

Todo list

Einführung mit Inhalt füllen	2
Hier vielleicht unsere Probleme an Anfang erwähnen? Frameworks liefen nicht auf Windows, wir haben Linux installiert und uns dann später in Palma und das Modul-/Bacthsystem reingefuchst	2
NAS-Unet mit Inhalt füllen	3
Deeplab mit Inhalt füllen	11
Durchschnitt ermitteln Pascal VOC 2012	18
Predictions fuer Task 100 CT alle Versionen auswerten	18
Stimmt das so mit dem CT Datensatz?	31
Scatterplot Haeufigkeiten CT	32
<i>schlecht</i> gegen etwas angemessenes austauschen	32
<i>einigermaßen akzeptable</i> ersetzen	32
Verweis auf Visualisierung des CT Datensatzes, Predictions raussuchen und veranschaulichen	32
Verschiedene Versionen diskutieren? 3dcascade, 3dfullres (1000 + 2000) und 2d	32
Gewichtung für Average als Formel darstellen	32
Ergebnisse ausformulieren, Bilder einfügen	32
anders formulieren, Ergebnisse sind halbwegs ok	33
Ergebnisse einfügen, diskutieren wie 3d_fullres und 2d sich unterscheiden, Ensemble auch	33
Guter Support: Github Issue Antwortzeit nur wenige Tage, schnelle bugfixes	33
Die anderen beiden Frameworks kritisieren, dieses ist deutlich besser / das einzige was überhaupt funktioniert...	33

Gesamt-Fazit mit Inhalt füllen	34
Anhang mit Inhalt füllen	35
Literaturverzeichnis ergänzen	35



AutoML für Segmentierung

PROJEKTSEMINAR
zum Thema
AUTOML

Westfälische Wilhelms-Universität Münster
Institut für Informatik

Eingereicht von:
Milan Blunk (418650)
Pia Nümann (454700)
Matthias Wolff (458766)

Münster, März 2021

Inhaltsverzeichnis

1. Einführung	2
2. NAS-Unet	3
2.1. Funktionsweise / Theorie	3
2.2. Unsere Arbeit / Praxis	7
2.3. Ergebnisse	7
2.4. Fazit	9
3. Auto-DeepLab	11
3.1. Funktionsweise / Theorie	11
3.2. Unsere Arbeit / Praxis	14
3.3. Fazit	14
4. nnUNet	15
4.1. Funktionsweise / Theorie	15
4.2. Unsere Arbeit / Praxis	18
4.2.1. Datensätze aus dem Paper	21
4.2.2. Larven-Datensatz	21
4.2.3. Retina 2D-Datensatz	26
4.2.4. CT-Datensatz	31
4.2.5. Pascal VOC2012	32
4.2.6. Retina 3D-Datensatz	33
4.3. Fazit	33
5. Gesamtfazit	34
A. Ein Anhangskapitel	35

1 | Einführung

Einführung
mit Inhalt
füllen

Hier
vielleicht
unsere
Probleme
an
Anfang
erwäh-
nen?
Frame-
works
liefen
nicht auf
Windows,
wir haben
Linux
installiert
und uns
dann
später in
Palma
und das
Modul-
/Bacthsystem
reinge-
fuchst

2 | NAS-Unet

NAS-Unet
mit Inhalt
füllen

2.1. Funktionsweise / Theorie

NAS-Unet ist einer der ersten Versuche NAS auf medizinische Bildsegmentierung anzuwenden. Es sollten MRT-, CT- und Ultraschallbilder segmentiert werden. Die Architektur von NAS-Unet wurde auf Pascal VOC2012 **PascalVOCDatensatz** gesucht und diese dann auf den unterschiedlichen medizinischen Datensätzen trainiert. Als medizinische Datensätze werden für die MRT-Bilder der Promise12 Datensatz **Promise12Datensatz**, für die CT-Bilder der Chaos Datensatz **ChaosDatensatz** und für die Ultraschallbilder der NERVE Datensatz **NerveDatensatz** verwendet.

Das vorrangige Ziel von NAS-Unet ist das automatische Finden einer geeigneten Zwei-Zell Architektur. Dabei wird parallel nach der Up-Sampling und nach der Down-Sampling Schicht gesucht, die beiden Schichten werden gleichzeitig upgedated. Dabei wird also immer eine Up-Sampling Zelle gleichzeitig mit der ihr gegenüberliegenden Down-Sampling Zelle aktualisiert. Die Architektur von NAS-Unet ist streng symmetrisch und es gibt keine zusätzliche Convolutionschicht in der Mitte (siehe Abbildung 2.1).

2. NAS-UNET

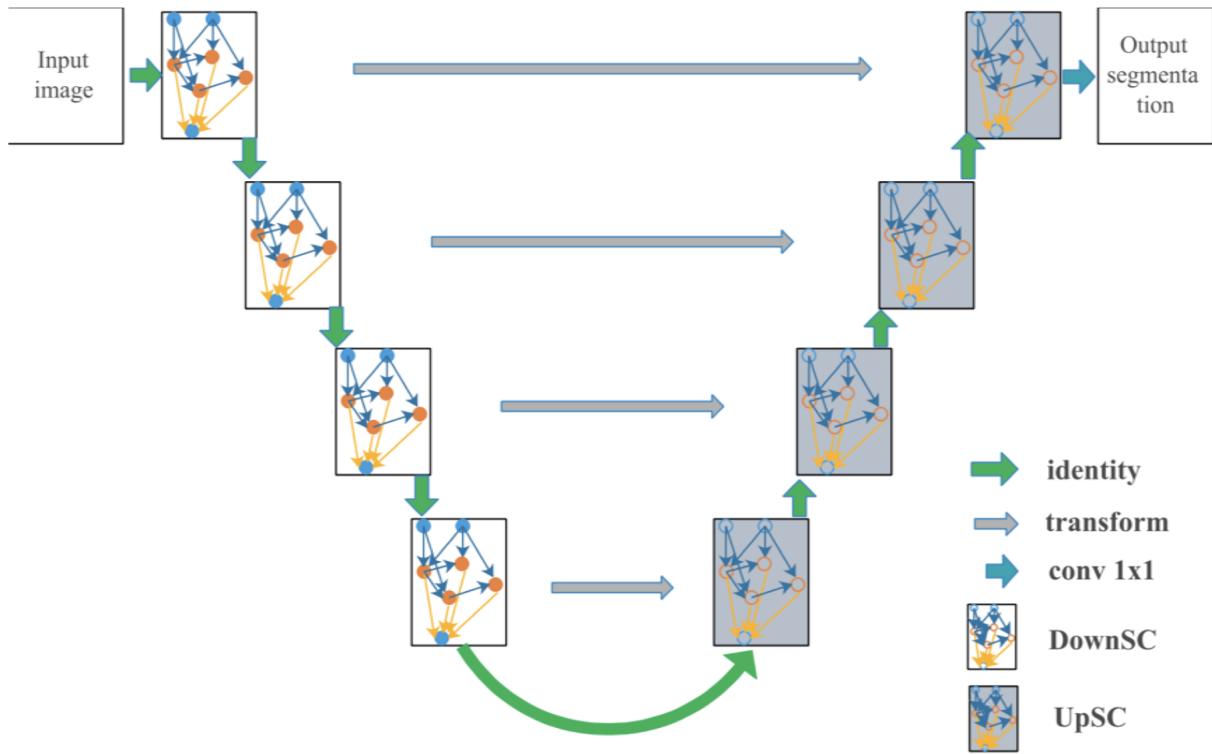
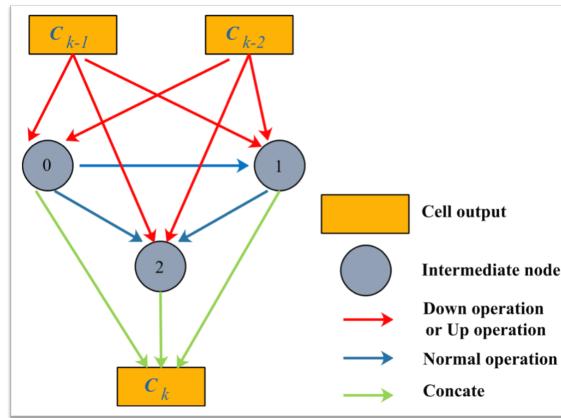


Abbildung 2.1.: Zellbasierte Netzarchitektur von NAS-Unet [nasunetPaper](#)

Der Suchraum, in dem die Architektur gesucht werden soll, enthält die möglichen Architekturen die prinzipiell verwendet werden können, sowie eine Auswahl von primitiven Operationen. Die möglichen Architekturen sind populäre Unet-Architekturen, von denen nur die mittlere Convolutionschicht entfernt wurde. NAS-Unet verwendet einen zell-basierten Architektursuchraum. Die zell-basierte Architektur (siehe Abbildung 2.1) soll die Generierungsmethode beschränken und so das Problem lösen, dass der Suchraum zu groß wird. Nachdem die beste Zellarchitektur (siehe Abbildung 2.2) gefunden wurde, wird sie im ganzen Netzwerk genutzt und im Rückrad des Netzes gestapelt. Dabei sind nicht nur die Convolutionschichten in die Zellen verlegt, sondern auch alle Up- und Down-sampling Operationen. Die Inputknoten einer Schicht sind definiert als die Outputknoten der vorherigen zwei Schichten (siehe Abbildung 2.2). Bei der Auswahl der primitiven Operatoren wurde zum einen darauf geachtet Redundanz zu vermeiden und zum anderen darauf, möglichst wenige Parameter zu haben, um möglichst wenig Memory zu verbrauchen.


 Abbildung 2.2.: Zellarchitektur von NAS-Unet **nasunetPaper**

Die Suchstrategie teilt sich in mehrere Schritte auf. Zunächst wird ein überparametrisiertes Netzwerk erstellt (Siehe Abbildung 2.3).

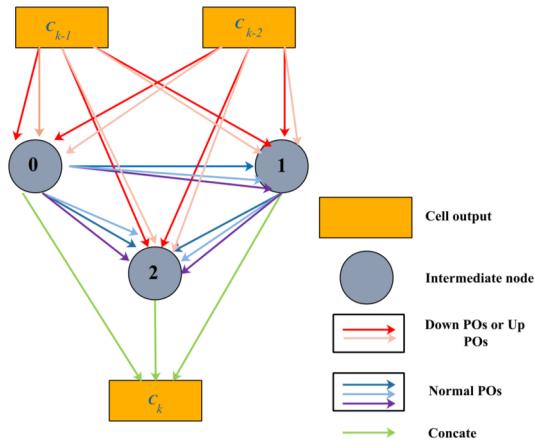


FIGURE 4. An example of Over-Parameterized Cell Architecture. each edge associate with N candidate operations from different primitive operation sets.

 Abbildung 2.3.: überparametrisierte Zellarchitektur von NAS-Unet **nasunetPaper**

In diesem überparametrisierten Netzwerk lässt sich der Output einer Kante aus der Kombinationen der unterschiedlichen primitiven Operatoren folgendermaßen als Formel darstellen:

$$MixO(x) = \sum_{i=1}^N w_i o_i(x)$$

Dabei ist $o(x)$ die Primitive Operation, w ist das zu der Operation gehörende Gewicht und N ist die Anzahl an primitiven Operationen.

Um die Parameter zu aktualisieren, wird eine effizienter Parameter-Update Strategie verwendet, damit GPU-Memory gespart wird. Da die Output-Feature-Maps nur berechnet werden können, wenn alle Operationen gleichzeitig im GPU-Memory sind, wird das N-fache an GPU-Memory benötigt, als wenn man ein kompaktes Modell trainieren würde. Daher wird hier ein binärer Ansatz verwendet, das heißt anstatt bei jedem Schritt alle Architekturparameter mit dem Gradientenabstiegsverfahren zu aktualisieren, wird immer nur ein Parameter aktualisiert (siehe Abbildung 2.4). Dadurch werden aber mehr Iterationen des Updatens benötigt.

