Ampfer Mampfer

Inhalt

[Kurzfassung 1](#_Toc92636415)

[Idee 1](#_Toc92636416)

[Bilderkennung 2](#_Toc92636417)

[Neuronale Netzwerke 2](#_Toc92636418)

[Grundlagen 2](#_Toc92636419)

[Lernen 3](#_Toc92636420)

[Convolutional Neuronal Networks 4](#_Toc92636421)

[Region Based Convolutional Neuronal Networks 6](#_Toc92636422)

[Umsetzung 6](#_Toc92636423)

[Klassifizierung 6](#_Toc92636424)

[Object Detection 7](#_Toc92636425)

[Positionsbestimmung 10](#_Toc92636426)

[Ergebnisse 12](#_Toc92636427)

[Ausblick 13](#_Toc92636428)

[Quellen 13](#_Toc92636429)

# 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.

# Idee

Wenn man als Landwirt z.B. Getreide anbaut, kann Unkraut ein großes Problemsein. Konventionelle Landwirte gehen gegen Unkraut mit Pflanzenschutzmitteln, wie z.B. Herbiziden, vor. Diese Pflanzenschutzmittel sind aber nicht gut für die Gesundheit und die Umwelt. Daher werden im ökologischen Landbau keine solche 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 -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 20cm Tiefe *[Q2]*. Die Probleme des Ampfers sind, dass er die Kulturpflanzen verdrängt und die Ernte verunreinigt, da dann auch Ampfer samen im Getreide sind. Aber man kann Ampfer 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.

Ein Bild, das Boden, Pflanze, Gemüse, Zement enthält.

Automatisch generierte Beschreibung

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

Deshalb ist meine Projektidee, eine Maschine zu entwickeln, die Ampfer auf einem Getreideacker gezielt mechanisch bekämpfen kann. In dem hier vorloegenden Teil dieses Projekts beschäftige ich mich mit der Software, um Ampfer auf Bildern von Drohnen und aus der Nähe zu lokalisieren und die Position des Ampfers zu berechnen.

# Bilderkennung

Um Ampfer auf Bildern zu erkennen habe ich Software implementriert, die auf der Grundlage von Neuronalen Netzwerken arbeitet. Jetzt skizziere ich die relevanten theorethischen Grundlagen für diesen

## Neuronale Netzwerke

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

### Grundlagen

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

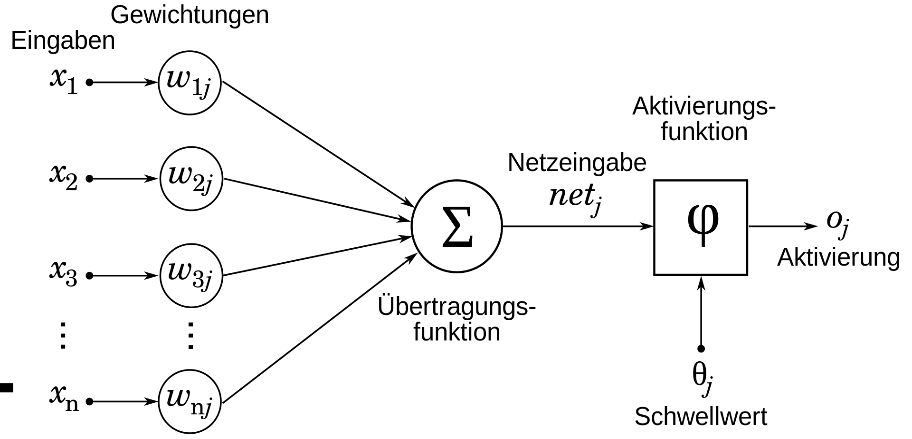
Links sieht man die Eingaben x (x1 bis xn) des Neurons, die mit der jeweiligen Gewichtung w (w1 bis wn) gewichtet werden. Diese Eingaben werden dann von einer Übertragungsfunktion (z.B. Summe) zusammengefasst. Dieser Wert wird an die Aktivierungsfunktion (z.B. die Sigmoid-Funktion) übergeben, die einen Ausgabewert (Aktivierung) erstellt, der dann anderen Neuronen als Eingabe übergeben wird.

