

Todo list

Einführung mit Inhalt füllen	2
NAS-Unet mit Inhalt füllen	3
Deeplab mit Inhalt füllen	11
Gesamt-Fazit mit Inhalt füllen	16
Anhang mit Inhalt füllen	17
Literaturverzeichnis ergänzen	18



AutoML für Segmentierung

PROJEKTSEMINAR

zum Thema

AUTOML

Westfälische Wilhelms-Universität Münster
Institut für Informatik

Eingereicht von:

Milan Blunk (Matrikelnummer)

Pia Nümann (454700)

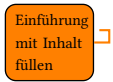
Matthias Wolff (458766)

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	8
3. Deeplab	11
3.1. Funktionsweise / Theorie	11
3.2. Unsere Arbeit / Praxis	11
3.3. Ergebnisse	11
3.4. Fazit	11
4. nnUNet	12
4.1. Funktionsweise / Theorie	12
4.2. Unsere Arbeit / Praxis	15
4.3. Ergebnisse	15
4.4. Fazit	15
5. Gesamtfazit	16
A. Ein Anhangskapitel	17

1 | Einführung



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 [1] gesucht und diese dann auf den unterschiedlichen medizinischen Datensätzen trainiert. Als medizinische Datensätze werden für die MRT-Bilder der Promise12 Datensatz [2], für die CT-Bilder der Chaos Datensatz [3] und für die Ultraschallbilder der NERVE Datensatz [4] 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).

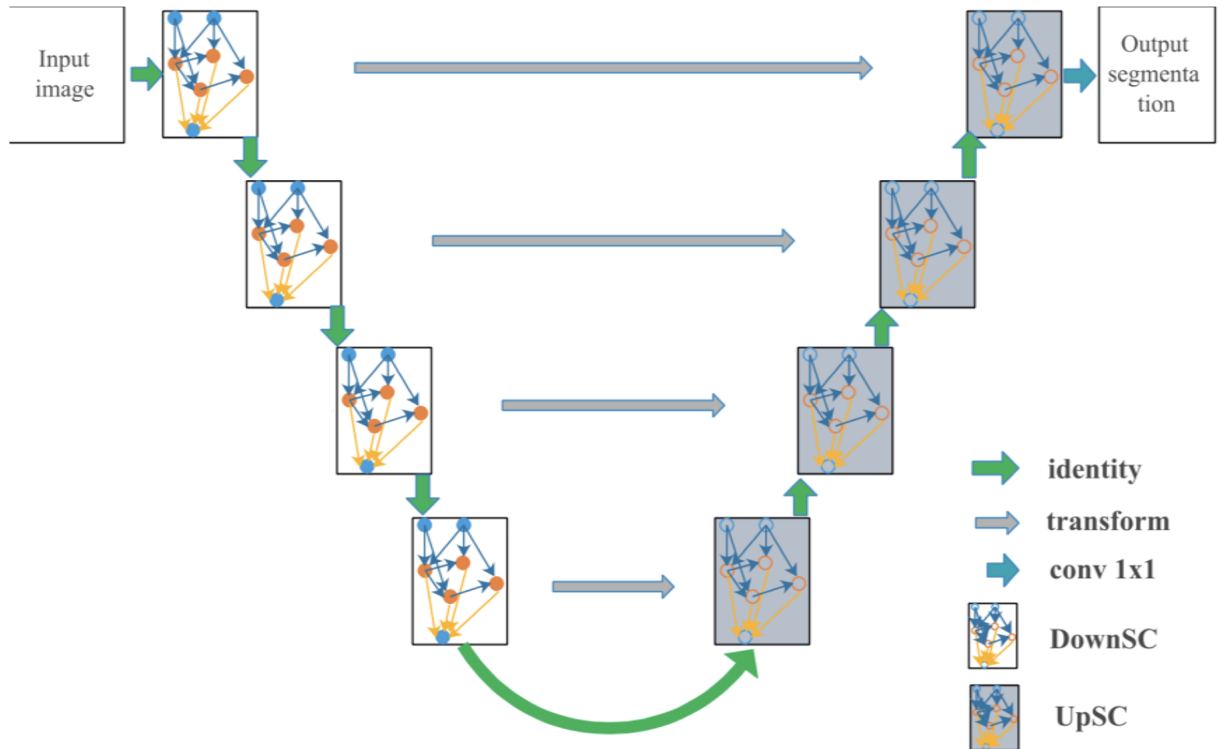


Abbildung 2.1.: Zellbasierte Netzarchitektur von NAS-Unet [5]

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.