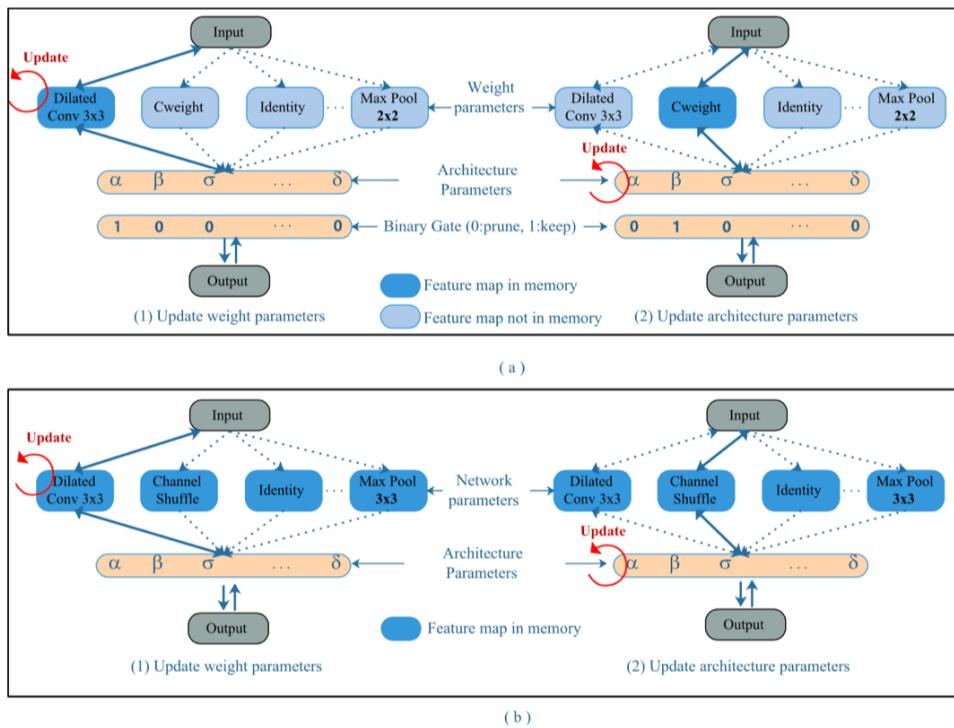


Abbildung 2.4.: Vergleich der binären Suchstrategie von NAS-Unet mit dem gleichzeitigen Updaten aller Parameter **nasunetPaper**

Zu den Implementierungsdetails gehört, dass es immer jeweils 4 Up- und Downsampling Zellen gibt. Die Bilder werden zufällig zur einen Hälfte in Trainingsbilder und zur anderen Hälfte in Testbilder eingeteilt. Das Paper konzentriert sich vorrangig darauf einen effizienten Suchraum zu konstruieren. Die Suchstrategie ist für die Autoren weniger wichtig, da laut ihrer Aussage (Paper, II. B. **nasunetPaper** und Page 44253, Satz2, **nasunetPaper**) jede differentielle Suchstrategie auf dem Suchraum funktionieren würde. In diesem konkreten Fall wird die DARTS-Update Strategie verwendet.

2.2. Unsere Arbeit / Praxis

Bei der praktischen Arbeit mit NAS-Unet sind wir auf verschiedene Schwierigkeiten und Hinderisse gestoßen. Diese belaufen sich vorrangig auf die Problematik, dass es keine Anleitung oder Einführung für Nas-Unet gibt und auch nahezu keine Dokumentation vorliegt. Zunächst einmal haben wir versucht NAS-Unet auf verschiedenen Datensätzen zum Laufen zu bekommen. Dies waren der Datensatz Pascal VOC, der Chaos Datensatz und der Promise Datensatz, welche alle auch im Paper verwendet werden. Bereits hier hatten wie einige Schwierigkeiten, die wir zum Teil auch nicht überwinden konnten.

Als erstes haben wir durch Fehlermeldungen und Suchen im Code herausgefunden, dass NAS-Unet den Datensatz in einem ganz bestimmten Ordner an einem ganz bestimmten Pfad erwartet. Da wir den Pfad auf Palma nicht einrichten konnten, da dieser schon im Wurzelverzeichnis beginnt, mussten wir die Stelle im Code entsprechend anpassen. Die Tatsache, dass der Code fast gar nicht kommentiert wurde, hat uns die Suche und Anpassung erheblich erschwert. Auch die Ordnernamen und die Struktur des Datensatzes mussten angepasst werden. Auch hierzu gab es keinerlei Hinweise oder Dokumentation.

Auf dem Chaos Datensatz, welcher auch im Paper verwendet wird, hatten wir das Problem, dass das Framework im Datensatz nach Bildern sucht, die in keiner öffentlich verfügbaren Version des Datensatzes **ChaosDatensatz** existieren. Wie das NAS-Unet auf diese Bilder kommt oder wie man verhindert, dass es nach diesen sucht, ist uns unklar geblieben.

Während wir den Promise Datensatz nicht öffentlich zugänglich finden konnten, ließ sich das Framework auf dem Pascal VOC Datensatz erfolgreich ausführen.

2.3. Ergebnisse

Erfolgreich zum Laufen bringen konnten wir das Netz lediglich auf dem Datensatz Pascal VOC2012. Unsere erzielten Ergebnisse waren jedoch leider sehr schlecht. Unsere mIoU auf dem Pascal Datensatz auf den Testbildern war <0.05 (siehe Abbildung 2.5).

2. NAS-UNET

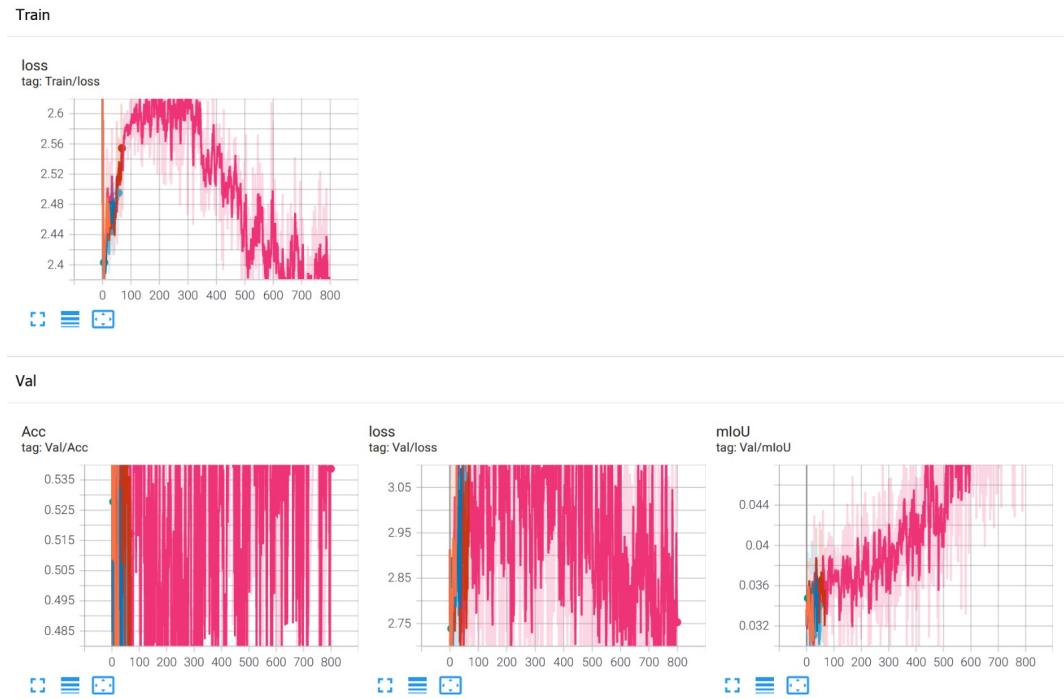


Abbildung 2.5.: überparametrisierte Zellarchitektur von NAS-Unet [nasunetPaper](#)

Wie man sieht, ist der Graphenverlauf sehr schwankend und ergibt nur stark gesmoothed eine sichtbare Tendenz. Die Werte sind schlecht bis sehr schlecht, und verbessern sich auch nur sehr langsam.

Auf dem Chaos-Datensatz haben wir NAS-Unet nicht zum Laufen bekommen, da NAS-Unet nach einem Bild sucht, welches in keiner der öffentlichen Versionen des Datensatzes (**ChaosDatensatz** unter Download) existiert. Da es uns nicht gelungen ist, dieses Problem zu lösen, haben wir das Framework leider nicht auf dem Chaos-Datensatz zum Laufen bekommen und können daher auch keine Ergebnisse vorstellen. Uns ist auch nach langer Fehlersuche nicht klar geworden, warum das Framework überhaupt nach einem Bild sucht, welches es eigentlich nie eingegeben bekommen hat.

Da wir den Promise Datensatz nicht öffentlich finden konnten, können wir hier leider auch über keine Ergebnisse berichten.

2.4. Fazit

Durch die fehlende Dokumentation des Codes und die fehlende Anleitung zur Nutzung des Frameworks ist es extrem schwierig und zeitaufwändig das Framework zu nutzen. Es war uns leider auch nicht möglich, den Code vollständig nachzuvollziehen. Wir konnten daher leider nicht nachvollziehen, was genau gemacht wurde. Auch im Internet, zum Beispiel auf Github, konnten wir leider niemanden finden, der den Code nachvollziehen konnte. Man findet auch hier leider nur viele Fragen. Ein Beispiel ist der folgende Versuch den Code nachzuvollziehen: Fraunhofer siehe Abbildung 2.6 Hier sieht man, dass sich auch die Frage der verschiedenen Metriken auftut. Auch darauf haben wir keine Antwort finden können.

```

... ...
@@ -23,6 +23,12 @@ from models import get_segmentation_model
import models.geno_searched as geno_types
from tensorboardX import SummaryWriter

+ '''Questions about the implementation.
+ 1. Why authors asked to run this file to train the network and search the architecture, when nowhere we
+ are using searching?
+ 2. what is the dimension of the nasunet output? whats aux_pred in preds vector?
+ 3. self.test_queue is not even defined anywhere but used in test function
+ 4. Just wondering why are we using two metrics, pixacc andd mIoU?''
+

```

Abbildung 2.6.: Beispielversuch den Code nachzuvollziehen **Frauenhofer_NasUnet**

Ein weiteres Problem ist das Verhalten von NAS-Unet auf dem Chaos Datensatz. Die Frage, warum es nach einem Bild sucht, welches im Datensatz nicht vorkommt, bleibt offen und damit auch die Frage, wie man NAS-Unet auf diesem Datensatz zum Laufen bekommen könnte. Auffällig ist dies vor allem daher, dass NAS-Unet im Paper auch angeblich auf dem Chaos Datensatz angewendet wird. Hier ist wiederum auffällig, dass das Paper zu NAS-Unet am 04.04.2019 veröffentlicht wurde, während der Datensatz erst am 11.04.2019 veröffentlicht wurde. Möglicherweise hatte die Autoren eine leicht andere Vorversion. Trotzdem erklärt dies nicht, warum das Netz nach mehr Bildern sucht, als ihm eingegeben werden.