Abbildung 2 Schema für ein künstliches Neuron

Ein Bild, das Unschärfe enthält.

Automatisch generierte BeschreibungUm ein Netzwerk zu formen, werden die Neuronen in verschiedenen Schichten miteinander verbunden. Als erste Schicht gibt es eine Input-Schicht (grün), bei der die Neuronen von „außen“ eine Eingabe bekommen. Danach folgen die versteckten Schichten (blau) und als Letztes die Output Schicht (gelb), bei der die Outputs der Neuronen ausgegeben werden.

Abbildung Vereinfachte Darstellung eines Künstlichen Neuronalen Netzwerks

### Lernen

Um Künstliche Neuronale Netzwerke zu trainieren, müssen die für die Aufgabe optimalen Gewichtungen w gefunden werden. 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 [*Q3*]:

: Fehler

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

: gewünschte Ausgabe/Zielwert (*target*)

: die errechnete Ausgabe des Netzes (*output*)

Der Faktor wird lediglich zur Vereinfachung der Ableitung hinzugenommen.

Das Zeil des Trainings ist es, die Fehlerfunktion zu minimieren, wobei meistens aber nur ein lokales Minimum erreicht wird. Das wird beim Backpropagation-Verfahren (anschaulich erklärt in [*Q4*]) durch die Veränderungen der Gewichte w, von denen die Ausgabe des Netzes abhängig ist, erreicht.

Die Ausgabe eines Neurons ist definiert durch:

Die Netzeingabe lässt sich wiederum definieren durch:

: Aktivierungsfunktion des Neurons (z.B. Sigmoid)

: Anzahl der Eingaben

: Eingabe

: Gewichtung zwischen Eingabe und Neuron

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

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.

: Änderung des Gewichts , der Verbindung von Neuron zu Neuron

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

: Fehler des Neurons , entspricht

: Soll-Ausgabe des Ausgabeneurons

: Ausgabe des Neurons

: Tatsächliche Ausgabe des Ausgabeneurons

: Index der Nachfolgenden Neuronen von

Jetzt können die Gewichte verändert werden:

: der neue Wert des Gewichts

: der alte Wert des Gewichts

: die vorher berechnete Änderung des Gewichts (basierend auf dem alten Wert des Gewichts

### Convolutional Neuronal Networks

[Convolutional Neuronal Networks](https://de.wikipedia.org/wiki/Convolutional_Neural_Network) (CNN) werden hauptsächlich für Bildanalyse (z.B. Klassifizierung) verwendet. Sie sind in cerschiedenen 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.

Ein Bild, das Text, Anzeigetafel enthält.

Automatisch generierte Beschreibung

Abbildung 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. Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

Abbildung 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.

### 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:

#### R-CNN

Bei dem originalen R-CNN werden als erstes 2000 Bereiche, in denen ein Objekt sein könnte, mithilfe eines Selective Search Algorithmus [*Genaueres in Q7*] ermittelt. Dieser Algorithmus ermittelt zuerst sehr viele 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 ein Objekt sein 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.

#### 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.

#### 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 von dem Bild erstellt. Dann 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.

# Umsetzung

Um die Ampferpflanzen auf Bildern zu erkennen, werden Künstliche neuronale Netzwerken verwendet. Die Netzwerke habe ich in Python mit [Pytorch](https://pytorch.org) ([torchvision](https://pytorch.org/vision/stable/index.html)) programmiert.

## Klassifizierung

Mein erstes Ziel war es, mithilfe eines CNNs Ampferbilder, die aus geringer Höhe (ca. 1m) 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).

Ein Bild, das Gras, Pflanze enthält.

Automatisch generierte BeschreibungEin Bild, das Pflanze, grün enthält.

Automatisch generierte Beschreibung

Abbildung Getreide Abbildung Ampfer

Diese Bilder habe ich dann mithilfe von einem selbstgeschriebenen Python Programm klassifiziert. 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. 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 5x und dann ein Pooling Layer mit der ReLu (Rectified Linear Unit) [*Q9*] als Aktivierungsfunktion. Diese Kombination wiederholt sich dreimal, sodass das Netz am Ende vier Convolutional Layer hat. 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 dann mit den Trainingsbildern trainiert. Auf den Testbildern, die das Netz noch nie „gesehen“ hat, wurde nach dem Lernen 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. Das ist für die Ampferbekämpfung aber essenziell. Deshalb habe ich zu einer anderen Methode gewechselt.

