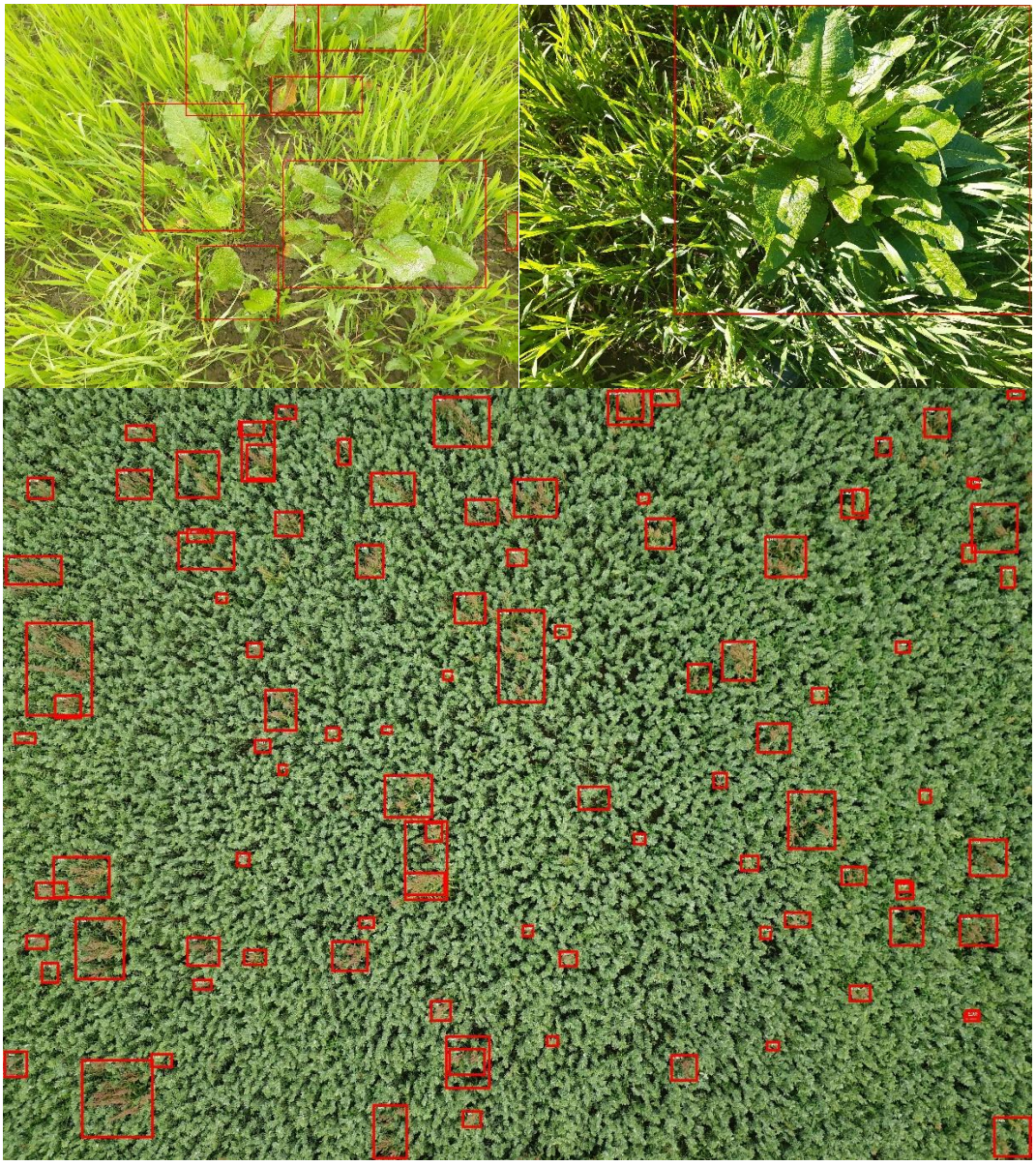
Ampfer Mampfer

Christian Krause

SFZ Südwürttemberg Standort Ochsenhausen



Kurzfassung

In meinem Projekt geht es darum, Unkraut (Ampfer) auf einem Getreideacker zu erkennen und mechanisch, ohne den Einsatz von chemischen Pflanzenschutzmitteln zu bekämpfen. Das Problem mit Ampfer ist, dass dieser sehr widerstandsfähig ist und auch weiterwächst, wenn man die Blätter abschneidet oder den Acker umpflügt. Es ist mir wichtig, dass die Bekämpfung des Ampfers mechanisch geschieht, da es ja bereits funktionierende chemische Pflanzenschutzmittel gibt, die aber schlecht für die Umwelt sind. Zurzeit fokussiere ich mich auf die Erkennung des Ampfers auf dem Acker. Um den Ampfer z. B. auf Drohnenbildern zu erkennen und zu lokalisieren, verwende ich Bilderkennung in Form von künstlichen Neuronalen Netzen, die ich mit Python (Pytorch) programmiere. Um den Ampfer dann auch zu bekämpfen, könnte man sich im weiteren Verlauf mit einer Methode zur autonomen mechanischen Bekämpfung von Ampfer beschäftigen. Das könnte z. B. ein Roboter, der Ampfer herauszieht, oder ein Anbau für Traktoren sein.

Inhalt

1: Idee

Wenn man als Landwirt zum Beispiel Getreide anbaut, kann Unkraut ein großes Problem sein. Konventionelle Landwirte gehen gegen Unkraut mit Pflanzenschutzmitteln, wie zum Beispiel Herbiziden, vor. Diese Pflanzenschutzmittel sind aber nicht gut für die Gesundheit und die Umwelt. Daher werden im ökologischen Landbau keine solchen Pflanzenschutzmittel verwendet. Hier wird gegen Unkraut mit mechanischen Verfahren vorgegangen, z. B. mit Striegeln oder Hackgeräten. Es gibt aber auch bestimmte, besonders hartnäckige Wurzelunkräuter, die man kaum mit mechanischen Geräten bekämpfen kann. Eine dieser Unkrautarten ist Ampfer, genauer gesagt der Stumpfblättrige Ampfer (Rumex obtusifolius) *[Q1]*. Er bildet Samen, die von Wind oder Wasser getragen werden und im Boden 40 bis 50 Jahre überleben und keimfähig bleiben. Im Winter stirbt der überirdische Teil des Ampfers und im Frühling erfolgt der Neuaustrieb aus der großen Pfahlwurzel aus bis zu 20 cm Tiefe *[Q2]*. Die Probleme des Ampfers sind, dass er die Kulturpflanzen verdrängt und die Ernte verunreinigt, da dann auch Ampfersamen ins Getreide gelangen können. Man kann Ampfer jedoch nicht einfach wie viele andere Unkräuter bekämpfen, indem man den Acker umpflügt oder grubbert, da die Ampferwurzel noch anwachsen kann – selbst, wenn sie zerteilt ist (siehe Abbildung 1).

Abbildung 1: Ampfer, der aus einer Wurzel, die untergepflügt wurde, wieder ausgetrieben ist

Deshalb ist meine Projektidee, eine Maschine („Ampfer Mampfer“) zu entwickeln, die Ampfer auf einem Getreideacker gezielt mechanisch bekämpfen kann. Aktuell beschäftige ich mich mit der Software, die notwendig ist, um Ampfer auf Bildern von Drohnen und aus der Nähe zu lokalisieren und die Position des Ampfers zu berechnen.