Zu den oben genannten Problemen bei der Arbeit mit dem Framework kommt hinzu, dass unsere Ergebnisse auf dem Datensatz Pascal VOC2012 sehr schlecht waren. Besonders auffällig ist dies auf dem Datensatz von Pascal VOC2012, da dieser auch im Paper genutzt wurde und es darauf spezialisiert ist. Leider wurden unserer Recherche nach nie fertig trainierte Modelle von NAS-Unet, welche Ergebnisse wie im Paper angegeben erzielen, veröffentlicht. Daher war es uns leider weder möglich die Ergebnisse zu reproduzieren noch sie nachzuvollziehen.

Im folgenden Github Issue: [Issue11 nasunetGithubIssue11](#) haben wir herausgefunden, dass man die trainierte Netzstruktur anscheinend händisch in den Code zum Trainieren hineinkopieren muss.

Per default verwendet NAS-Unet aber die auf Pascal VOC2012 ausgesuchte Netzstruktur. Daher sollten unsere Ergebnisse auf Pascal VOC 2012 eigentlich davon nicht negativ beeinflusst werden.

Auch beim Durchsuchen der Github Issues auf der zugehörigen Github Seite **nasunetGithub** sind wir mehrfach darauf gestoßen, dass die Ergebnisse, die im Paper angegeben wurden, nicht reproduziert werden konnten(zum Beispiel Issue 31 **nasunetGithubIssue31**). Da wir auch nicht genau nachvollziehen können, wie diese Ergebnisse entstehen, haben wir uns, auch auf Grund der oben genannten Probleme, dazu entschlossen, nicht länger mit diesem Framework zu arbeiten. Anstelle von NAS-Unet haben wir uns selbstständig ein neues Framework rausgesucht und mit ihm weitergearbeitet (siehe nnU-Net, Kapitel 4).

3 | Auto-DeepLab

Deeplab
mit Inhalt
füllen

3.1. Funktionsweise / Theorie

Auto-DeepLab ist ein NAS (Neural Architecture Search) Programm, welches im Jahr 2019 zur Segmentierung von Bildern entwickelt wurde. Auto-DeepLab verfolgt den Ansatz, die Netzstruktur sowie die Zellstruktur des Convolutional Netzes zu suchen. Die Architektur kann mittels Gradient Descent in 3 GPU Tagen gesucht werden. Anders als bei nicht-differenzierbaren Suchtechniken arbeitet Auto-DeepLab somit sehr effektiv: Es wird nicht in einer diskreten Menge von Kandidaten nach der besten architektur gesucht. Stattdessen wird mit Hilfe von Gradient Descent (also mittels Differenzierbarkeit, also stetig) gearbeitet. Die Optimierung geschieht in Hinblick auf die Validierungsperformance. Diese Effektivität ist auch notwendig, da Bildsegmentierung auf hochauflösenden Bildern funktionieren muss und auch das segmentierte Ausgabebild im besten Fall die gleiche Auflösung wie das zu segmentierende Eingabebild haben sollte.

Zum Zeitpunkt der Veröffentlichung von Auto-DeepLab war der Stand der Entwicklung der NAS Programme so, dass oft nur das Innere der Zellen automatisiert gesucht wurde und dann anhand dieses Resultates das Netz bestimmt wurde. Hier unterscheidet sich Auto-DeepLab, da sowohl Netz als auch die Zellstruktur automatisiert ermittelt werden. Im Folgenden wird die Suche im hierarchischen Suchraum beschrieben. Wir beginnen mit dem Suchraum bezüglich des Zellinneren.

Zunächst einmal definieren wir, was eine Zelle ist:

Eine Zelle ist ein gerichteter, azyklischer Graph, der aus einer geordneten Sequenz aus n Knoten besteht. Jeder Knoten ist hierbei ein sogenannter Block. Mehrere Zellen miteinander verkettet bilden dann das gesamte neuronale Netz.

Die Blöcke in den Zellen sind Strukturen, die zwei Tensoren als Input entgegennehmen und einen Output-Tensor ausgeben. Block i in Zelle l kann mit einem 5-Tupel charakterisiert werden: (I_1, I_2, O_1, O_2, C) mit $I_1, I_2 \in \mathcal{I}_i^l$, wobei \mathcal{I}_i^l die Menge aller möglichen Input-Tensoren darstellt: \mathcal{I}_i^l besteht aus dem Output der vorherigen Zelle, H^{l-1} , dem Output der Zelle zwei Zellen vor der aktuellen Zelle l , H^{l-2} , und jeweils dem Output der Blöcke der aktuellen Zelle, die sich in dem gerichteten, azyklischen Graphen vor dem aktuellen Block i befinden: H_1^l, \dots, H_{i-1}^l . Dadurch haben die Blöcke einer Zelle, die im gerichteten Graphen weiter hinten sind, mehr mögliche Inputs.

Zurück zu dem charakterisierenden 5-Tupel bilden O_1 und O_2 die Layer-Typen jeweils für die beiden Inputs I_1 und I_2 mit $O_1, O_2 \in \mathcal{O}$. Die Menge \mathcal{O} besteht wiederum aus 8 möglichen Operatoren: 3 x 3 depthwise-separable conv, 5 x 5 depthwise-separable conv, 3 x 3 atrous conv with rate 2, 5 x 5 atrous conv with rate 2, 3 x 3 average pooling, 3 x 3 max pooling, skip connection, no connection (zero). Das C stellt den Operator dar, mit dem die beiden auf die Layers angewandten Inputs zu einem Output gemacht werden: $O_1(I_1)$ und $O_2(I_2)$ werden immer einfach elementweise addiert. Damit könnte man die Charakterisierung eigentlich auch auf ein 4-Tupel bestehend aus I_1, I_2, O_1, O_2 beschränken.

Der Outpur Tensor der Zelle i , H^l , ist einfach die Konkatenation von H_1^l, \dots, H_n^l . D.h. die Outputs der Blöcke der Zelle i werden konkateniert in der Reihenfolge des Auftreten im Graphen.

Nachdem die Struktur der Zellen nun erklärt ist, fahren wir fort mit der Beschreibung, wie sich die Architektur des Netzes zusammenstellt. Dies ist am besten graphisch mit Abbildung 3.1 möglich. Links in Abbildung 3.1 ist ein Gitter dargestellt. Die blauen Punkte des Gitters repräsentieren die Zellen. Die Zweierpotenzen links stehen für die Anzahl an Downsamplings. D.h. eine hohe Zahl (z.B. die 32) steht für eine starke Reduzierung der Bildauflösung. Die Zahlen oben stehen für die Anzahl an Schichten im Netz. Bei Auto-DeepLab gilt $L = 12$, d.h. nach den initialen Zellen zum Anfang des Netzes (grau gefärbt) gibt es 12 weitere Zellen im Netz. Ein Pfad durch das Gitter stellt nun eine mögliche Architektur dar. Von allen Möglichkeiten sucht Auto-DeepLab den Pfad, also die Architektur, die auf den Validierungsdaten am besten performt. Als Verlustfunktion verwendet Auto-DeepLab Cross-Entropy. Ein nächster Schritt in dem Gitter ist entweder immer ein Schritt schräg nach oben (die Auflösung verdoppelt sich), ein Schritt waagerecht (gleiche Auflösung) oder ein Schritt schräg nach unten (die Auflösung halbiert sich). Mit jeder Halbierung der Auflösung verdoppelt sich die Anzahl an Feature Maps.

Die drei Schritte (schräg nach oben, waagerecht, schräg nach unten), die durch die hellgrauen Pfeile dargestellt sind, können als Übergangswahrscheinlichkeiten von einem Zustand (einer Zelle) in den nächsten (nächste Zelle) interpretiert werden. Die drei Skalare sind somit alle nicht-negativ und bilden in der Summe einen Wert von 1. Als beste Architektur wird der Pfad bestimmt, der die Übergangswahrscheinlichkeiten maximiert.

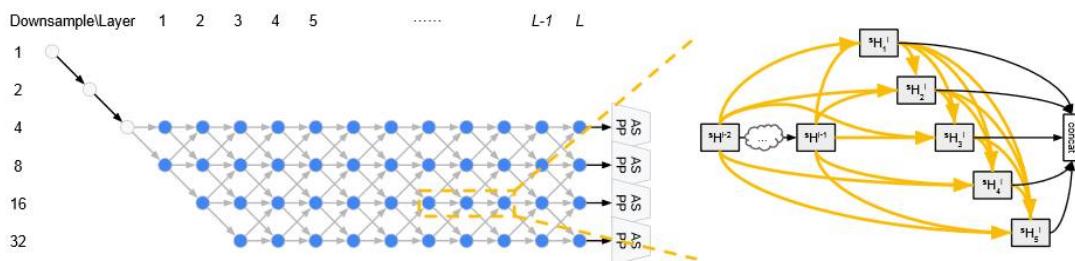


Abbildung 3.1.: Auto-DeepLab Architektur. [autodeeplabPaper](#)

Rechts in Abbildung 3.1 sind insgesamt drei (blaue) Zellen dargestellt: Zelle $l - 2$ (${}^s H^{l-2}$), Zelle $l - 1$ (${}^s H^{l-1}$) und Zelle l (${}^s H^l$). Zelle l ist anders dargestellt als $l - 1$ und $l - 2$: Hier wird das Innere der Zelle, also die Blöcke, dargestellt. Bei Auto-DeepLab besteht jede Zelle aus genau 5 Blöcken. Wie bereits oben beschrieben, bestehen die möglichen Inputs eines Blocks aus den Outputs der zwei Vorgängerzellen und den Outputs der Vorgängerblöcke in der Zelle l . Das hochgestellte s steht für die Stufe an Downsamplings, also $s \in \{4, 8, 16, 32\}$ (es wird bei 4 angefangen, da initial bereits zwei Downsamplings durchgeführt werden).

Zwei beispielhafte, gefundene Architekturen sind in Abbildung 3.2 dargestellt.

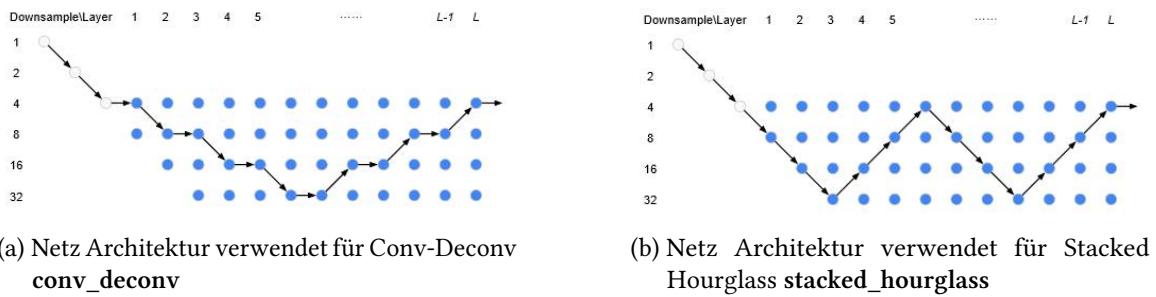


Abbildung 3.2.: Architekturen ermittelt durch Auto-DeepLab für zwei verschiedene Bilddatensätze.

In Abbildung 3.2a sieht man eine fast U-förmige Architektur. Diese Form werden wir spezifischer und - anders als hier - bewusst forcieren in Kapitel 4 zum nnU-Net genauer betrachten. Beide hier dargestellten Architekturen enden auf Stufe 4 der Downsamplings (was allerdings keineswegs immer der Fall ist). Damit die Auflösung des Eingabebildes zurückgewonnen werden kann, folgen auf jeder Downsampling Stufe nach der L-ten Schicht die sogenannten Atrous Spatial Pyramid Pooling Module (in Abbildung 3.1 als ASPP abgekürzt). Mit Hilfe dieser Module wird durch Upsampling die ursprüngliche Auflösung wieder gewonnen.

