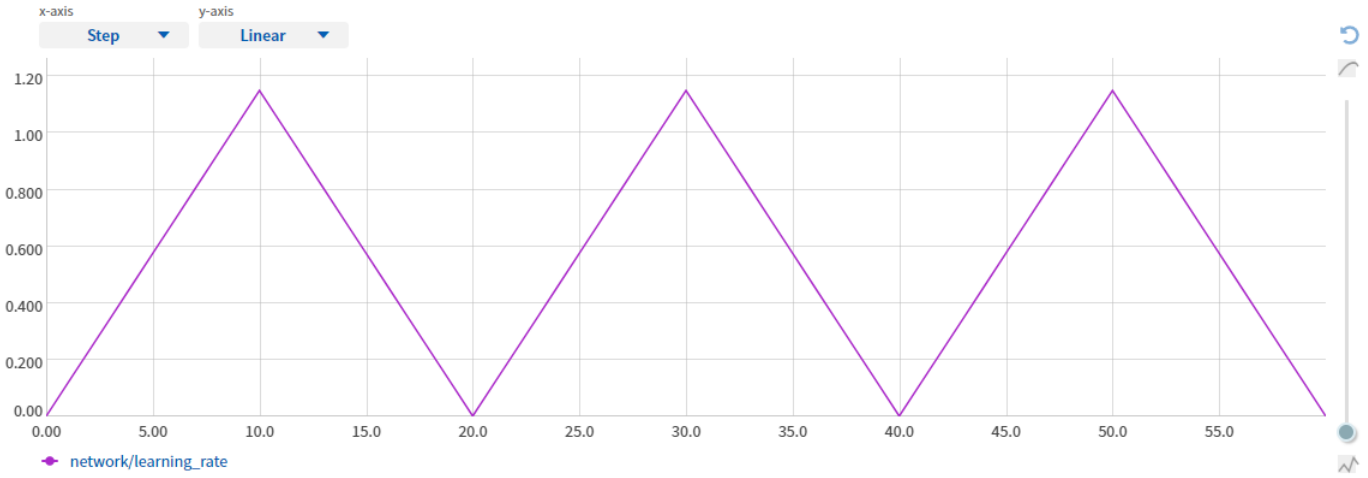


Figure 14: Learning rate gennem epokerne for netværk_final

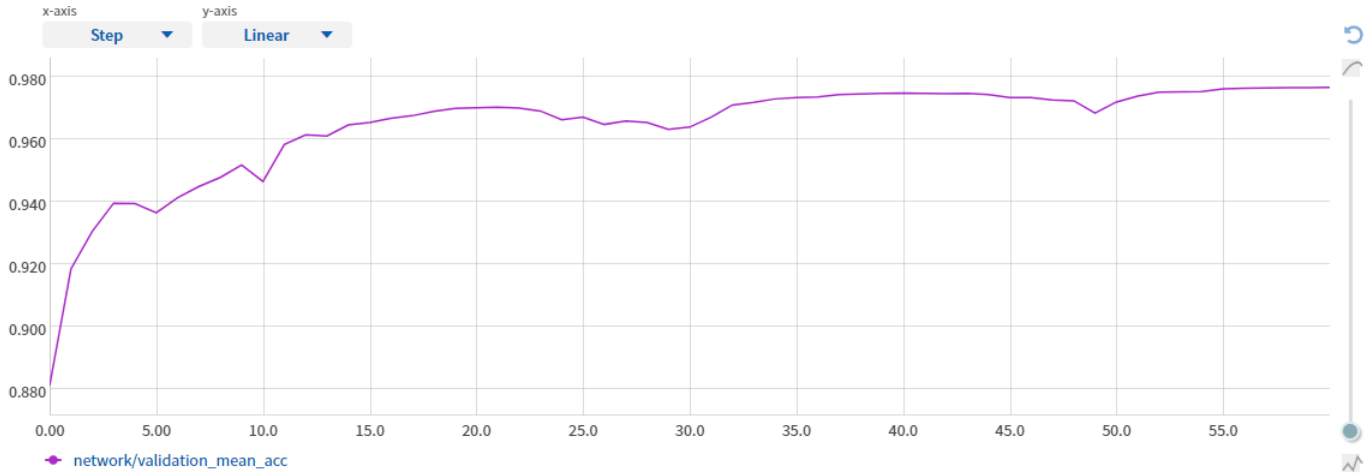


Figure 15: Den gennemsnittelige pixel accuarcy for netværk_final (på validation dataet)