2: Bilderkennung

Um Ampfer auf Bildern zu erkennen habe ich Software implementiert, die auf der Grundlage von Neuronalen Netzwerken arbeitet. Im Folgenden skizziere ich die relevanten theoretischen Grundlagen für diese Anwendung.

2.1: Neuronale Netzwerke

Künstliche Neuronale Netzwerke sind eine Form der Künstlichen Intelligenz, die dem Vorbild aus der Natur, also dem Nervensystem von Lebewesen, nachempfunden sind.

2.1.1: Grundlagen

Künstliche Neuronale Netzwerke haben Neuronen, die mit Gewichtungen miteinander verbunden sind. Die einzelnen Neuronen sind folgendermaßen aufgebaut:

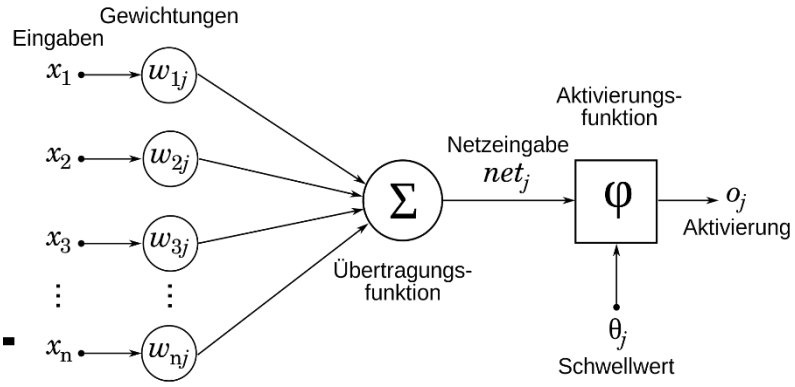


Abbildung 2: Schema für ein künstliches Neuron

In Abbildung 2 sind die Eingaben x (x_1 bis x_n) des Neurons zu sehen, die mit der jeweiligen Gewichtung w (w_1 bis w_n) gewichtet werden. Diese Eingaben werden dann von einer Übertragungsfunktion (z. B. Summe) zusammengefasst. Dieser Wert wird an die Aktivierungsfunktion (z. B. die Sigmoid-Funktion) übergeben, die einen Ausgabewert (Aktivierung) erstellt, der dann anderen Neuronen als Eingabe übergeben wird.

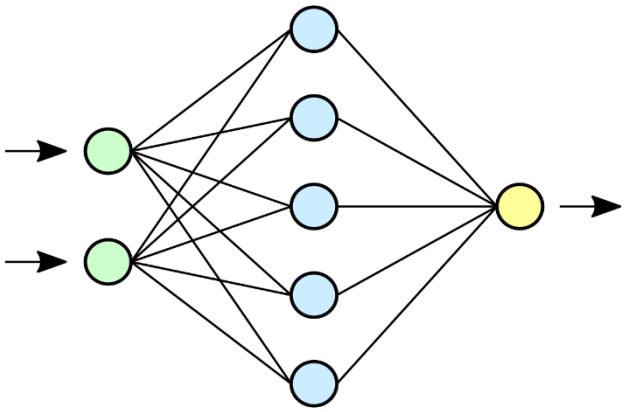


Abbildung 3: Vereinfachte Darstellung eines Künstlichen Neuronalen Netzwerks

Um ein Netzwerk zu formen, werden die Neuronen in verschiedenen Schichten miteinander verbunden (siehe Abbildung 3). Als erste Schicht gibt es eine Input-Schicht (grün), bei der die Neuronen von „außen“ eine Eingabe bekommen. Danach folgen eine oder mehrere versteckte Schichten (blau) und als Letztes die Output-Schicht (gelb), bei der die Outputs der Neuronen ausgegeben werden.

2.1.2: Lernen

Um Künstliche Neuronale Netzwerke zu trainieren, müssen die für die Aufgabe optimalen Gewichtungen w gefunden werden [*Q3*]. Hierzu wird als Erstes der Fehler der Gewichtungen w für einen zufälligen Satz von Ausgangswerten berechnet. Dafür wird ein Eingabemuster erstellt, das dann durch das Netz propagiert wird. Die Ausgaben des Netzwerks werden dann mit den gewünschten Ausgaben verglichen. Der Fehler wird mit dieser Fehlerfunktion berechnet:

$$E = \frac{1}{2}\sum_{i=1}^{n}(t_i - o_i)^2$$

E: Fehler

n: Anzahl der Muster, die dem Netz als Input gegeben wurden

t_i: gewünschte Ausgabe/Zielwert (*target*)

o_i: die errechnete Ausgabe des Netzes (*output*)

Der Faktor $\frac{1}{2}$ wird nur zur Vereinfachung der später erfolgenden Ableitung hinzugenommen.

Das Ziel des Trainings ist es, die Fehlerfunktion zu minimieren, wobei meistens aber nur ein lokales Minimum erreicht wird. Diese Minimierung wird beim Backpropagation-Verfahren (anschaulich

erklärt in [*Q4*], [*Q3*]) durch die Veränderungen der Gewichte w, von denen die Ausgabe des Netzes abhängig ist, erreicht.

Die Ausgabe o_j eines Neurons ist definiert durch:

$$o_j = \varphi(net_j)$$

Die Netzeingabe net_j lässt sich wiederum definieren durch:

$$net_j = \sum_{i=1}^{n} x_i w_{ij}$$

φ: Aktivierungsfunktion des Neurons (z. B. Sigmoid)

n: Anzahl der Eingaben

x_i: Eingabe i

w_{ij}: Gewichtung zwischen Eingabe i und Neuron j

Damit kann man dann die Auswirkung jedes Gewichts auf den Fehler bestimmen, indem man die Fehlerfunktion E unter der Verwendung der Kettenregel partiell ableitet:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Aus den einzelnen Termen kann jetzt die Formel zur Veränderung der Gewichte berechnet werden. Die Herleitung hängt von zwei Fällen ab:

1. Wenn das Neuron direkt in der Ausgabeschicht liegt, dann hat es direkten Einfluss auf die Ausgabe.
2. Wenn das Neuron aber in einer versteckten Schicht liegt, dann kann die Anpassung nur indirekt berechnet werden.

$$\delta_j = \begin{cases} \varphi'(net_j)(o_j - t_j) & falls\ j\ ein\ Ausgabeneuron\ ist \\ \varphi'(net_j) \sum_k \delta_k w_{jk} & falls\ j\ ein\ verdecktes\ Neuron\ ist \end{cases}$$

$$\Delta w_{ij} = -n \frac{\partial E}{\partial w_{ij}} = -n \delta_j o_i$$

Δw_{ij}: Änderung des Gewichts w_{ij}, der Verbindung von Neuron i zu Neuron j

n: Lernrate, um die Stärke der Veränderungen der Gewichte zu kontrollieren

δ_j: Fehler des Neurons j, entspricht $\frac{\partial E}{\partial net_j}$

t_j: Soll-Ausgabe des Ausgabeneurons j

o_i: Ausgabe des verdeckten (oder eingabe) Neurons i

o_j: Tatsächliche Ausgabe des Ausgabeneurons j

k: Index der nachfolgenden Neuronen von Neuron j

Jetzt können die Gewichte verändert werden:

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij}$$

w_{ij}^{neu}: der neue Wert des Gewichts w_{ij}

w_{ij}^{alt}: der alte Wert des Gewichts

Δw_{ij}: die vorher berechnete Änderung des Gewichts (basierend auf dem alten Wert des Gewichts)

Wenn man diese Berechnungen immer wieder wiederholt, wird die Fehlerfunktion immer kleiner (d. h. das Netz wird immer genauer).