Für die Suche nach der perfekten Architektur nutzt Auto-DeepLab 40 Epochen. Die Batch-Größe beträgt 2. Wie bereits erwähnt, wird mit Gradient Descent gearbeitet. Genauer wird Stochastic Gradient Descent angewandt, um schneller Ergebnisse erzielen zu können. Gestartet wird mit einer Lernrate von 0.025, diese reduziert sich allerdings im Verlauf der Suche immer weiter bis schließlich auf einen Wert von 0.001.

Auffällig ist, dass in den ersten 75% der Layer (also in den ersten 9 Layer) eher Downsampling dominiert und in den letzten 25% (also letzten 3 Layer) eher Upsampling stattfindet.

Zudem ist zu bemerken, dass atrous und depthwise convolution häufig gewählt werden als Operatoren aus O .

3.2. Unsere Arbeit / Praxis

3.3. Fazit

4 | nnUNet

4.1. Funktionsweise / Theorie

Das nnU-Net ist ein Framework, welches sich mit der Segmentierung von medizinischen 3D-Aufnahmen mit Hilfe von automatisiertem maschinellem Lernen beschäftigt. Es wurde im Rahmen des Medical Segmentation Decathlon Wettbewerb entwickelt und gewann diesen sowie im Anschluss auch viele weitere Segmentierungs-Wettbewerbe.

Das nnU-Net verwendet eine klassische und nicht neue U-Net Architektur (not new U-Net). Es konzentriert sich kaum auf Architekturdesign und -suche, sondern vorrangig auf die Suche von guten Hyperparametern. Es wird eine gleiche oder sehr ähnliche U-Net Struktur immer durch eine individuell auf die individuellen Daten angepasste Trainingspipeline zur Optimierung geschickt (siehe Abbildung 4.1). Das Training der Parameter des Netzes ist also individuell zugeschnitten, während die Architektur sich bei unterschiedlichen Daten nicht oder kaum unterscheidet. Es wird sich also vorrangig auf das Training des Netzes und die Suche individueller Hyperparameter für die Trainingspipeline konzentriert und nicht auf die Suche nach der Architektur.

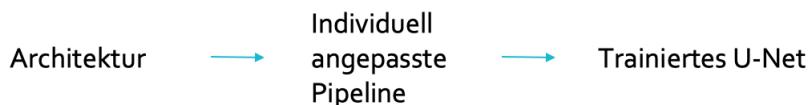


Abbildung 4.1.

Das nnU-Net verwendet 3 Standardarchitekturen, welche 2D U-Net, 3D full resolution U-Net und 3D U-Net Cascade sind. Vor dem Training kann man einstellen, wie viele und welche Architekturen man trainieren möchte. Per default trainiert nnU-Net alle und wählt am Ende die beste oder die beste Kombination aus maximal zwei Architekturen aus. 2D U-Net eignet sich besonders für 2D-Daten und läuft gut auf anisotropen Daten. Es arbeitet auf den Bildern in Originalauflösung. 3D full resolution U-Net eignet sich für kleine 3D-Daten und arbeitet auch auf den Bildern in Originalauflösung. Bei größeren Bildern werden jedoch die Patches sehr klein, was zum immer größer werdenden Verlust von Kontextdaten führt. Daher gibt es das 3D U-Net Cascade für große 3D-Daten. Es besteht aus 2

hintereinander gereihten U-Nets. Das erste U-Net arbeitet auf den Bildernals ganzes, also ohne Aufteilung in Patches, in geringerer Auflösung. Diese grobe Vorsegmentierung wird zusammen mit dem Bild in Originalgröße an das zweite U-Net weitergegeben. Dieses arbeitet dann wieder auf der vollen Bildauflösung und mit Patches und erstellt eine endgültige und verfeinerte Segmentierung. Durch diesen Übergabeschritt zwischen den beiden U-Nets bleiben die Kontextdaten erhalten (siehe Abbildung 4.2).

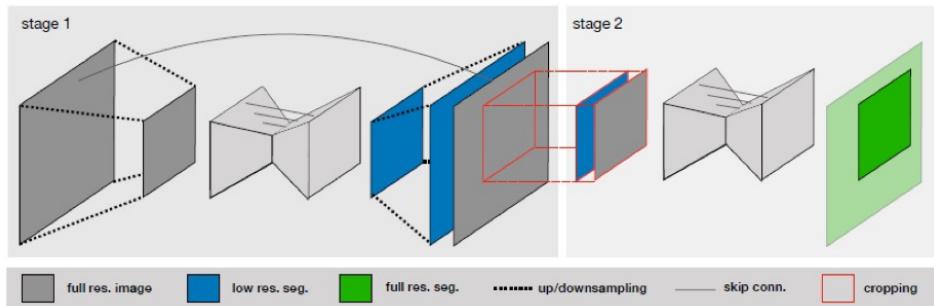


Abbildung 4.2.: nnU-Net Cascade nnunetPaper

Um die Konzentration auf die Anpassung der Hyperparameter der Trainingspipeline an den individuellen Datensatz zu erreichen, wird zunächst ein Datafingerprint aus den Eigenschaften der Trainingsdaten erstellt (siehe Abbildung 4.3). Die hierbei genutzten Eigenschaften sind unter anderem die Imgasize, das Pixelvolumen oder die Farbkanäle, die spacing Anisotropie sowie die Anzahl der Klassen und deren Häufigkeitsverteilung.

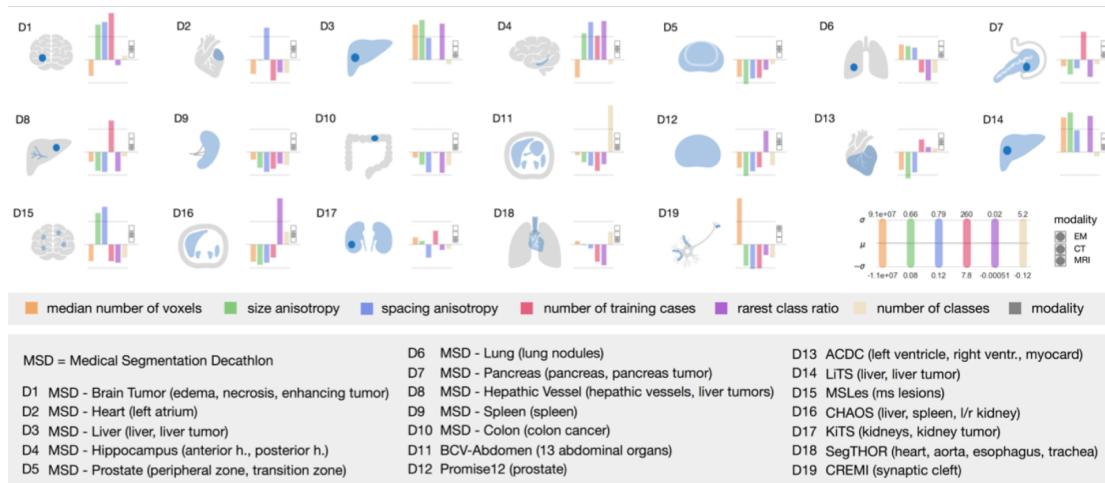


Abbildung 4.3.: Datafingerprint nnunetPaperB

Aus dem Datafingerprint werden, mit Hilfe von heuristischen Regeln, die Inferred Parameter berechnet. Die Inferred Parameter umfassen die Patch Size, die Batch size, wichtige Parameter zur

dynamischen Anpassung der Netzwerktopologie, wie zum Beispiel die Anzahl der Max. Poolings und Downsamplings, sowie Parameter zur Bild Vorverarbeitung.

Die Bestimmung der Patch size erfolgt zunächst initial über den Median der Bildgröße nach dem Resampling. Anschließend wird mit dieser Patch Size die Architektur konfiguriert und geschaut ob ausreichend GPU-Memory zur Verfügung steht. Steht nicht ausreichend GPU-Memory zur Verfügung, so wird die Patch Size reduziert und die Architektur darauf aufbauend neu konfiguriert. Dies wird so oft wiederholt, bis ausreichend GPU-Memory verfügbar ist. Anschließend wird die Batch Size angepasst und das Netzwerk abschließend konfiguriert (Siehe Abbildung 4.4). Dabei muss beachtet werden, dass die Patch Size immer durch 2^i teilbar sein muss (mit $i = \text{Anzahl an DownSampling Operationen}$) da sich die Patch Size pro DownSampling Operation halbiert. Ist das nicht gegeben, so wird die Patch-Size entsprechend vergrößert oder verkleinert bis sie in allen Dimensionen durch 2^i teilbar ist.

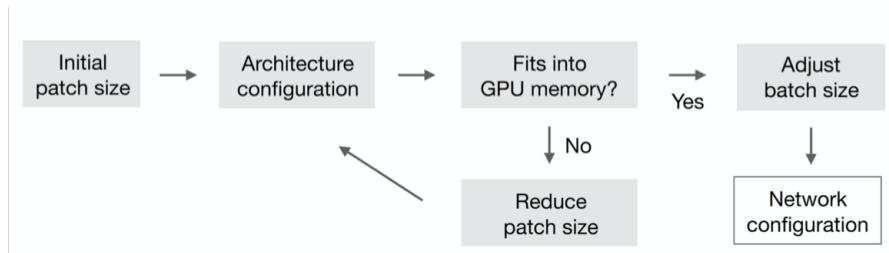


Abbildung 4.4.: Patch Size Ermittlung nnunetPaperB

Im Anschluss an die Inferred Parameter wird der Pipelinefingerprint erstellt, welcher sich aus den Inferred Parametern, den Blueprint Parametern und den empirischen Parametern zusammensetzt. Während die bereits beschriebenen Inferred Parameter für die entscheidende Anpassung an einen neuen Datensatz sorgen, sind die Blueprint Parameter unabhängig von dem Datensatz. Sie enthalten die drei möglichen Architekturen, sowie Hyperparameter mit festen default Werten, wie Verlustfunktion, Training Schedule, Data Augmentation, Normalisierung, stochastic Gradient oder Aktivierungsfunktion (siehe Abbildung 4.5). Die Verlustfunktion wird als die Summe von Dice-Verlustfunktion und Cross-Entropy-Verlustfunktion gewählt. Dies wird gemacht, da medizinische Bilddaten oft Probleme mit einer großen Disbalance im Vorkommen der einzelnen Klassen haben und darum im Training seltener vorkommende Klassen unterrepräsentiert sind und gleichzeitig durch die Lösung dieses Problems die Verteilung der Klassen verzerrt wird. An der Zusammensetzung dieser beiden Verlustfunktionen könnte man also auch arbeiten, wenn man das Framework auf andere Arten von Datensätzen anpassen wollte. Das Training läuft über 1000 Epochen mit jeweils 250 Trainingsiterationen.