## Object Detection

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 brauchen, die sogar teilweise nicht unter Windows erhältlich sind, verwende ich zum Ausführen meiner Programme ab jetzt Google Colaboratory. Das ist ein 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 von diesen Angaben kann das Netz dann lernen. Die Label für die Trainingsbilder habe ich mit LabelImg (<https://github.com/tzutalin/labelImg>), einem kostenlosen Programm, mit dem man in Bilder die Positionen der gesuchten Objekte einzeichnen kann, erstellt. Diese werden dann automatisch nach dem gewählten Standard abgespeichert, hier Pascal VOC (Visual Object Classes) [*genaueres 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. Hierzu werden rote Rechtecke (Boxen) von einem Python Skript anhand der Ausgabe des Netzes eingezeichnet.

Ein Bild, das Pflanze, Baum enthält.

Automatisch generierte BeschreibungEin Bild, das Baum, draußen, grün, Pflanze enthält.

Automatisch generierte Beschreibung

Abbildung Ampfer Abbildung 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. 10m bis 20m macht. Zu dem Zeitpunkt dieser Aufnahmen war der Ampfer schon fast reif. Deshalb sieht man den Ampfer auf den Drohnenbildern orangebraun.

Ein Bild, das Gras, draußen, Pflanze, grün enthält.

Automatisch generierte BeschreibungEin Bild, das Baum, draußen, Pflanze enthält.

Automatisch generierte Beschreibung

Abbildung Ampfer reif Abbildung 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 aber 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 die 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 Testbildern schon sehr gut.

