Seminararbeit “EnBeeMo”

Schriftliche Ausarbeitung zur Entwicklung und Auswertung eines CNN's zur Objekterkennung von Bienen

Hiob Gebisso
Hochschule München
Fakultät für Informatik und Mathematik
80335 München
Email: gebisso@hm.edu

Johannes Pagel
Hochschule München
Fakultät für Informatik und Mathematik
80335 München
Email: jpagel@hm.edu

Laura Stangl
Hochschule München
Fakultät für Informatik und Mathematik
80335 München
Email: lstangl@hm.edu

***Abstract*—Innerhalb dieser Arbeit wird die Vorgehensweise des Projektes „EnBeeMoo“ erläutert. Im Zuge des Projektes wurde ein neuronales Netz zur Erkennung von Bienen in Bilddaten implementiert, mit einem Fokus auf die Optimierung von Hyperparametern und einer Sensitivitätsanalyse ebendieser.**

I. Grundlegendes Vorgehen

Die vorliegende Aufgabenstellung wird aus dem Abstrakt ersichtlich. Es wurde eine Sammlung an vorgelabelten Bilddateien von Bienen zur Verfügung gestellt. Zur Lösung der Problemstellung wurde die letzte Schicht eines vortrainierten Netzwerks mit Hilfe der Bilddaten neu trainiert. Um hier die bestmöglichen Ergebnisse hinsichtlich der Präzision zu erzielen, wurde im weiteren Vorgehen ein Hyperparametertuning vorgenommen.

A. Datenaufbereitung

Während der Datenaufbereitung ist aufgefallen, dass einzelne Bilder falsch annotiert sind. Ein Bild, welches fälschlicherweise keine Bounding Boxes enthielt, wurde deswegen aus dem Datensatz entfernt. Im Rahmen der Vorverarbeitung wurde eine Datentransformation vorgenommen. Das Ziel ist es hierbei, einzelne Bilder zu modifizieren, um Overfitting zu vermeiden und die Generalisierung des Modells zu verbessern. In der vorliegenden Implementierung wurde eine horizontale Spiegelung angewandt. Hierzu wurde der Datensatz aus dem AWS-Bucket importiert und innerhalb einer eigens erstellten Pytorch Image Dataset-Klasse entsprechend aufbereitet. Die Dataset-Klasse wandelt die Bilddateien in Tensoren um, extrahiert die Koordinaten und Fläche der Bounding Boxes aus den Annotations und aggregiert diese mit weiteren Metadaten in eine Target Dictionary. Ein Datenpunkt entspricht dem Tupel aus Bildtensor und der Target Dictionary. Anschließend wurde eine randomisierte 90/10 Aufspaltung in Trainingset und Validierungsset vorgenommen und die beschriebene Transformierung auf den Trainingset angewandt. [1]

B. Netzwerkarchitektur

Es wurde der Entschluss gefasst, die Faster RCNN-Architektur zu verwenden. Eine Besonderheit von Faster RCNN im Vergleich zu R-CNNs liegt in dem Geschwindigkeitszuwachs. Dieser resultiert daraus, dass nicht jedes Mal 2000 Regionsvorschläge an das neuronale Faltungsnetzwerk gesendet werden müssen, sondern die Faltungsoperation nur einmal pro Bild ausgeführt wird und hieraus eine Feature-Map generiert wird. Als Backbone wurde der Einsatz von ResNet50 gewählt. ResNet, kurz für Residual Networks, ist ein klassisches neuronales Netzwerk, das als Backbone für zahlreiche Computer-Vision-Aufgaben verwendet wird. ResNet verwendet die Sprungverbindung/Identitätsverbindung, um die Ausgabe einer früheren Ebene zu einer späteren Ebene hinzuzufügen. Dieses Vorgehen hilft dabei, das Problem des verschwindenden Gradienten massiv zu mildern. Pytorch bietet für den Einsatz dieser Architektur ein vortrainiertes Modell an. Dieses wurde auf den Datensatz COCO 2017, bestehend aus 118.000 Bildern von 172 Klassen, trainiert. Das vortrainierte Netz kann aufgrund der Diversität des COCO 2017 Datensatzes für verschiedene Problemstellungen der Objekterkennung, wie auch die Erkennung von Bienen in diesem Projekt, eingesetzt werden. Hierzu musste der Klassifikator in der letzten Schicht der Netzwerkarchitektur redefiniert und neu trainiert werden. [2]