2.1.3: Convolutional Neuronal Networks

Convolutional Neuronal Networks (**CNN**) [*Q5*] werden hauptsächlich für Bildanalyse (z. B. Klassifizierung) verwendet. Sie sind in verschiedenen Schichten aufgebaut:

Als erste Schicht wird ein **Convolutional Layer** verwendet. Die Aktivität jedes Neurons wird über diskrete Faltung berechnet (in der Mathematik auch als Konvolution (von lat. convolvere „zusammenrollen“) bezeichnet – daher der engl. Begriff convolutional). Dafür wird eine Faltungsmatrix (Filter-Kernel) von geringer Größe über die Eingabe bewegt. Die Summe der gewichteten Pixel des entsprechenden Bildausschnitts ist dann die Eingabe des Neurons.

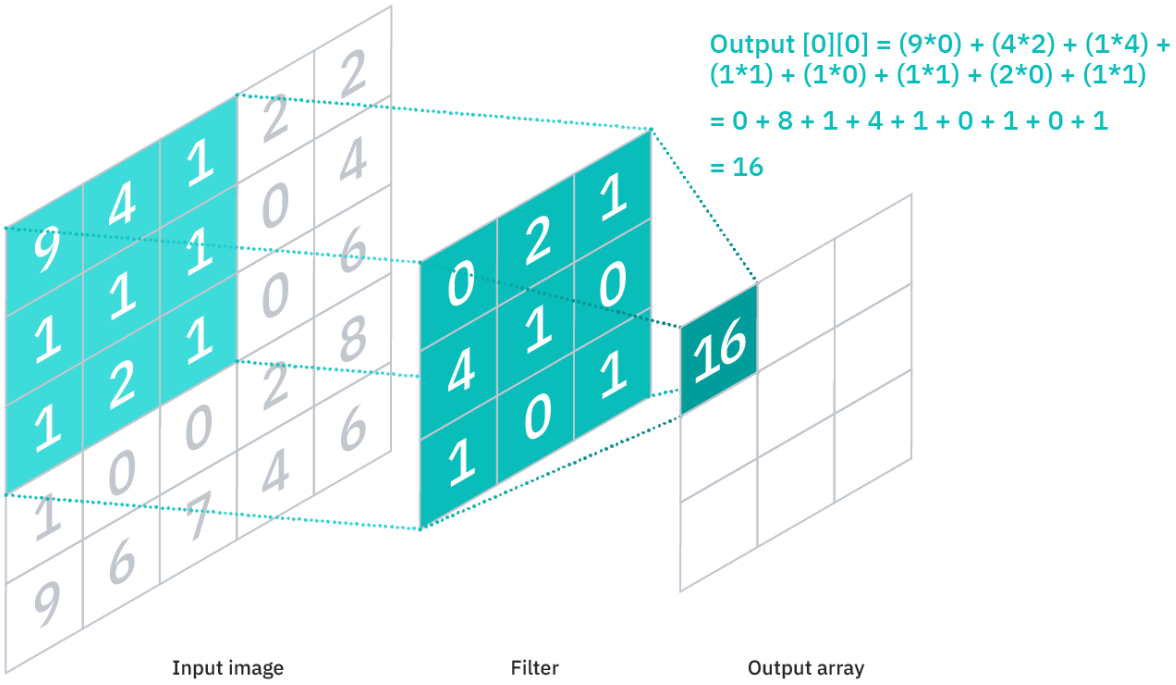


Abbildung 4: Schema convolution ("Faltung")

Ein großer Vorteil des **Convolutional Layers** ist, dass alle Neuronen die Gewichtungen im Filter-Kernel teilen. Es müssen also nur sehr wenige Gewichte berechnet werden.

Danach folgt ein **Pooling Layer**, in dem das Bild verkleinert, also auf die wesentlichen Informationen reduziert wird. Am weitesten verbreitet ist das Max-Pooling, bei dem aus jedem 2 x 2 (oder auch 5 x 5

etc.) Quadrat im vorherigen Layer nur die Aktivität des aktivsten Neurons (daher „Max-“) übernommen wird.

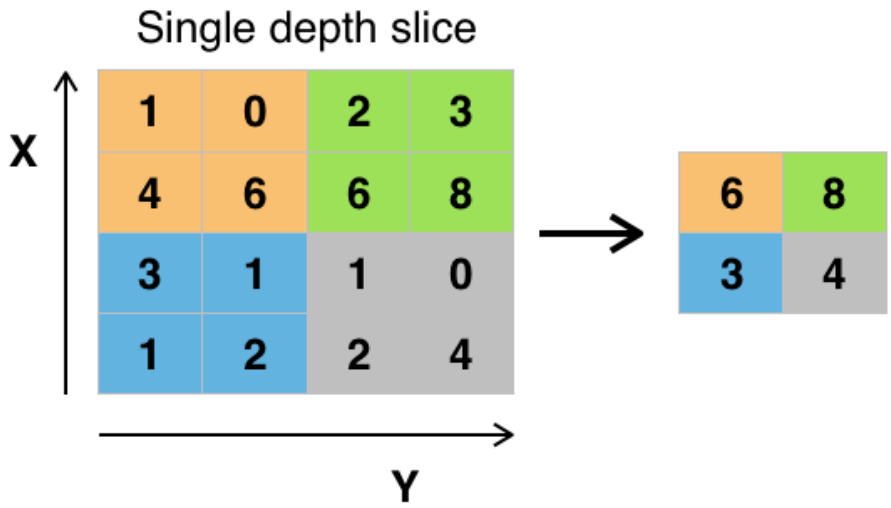


Abbildung 5: Max pooling mit einem 2 x 2 Filter

Die Kombination aus **Convolutional-** und **Pooling Layer** kann sich beliebig oft wiederholen.

Am Schluss folgt ein (oder mehrere) **Fully-connected Layer**, bei dem alle Neuronen mit jedem Neuron aus dem vorherigen Layer verbunden sind. Die Anzahl der Neuronen im letzten Layer entspricht bei einer Klassifizierung meistens der Anzahl der Klassen, die das Netz unterscheiden soll.

2.1.4: Region Based Convolutional Neuronal Networks

Um auch die Position von Objekten auf Bildern zu erfassen, werden **Region Based Convolutional Neuronal Networks** (R-CNN) [*Q6*] verwendet. Es gibt drei Generationen dieses Netzwerktyps, von denen ich die letzte eingesetzt habe:

2.1.4.1: R-CNN

Bei dem originalen R-CNN werden als Erstes 2000 Bereiche, in denen sich ein Objekt befinden könnte, mithilfe eines Selective Search Algorithmus [*Genaueres in Q7*] ermittelt. Dieser Algorithmus ermittelt zuerst viele unterschiedliche mögliche Regionen, basierend auf der Intensität der Pixel. Dann werden ähnliche Regionen rekursiv zu größeren zusammengefügt. Mit diesen Regionen werden dann die finalen Regions-Vorschläge, in denen sich ein Objekt befinden könnte, berechnet. Diese Regionen werden dann jeweils von einem CNN klassifiziert. Außerdem gibt das CNN vier Werte aus, die anzeigen, wie die Box verändert werden muss, sodass sie so gut wie möglich auf das Objekt passt. Die Probleme bei dieser Methode sind einerseits, dass sie sehr langsam ist und andererseits, dass der Selective Search Algorithmus ein fester Algorithmus ist und nicht mit dem Netz mitlernt, was zu schlechten Regions-Vorschlägen führen kann.