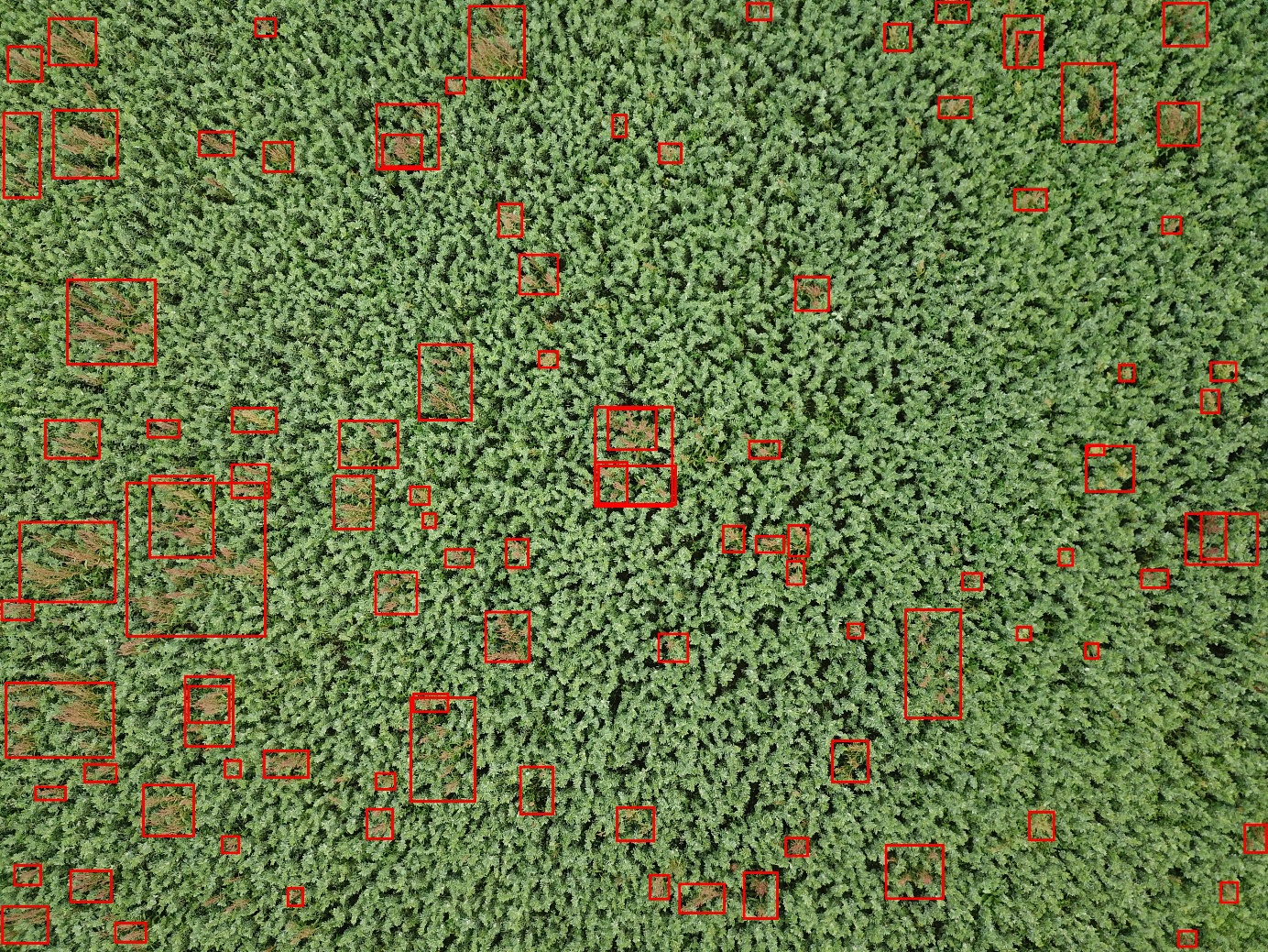


Abbildung Bild vom Acker mit einer Drohne aufgenommen, Ampfer von dem Faster R-CNN lokalisiert (Höhe: 15,5m)

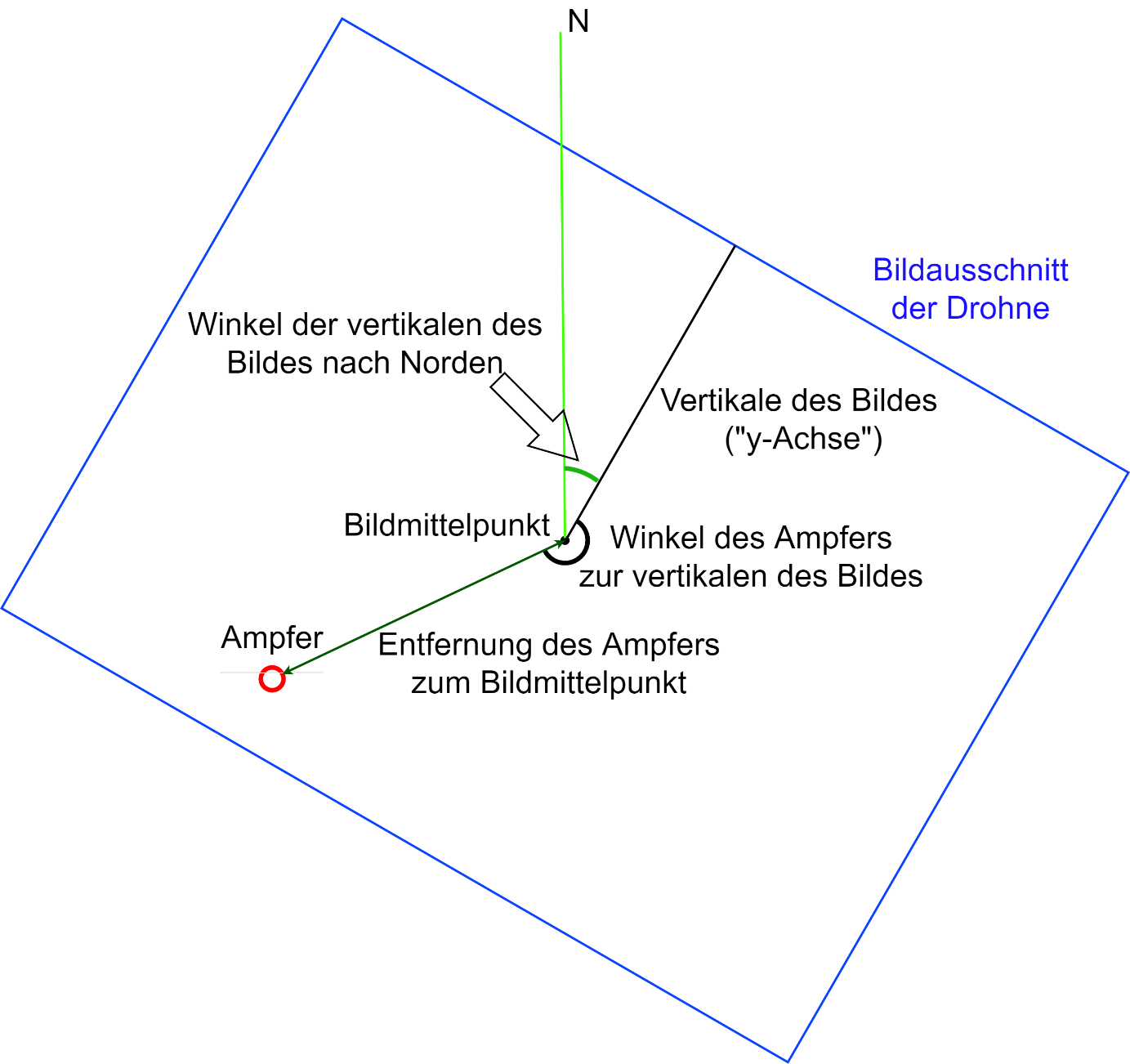
Wie man auf diesem Bild sieht, überlappen sich manche der erkannten Regionen stark. Um diese Überlappung zu entfernen habe ich einen Algorithmus in Python implementiert, der für alle Boxen bestimmt, wie weit sie sich mit jeder anderen in x- und y-Richtung überlappen. Anhand dieser Werte kann die Fläche 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 Bild vom Acker mit einer Drohne aufgenommen, Ampfer von dem Faster R-CNN lokalisiert (Höhe: 15,5m)