Das Erkennen von Objekten in verschiedenen Maßstäben ist insbesondere für kleine Objekte eine Herausforderung. Es kann eine Pyramide desselben Bildes in unterschiedlichen Maßstäben verwendet werden, um Objekte zu erkennen. Die Verarbeitung von Bildern mit mehreren Maßstäben ist jedoch zeitaufwändig und der Speicherbedarf ist zu hoch, um gleichzeitig Ende-zu-Ende trainiert zu werden. Deshalb wurde eine Top-Down-Architektur mit seitlichen Verbindungen entwickelt, um semantische Feature-Maps auf hoher Ebene in allen Maßstäben zu erstellen. Diese Architektur wird als Feature Pyramid Network (FPN) bezeichnet und zeigt in mehreren Anwendungen signifikante Vorteile gegenüber generischen Feature-Extraktoren. [3]

C. Beschreibung des Trainings

Pytorch stellt im Zusammenhang mit dem Faster RCNN Resnet-50 FPN verschiedene Klassen und Methoden zur Verfügung, die das Training der letzten Schicht auf den eigenen Datensatz erleichtern. [4]

Während des Trainings einer Epoche kann nach jeder Iteration der Trainingsloss ausgegeben werden. Am Ende jeder Epoche können die typischen Metriken einer Klassifikation betrachtet werden, indem anhand unseres Validationssatzes

die Intersection over Union (prozentuale Schnittfläche der Ground-Truth Boxen und den geschätzten Boxen) errechnet wird und im Anschluss Schwellenwerte für diesen Anteil definiert werden. Es ergibt Sinn, sich hierbei auf die Average-Precision zu beschränken, da die Klassen (Biene oder keine Biene) unbalanciert sind und somit die Accurary keine sinnvollen Ergebnisse liefert.

Die Average-Precision (aP) ist die durchschnittliche Precision über alle definierten Recall Thresholds. Hierzu werden die geschätzten Boxen nach ihren Klassifizierungsscores abwärts sortiert und für jeden der 101 Recall Thresholds von 0-1 die maximale Precision berechnet und gemittelt. [5]

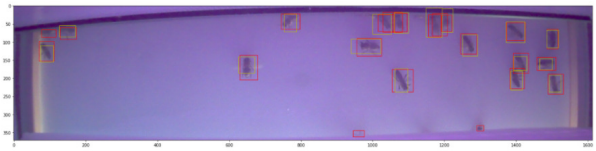

Fig. 1. Bounding Boxes (Predictions = rot vs. Ground Truth = gelb) vor der Hyperparameteroptimierung. Quelle: Eigene Darstellung

Außerdem fiel die Entscheidung darauf, bereits ab einer Schnittfläche von 50% von einer richtigen Zuordnung zu sprechen, da die geschätzten Boxen teils präziser an die Biene angepasst waren und sich in einzelnen Fällen von der Fläche der Grund-Truth Boxen unterschieden. (Fig. 1)

Der Trainingsloss und die Average-Precision wurden anhand des Tensorboard-Writers für jede Epoche mitgeschnitten, um die Verbesserungen des Netzes im Verlauf des Trainings untersuchen zu können und die Zielgröße der Hyperparameteroptimierung im Folgenden quantifizieren zu können. [6]

II. Hyperparametersuche

In der folgenden Sektion wird die Wahl der zu optimierenden Hyperparameter genauer erläutert.