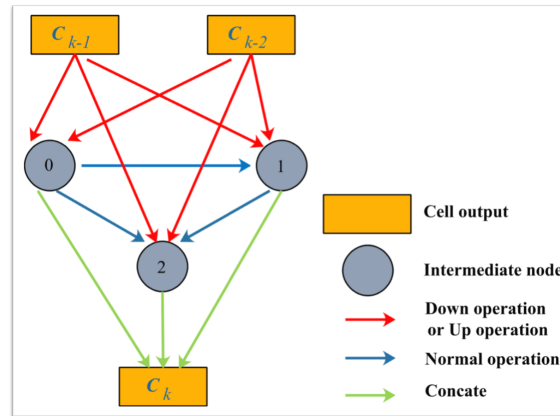


Abbildung 2.2.: Zellarchitektur von NAS-Net [5]

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

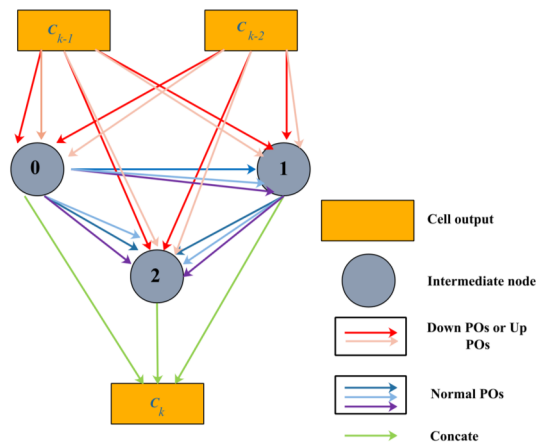


FIGURE 4. An example of Over-Parameterized Cell Architecture. each edge associate with N candidate operations from different primitive operation sets.

Abbildung 2.3.: überparametrisierte Zellarchitektur von NAS-Net [5]

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.