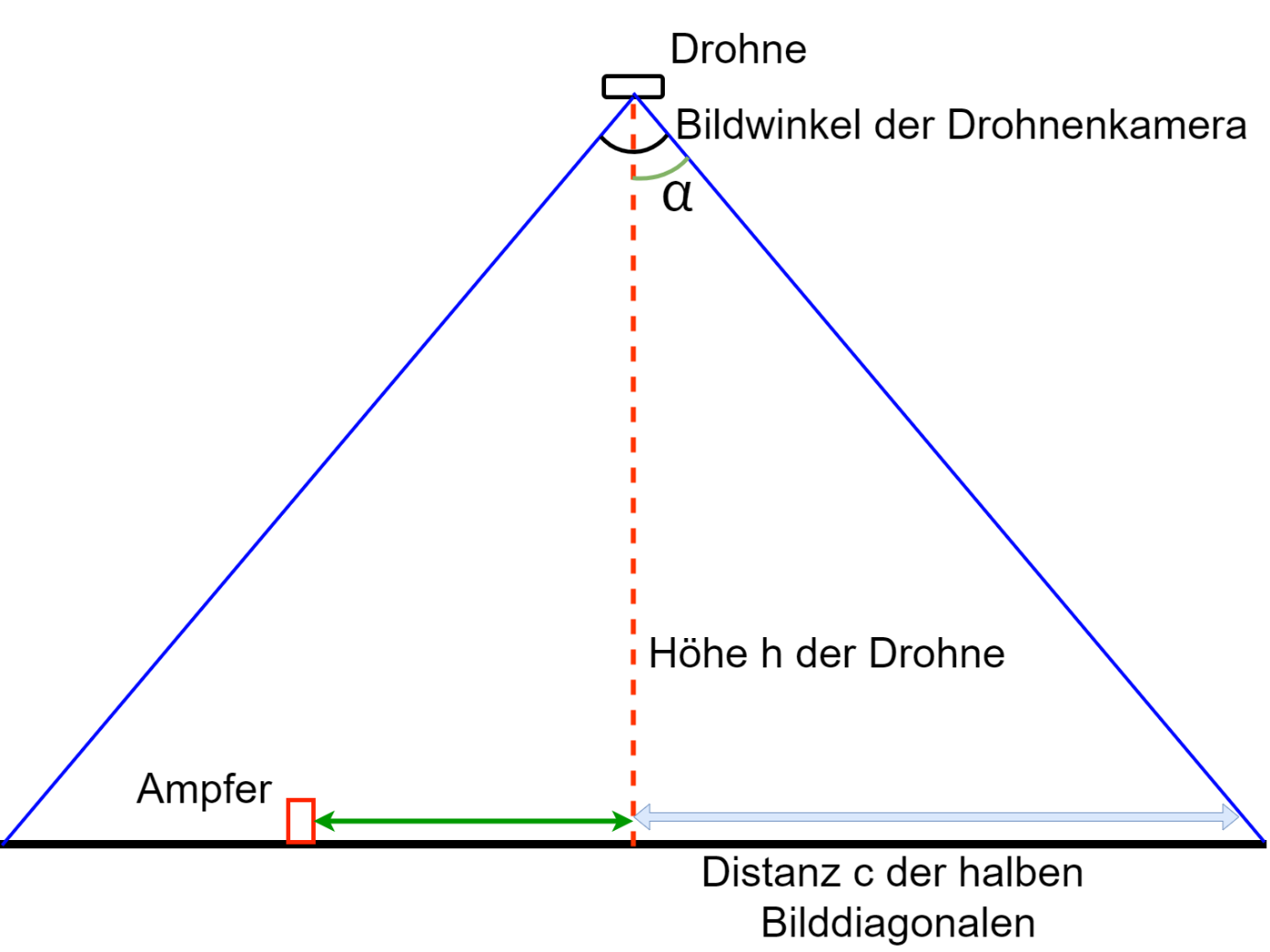
## Positionsbestimmung

Da in den Metadaten der Bilder die GPS-Position und Höhe der Drohne zum Zeitpunkt der Aufnahme enthalten sind, kann man die GPS-Position der Ampfer bestimmen, wenn man die Position der Ampfer auf dem Bild kennt. Dafür werden zwei Werte bestimmt:



Als Erstes soll der Winkel von der Verbindung zwischen dem Ampfer und dem Mittelpunkt nach Norden bestimmt werden. Dafür muss zuerst 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 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 als erstes ein Maßstab berechnet, welche der realen Distanz in Metern ein Pixel im Bild entspricht.

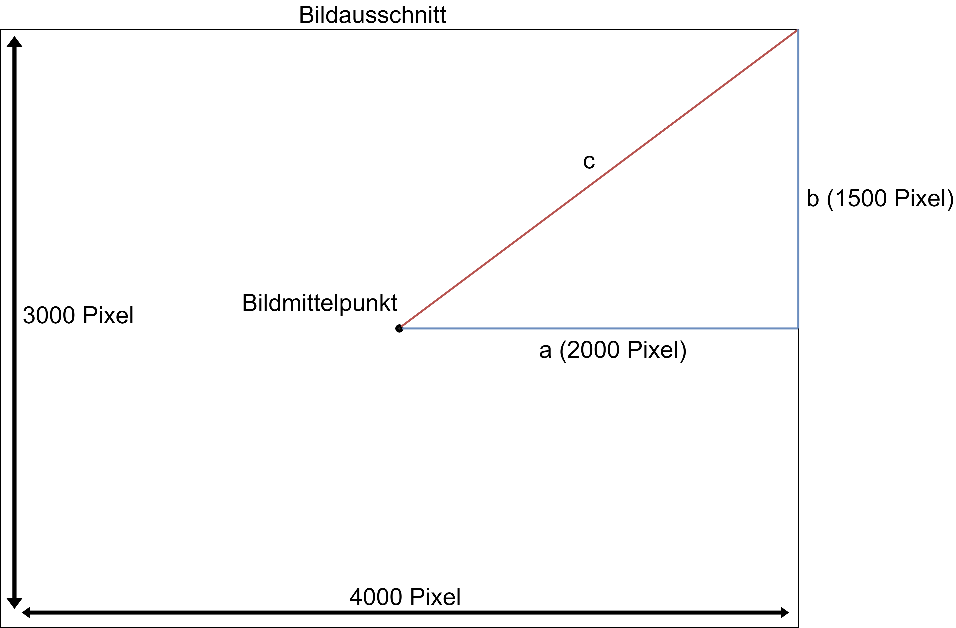


Hälfte des Bildwinkels der Drohnenkamera

Höhe der Drohne

Reale Distanz der Hälfte der Bilddiagonalen

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



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.