A. Stochastic Gradient Descent & Wahl der Hyperparameter

Die Autoren von Pytoch empfehlen zum Training der letzten Schicht des Netzes den Stochastic Gradient Descent. Bei Wahl dieses Optimierers ist die Lernrate ein maßgeblicher Hyperparameter, da diese in jeder Iteration die Schrittweite festlegt. Diese sollte für jedes Optimierungsproblem individuell festgelegt werden, da die Kostenfunktion immer unterschiedliche Strukturen und Eigenschaften aufweist. [7]

Der Weight-Decay ist ein weiter wichtiger Hyperparameter: Er wird in den Optimizer integriert, welcher wiederum in den Learning-Rate-Scheduler übergeben wird und für jede Epoche eine Gewichtsanpassung vornimmt. Als weiterer Parameter wurde das Momentum untersucht. Es kann die Geschwindigkeit des Optimierungsprozesses zusammen mit der Schrittgröße verbessern und die Wahrscheinlichkeit erhöhen, dass in weniger Trainingsepochen ein besserer Satz von Gewichten entdeckt wird. Außerdem wurde die Batch-Size in die Optimierung inkludiert. Die optimale Batch-Size ist maßgeblich für die Konvergenz des Modells. Zu große Bach-Sizes führen zwar zu genauen Schätzungen, konvergieren jedoch sehr langsam. Wird die Batch-Size hingegen zu klein gewählt, konvergiert das Modell zwar schnell, jedoch unter einer hohen Varianz der Zielfunktion. [8]

III. Hyperparameteroptimierung

Für die Hyperparameteroptimierung wurde ihm Rahmen der Studienarbeit der Entschluss gefasst, einen automatisierten Ansatz zu verwenden. Im Folgenden wird das gewählte Vorgehen detaillierter erläutert.

A. ClearML + Optimizer Optuna

Für die Hyperparameteroptimierung wurde das Framework ClearML verwendet, welches eine Open-Source Machine Learning Suite ist und ein automatisiertes Finetuning der Hyperparameter ermöglicht. Der hierfür zuständige Experimentmanager läuft auf einem eigenen ClearML Server. Die Wahl der Hyperparameterkombination wurde mittels der Optuna Strategie ermittelt. Diese, von Takiba et al. entwickelte Strategie, kombiniert die Konzepte der Bayes'schen Optimierung und evolutionsbasierten Optimierung, um möglichst effiziente Hyperparameterkombinationen aus den definierten Intervallen auszuwählen. Die Bayes'sche Optimierung setzt ein Gaus'sches Surrogatmodell an, welches eine Approximierung der mehrdimensionalen Fehlerfunktion der Hyperparameter ist und nach Bayes' Satz mit jedem neuen Experiment laufend aktualisiert wird. Das Minimum dieser Funktion stellt die optimale Hyperparameterkombination dar. Bei der evolutionsbasierten Optimierung wird ein Ranking der Hyperparametertuples erstellt. Hierbei werden die am schlechtesten performenden Tupels durch neue Hyperparameterkombinationen, die Korrelationen der Hyperparameter miteinbeziehe, ersetzt. Beide Optimierungsalgorithmen sind im Gegensatz zu einer Random Search oder Grid Search informativ und beschleunigen daher die Optimierung. [9]

B. Sensitivitäsanalyse

Für die folgende Sensitivitätsanalyse musste aufgrund des geringen Arbeitsspeichers der Instanz die Intervalle der Hyperparameter, die Anzahl der Epochen und Anzahl der Durchläufe stark eingegrenzt werden. Dies bremst leider auch den großen Vorteil vom Optimizer Optuna und schränkt die Aussagekraft der Ergebnisse ein. Für die Intervallbestimmung der Hyperparameter wurden die von Pytorch festgesetzten Default Werte als Richtlinie genommen und wie folgt festgelegt.

- Learning-Rate: [0,003-0,007, step size=0,002]
- Batch Size: [2,4,8]
- Momentum: [0,85-0,95, step size=0,05]
- Weight Decay: [0,0003-0,0007, step size = 0,0002]

Um eine Vergleichsbasis zwischen den verschiedenen Modellen und deren Hyperparameterkonfigurationen zu schaffen, wurde während des Trainings der Trainingsloss und die Average-Precision gespeichert. Diese sollen in den folgenden Untersuchungen die Performance der Modelle quantifizieren.