2.1.4.2: Fast R-CNN

Bei dem Fast R-CNN ist die Herangehensweise ähnlich zu der des R-CNN, am Anfang werden die Regions-Vorschläge mithilfe eines Selective Search Algorithmus ermittelt. Diese werden dann aber nicht einzeln von dem CNN klassifiziert, sondern das ganze Bild wird dem CNN als Input gegeben. Auf der errechneten Feature-Map (Ausgabe eines Convolutional Layers) werden dann die Regions-Vorschläge lokalisiert und von einem Fully connected Layer klassifiziert, nachdem sie von einem Pooling Layer auf eine feste Größe gebracht wurden. Dieser Netztyp ist deutlich schneller als der vorherige, da die convolution-Operation nur einmal pro Bild durchgeführt wird.

2.1.4.3: Faster R-CNN

Bei dem Faster R-CNN wird im Gegensatz zu den vorherigen Netzen kein Selective Search Algorithmus verwendet, sondern es wird gleich am Anfang mit einem CNN eine Feature-Map des Bildes erstellt. Daraufhin werden anhand von dieser Feature-Map mit einem separaten Netzwerk die

Regions-Vorschläge ermittelt. Diese Vorschläge werden dann mit einem Pooling Layer auf ein einheitliches Format gebracht und anschließend klassifiziert. Außerdem wird bestimmt, wie die Boxen, die die Positionen der Objekte angeben, verschoben werden sollen. Das Faster R-CNN ist deutlich schneller als seine Vorgänger, es ist sogar Object-Detection in Echtzeit möglich.

3: Umsetzung

Um die Ampferpflanzen auf Bildern zu erkennen, werden Künstliche Neuronale Netzwerke verwendet. Die Netzwerke habe ich in Python mit Pytorch (torchvision) programmiert. Die Programme sind in meinem Github Repository zu finden:
https://github.com/Bergschaf/Ampfer_Mampfer_public.git

3.1: Klassifizierung

Mein erstes Ziel war es, mithilfe eines CNNs Ampferbilder, die aus geringer Höhe (ca. 1 m) aufgenommen wurden, zu klassifizieren. Dafür habe ich als Erstes über einen Zeitraum von zwei Wochen auf verschiedenen Äckern Trainingsbilder von Ampfer und Getreide aufgenommen (Insgesamt 673 Trainingsbilder und 98 Testbilder). Testbilder sind Bilder, mit denen das Netz nicht trainiert wird und die zur Evaluation des Netzes während oder nach dem Training verwendet werden.

Abbildung 6: Getreide

Abbildung 7: Ampfer

Um das Netz zu trainieren sind Trainingsbilder nötig, die ein Mensch gelabelt hat. Das bedeutet, er hat für jedes Bild entschieden, was die Ziel-Ausgabe des Netzes für dieses Bild sein soll (also ob auf dem Bild ein Ampfer ist oder nicht). Anhand von diesen Bildern kann das Netz lernen, auch andere Bilder zu klassifizieren.

Die Trainingsbilder habe ich mithilfe eines selbstgeschriebenen Python-Programms gelabelt. Die Bilder wurden immer einzeln angezeigt und der Benutzer kann mit einer Eingabe in die Konsole bestimmen, ob das Bild als Ampfer- oder Getreidebild (ohne Ampfer) gespeichert werden soll. Um zu erkennen, welche Bilder wie gelabelt wurden, werden die Bilder nach der Klasse benannt. Mit diesen Trainingsbildern wurde dann ein CNN mit der Python-Bibliothek Pytorch trainiert [*anhand von Q8*]. Das Netz ist folgendermaßen aufgebaut:

Als ersten Layer verwende ich einen Convolutional Layer mit einer Filter-Kernel-Größe von 5x5 und dann ein Pooling Layer mit der ReLu-Funktion (Rectified Linear Unit) [*Q9*] als Aktivierungsfunktion. Diese Kombination wiederholt sich dreimal, sodass das Netz am Ende vier Convolutional Layer besitzt. Am Ende folgen zwei fully-connected Layer, der erste mit einer ReLu Aktivierungsfunktion, der zweite mit einer Sigmoid Aktivierungsfunktion und zwei Output-Neuronen, von denen eines für „Ampfer“ und eines für „keinen Ampfer“ steht. Um die passende Neuronenanzahl der Layer zu ermitteln, habe ich Tests mit verschiedenen Konfigurationen der Layer gemacht.

Dieses Netz wurde anschließend mit den Trainingsbildern trainiert. Auf den Testbildern, die das Netz noch nie „gesehen“ hat, wurde nach dem Training dann eine Genauigkeit von ca. 90 % erreicht. Dies könnte man durch mehr Training oder leichten Änderungen am Netz bestimmt noch weiter optimieren,

aber das Problem an dieser Methode ist, dass man nicht feststellen kann, wo sich der Ampfer im Bild befindet. Dies ist für die Ampferbekämpfung aber essenziell. Deshalb habe ich zu einer anderen Methode gewechselt, die im Folgenden beschrieben wird.

3.2: Object Detection

Object Detection (auch Objekterkennung) bedeutet, Objekte auf Bildern zu erkennen und zu identifizieren.

Um die Position von Ampfer auf Bildern zu bestimmen, verwende ich ein pre-trained (bereits auf Object Detection trainiertes) Faster R-CNN von Pytorch. Da diese Netze sehr leistungsintensiv sind und viele Bibliotheken benötigen, die sogar teilweise nicht unter Windows erhältlich sind, verwende ich zum Ausführen meiner Programme ab jetzt Google Colaboratory. Dabei handelt es sich um einen Dienst von Google, der kostenlose CPU- und GPU-Leistung z. B. für maschinelles Lernen zur Verfügung stellt. Das Netz habe ich [*anhand von Q10*] mit Pytorch implementiert. Um das Netz zu trainieren, mussten zuerst Trainingsbilder mit Labeln erstellt werden. Das bedeutet, dass von Menschen auf den Trainingsbildern die Positionen der gesuchten Objekte bestimmt werden. Anhand dieser Angaben kann das Netz dann lernen. Die Label für die Trainingsbilder habe ich mithilfe des Tools LabelImg (https://github.com/tzutalin/labelImg) erstellt, einem kostenlosen Programm, mit dem man in Bildern die Positionen der gesuchten Objekte einzeichnen kann. Diese werden dann automatisch nach dem gewählten Standard abgespeichert, hier Pascal VOC (Visual Object Classes) [*Genaueres siehe Q11*], also als .xml Datei. Nach dem Training kann das Netz Ampfer auch auf Testbildern, in denen der Ampfer nicht sehr offensichtlich ist, mit hoher Genauigkeit erkennen. Es wird aber auch erkannt, wenn kein Ampfer auf dem Bild ist, d. h. es werden keine Ampfer auf Bildern ohne Ampfer erkannt. Um die erkannten Positionen zu visualisieren werden rote Rechtecke (Boxen) von einem Python Skript anhand der Ausgabe des Netzes eingezeichnet.