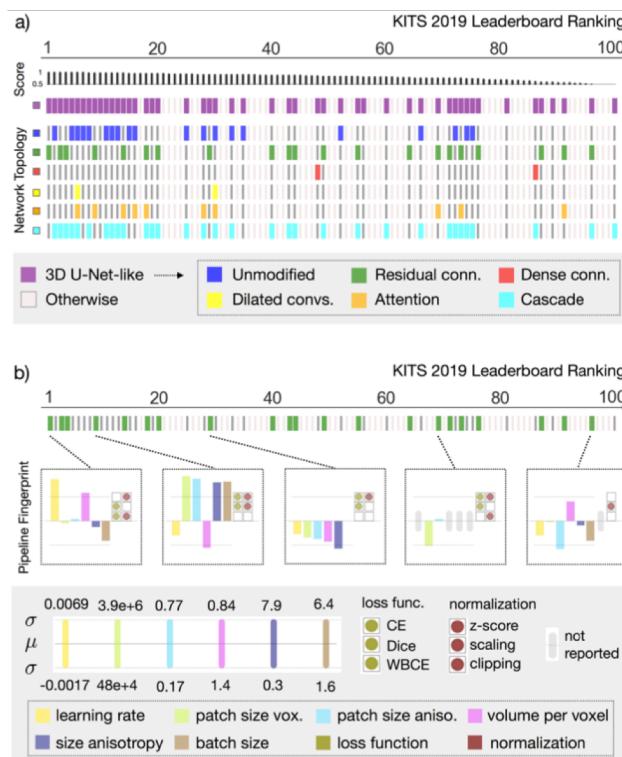


Abbildung 4.5.: Pipelinefingerprint nnunetPaperB

Da die empirischen Parameter nicht direkt aus dem Datensatz erschlossen werden können, werden sich nach dem Training empirisch bestimmt. Sie werden zur Nachbearbeitung und bei der Auswahl der besten Netzstruktur genutzt.

4.2. Unsere Arbeit / Praxis



Datensatz	Split (Train:Test)	verwendete nnUNet-Variante	Trainingszeit (h)	Train- Accuracy (Dice)	Test- Accuracy (Dice)
Larven	265:0 ≈ 1:0	2D	8:30	0.99970	-
Larven	173:92 ≈ 2:1	2D	6:45	0.99982	0.94459
Pascal VOC12	2516:340 ≈ 7:1	2D	26:00	0.90266	0.34953
Retina-2D (manuelle Data-Augmentation)	56:34 ≈ 2:1	2D	23:20	0.99977	0.93606
Retina-2D (minimal)	13:32 ≈ 1:2	2D	21:15	0.99999	0.83013
CT	19:0 ≈ 1:0	3D_fullres	58:30	?	-
CT	19:0 ≈ 1:0	3D_fullres	≈ 89:30 (2000 Epochen)	?	-
CT	19:0 ≈ 1:0	2D	19:00	?	-
CT	19:0 ≈ 1:0	3D_cascade	≈ 53:00 + 49:30 = 102:30	?	-
Retina-3D	14:7 ≈ 2:1	3D_fullres	45:15	0.91863	0.83759
Retina-3D	14:7 ≈ 2:1	2D	19:00	0.98574	0.78931
Retina-3D (Ensemble)	14:7 ≈ 2:1	2D & 3D_fullres	-	0.97775	0.82363

Abbildung 4.6.: Datensätze, auf denen wir trainiert haben (jeweils 1000 Epochen auf GPUv100)

Grundsätzlich haben wir für jeden folgenden Datensatz die Ursprungsdateien in Nifti-Dateien umgewandelt und meistens zufällig in einen Train- und Testsplit aufgeteilt, außer bei dem 3D-CT Datensatz, da wird dort auch mit allen Samples im Trainsplit keine sonderlich guten Ergebnisse erzielen konnten, und dem ersten Versuch auf dem Larvendatensatz.

Der generelle Arbeitsablauf bestand bei uns aus der Befehlsabfolge:

```
nnUNet_plan_and_preprocess -t <TASK-ID> --verify_dataset_integrity
```

bzw. für 2D-Datensätze, da dort kein 3D-Modell anwendbar ist:

```
nnUNet_plan_and_preprocess -t <TASK-ID> -pl3d None
```

Dieser Befehl bereitet das Training vor und prüft, ob der gegebene Datensatz als Nifti-Dateien korrekt ist, indem Wertebereiche und das Vorhandensein aller Dateien geprüft wird. Da dieser Befehl das Training vorbereitet, muss er mit den gleichen verfügbaren Ressourcen wie auch später das Training aufgerufen werden, in unserem Fall auf dem GPUv100 Knoten von Palma II.

Danach kann das Training für die einzelnen Netzvarianten (2d, 3d_fullres, 3d_cascade gestartet werden:

```
# 2d
nnUNet_train 2d nnUNetTrainerV2 <TASK-ID> all --npz
```

```
# 3d_fullres
nnUNet_train 3d_fullres nnUNetTrainerV2 <TASK-ID> all --npz
# Cascade
nnUNet_train 3d_lowres nnUNetTrainerV2 <TASK-ID> all --npz
nnUNet_train 3d_cascade_fullres nnUNetTrainerV2CascadeFullRes
    <TASK-ID> all --npz
```

Dabei verwenden wir für 3D-Datensätze immer 3d_fullres und die 2d-Variante, für 2D-Datensätze immer nur die 2d Variante. Falls das Framework es für einen 3D-Datensatz notwendig hält bzw. überhaupt zulässt, wird auch 3d_cascade verwendet. Der Parameter –npz sorgt dafür, dass die Softmax-Ausgaben zusätzlich gespeichert werden, was zwar sehr viel Festplattenspeicher benötigt, uns aber später ein eventuelles Ensembling der Predictions ermöglicht.

Der Parameter „all“ gibt an, welcher der 5 Folds, die das Framework automatisch erstellt, zur Validierung benutzt wird, während die anderen 4 dem Training dienen. Der Autor des Frameworks vermutet, dass wenn man statt „all“ alle Werte 0 bis 4 verwendet und hinterher aus den 5 verschiedenen Folds ein Ensemble bildet die finale Performance besser ist im Vergleich zu „all“, jedoch hat auch er keine empirischen Beweise dafür **nnunetGithub-Folds**. Wir haben uns für „all“ entschieden, da die Handhabung dann etwas einfacher wird und wir vermuten, dass die zum Training benötigte Zeit deutlich geringer ist, da lediglich ein Modell aus allen Trainingsdaten trainiert wird, und nicht 5 verschiedene basierend auf einer verschiedener Aufteilung der Folds. Außerdem sind unsere Ergebnisse trotz der nicht optimalen Wahl der Parameter ziemlich gut (s. Tabelle 4.6).

Nach dem Beenden des Trainings lassen wir uns von dem Framework die Predictions zu dem Train- und Testsplit erzeugen:

```
nnUNet_predict -i <Pfad zu Original-Niftis> -o <Prediction-Pfad>
-m <2d, 3d_fullres oder 3d_cascade_fullres> -t <TASK-ID> -f all -z
```

Der Parameter -z sorgt auch hier wieder für das Speichern der Softmax-Werte, um später eventuell ein Ensembling aus verschiedenen Netzvarianten zu bilden.

Abschließend erstellen wir mit dem Framework eine Auswertung der Performance auf dem Datensatz, indem wir für den Train- und Testsplit die Ground-Truth Segmentierung mit den Predictions vergleichen:

```
nnUNet_evaluate_folder -ref <Ground-Truth-Pfad>
-pred <Prediction-Pfad> -l <Klassennummern>
```

<Klassennummern> ist hierbei eine Liste aller Klassennummern, die in der Auswertung berücksichtigt werden. Da uns die Performance auf dem Hintergrund (0) nicht interessiert, starten wir bei 1

und gehen z.B. bei Pascal VOC12 **PascalVOCDatensatz** alle Klassen durch -*l 1 2 3 4 ... 20*. Die erzeugt in dem Ordner, in dem die Predictions liegen eine JSON-Datei mit ausführlichen Informationen über die Güte der Predictions je Sample und je vorhandener Klasse, aus der wir einen Scatterplot erstellen.

4.2.1. Datensätze aus dem Paper

Nach unseren schlechten vorherigen Erfahrungen mit (Auto-) Deeplab **deeplabGithub** und insbesondere NAS-Unet **nasunetGithub** haben wir zuerst versucht die bemerkenswert guten Ergebnisse [**nnunetPaper**] auf den Datensätzen der Medical Segmentation Decathlon Challenge **msdChallenge** mit dem Framework zu reproduzieren. Wir haben die 3D-Datensätze Spleen, Lung und Heart **msdChallenge** ausprobiert und konnten auf allen ähnliche Ergebnisse wie im Paper erzielen.

4.2.2. Larven-Datensatz

Larven-Datensatz ohne Testsplit

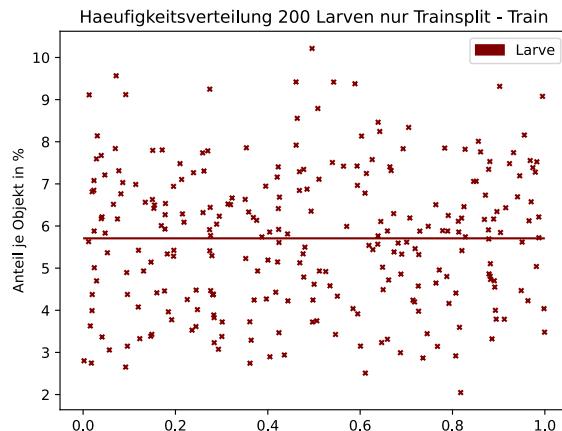


Abbildung 4.7.: Anteil von Objekt (Larve) je Sample im Trainsplit (alle 265 Samples) mit Durchschnitt $\approx 5,8\%$

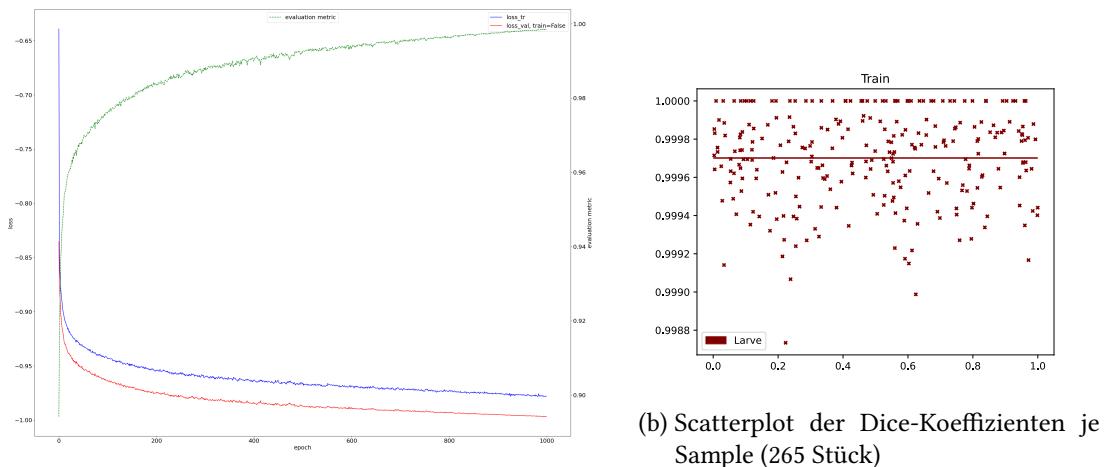
Nachdem wir die Ergebnisse im Paper erfolgreich reproduzieren konnten haben wir versucht einen eigenen Datensatz in das Framework zu geben. Dabei ist zu erwähnen, dass unser Larven-Datensatz ein 2D Datensatz ist. Er beinhaltet Graustufen-Bilder von Larven auf einer Glasscheibe, die mittels Frustrated Total Internal Reflection abgelichtet wurden. Ziel ist es, die Larven zu segmentieren und

die Verschmutzungen um die Larven herum dabei zu ignorieren. Jedoch ist nnUNet nicht dafür entworfen worden auf 2D-Datensätze angewendet zu werden, besonders wenn die Datensätze aus der „non-biomedical domain“ **nnunetGithub2D-Daten** stammen.

Dies ist jedoch nicht als Einschränkung des Funktionsumfangs zu verstehen, sondern lediglich als Vorwarnung, dass die Ergebnisse eventuell nicht gut ausfallen werden.