Im ersten Durchlauf wurden 5 Experimente mit jeweils einer Epoche durchgeführt, indem mittels der Optuna Strategie 5 Hyperparameterkombinationen innerhalb der definierten

Intervalle gewählt wurden. Hierbei war das Ziel, eine erste Aussage über gute und schlechte Hyperparameterwerte treffen zu können.

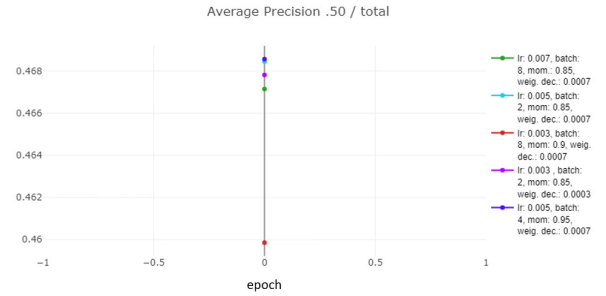

Fig. 2. Run 1 (aP): 5 Experimente über eine Epoche. Quelle: Eigene Darstellung

Die graphische Darstellung (Fig. 2) der jeweiligen Average-Precisions zeigt, dass 4 der 5 Experimente ziemlich ähnliche Ergebnisse liefern (0.467-0.469). Ein Experiment sticht mit der geringen aP von 0,459 heraus. Hieraus wurde schlussgefolgert, dass eine geringe Learning-Rate (0,003) in Kombination mit einer größeren Batch-Size von 8 eine langsamere Konvergenz innerhalb der ersten Epoche zur Folge hat. Dies lässt sich mit der Tatsache erklären, dass bei einer größeren Batch-Size die Modellparameter beim Gradientenabstiegsverfahren innerhalb einer Epoche weniger häufig aktualisiert werden und ein kleiner Lernschritt den Betrag der Gewichtsänderungen zusätzlich minimiert. Über das Momentum und den Weight-Decay konnten in diesem Durchlauf keine Rückschlüsse gezogen werden.

Als Nächstes wurde analysiert, wie sich die Leistung des Modells über mehrere Epochen entwickelt. Diese Untersuchung wurde anhand des initial besten (Modell 1) und schlechtesten (Modell 2) angestellt, um zu überprüfen, ob mehrere Epochen überhaupt einen Mehrwert in das Training bringen. Diese Frage war unter dem begrenzten Arbeitsspeicher besonders interessant, da ein Training über 6 Epochen nur für maximal 2 Experimente möglich war.

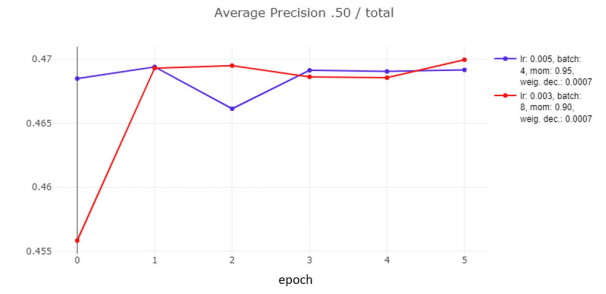

Fig. 3. Run 2 (aP): 6 Epochen für Modell 1 (blau) und Modell 2 (rot). Quelle: Eigene Darstellung

Bei Vergleich der beiden Experimente fällt auf (Fig. 3), dass die aP bei Modell 1 über die Epochen fast konstant bleibt. Modell 2 performt scheinbar nur in der ersten Epoche schlechter und schmiegt sich danach ebenso an ein denkbar globales Optimum an. Diese Ergebnisse bestätigen die Vermutungen des ersten Durchlaufs: Bei einer geringen Lernrate in Kombination mit großen Batches benötigt das Modell länger um gegen das Optimum zu konvergieren.