Abbildung 8: Ampfer

Abbildung 9: Ampfer

Um im nächsten Schritt die Position aller Ampfer auf einem Acker zu bestimmen, verwende ich eine Drohne, die Aufnahmen von dem Acker in einer Höhe von ca. 10 m bis 20 m macht. Zu dem Zeitpunkt dieser Aufnahmen war der Ampfer schon fast reif. Deshalb sieht man den Ampfer auf den Drohnenbildern orangebraun.

Abbildung 10: Ampfer reif

Abbildung 11: Ampfer reif (Ausschnitt aus einem Drohnenbild)

Mit diesen Bildern habe ich wieder mit Google Colaboratory ein Faster R-CNN trainiert. Um das Netz zu trainieren, werden jedoch viele Trainingsbilder mit Labeln benötig. Da es sehr viel Arbeit wäre, alle Bilder mit LabelImg manuell zu labeln, habe ich begonnen das Netz zuerst mit wenigen Trainingsbildern zu trainieren. Dann habe ich dieses Netz verwendet, um bereits einen Teil der Ampfer in den noch nicht gelabelten Trainingsbildern zu lokalisieren. Die Ausgabe des Netzes wird dann nach dem Pascal VOC Standard in eine .xml Datei geschrieben. Diese Dateien kann man dann zusammen mit den entsprechenden Bildern in LabelImg öffnen. So kann man die Ausgabe des Netzes manuell korrigieren bzw. ergänzen. Nach dem Training des Netzes sind auch die Ergebnisse auf den Testbildern schon sehr gut.

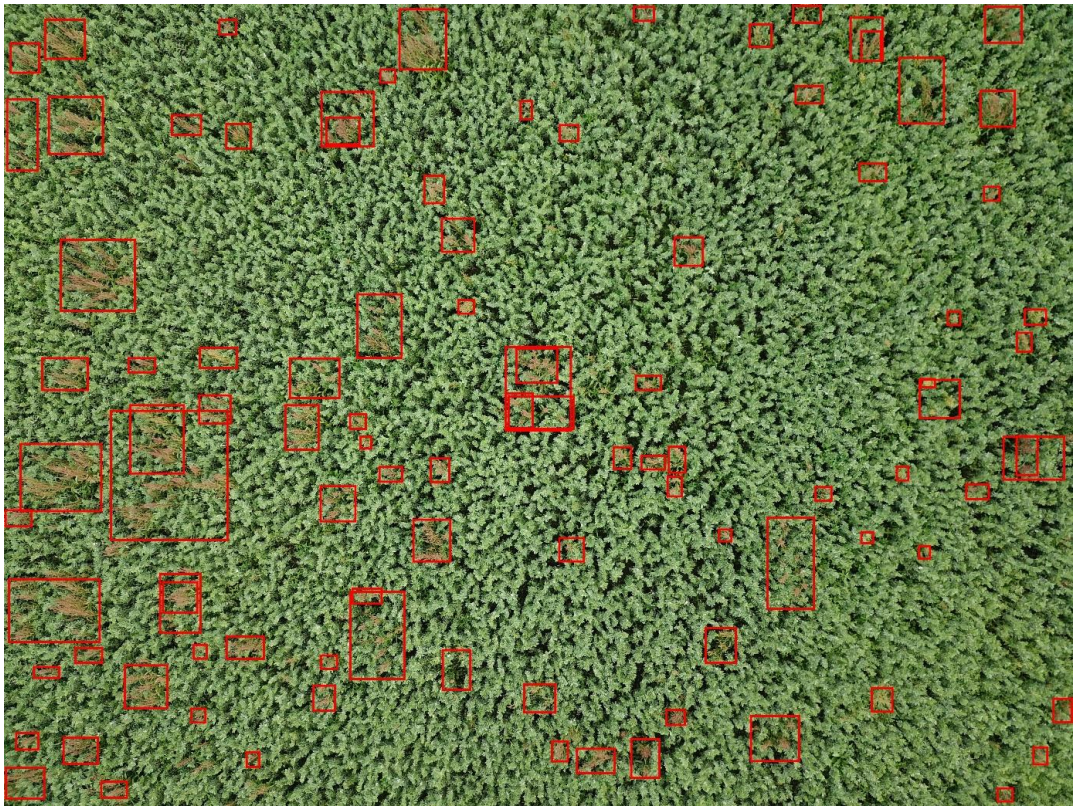


Abbildung 12: Bild vom Acker mit einer Drohne aufgenommen, Ampfer von dem Faster R-CNN lokalisiert (Höhe: 15,5m)

Wie man in Abbildung 12 sieht, überlappen sich manche der erkannten Regionen stark. Um diese Überlappung zu entfernen habe ich einen Algorithmus in Python implementiert, der für alle Boxen

bestimmt, wie weit sie sich mit jeder anderen in x- und y-Richtung überlappen. Anhand dieser Werte kann die Fläche der Überlappung bestimmt werden. Wenn diese Fläche mehr als 80 % der Fläche der kleineren Box ausmacht, dann wird die kleinere Box gelöscht, da sie fast vollständig in der großen Box enthalten ist.

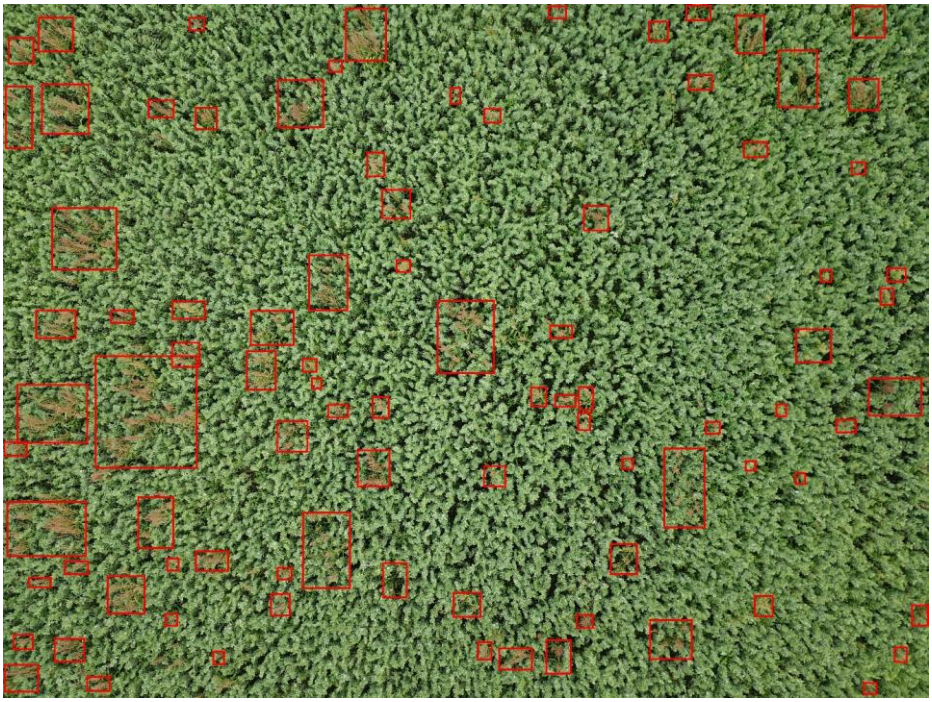


Abbildung 13: Abbildung 12 ohne Überlappungen

3.3: Positionsbestimmung

3.3.1: Verfahren

Da in den Metadaten der Bilder die GPS-Position und Höhe der Drohne (DJI Mavic Pro) zum Zeitpunkt der Aufnahme enthalten sind, kann man die GPS-Position der Ampfer bestimmen, wenn man die Position der Ampfer auf dem Bild kennt. Die Metadaten der Bilder lese ich mithilfe der Python-Bibliothek exif aus den Bilddateien aus. Um die GPS-Position des Ampfers zu bestimmen, werden zunächst zwei Werte ermittelt:

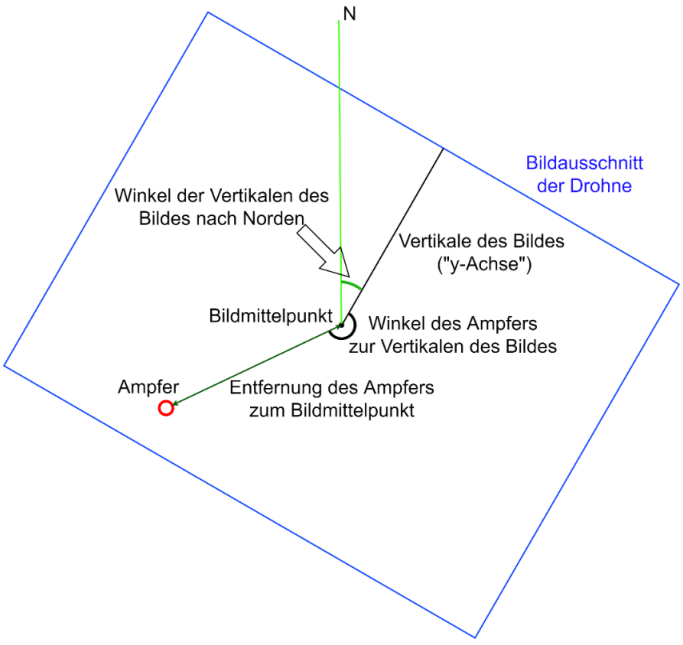


Abbildung 14: Grafik 1 zur Besimmtung der Ampferposition

Als Erstes soll der Winkel von der Verbindung zwischen dem Ampfer und dem Mittelpunkt nach Norden bestimmt werden. Dafür muss zunächst der Winkel zwischen der Verbindung vom Ampfer zum Mittelpunkt und der Horizontalen („x-Achse“) des Bildes ermittelt werden. Dafür verwende ich die trigonometrische Funktion Arkustangens [*Q12*], die in der „atan2“ Funktion der „math“ Python Bibliothek [*Q13*] bereits so implementiert ist, dass aus den kartesischen x- und y-Koordinaten eines Punktes die Winkelkoordinate der Polarkoordinaten [*Q14*] dieses Punktes im Bogenmaß berechnet werden kann. Da sich der berechnete Winkel auf die x-Achse des Koordinatensystems bezieht, wird auf den errechneten Winkel $\frac{1}{2}\pi$ addiert, um den Winkel zur y-Achse (schwarz), also der Vertikalen des Bildes, zu ermitteln. Da in den Metadaten des Bildes der Winkel der Drohne in Richtung Norden (grün) zum Aufnahmezeitpunkt des Bildes gespeichert ist, wird dieser Winkel einfach auf den errechneten Winkel addiert, um den Winkel in Richtung Norden zu erhalten.

Als Nächstes muss die Entfernung des Ampfers zum Bildmittelpunkt in Metern bestimmt werden. Dafür wird zunächst ein Maßstab berechnet, welcher realen Distanz in Metern ein Pixel im Bild entspricht.

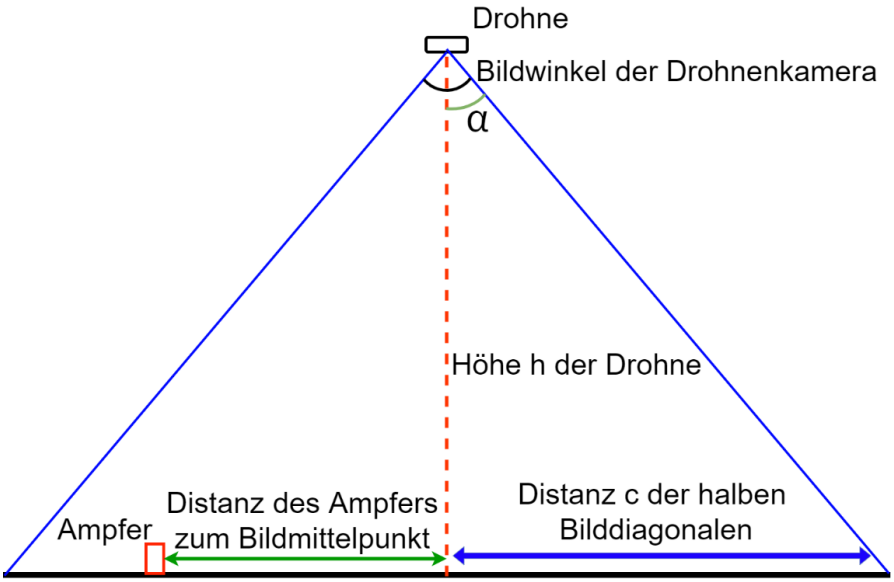


Abbildung 15: Grafik 2 zur Besimmtung der Ampferposition

$$\tan \alpha = \frac{c_{real}}{h}$$

$$c_{real} = h \times \tan \alpha$$

α: Hälfte des Bildwinkels der Drohnenkamera (hier 78,8° / 2 = 39,4°)

h: Höhe der Drohne

c_{real}: Reale Distanz in Metern der Hälfte der Bilddiagonalen

So kann für jedes Bild die Distanz der halben Bilddiagonalen c in Metern bestimmt werden.

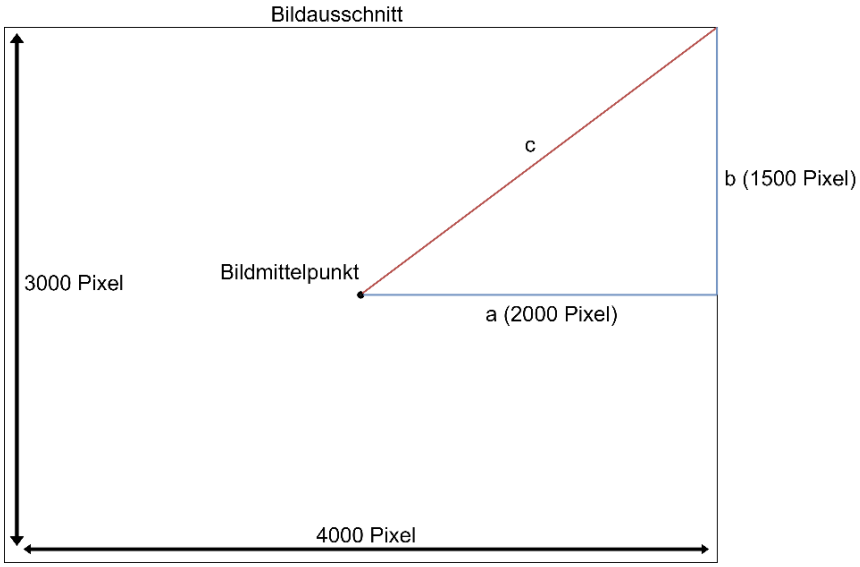


Abbildung 16: Grafik 3 zur Besimmtung der Ampferposition

Um einen Maßstab zu erstellen kann jetzt mithilfe des Satzes des Pythagoras bestimmt werden, wie lang die halbe Bilddiagonale c im Bild in Pixeln ist.