Wir haben das zum Framework gehörige Python-Script **nnunetGithub2D-Pythonscript** nur leicht modifizieren müssen und konnten den 2D Datensatz in Nifti-Dateien (.nii.gz) konvertieren und so in das Framework geben.

Da es sich hierbei um unseren ersten Test des Frameworks mit eigenem Datensatz handelte haben wir zuerst keinen Test-Split vorgesehen und alle 265 Bilder als Trainingsdaten benutzt.



(a) Verlauf des Dice-Koeffizienten beim Training über 1000 Epochen

Abbildung 4.8.: Dice-Koeffizienten auf dem Trainsplit zum Larvendatensatz ohne Testspli

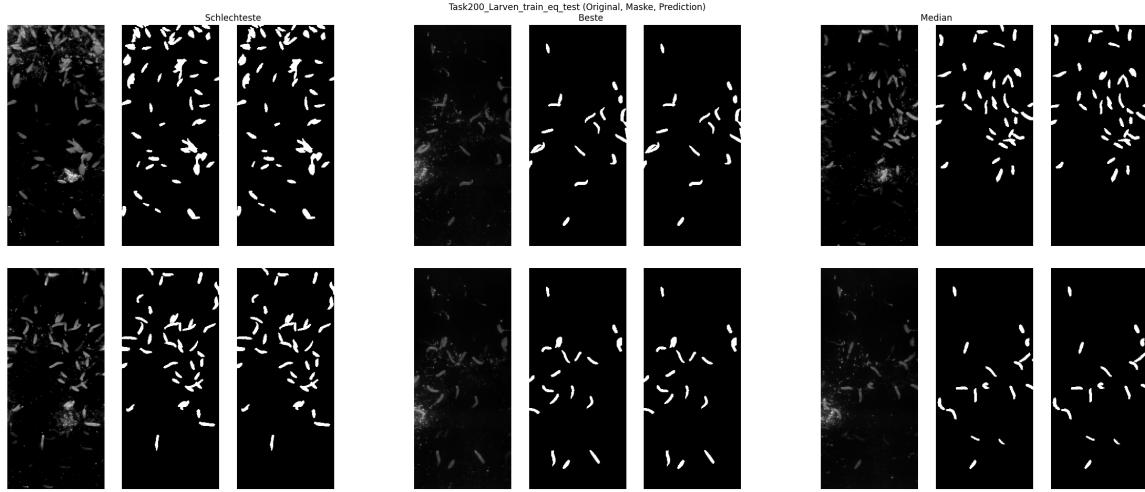


Abbildung 4.9.: Visualisierung des Trainsplits auf dem Larvendatensatz ohne Testsplit (links: schlechteste Ergebnisse, mitte: beste Ergebnisse, rechts: Ergebnisse im Median; jeweils Original, Ground-Truth und Prediction)

Wir konnten bei unserem ersten Versuch einen eigenen Datensatz in das Framework zu geben einen Dice-Wert von durchschnittlich > 0.999 erzielen und auch die am schlechtesten segmentierten Samples liegen weit über 0.99, wie in Abbildung 4.8b zu erkennen ist. Auch bei der Visualisierung der besten, schlechtesten und mittleren Ergebnisse (Abbildung 4.9) kann man zwischen Ground-Truth und der Prediction mit bloßem Auge keine Unterschiede erkennen.

Larven-Datensatz mit $\frac{2}{3}$ Train- und $\frac{1}{3}$ Testspli

Anschließend haben wir die Larvenbilder zufällig in $\frac{2}{3}$ Trainingsdaten und $\frac{1}{3}$ Testdaten aufgeteilt und erneut trainieren lassen. Wir haben uns vergewissert, dass Train- und Testspli möglichst allgemein und zueinander ähnlich sind, und nicht zufällig in einem Split nur die Samples mit hohem Objektanteil und im anderen mit wenig Objektanteil vorhanden sind. Sowohl im Train- als auch im Testspli ist der Objektanteil in den Samples ähnlich (s. Abbildung 4.10).

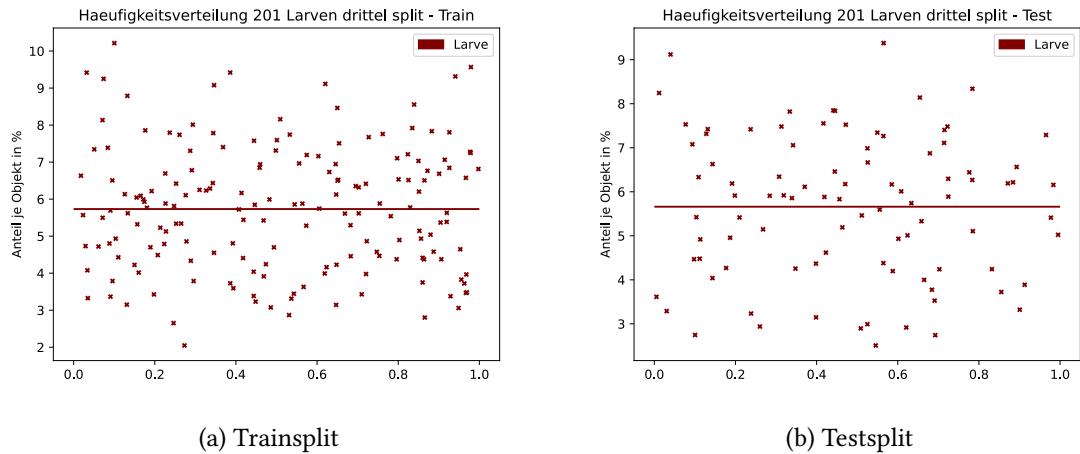
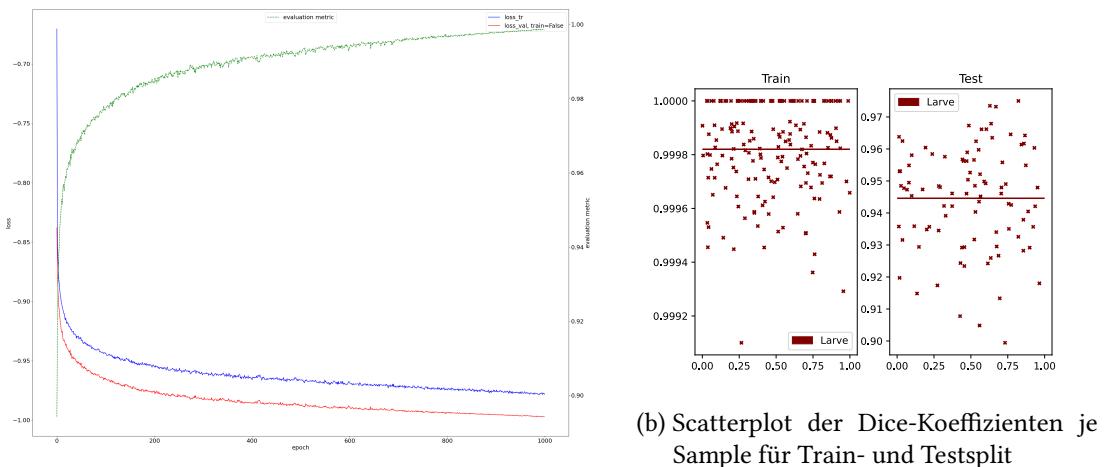


Abbildung 4.10.: Anteil von Objekt (Larve) je Sample mit Durchschnitt je Split $\approx 5,8\%$

Hierbei konnten wir nach dem Training mit Hilfe des Testsplits die Performance des erlernten Modells evaluieren und prüfen, ob das Framework tatsächlich gelernt hat die Strukturen und Merkmale einer Larve zu erkennen und von ähnlich aussehender Verschmutzung zu unterscheiden. Auf dem Testsplits ist die Performance etwas schlechter als auf dem Trainsplit, aber mit einem Dice-Koeffizienten von über 0.94 im Durchschnitt trotzdem noch erstaunlich gut und selbst das am schlechtesten segmentierte Sample im Testsplits hat mit 0.9 auch noch eine akzeptable Genauigkeit (s. Abbildung 4.11b).



(a) Verlauf des Dice-Koeffizienten beim Training über 1000 Epochen

Abbildung 4.11.: Dice-Koeffizienten zum Larvendatensatz mit einem Drittel als Testsplitt

Bei der Visualisierung der Predictions fällt erneut auf, dass auf dem Trainsplit mit bloßem Auge keine Unterschiede zu Ground-Truth vorhanden sind (s. Abbildung 4.12), bei dem Testsplit kommt es jedoch bei den schlechtesten Beispielen zu Fehlern, die auch deutlich erkennbar sind. Es werden teilweise komplette Larven nicht oder nur teilweise erkannt und zudem wird auch besonders in den stark verschmutzen Bildern die Verschmutzung als Larve erkannt. Hierbei stellt sich die Frage, ob die Ground-Truth Segmentierung korrekt ist, da besonders die angeblichen Verschmutzungen, die das Modell als Larve erkannt hat, im Originalbild tatsächlich eher wie eine Larve aussehen als Verschmutzung. Intuitiv würde man die betroffenen Stellen im Originalbild vermutlich auch, wie das Modell, als Larve interpretieren. Die Larven die nicht erkannt wurden sind sehr schwach bis gar nicht mit dem Auge erkennbar bzw. ähneln stark einer Verschmutzung (s. Abbildung 4.13).

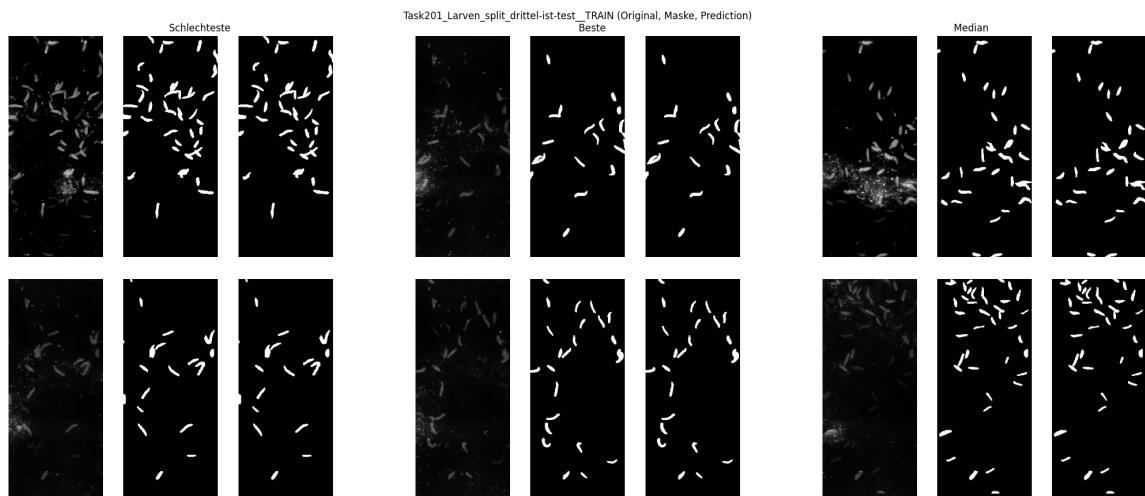


Abbildung 4.12.: Visualisierung des Trainsplits auf dem Larvendatensatz mit $\frac{1}{3}$ Testsplit (links: schlechteste Ergebnisse, mitte: beste Ergebnisse, rechts: Ergebnisse im Median; jeweils Original, Ground-Truth und Prediction)