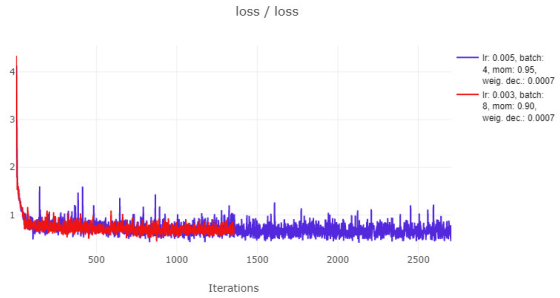

Fig. 4. Run 2 (loss): 6 Epochen für Modell 1 (blau) und Modell 2 (rot). Quelle: Eigene Darstellung

Des Weiteren wurde der Trainingsloss für jede Iteration innerhalb eines Batches beobachtet (Fig. 4). Dabei ist auffällig, dass der Trainingsloss für Modell 1 über die Iterationen eine deutliche größere Varianz aufweist. Diese Beobachtung lässt sich darauf zurückführen, dass eine höhere Anzahl an Bildern pro Iteration mehr Information und somit stabilere Lösungen mit sich bringen. [10]

Trotz unterschiedlicher Varianz konvergieren beide Modelle bereits nach weniger als 200 Iterationen gegen ihr Optimum.

Im nächsten Schritt wurde Modell 1 dupliziert und alle Hyperparameter bis auf das Momentum wurden in ihrer ursprünglichen Konfiguration belassen. Das Momentum wurde von 0,95 auf 0,5 gesenkt (Modell 3).

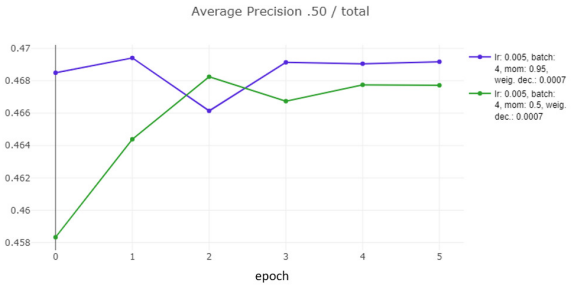

Fig. 5. Run 3 (aP): 6 Epochen für Modell 1 (blau) und Modell 3(grün). Quelle: Eigene Darstellung

Bei der Betrachtung der Average-Precisions aus (Fig. 5) zeichnet sich das folgende Bild ab: Im Durchschnitt über die Epochen weist Modell 1 die bessere aP auf, so auch in der ersten und letzten Epoche. Modell 3 startet hier in Epoche 1 vergleichsweise schlecht, verringert aber im Laufe der Epochen die Differenz zu Modell 1 signifikant. Somit kann schlussfolgernd festgehalten werden, dass die Gradientenvektoren mit einem höheren Momentum schneller in die richtigen Richtungen beschleunigt werden. Daher sollte man

das Momentum so groß wie möglich setzen, ohne während des Trainings Instabilitäten zu verursachen.

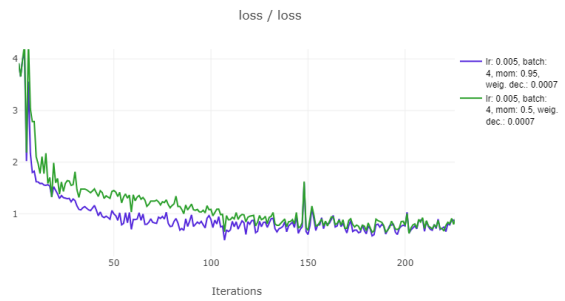

Fig. 6. Run 3 (loss): 6 Epochen für Modell 1 (blau) und Modell 3 (grün) Ergebnis. Quelle: Eigene Darstellung

Durch diese Modifikation kann anhand der graphischen Auswertung aus Fig. 6 die These aufgestellt werden, dass Modell 1 eine schnellere Konvergenz zum (vermuteten) globalen Optimum innerhalb einer Iterationsmenge sowie über den Verlauf multipler Epochen hinweg erreicht. Hier sei zu berücksichtigen, dass Fig. 6 nur ein Ausschnitt des gesamten Trainingsdurchlaufs ist, um die frühe Konvergenz im Trainingsloss in einem anderen Maßstab veranschaulichen zu können. Modell 3 hingegen befindet sich möglicherweise in einem lokalen Optimum, da sich hier ein stagnierendes Verhalten nach der 4. Epoche abzeichnet. (Fig. 6)

In einem letzten Durchlauf sollte der Einfluss des Weight-Decays genauer beleuchtet werden. Dazu wurde als Vergleichsbasis erneut das Modell 1 (blau) herangezogen und einerseits der Weight-Decay stark angehoben (Modell 4), andererseits stark gesenkt (Modell 5).