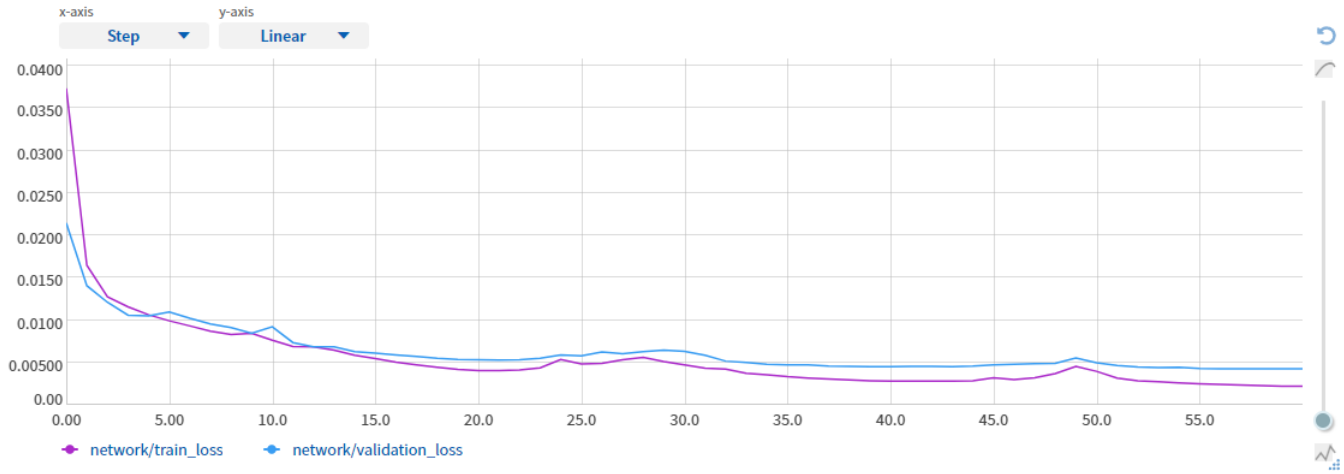


Figure 16: Validation og træning loss for netværk_final

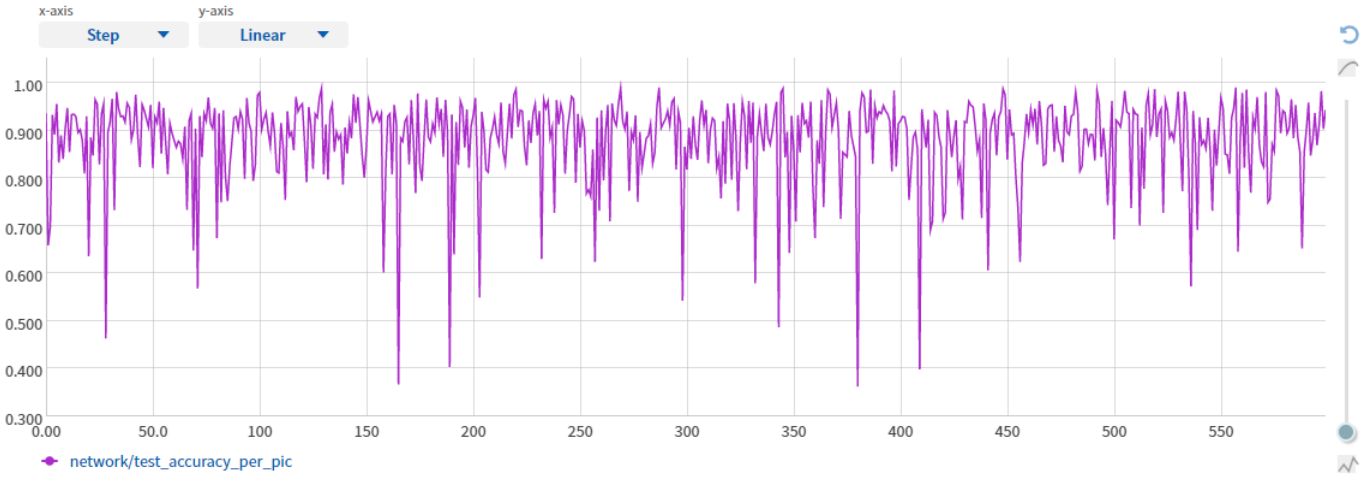


Figure 17: Pixel accuarcy for test-billederne

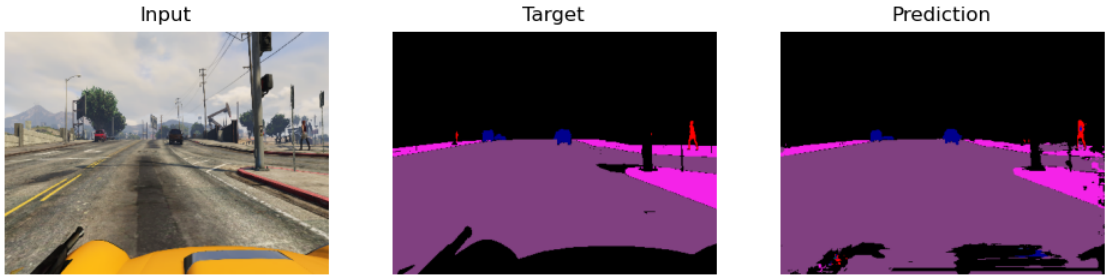


Figure 18: Input, target og en prædiktion fra netværk_final