Hochschule Düsseldorf
University of Applied Sciences

Hochschule Düsseldorf

Where to look?

Projektdokumentation

Verantwortlicher Dozent: Prof Dr. Dennis Müller

Vorgelegt von: Julian Mäder

Matrikelnummer: 888916

Studienfach: Advances in Intelligent Systems (ISS)

Wintersemester 2023/24

INHALT

1 EINLEITUNG

Die Erklärbarkeit von Modellvorhersagen war lange Zeit ein Nischenthema, denn der Weg von einem Eingabe Bild zu einer Vorhersage ist meist intransparent. Die vom ML-Modell gelernten Zusammenhänge befinden sich oftmals in einer Blackbox, welche nicht einsehbar ist. Genau da kommt Class Activation Mapping (CAM) ins Spiel, wodurch die Regionen in einem Bild hervorgehoben werden können, die das Netz dazu verleitet haben sich zu entscheiden. Die Information, woran sich das Netzwerk entschieden hat, kann des Weiteren zur Objektlokalisierung genutzt werden. So können beispielsweise Bilder von mit Malaria infizierten Zellen nicht nur klassifiziert werden, sondern auch Den Bereich der Infektion hervorheben. Genau dieses Beispiel, sowie eine Geschlechter Klassifizierung, werde ich im Folgenden behandeln. In dieser Dokumentation werden nur kleine Codeausschnitte aufgeführt, um die Dokumentation nicht zu überlasten. Das dazugehörige Python Programm steht auf GitHub[1] bereit.

1.1 STATE OF THE ART

Wie eingangs beschrieben gab es lange Zeit kein großes Interesse daran die Entscheidungen eines Neuronalen Netzes nachzuvollziehen. Auch heute noch ist erstaunlich wenig Verschiedenes zu CAM verbreitet. Was man dafür sehr häufig findet, ist der Ansatz mit dem Resnet50 Modell die Aktivierungen der letzten Convolutional Layer zu visualisieren. Darüber hinaus geht aber kaum ein z findendes Beispiel. CAM auf allen Ebenen des Netzwerkes zu visualisieren, sowie auf ein komplett eigenes Netzwerk anzuwenden, ist weitestgehend unerforscht. Die reine Bildbasierte Objektlokalisierung hingegen ist umso erforschter.

1.2 ZIEL DES PROJEKTES

Das Ziel dieses Projektes ist die Objektlokalisierung mittels Class Activision Mapping. Dabei sollen mit Malaria befallene Zellen identifiziert werden und die Infektion durch eine Heatmap hervorgehoben werden. Dieses Verfahren soll anschließend um die Aktivierungen der Zwischenebenen des Netzwerkes erweitert werden, sodass nicht nur die Aktivierungen der letzten Faltungsschicht, sondern auch die Aktivierungen der Vorherigen sichtbar gemacht werden.

2 VORARBEIT

2.1 VERWENDETE METHODEN UND TECHNOLOGIEN

Diesem Projekt liegt insbesondere eine bestimmte Technik zugrunde: Class Activision Mapping (CAM). CAM ist ein Verfahren, um Objekte mittels eines vortrainierten Netzes in Bildern zu lokalisieren, indem die Regionen gekennzeichnet werden, in denen das Netzwerk eine hohe Aktivierung aufweist. Die Implementierung von CAM wird in Punkt 3.2.1 genauer erläutert. Essentiell für CAM ist, dass die Featuremaps des Convolutional Neural Networks ihren räumlichen Bezug zum Bild beibehalten. Das ist auch der Grund, weshalb Global Average Pooling für das Pooling genutzt werden muss: Es behält den räumlichen Bezug zum Bild bei. Punkte, die z.B. oben links im Bild sind, bleiben auf der Featuremap oben links.

Der Code zum Trainieren der Modelle basiert auf dem Cats vs. Dogs Beispiel aus der Vorlesung.