$$a^2 + b^2 = c^2$$

$$\sqrt{a^2 + b^2} = c$$

$$\sqrt{2000^2 + 1500^2} = 2500$$

Mit diesen Werten kann für jedes Bild ein Maßstab errechnet werden, welcher Distanz in Metern ein Pixel im Bild entspricht.

$$\frac{c_{real}}{2500} = Umrechnungsfaktor\ in\ Metern\ pro\ Pixel$$

c_{real}: Die vorher berechnete reale Distanz der halben Bilddiagonalen in Metern

Mit dieser Anwendung des Satzes des Pythagoras kann auch die Distanz in Pixeln eines Ampfers, dessen Position in Pixelkoordinaten auf dem Bild bekannt ist, zum Mittelpunkt berechnet werden. Mithilfe des Umrechnungsfaktors kann nun die Distanz des Ampfers zum Bildmittelpunkt in Metern ermittelt werden.

Mit der beschriebenen Methode kann für einen Ampfer, dessen Position auf dem Bild bekannt ist, die reale Position relativ zu der Position der Drohne bestimmt werden. Da die GPS-Position der Drohne bekannt ist, kann mithilfe der inverse_haversine Funktion der haversine Python-Bibliothek [*Q15*] die Position des Ampfers anhand der Position der Drohne, der Entfernung von dieser Position (die Entfernung vom Bildmittelpunkt) und des Winkels nach Norden bestimmt werden.

Die reale Position der Ampfer, deren Position auf einem Bild bekannt ist, wird dann mithilfe der Position und Höhe der Drohne zum Zeitpunkt der Aufnahme des Bildes ermittelt.

3.3.2: Genauigkeit

Um die Genauigkeit dieses Algorithmus zu überprüfen habe ich mehrere Bilder vom Schulhof in Ochsenhausen aufgenommen und mit dem Algorithmus die Position des Torpfostens als Referenzpunkt ermittelt. Diese Positionen habe ich auf Google Maps eingezeichnet.

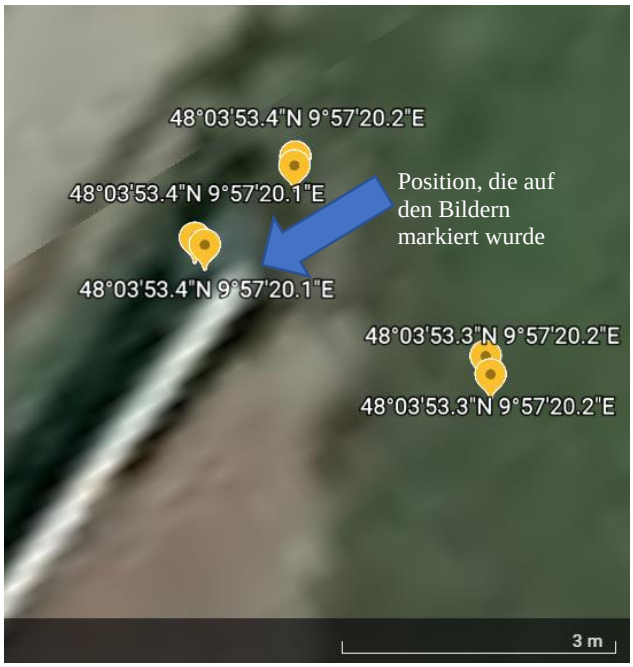


Abbildung 17 Google Earth

Die Bilder wurden aus einer Höhe von ca. 20 m bis 100 m aufgenommen, um auch die Ungenauigkeit auf größeren Höhen zu ermitteln. Auf einer geringen Höhe wie 20 m ist die Positionsbestimmung also recht genau (ca. 1 m Ungenauigkeit). Um die Verzeichnung [*Q17*] des Weitwinkelobjektivs der Drohne als Fehlerfaktor auszuschließen wurde die Positionsbestimmung für jedes Bild einmal mit Objektivkorrektur und einmal ohne Objektivkorrektur durchgeführt. Die Objektivkorrektur kann mithilfe von Adobe Lightroom Classic für die Bilder nachträglich berechnet werden. Da die Positionen der Bilder mit und ohne Objektivkorrektur immer sehr eng zusammen liegen, sieht man, dass die Verzeichnung keinen großen Fehler verursacht. Deshalb vermute ich, dass die Ungenauigkeit durch die Ermittlung der GPS-Position und der Höhe zustande kommt welche bis zwölf Meter über Ultraschall und darüber über den Luftdruck gemessen wird. Ein weiterer Faktor könnte sein, dass die Drohne etwas geneigt ist, was aber durch ein Gimbal ausgeglichen wird. Die Neigung der Drohne und des Gimbals kann man aus den Metadaten der Bilder auslesen, ich habe diese Angaben aber noch nicht berücksichtigt, da die Drohne meistens kaum geneigt ist. Um die Ungenauigkeit der Höhe zu minimieren, könnte man die Bilder aus niedrigeren Höhen aufnehmen, was allerdings zu deutlich mehr Bildern führen würde.

4: Ergebnisse

Auf Bildern, die aus niedriger Höhe (ca. 1 m) aufgenommen wurden, können jetzt mit dem ersten Faster R-CNN die Ampferpflanzen erkannt und auf dem Bild lokalisiert werden. Um die Genauigkeit dieses Netzes zu testen, verwende ich die **mAP** (mean Average Precision) [*Q16*]. Das Netz erreicht auf Testbildern einen Wert von ca. 91 %, was selbst für diese eher einfache Aufgabe mit nur einer Klasse sehr gut ist. Um einen Überblick über den ganzen Acker zu erreichen, können mit dem zweiten Faster R-CNN Ampfer auf Drohnenbildern aus ca. 10m bis 20m Höhe lokalisiert werden. Dieses Netz erreicht auf Testbildern eine mAP von ca. 24 %. Diese eher niedrige Erkennungsrate kommt daher, dass auf diesen Bildern sehr viele, sehr kleine Objekte gefunden werden sollen. So sind selbst Ungenauigkeiten von wenigen Pixeln ausschlaggebend. Eine weitere Ursache für diese Ungenauigkeiten kann aber auch sein, dass die Position des Ampfers nicht ganz eindeutig ist. So ist es zum Beispiel möglich, dass ein Ampfer so markiert ist, dass in der Box um den Ampfer herum noch etwas Platz ist, während bei einem anderen Ampfer kein Platz mehr zwischen Box und Ampfer ist. Da bei der mAP die Ausgabe des Netzes mit den Labeln der Bilder verglichen wird, werden eigentlich richtige Boxen aufgrund von leichter Inkonsistenz der Label als falsch gewertet.