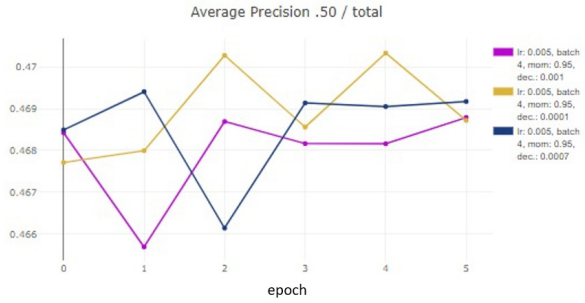

Fig. 7. Run 5 (aP): 6 Epochen für Modell 1 (blau), Modell 4 (gelb) und Modell 5 (lila). Quelle: Eigene Darstellung

Leider lässt diese graphische Untersuchung aus Fig. 7 keine Rückschlüsse über den Einfluss dieses Parameters zu. Die einzige Auffälligkeit ist, dass die aP von Modell 5 die besten Ergebnisse erreicht und für Epoche 1 bis 4 dauerhaft über der aP des Modell 4 liegt.

IV. Zusammenfassung

Im Abschluss an die vorherigen durchgeführten Modifikationen und die Interpretationen eben dieser lässt sich festhalten, dass die Performance eines neuronalen Netzes auch dann nachhaltig beeinflussbar ist, wenn nur die letzte Schicht eines vortrainierten Netzes mittels einer Hyperparameteroptimierung modifiziert wird (Fig. 8). Hier müssen potenzielle Relationen zu anderen Hyperparametern identifiziert und genauer beleuchtet werden. Es gilt jedoch zu beachten, dass nicht „der eine“ Weg zur Optimierung der Hyperparameter existiert, da es hier individuelle Problemstellungen, sowie einen vielfältigen Pool an Ansätzen gibt.

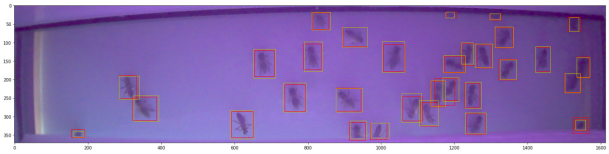

Fig. 8. Bounding Boxes (Predictions = rot vs. Ground Truth = gelb) nach der Hyperparameteroptimierung. Quelle: Eigene Darstellung

Die vorliegende Studienarbeit und dazugehörige Implementierung wurde von allen Teammitgliedern zu gleichen Teilen erarbeitet.

References

[1] “Torchvision object detection finetuning tutorial.” [Online]. Available: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

[2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” Dec 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[3] J. Hui, “Understanding feature pyramid networks for object detection (fpn),” Apr 2020. [Online]. Available: https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c

[4] Pytorch, “pytorch/vision.” [Online]. Available: https://github.com/pytorch/vision/tree/master/references/detection

[5] “Common objects in context.” [Online]. Available: https://cocodataset.org/#detection-eval

[6] “torch.utils.tensorboardÂ¶.” [Online]. Available: https://pytorch.org/docs/stable/tensorboard.html#:~:text=Writes%20entries%20directly%20to%20event%20files%20in%20the,it.%20The%20class%20updates%20the%20file%20contents%20asynchronously.

[7] “1.5. stochastic gradient descentÂ¶.” [Online]. Available: https://scikit-learn.org/stable/modules/sgd.html

[8] J. Nabi, “Hyper-parameter tuning techniques in deep learning,” Mar 2019. [Online]. Available: https://towardsdatascience.com/hyper-parameter-tuning-techniques-in-deep-learning-4dad592c63c8

[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” Jul 2019. [Online]. Available: https://arxiv.org/abs/1907.10902

[10] I. Kandel and M. Castelli, “The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset,” May 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405959519303455