[1] https://github.com/Julimaeder/Where-to-look

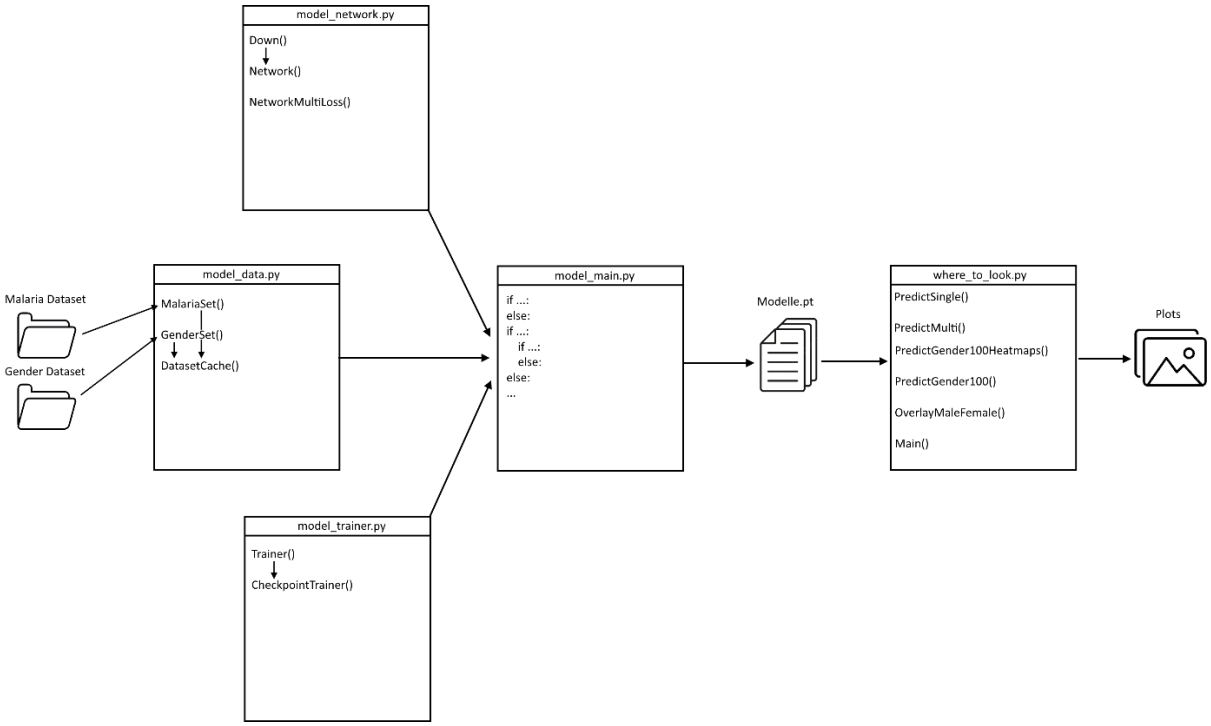
2.2 VERWENDETE DATENSÄTZEN

Es wurden zwei Datensätze verwendet: Einer für die Genderklassifikation und einer zur Erkennung von Malaria infizierten Zellen. Beide sind in den Quellen aufgeführt. Der Gender Datensatz ist sehr geeignet, da zum einen, die Bilder alle auf das Gesicht zentriert sind, und zum anderen, die Ordnerstruktur schon perfekt zum Trainieren von Modellen ist mit einer Unterteilung in Test-, Training- und Validation- Bilder.

Im Malaria Datensatz sind die Bilder nur in gesunde und infizierte Zellen unterteilt. Zudem kam es teilweise zu falschen Vorhersagen, da im ‚uninfected' Ordner auch Zellen von augenscheinlich infizierten Zellen sind.

3 MEINE ARBEIT

Meine Arbeit besteht im Wesentlichen aus zwei Teilen: Dem trainieren des Netzwerkes und der Anwendung der CAMs. Eine grobe Übersicht liefert folgende Visualisierung der Dateien und Funktionen:



3.1 NETZWERK TRAINIEREN

Da, wie oben bereits genannt, meist das ResNet50 Modell genutzt wird, und sich die Architektur des Modelles nicht anbietet um CAM auf die verschiedenen Faltungsebenen anzuwenden, müssen eigene Netze trainiert werden. Dazu sind die vier Dateien model_data.py, model_network.py, model_trainer.py und model_main.py zuständig. Der Aufbau basiert auf dem, zuvor genannten, Cats vs. Dogs Code[2], wobei er inzwischen an vielen Stellen davon abweicht. Es können vier verschiedene Modelle trainiert werden, welche unterschiedliche Nutzen haben.