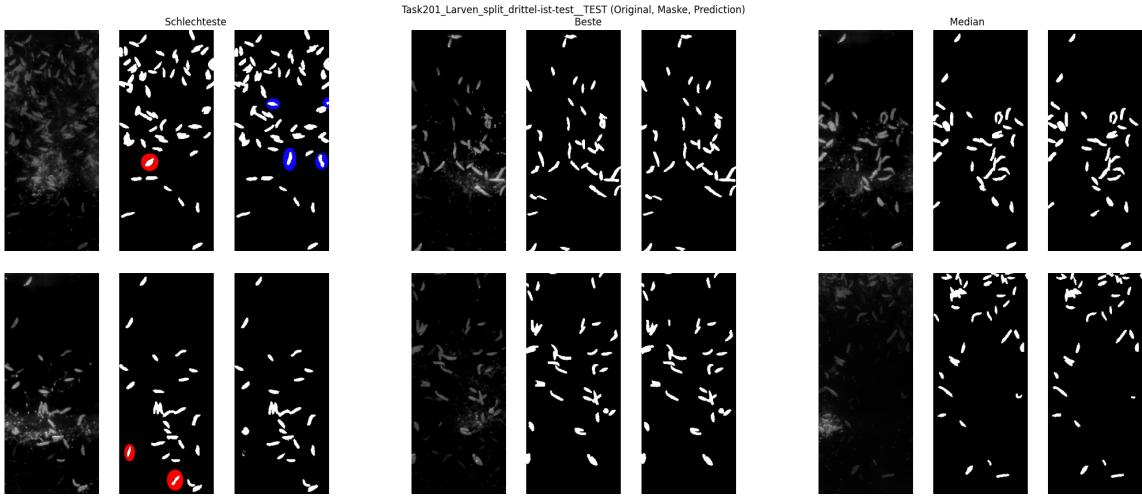


Abbildung 4.13.: Visualisierung des Testsplits auf dem Larvendatensatz mit $\frac{1}{3}$ Testsplits (links: schlechteste Ergebnisse, Mitte: beste Ergebnisse, rechts: Ergebnisse im Median; jeweils Original, Ground-Truth und Prediction). Rot eingefärbt sind nicht erkannte Larven, blau als Larven erkannte Verschmutzungen

4.2.3. Retina 2D-Datensatz

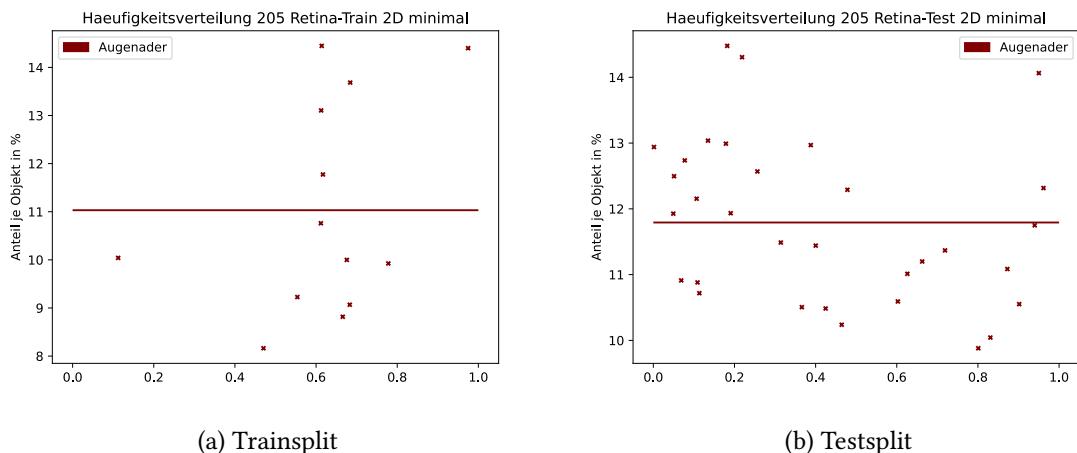


Abbildung 4.14.: Anteil von Objekt (Ader) je Sample mit Durchschnitt je Split $\approx 11\text{-}12\%$

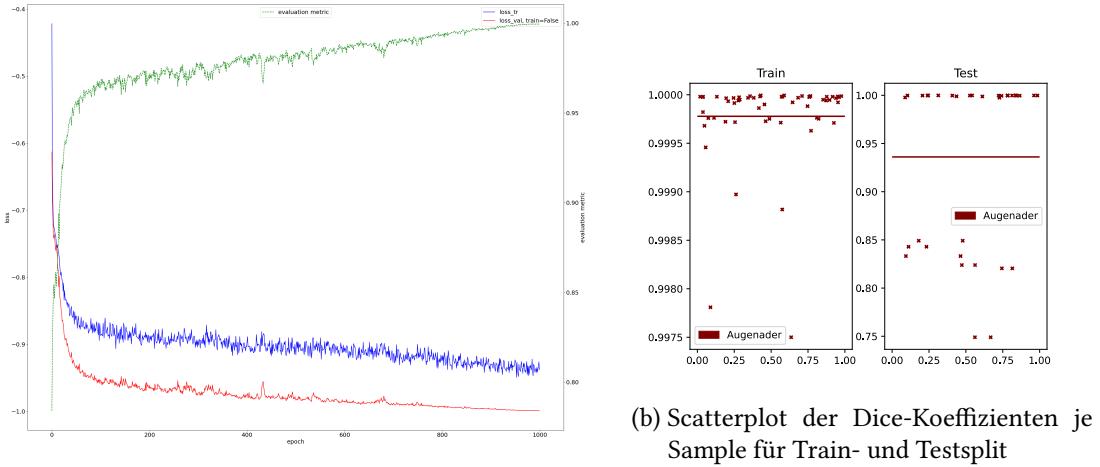
Da wir auf dem Larven-Datensatz (Graustufen mit einer einzigen Objekt-Klasse) so gute Ergebnisse erzielen konnten, obwohl das Framework für solche Aufgaben eigentlich nicht gemacht ist, wollten wir einen Schritt weiter gehen und statt graustufen Bildern farbige Bilder verwenden.

Dazu verwenden wir den Retina-2D Datensatz **retina2d**. Dieser besteht aus jeweils 15 Aufnahmen von gesunden Retinae, Retinae von Augen mit Glaukom und diabetischer Retinopathie, also insgesamt 45 hochauflösenden RGB-Bildern. Unser Ziel der Segmentierung ist es, unabhängig von der Erkrankung, die Adern in der Retina zu markieren.

Diese Bilder konnten auch wie bei den Larven mit dem zur Verfügung gestellten Python-Script **nnunetGithub2D-Pythonscript** in Nifti-Dateien konvertiert werden. Jedoch ergab sich beim Ausführen des Trainings das Problem, dass die automatisch ermittelte Batch-Size des Frameworks angeblich zu niedrig ist. Nach etwas Ausprobieren und Nachschauen im Code sind wir auf eine „estimated GPU-RAM consumption“ **nnunetGithub** gestoßen, die die Batch-Size vorgibt. Durch die hohe Auflösung der Bilder ($\approx 3500 \times 2300$) wird dieser geschätzte GPU-Ram Verbrauch zu groß und als Folge dessen die Batch-Size mit 1 zu klein, da in einem Batch per Definition des Frameworks immer mindestens 2 Samples enthalten sein müssen.

Durch Ausgeben des geschätzten GPU-Ram Verbrauchs haben wir herausgefunden, dass dieser ungefähr linear mit der Anzahl an Pixeln wächst und konnten so ausrechnen, dass eine Verkleinerung der Bilder auf mindestens 42% ausreicht, damit 2 Samples in ein Batch gelangen können. Diese Verkleinerung der Auflösung muss nur für die Trainingsdaten vorgenommen werden. Auf den Testdaten, von denen lediglich eine Prediction erstellt werden muss, kann die Auflösung höher sein.

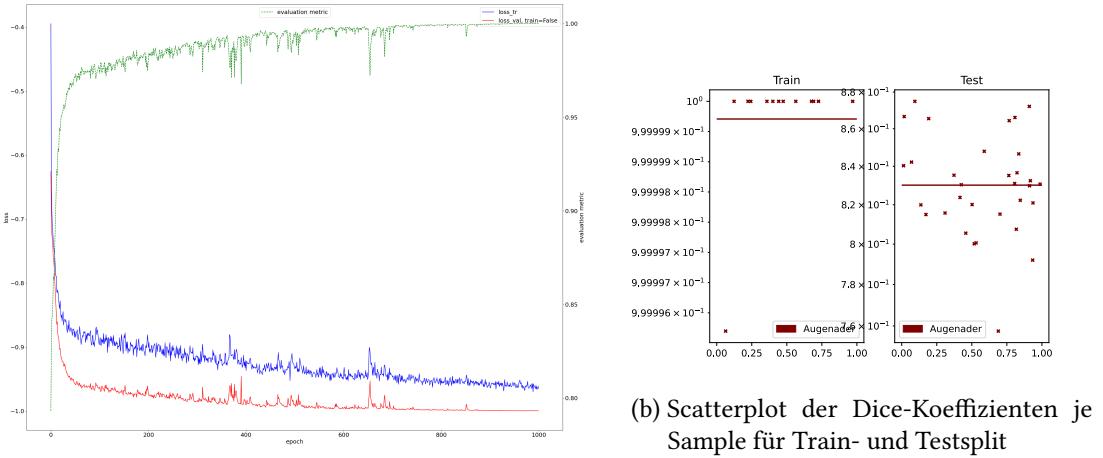
Außerdem haben wir, bevor wir mit dem geschätzten GPU-Ram Verbrauch gespielt haben, manuelle Data-Augmentation betrieben indem wir die Bilder rotiert und gespiegelt haben, in der Hoffnung dadurch mehr Samples in einem Batch zu erhalten. Dies hat sich im Nachhinein jedoch als nicht nötig herausgestellt und hat dem Framework das Training im 1. Durchlauf eventuell unnötig erschwert. Später im 2. Durchlauf mit minimaler Trainingssample-Anzahl haben wir diesen Fehler nicht gemacht.



(a) Verlauf des Dice-Koeffizienten beim Training über 1000 Epochen

Abbildung 4.15.: Dice-Koeffizienten zum Retina-2D Datensatz mit einem Drittel als Testsplitt

Da wir auch hier relativ gute Ergebnisse erzielen konnten, wollten wir das Framework an seine Grenzen bringen und so wenig Trainingsamples wie möglich zur Verfügung stellen. Durch Ausprobieren und schrittweises Annähern haben wir herausgefunden, dass nnUNet, jedenfalls bei diesem Datensatz, mindestens 13 Trainingsbeispiele benötigt, da bei weniger Trainingsbeispielen später beim Training ein *IndexOutOfBoundsException* Fehler auftritt. Leider ist beim Aufteilen in Train- und Testsplitt der Objektanteil in den Samples nicht ganz ausbalanciert, da im Trainsplit nur 11% der Pixel Adern sind und im Testsplitt knapp 12% (s. Abbildung 4.14).



(a) Verlauf des Dice-Koeffizienten beim Training über 1000 Epochen

Abbildung 4.16.: Dice-Koeffizienten zum Retina-2D Datensatz mit einem minimalen Trainsplit von 13 Samples

Es fällt auf, dass bei dem minimalen Trainingssplit Overfitting stattfindet, da alle Samples bis auf ein einziges einen Dice-Koeffizienten von genau 1 besitzen (s. Abbildung 4.16b). Auch der Progress-Graph (Abbildung 4.16a) steigt am Anfang schneller als bei $\frac{2}{3}$ Trainingssplit.

Auf dem Testsplit fällt auf, dass bei Durchlauf 1 (Abbildung 4.15b) eine Gruppierung stattfindet. Viele Samples liegen sehr nah bei 1 und eine zweite, etwa gleich große Gruppe liegt um 0,85 herum. Dies kommt sehr wahrscheinlich von unserer, fälschlicherweise durchgeführten, manuellen Data-Augmentation, bei der wir das gleiche Bild mehrmals in rotierter und gespiegelter Form in das Framework gegeben haben. Beim 2. Durchlauf mit minimalem Trainsplit sind die Samples im Testsplit relativ gleichmäßig um den Durchschnitt von 0,83 verteilt (s. Abbildung 4.16b).