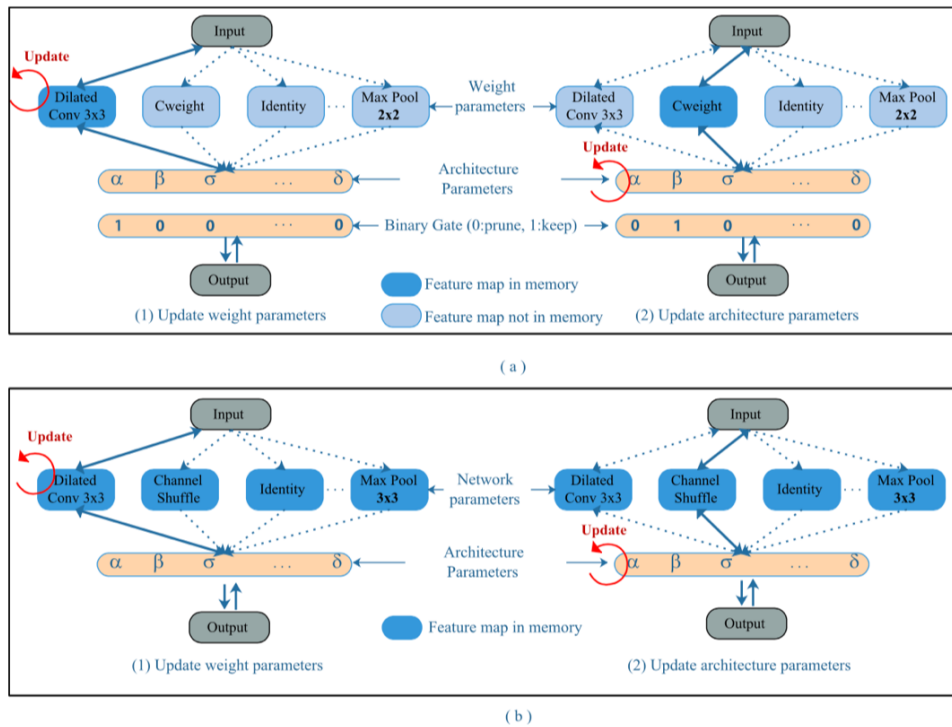


Abbildung 2.4.: Vergleich der binären Suchstrategie von NAS-Unet mit dem gleichzeitigen Updaten aller Parameter [5]

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. [5] und Page 44253, Satz2, [5]) 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 Hindernisse 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 wir 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 Ordernamen 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 [3] 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).



Abbildung 2.5.: überparametrisierte Zellarchitektur von NAS-Unet [5]

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.