3.1.1 model_data.py

Die Datei dient dazu die Datensätze zu laden und im Zwischenspeicher zu halten. Für das Laden der Datensätze sind die Klassen MalariaSet und GenderSet zuständig. So werden in MalariaSet zu Beginn

[2] https://github.com/dmu1981/pythoncourse/tree/master/advances

zwei Listen mit allen Pfaden der infizierten, bzw. gesunden Zellen gespeichert und anschließend in Trainings und Validation Daten unterteilt. Zudem werden die Pfade zu Testbildern in einer separaten JSON-Datei gespeichert, um im späteren Verlauf Bilder zum Testen zu haben, welche das Neuronale Netz noch nie gesehen hat. Die Liste wird anschließend entsprechend der gewünschten Länge angepasst. Und um die Label der Datenpunkte erweitert. So entsteht eine Liste an Tupel mit einer gewünschten Länge. Da die Klasse auf der Dataset Klasse von Pytorch basiert, muss die Anzahl der Datenpunkte in __len__() angegeben werden. In der __getitem__ Methode wird der Datensatz indexiert und Bilder, die bei Aufruf eingeladen werden, verarbeitet, sodass im weiteren Verlauf mit ihnen gearbeitet werden kann. Zurückgegeben wird das Bild mitsamt Label.

Die GenderSet Klasse ist gleich aufgebaut, wie die MalariaSet Klasse, nur mit dem Unterschied, dass aufgrund der Ordnerstruktur des Gender Datensatzes kein Split in Training, Test und Validation durchgeführt werden muss. Die Bilder sind Bereits in den Ordnern richtig unterteilt.

Zuletzt folgt die DatasetCache Klasse, welche dazu dient, Biller zwischenzuspeichern. Dies geschieht, indem zu Beginn eine leere Liste mit der Länge des Datensatzes erstellt wird. Wenn ein Datenpunkt von der __getitem__ Methode angefragt wird, wird überprüft, ob an diesem Index in der Liste schon ein Wert steht. Wenn nicht, wird der Datenpunkt am passenden Index gespeichert und kann beim nächsten Mal direkt aus der Liste aufgerufen werden.

3.1.2 model_network.py

Diese Datei beherbergt die Netzwerke, die zum Trainieren genutzt werden. Dazu dienen die beiden Klassen Network und NetworkMultiLoss, sowie die Klasse Down, welche zu Network gehört. Die Klasse Network besteht im Wesentlichen, da sie auf dem torch.nn.Module basiert, aus zwei Komponenten: Den Rechenoperationen, die mit dem Eingabebild durchgeführt werden sollen in der __init__ Methode, und die forward Methode, welche definiert, wie die Eingabedaten durch das Netz fließen. So wird das Eingabe Bild zu Beginn mehrfach durch folgende Sequenz geschickt, welche in der Klasse Down definiert, ist:

```
torch.nn.Conv2d(in_features, out_features, kernel_size=(5,5), padding="same"),
torch.nn.MaxPool2d(kernel_size=(2,2), stride=(2,2)),
torch.nn.BatchNorm2d(num_features = out_features),
torch.nn.ReLU()
```

In diesen Operationen wird zu Beginn eine Faltung mit einem 5x5 Filter durchgeführt. Daraufhin wird Maxpooling angewandt, mit einer kernel_size von 2x2 und einem Stride von 2x2, was bedeutet, dass der immer ein 2x2 großer Bereich betrachtet wird, aus diesen 4 Werten der maximale Wert auf die neue Matrix projiziert wird und dann zwei Schritte weiter gegangen wird. Das wird für die komplette Eingangsmatrix wiederholt, was eine Reduzierung der Größe der Featuremaps zur Folge hat. Anschließend folgt eine Batchnormalisierung, welche den Mittelwert jedes Batches auf 0 und die Standardabweichung auf 1 setzt. Die Folge davon ist ein schnelleres, sowie stabileres Training. Abschließend gehen die Daten durch eine Aktivierungsfunktion, in diesem Fall die ReLU-Funktion. Diese setzt alle negativen Werte auf Null und behält die positiven bei. Nach diesem Block folgt ein Dropout mit der Wahrscheinlichkeit von 0.2. Das Bedeutet, dass 20% aller Neuronen ausgeschaltet, also nicht beim Training berücksichtigt werden. Dadurch wird die Gefahr von Overfitting minimiert, da sich Das Netzwerk nicht zu stark auf bestimmte Neuronen Verbindungen verlassen kann.