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

Mit dieser Anwendung des Satzes des Pythgoras kann auch die Distanz eines Ampfers, dessen Position auf dem Bild bekannt ist, zum Mittelpunkt berechnet werden.

Dann kann mithilfe des Satz des Pythagoras bestimmt werden, welche Distanz in Pixeln die Hälfte der Bildschirmdiagonale entspricht. Da diese Distanz dann in der Realität und im Bild bekannt ist, kann errechnet werden, welcher realen Distanz ein Pixel entspricht.

So 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 und des Winkels nach Norden bestimmt werden.

# Ergebnisse

Auf Bildern, die aus niedriger Höhe (ca. 1m) gemacht wurden, können jetzt mit dem ersten Faster R-CNN die Ampferpflanzen 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 den 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 %. Das liegt daran, dass auf diesen Bildern sehr viele, sehr kleine Objekte gefunden werden sollen. So sind selbst Ungenauigkeiten von wenigen Pixeln ausschlaggebend. Diese Ungenauigkeiten können aber auch daherkommen, dass die Position des Ampfers nicht ganz eindeutig ist, weshalb eigentlich richtige Boxen aufgrund von leichter Inkonsistenz der Label als Falsch gewertet wird.

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



Position, die auf den Bildern markiert wurde

Abbildung Google Earth

Die Bilder wurden aus einer Höhe von ca. 20m bis 100m aufgenommen. Auf einer geringen Höhe wie 20m ist die Positionsbestimmung also sehr genau (ca. 1m 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. Da diese Positionen immer sehr eng zusammen liegen, sieht man, dass die Verzeichnung kein großer Fehlerfaktor ist. Das lässt darauf schließen, dass die Ungenauigkeit von der ungenauen PGS Position und Höhe der Drohne kommt. Die Höhe der Drohne wird über ein Barometer ermittelt, was teilweise etwas ungenau ist.

Diese Algorithmen habe ich in einem Python Programm auf Google Colaboratory umgesetzt. Man kann Drohnenbilder von einem Acker in einen Google Drive Ordner hochladen und das Programm (ca. 8s pro Bild) speichert die ermittelten Ampferpositionen in einer .json oder in einer .kml Datei, die man in Google Earth öffnen kann, ab.

# Ausblick

In der Zukunft könnte man diese Algorithmen auch für Statistiken verwenden. Man könnte von einem Acker Bilder aufnehmen und ausrechnen, wie groß der Ampferbestand auf dem Acker ist. Dafür bräuchte man allerdings einen Algorithmus, der erkennt wie sich die Bilder überlappen. Das könnte über die Koordinaten der Bildecken gelöst werden, was aber nicht exakt ist. Eine weitere Möglichkeit wäre, einen Algorithmus zu implementieren, der die Überlappung der Bilder anhand des Musters der erkannten Ampfer erkennt. Dieser Algorithmus könnte auch die Genauigkeit der GPS-Positionsbestimmung der Ampfer erhöhen, da für jeden Ampfer so Mittelwerte aus mehreren Positionen errechnet werden können, was die Genauigkeit weiter erhöhen würde.

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

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

Um einen funktionierenden Ampfer Mampfer zu bauen kann man sich auch noch damit beschäftigen, wie man einen Roboter baut, der zu den ermittelten Positionen der Ampferpflanzen fährt und diese vernichtet. Dieser Roboter sollte auf jeden Fall eine Kamera haben, sodass er den Ampfer noch genauer lokalisieren kann, da die Positionsbestimmung über die Bilder und über GPS 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 können sollte, um den Ampfer zu bekämpfen. Eine Möglichkeit wäre, die Ampferwurzel 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 Roboterarm, der so nach links und rechts fahren kann, montiert ist. An diesem Anbau könnte man dann auch eine Kamera montieren, 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.

# 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>

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>

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 1,6,7,8,9,11,12,13 : Eigene Aufnahmen

Allgemein:

<https://pytorch.org>

<https://pytorch.org/vision/stable/index.html>

<https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz>

<https://de.wikipedia.org/wiki/Convolutional_Neural_Network>

<https://colab.research.google.com>