Diese Methoden zur Positionsbestimmung der Ampfer ich in einem Python-Programm auf Google Colaboratory umgesetzt. Man kann Drohnenbilder von einem Acker hochladen und das Programm

(Dauer: ca. 8 s pro Bild) gibt die ermittelten Ampferpositionen in einer .json oder in einer .kml Datei, die man in Google Earth öffnen kann, aus.

5: Ausblick

Um die Genauigkeit der Lokalisierung der Ampfer auf den Bildern zu erhöhen, könnte man die Netze auch mit noch mehr Trainingsbildern trainieren. Eine andere Möglichkeit wäre, noch andere Neuronale Netzwerktypen auszuprobieren und zu optimieren.

Außerdem kann man sich damit beschäftigen, wie die Bilderfassung am Ende funktionieren soll, es könnte z. B. eine autonom fliegende Drohne eingesetzt werden, die die Bilder automatisch hochlädt.

Der nächste Schritt in diesem Projekt nach der Softwareentwickleung ist, einen funktionierenden „Ampfer Mampfer“ zu bauen. Dieser Roboter sollte zu den ermittelten Positionen der Ampferpflanzen fahren können und diese vernichten. Er sollte auf jeden Fall eine Kamera haben, sodass er den Ampfer zur Bekämpfung genauer lokalisieren kann, da die Positionsbestimmung über die Drohenbilder und die verfügbaren Drohnendaten nicht exakt ist. Das Problem an einem Roboter, der über den Acker fährt, ist, dass dieser Roboter einerseits möglichst wenig Getreide zerstören sollte, aber trotzdem schwer genug sein sollte und genug Kraft aufbringen sollte, um den Ampfer zu bekämpfen. Eine Möglichkeit wäre, die Ampferwurzeln mit einer Fräse zu vernichten. Dafür braucht man aber schon einen eher massiven und schweren Roboter, der natürlich auch viel Getreide einfach plattdrücken würde. Eine mögliche Lösung für dieses Problem wäre, wenn der „Ampfer Mampfer“ kein Roboter, sondern ein Traktoranbau wäre. Um mit dem Traktor kein Getreide zu zerdrücken, könnte man im Acker Fahrspuren lassen, in denen gar kein Getreide angesät wird. Dieses Verfahren wird schon in der konventionellen Landwirtschaft angewendet. So können Pflanzenschutzmittel in der Wachstumsphase des Getreides ausgebracht werden, ohne viel zu zerstören. Der Anbau könnte aus einer Schiene bestehen, auf der z. B. ein, mit einem Fräskopf versehener, Roboterarm, der nach links und rechts fahren kann, montiert ist. An diesem Anbau könnte auch eine Kamera montiert sein, die die Ungenauigkeit der Positionsangaben der Drohne ausgleicht.

Der Vorteil von einem Traktoranbau wäre, dass dieser durch den Traktor stabil genug ist und genug Kraft aufbringen kann, um den Ampfer durch Fräsen oder Herausziehen zu bekämpfen.

So könnte eine Drohne autonom über einen Acker fliegen und Bilder aufnehmen. Diese Bilder könnten dann automatisch auf einen Server hochgeladen werden, wo die ungefähren Positionen der Ampfer anhand der Bilder ermittelt werden können. Diese Positionen könnten dann (evtl. gleich in Kombination mit der optimalen Route, die der Roboter fahren soll) an einen Roboter (oder einen Traktor mit Anbau) übermittelt werden. Dieser Roboter würde dann die Ampferpflanzen anfahren und mit einer eigenen Optik die exakte Position der Ampferpflanze bestimmen, um diese dann zu bekämpfen.

In der Zukunft könnte man dieses Bilderkennungsverfahren auch für Statistiken verwenden. Man könnte von einem Acker Bilder aufnehmen und ausrechnen, wie groß der Ampferbestand auf dem Acker ist. Anhand von diesen Daten könnten auch Schadschwellen bestimmt werden, die angeben, ab welchem Ampferbestand man beginnen muss, die Ampfer zu bekämpfen.

6: Zusammenfassung

Mit dem aktuellen Programm kann man Drohnenbilder einfach in einem Google Drive Ordner hochladen und die Koordinaten der Ampfer werden berechnet. Die Ampfer auf Bildern, die in niedriger Höhe (ca. 1 m) aufgenommen wurden, können auch zuverlässig erkannt werden. Das bildet die Grundlage für den nächsten großen Schritt im Projekt. Dieser ist, einen Roboter (oder Traktoranbau) zu bauen, der die ermittelten Positionen der Ampfer anfahren kann und die Ampferpflanzen bekämpft.

7: Danksagung

An dieser Stelle möchte ich Florian Hölz dafür danken, dass er seine Drohne zur Aufnahme der Ampferbilder zur Verfügung gestellt hat. Mein Dank gilt auch meinen Projektbetreuern Benno Hölz und Matthias Ruf, die mich bei meinem Projekt unterstützt haben.

8: Quellen

Q1: https://de.wikipedia.org/wiki/Stumpfblättriger_Ampfer
Q2: https://bit.ly/3rOrpcS
Q3: https://de.wikipedia.org/wiki/Backpropagation
Q4: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
Q5: https://de.wikipedia.org/wiki/Convolutional_Neural_Network
Q6: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e
Q7: https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013/UijlingsIJCV2013.pdf
Q8: https://www.youtube.com/watch?v=pylvlMuRKY0&list=PLNmsVeXQZj7rx55Mai21reZtd_8m-qe27
Q9: https://de.wikipedia.org/wiki/Rectifier_(neuronale_Netzwerke)
Q10: https://towardsdatascience.com/building-your-own-object-detector-pytorch-vs-tensorflow-and-how-to-even-get-started-1d314691d4ae
Q11: https://towardsdatascience.com/coco-data-format-for-object-detection-a4c5eaf518c5
Q12:https://de.wikipedia.org/wiki/Arkustangens_und_Arkuskotangens#Umrechnung_ebener_kartesischer_Koordinaten_in_polare
Q13: https://docs.python.org/3/library/math.html#math.atan2
Q14: https://de.wikipedia.org/wiki/Polarkoordinaten
Q15: https://github.com/mapado/haversine#inverse-haversine-formula
Q16: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173
Q17: https://de.wikipedia.org/wiki/Verzeichnung
Q18: https://www.adobe.com/de/products/photoshop-lightroom-classic.html
Alle Links wurden zuletzt am 14.1.22 überprüft.

Abbildung 2: https://commons.wikimedia.org/wiki/File:NeuronModel_deutsch.svg
Abbildung 3: Von Dake, Mysid - Vectorized by Mysid in CorelDraw on an image by Dake., CC BY 1.0, https://commons.wikimedia.org/w/index.php?curid=1412126
Abbildung 4: https://www.ibm.com/cloud/learn/convolutional-neural-networks
Abbildung 5: Von Aphex34 - Eigenes Werk, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45673581
Abbildung 10: Von Boris Gaberšček - http://www2.arnes.si/~bzwitt/flora/rumex_obtusifolius.html, CC BY 2.5 si, https://commons.wikimedia.org/w/index.php?curid=41420986
Abbildung 17: https://earth.google.com/web/
Abbildung 1,6,7,8,9,11,12,13: Eigene Aufnahmen
Abbildung 14,15,16: selbst erstellt mit Drawio