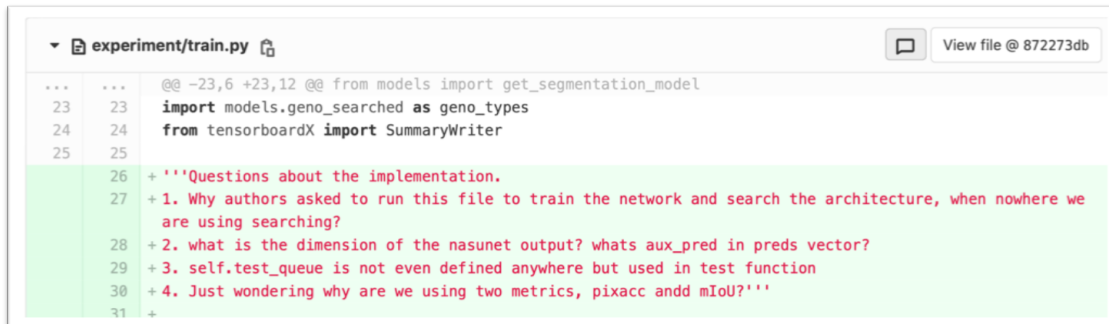
Auf dem Chaos-Datensatz haben wir NAS-Unet nicht zum Laufen bekommen, da NAS-Unet nach einem Bild sucht, welches in keiner der öffentlichen Versionen des Datensatzes ([3] unter Download) existiert. Da es uns nicht gelungen ist, dieses Problem zu lösen, haben wir das Framework leider nicht auf dem Chaos-Datensatz 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 @@ -23,6 +23,12 @@ from models import get_segmentation_model
23 23 import models.geno_searched as geno_types
24 24 from tensorboardX import SummaryWriter
25 25
26 + '''Questions about the implementation.
27 + 1. Why authors asked to run this file to train the network and search the architecture, when nowhere we
28 + are using searching?
29 + 2. what is the dimension of the nasunet output? whats aux_pred in preds vector?
30 + 3. self.test_queue is not even defined anywhere but used in test function
31 + 4. Just wondering why are we using two metrics, pixacc and mIoU?'''

```

Abbildung 2.6.: Beispielversuch den Code nachzuvollziehen [6]

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 [7] 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 [8] sind wir mehrfach darauf gestoßen, dass die Ergebnisse, die im Paper angegeben wurden, nicht reproduziert werden konnten (zum Beispiel Issue 31 [9]). 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 | **Deeplab**

Deeplab
mit Inhalt
füllen

3.1. Funktionsweise / Theorie

3.2. Unsere Arbeit / Praxis

3.3. Ergebnisse

3.4. 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 Architekturdisein 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.

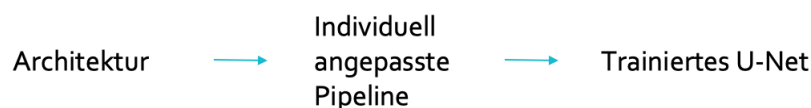


Abbildung 4.1.

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 Bildern als 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).

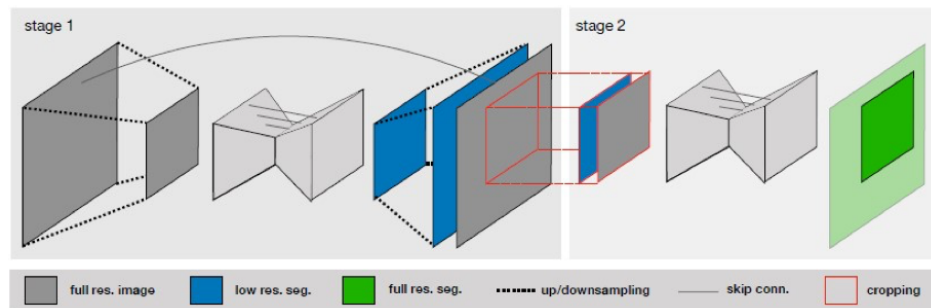


Abbildung 4.2.: nnU-Net Cascade [10]

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 Imagegröße, das Pixelvolumen oder die Farbkanäle, die spacing Anisotropie sowie die Anzahl der Klassen und deren Häufigkeitsverteilung.

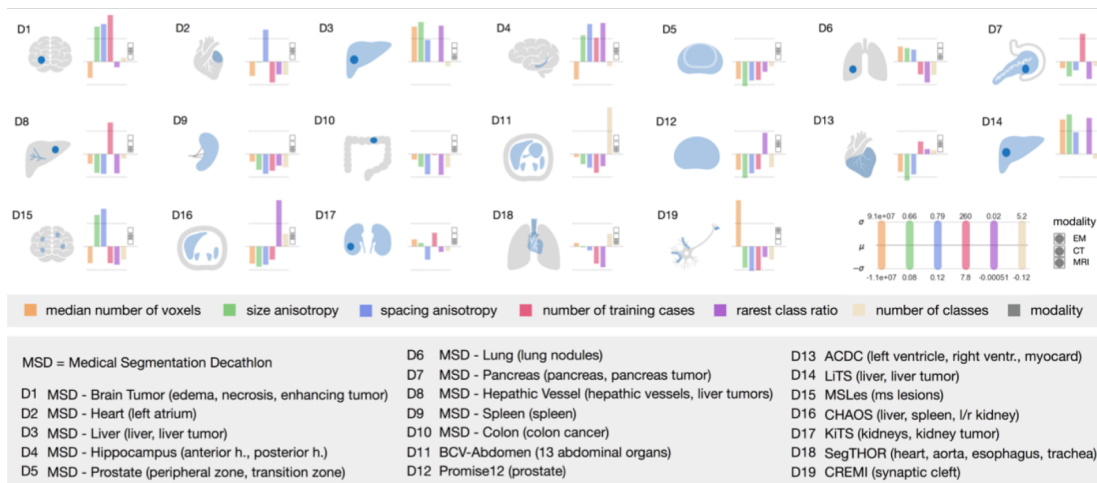


Abbildung 4.3.: Datafingerprint [11]

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 = 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.