Beim Betrachten der Visualisierung der besten, schlechtesten und mittleren Predictions je Split (Abbildungen 4.17, 4.18) fällt auf, dass die Adern in den Predictions generell breiter sind als in Ground-Truth. Das kommt von der leider notwendigen Verkleinerung der Auflösung der Bilder, da dann bei feinen Adern anstatt nur schwarzer oder weißer Pixel in der Ground-Truth Segmentierung auch graue Pixel entstehen, da z.B. Adern mit einer Breite von einem Pixel nicht weiter in der Auflösung reduziert werden können. Wir haben uns dafür entschieden, die dann grauen Pixel auch als weiße Pixel, also als Ader-Segmentierung, zu zählen, da ansonsten feine Adern Lücken bekommen und die Segmentierung insgesamt schlechter ausfällt. Ansonsten findet man keine groben Fehler, die meisten Adern werden sehr genau erkannt.

4. nnUNet

Abbildung 4.17.: Visualisierung des Trainsplits auf dem Larvendatensatz mit minimalem Trainsplit
 (links: schlechteste Ergebnisse, mitte: beste Ergebnisse, rechts: Ergebnisse im Median; jeweils Original, Ground-Truth und Prediction)

Abbildung 4.18.: Visualisierung des Testsplits auf dem Larvendatensatz mit minimalem Trainsplit
 (links: schlechteste Ergebnisse, mitte: beste Ergebnisse, rechts: Ergebnisse im Median; jeweils Original, Ground-Truth und Prediction)

Da wir bei diesem Datensatz relativ gute Ergebnisse produzieren konnten, haben wir dem Framework komplett fremde Bilder **retina2dExtra** in verschiedenen Auflösungen und Zoom-Stufen zum

Segmentieren gegeben. Zu diesen Bildern gab es leider keine Ground-Truth Segmentierung, jedoch kann man auch so grob abschätzen wie robust das Modell ist, und dass es tatsächlich die Merkmale einer Ader in der Retina erlernt hat und sogar mit verschiedenen Auflösungen, Ausschnitten und Färbungen der Retina umgehen kann (s. Abbildung 4.19 und 4.20).

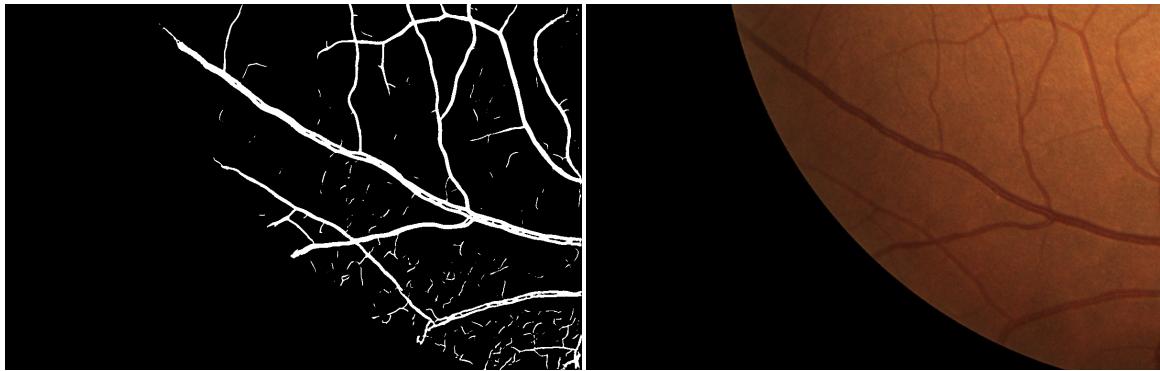


Abbildung 4.19.: Bild 13a_right aus **retina2dExtra** in starkem Zoom - Prediction und Original

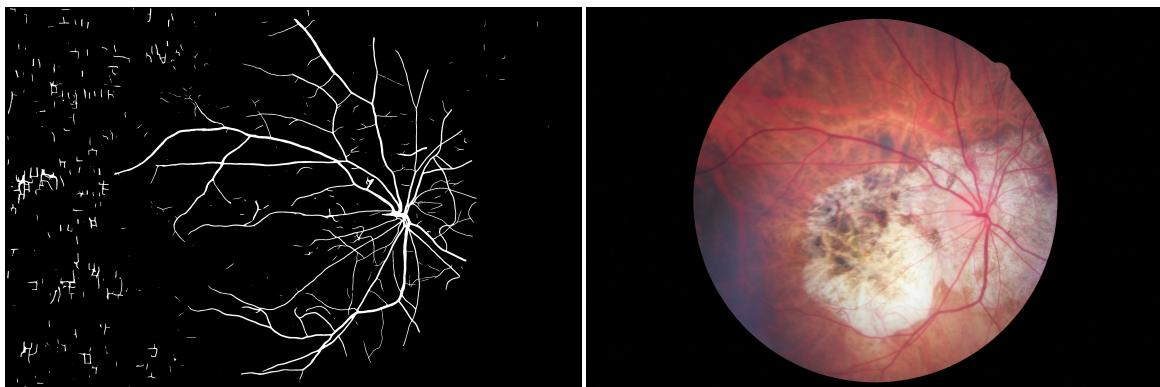


Abbildung 4.20.: Bild 1170_right aus **retina2dExtra** mit Wucherung im Hintergrund - Prediction und Original

4.2.4. CT-Datensatz

Da wir bisher nur eigene 2D-Datensätze in das Framework gegeben haben und es nur wenig öffentlich zugängliche 3D-Datensätze zur Segmentierung gibt, die nicht Teil der MSD-Challenge **msdChallenge** sind, wurde uns ein Datensatz mit 19 Ganzkörper CT-Aufnahmen zur Verfügung gestellt, in denen Kalzium-Ablagerungen am Rand der Gefäße segmentiert werden sollen. Auffällig ist hierbei, dass trotz der hohen Auflösung des Datensatzes (512x512 mit $\approx 400\text{-}600$ Slices je Sample) nur *sehr* wenig Pixel in der Segmentierung markiert sind. Bestenfalls sind ≈ 13000 Pixel im kompletten 3D Volumen bestehend aus 570 Slices mit je einer Auflösung von 512x512, was einem

Stimmt
das so
mit dem
CT Daten-
satz?

Anteil von *maximal* 0,008%. Außerdem gibt es auch Samples mit deutlich weniger oder gar keinen markierten Pixeln in der Ground-Truth Segmentierung. Dies erschwert das Training und könnte eine Begründung für die schlechten Ergebnisse sein.

Um den Datensatz in nnUNet zu geben mussten wir erst die zur Verfügung gestellten .nrrd (Ground-Truth) und .dcom (3D-CT-Scan) Dateien mit eigenen kleinen Pythonscripten **autoMLGithub** in Nifti-Dateien umwandeln. Beim Trainieren ist uns erst nach zweimaligem Neustarten ohne Erfolg, mit sehr schlechten Ergebnissen bei genauerem Lesen des Papers aufgefallen, dass wir das Preprocessen in nnUNet bisher immer auf dem Login-Node des Clustercomputers Palma II durchgeführt haben, der keine Grafikkarte besitzt, um Wartezeiten in der Warteschlange zu vermeiden. Das Preprocessen ist aber abhängig von den zu dem Zeitpunkt zur Verfügung gestellten Ressourcen, insbesondere des verfügbaren GPU Speichers. Nachdem wir diesen Denkfehler behoben haben und den CT-Datensatz sowohl auf einer GPUv100 Karte preprocessed und trainiert haben, gelang es uns einigermaßen akzeptable Ergebnisse zu erzielen.

4.2.5. Pascal VOC2012

Nachdem wir bei den bisherigen 2D-Datensätzen nur Graustufen-Bilder mit einer Klasse (Larven) und farbige Bilder mit einer Klasse (Retina 2D) verwendet haben wollten wir auch noch einen 2D-Datensatz in Farbe mit mehreren Objekt-Klassen ausprobieren. Die Entscheidung fiel relativ schnell auf Pascal VOC2012 **PascalVOCDatensatz**, da dieser sehr viele Klassen (20 Klassen + Background) und viele segmentierte Bilder (2856 Stück) in verschiedenen Formaten und Auflösungen zur Verfügung stellt und von vielen anderen Frameworks zum Vergleichen verwendet wird. Außerdem ist dieser Datensatz mit Fotografien nochmal wesentlich weiter von der „biomedical-domain“ **nnunetGithub2D-Daten**, für die das Framework eigentlich erstellt wurde, entfernt und dient somit als Test, wie robust das Framework mit verschiedenen Daten umgeht.

Um die Pascal-VOC 2012 Bilder in nnUNet zu geben mussten erst die Farbkodierungen in der Ground-Truth-Segmentierung in Indizes umgewandelt werden **autoMLGithub**, da nnUNet bei 0=Background beginnend aufsteigende Integer für die Klassen erwartet aber in dem Pascal-Datensatz **PascalVOCDatensatz** die Segmentierung zur besseren Erkennbarkeit farblich gekennzeichnet ist. Außerdem mussten 57 Bilder, die nicht farbig sondern nur in Graustufen in Pascal-VOC 2012 **PascalVOCDatensatz** vorhanden sind entfernt werden, da das Framework nicht mit einer variablen Anzahl an (Farb-) Kanälen umgehen kann.

Bei der Auswertung ist uns aufgefallen, dass die Accuracy (Dice) zwischen den verschiedenen Klassen stark schwankt und das Framework manche Klassen deutlich besser erkennt als andere.

4.2.6. Retina 3D-Datensatz

Abschließend haben wir, da wir auf dem anderen 3D-Datensatz mit CT-Aufnahmen keine guten Ergebnisse erzielen konnten, einen weiteren 3D-Datensatz mit Retinae zur Verfügung gestellt bekommen. Er besteht aus 21 Samples von 3D-Scans der Retina mit Segmentierungen. Diese Samples liegen sowohl in ihrer ursprünglichen, gekrümmten Form vor als auch in einer geplätteten Form, die die Krümmung der Netzhaut heraus rechnet. Wir haben uns auf die gekrümmte Version beschränkt.

anders formulieren, Ergebnisse sind halbwegs ok

Wir haben wieder die zur Verfügung gestellten .mat Dateien in Niftis konvertiert **autoMLGithub** und das Training gestartet.

Ergebnisse einfügen, diskutieren wie 3d_fullres und 2d sich unterscheiden, Ensemble auch

4.3. Fazit

Besonders gut gefallen hat uns der einfache Umgang mit dem Framework. Alle Schritte sowohl zur Installation als auch zum Umgang mit neuen und eigenen Datensätzen wurden detailliert in einer Anleitung erklärt und auch der Code ist sehr gut dokumentiert, was zum Verständnis deutlich beiträgt **nnunetGithub**.

Guter Support: Github Issue Antwortzeit nur wenige Tage, schnelle bugfixes

Das Framework machte von Anfang an einen durchdachten, anwenderfreundlichen Eindruck. Die Bedienung ist ziemlich einfach und das Framework nimmt dem Anwender jegliche Arbeit ab, so wie es sein sollte.

Die anderen beiden Frameworks kritisieren, dieses ist deutlich besser / das einzige was überhaupt funktioniert...

5 | Gesamtfazit

Gesamt-
Fazit mit
Inhalt
füllen

A | Ein Anhangskapitel

Anhang
mit Inhalt
füllen

Literaturverzeichnis
ergänzen