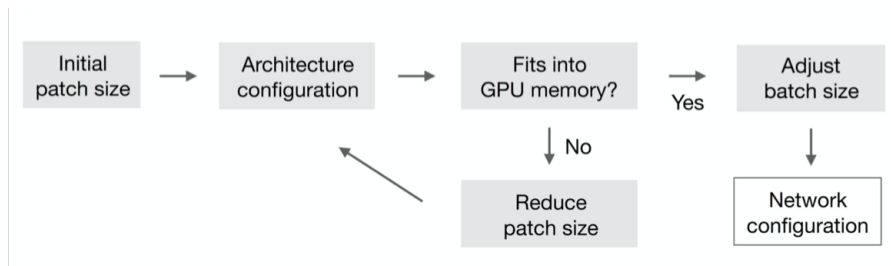


Abbildung 4.4.: Patch Size Ermittlung [11]

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.

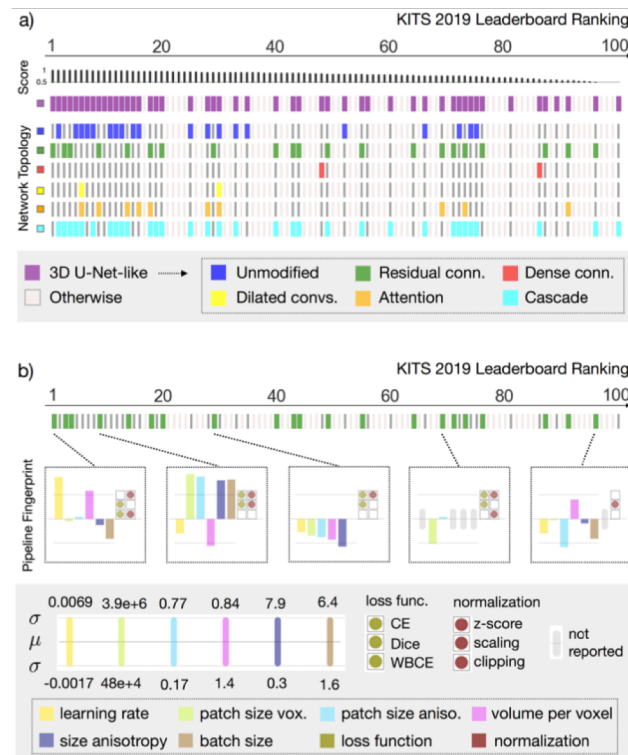


Abbildung 4.5.: Pipelinefingerprint [11]

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

4.2. Unsere Arbeit / Praxis

4.3. Ergebnisse

4.4. Fazit

5 | Gesamtfazit

Gesamt-
Fazit mit
Inhalt
füllen

A | Ein Anhangskapitel

Anhang
mit Inhalt
füllen

Literatur

- [1] M. Everingham, S. Eslami, L. V. Gool, C. Williams, J. Winn und A. Zisserman, *The Pascal visual object classes challenge: A retrospective*, Jan. 2015.
- [2] G. L. et al., *Evaluation of prostate segmentation algorithms for MRI: the PROMISE12 challenge*, Feb. 2014. Adresse: <http://www.sciencedirect.com/science/article/pii/S1361841513001734>.
- [3] *CHAOS-Combines (CT-MR) Healthy Abdominal Organ Segmentation*. 2019. Adresse: https://chaos.gand-challenge.org/Combined_Healthy_Abdominal_Organ_Segmentation/.
- [4] *Ultrasound Nerve Segmentation Kaggle*, 2016. Adresse: <https://www.kaggle.com/c/ultrasound-nerve-segmentation>.
- [5] Y. Weng, T. Zhou, Y. Li und X. Qiu, „NAS-Unet: Neural Architecture Search for Medical Image Segmentation“, 2019. Adresse: https://www.researchgate.net/publication/332216927_NAS-Unet_Neural_Architecture_Search_for_Medical_Image_Segmentation.
- [6] *NAS-Unet, Fraunhofer, Kommentar*. Adresse: <https://gitlab.itwm.fraunhofer.de/sharad/nasunet/-/commit/872273db3153fdb8354ddf60deb73702c0fc126d>.
- [7] *NAS-U-Net Github Issue 11*. Adresse: <https://github.com/tianbaochou/NasUnet/issues/11>.
- [8] *NAS-Unet Github Repository*. Adresse: <https://github.com/tianbaochou/NasUnet>.
- [9] *NAS-U-Net Github Issue 31*. Adresse: <https://github.com/tianbaochou/NasUnet/issues/31>.
- [10] F. Isensee, J. Petersen, A. Klein, D. Zimmerer, P. F. Jaeger, S. Kohl, J. Wasserthal, G. Köhler, T. Norajitra, S. Wirkert und K. H. Maier-Hein, „nnU-Net: Self-adapting Framework for U-Net-Based Medical Image Segmentation“, 2018. Adresse: <https://arxiv.org/pdf/1809.10486.pdf>.
- [11] F. Isensee, P. F. Jaeger, S. A. A. Kohl, J. Petersen und K. H. M. Hain, „Automated Design of Deep Learning Methods for Biomedical Image Segmentation“, 2020. Adresse: <https://arxiv.org/pdf/1904.08128.pdf>.