Nach vier Durchläufen haben wir 256 Feature Maps. Diese werden zur Klassifikation durch ein Global Average Pooling (GAP) Layer geschickt, welches den Durchschnittlichen Wert jeder Featuremap bildet und so einen Eindimensionalen Vektor erzeugt, welcher durch den classifier Teil geht:

```
torch.nn.Flatten(),
torch.nn.Dropout(0.5),
torch.nn.Linear(256, 2)
```

So geht das Netz auf zwei Neuronen runter, was das Netz auch nur für Binärklassifikationen brauchbar macht.

Die forward Methode der Klasse gibt daraufhin das Ergebnis der Binärklassifikation zurück, wenn return_cam = False ist. Wenn der Boolean auf True gesetzt wird, gibt er außerdem die Featuremaps der letzten Faltungsebene zurück, bevor GAP und die Klassifikation angewandt werden.

Die Klasse NetworkMultiLoss sieht sehr ähnlich aus, nur, dass hier die Operationen nicht in einer eigenen Klasse ausgelagert sind, da man ansonsten schwerer drauf zugreifen kann. Auch hier ist der Ablauf Conv2d, MaxPool2d, BatchNorm2d und ReLU, nur, dass hierbei die Features jeder Faltungsebene zurückgegeben werden, nicht nur die der letzten, was man der forward Methode erkennen kann:

```
def forward(self, x, return_features=False, return_features2=False):
        x1 = self.relu1(self.bn1(self.pool1(self.conv1(x))))
        x2 = self.relu2(self.bn2(self.pool2(self.conv2(x1))))
        x3 = self.relu3(self.bn3(self.pool3(self.conv3(x2))))
        x4 = self.relu4(self.bn4(self.pool4(self.conv4(x3))))
        if return_features:
            return x1, x2, x3, x4  # Return raw feature maps
```

Um das Netzwerk so zu trainieren, dass ein möglichst geringer Loss in allen Faltungsebenen entsteht[3], muss zudem die Möglichkeit bestehen, die Klassifikation, die in der Network Klasse am Ende erfolgt, hier auch für alle Zwischenebenen durchzuführen. Demnach geht jeder Output eines Convolution blockes (x1, x2, x3, x4) durch folgende Operationen und wird bei Bedarf zusammen mit den Features zurückgegeben:

```
gap1 = self.gap4(x1)
flat1 = torch.flatten(gap1, 1)
t1 = self.transform1(flat1)
out1 = self.classifier(t1)
```

3.1.3 model_trainer.py

Diese Datei trägt zwei Klassen in sich: Trainer und CheckpointTrainer. Die Trainer Klasse erstellt dabei die Trainingsschleife für das ausgewählte Modell.

Zu Beginn wird in der Methode init_optimizer der Optimierer, der für die Anpassung der Gewichte des Netzes während des Trainings verwendet wird, initialisiert. In diesem Fall wird der Adam Optimierer mit einer Lernrate von 0.0001 verwendet.

Des Weiteren wird in der Methode init_scheduler der Scheduler für die Lernrate initialisiert. Hier wird ein ExponentialLR Scheduler mit einem Faktor von 0.95 verwendet, was bedeutet, dass die Lernrate von Epoche zu Epoche sinkt.

In der Methode Epoche findet das Training statt: Hier wird über alle batches mit deren Labels aus dem Dataloader iteriert. Wenn der Loss für jede Ebene berechnet werden soll, werden die Features, sowie Outputs aller vier Ebenen des Netzwerkes angefragt. Daraufhin wird der Loss für jede Ebene berechnet und addiert. Soll der Loss nur für die letzte Ebene berechnet werden, so wie es üblich ist, wird nur die Vorhersage der letzten Ebene des Netzes angefragt, aus welcher ebenso der Loss berechnet wird. Beiden Varianten folgen Zähler zum Erfassen der Genauigkeit des Trainings, welche in einer tqdm Leiste angezeigt werden. Zuletzt werden die Gradienten je nach ausgewähltem Loss mittels Backpropagation berechnet und die Netzwerkparameter mit dem eingangs genannten Optimizer angepasst.

[3] Siehe Abschnitt…

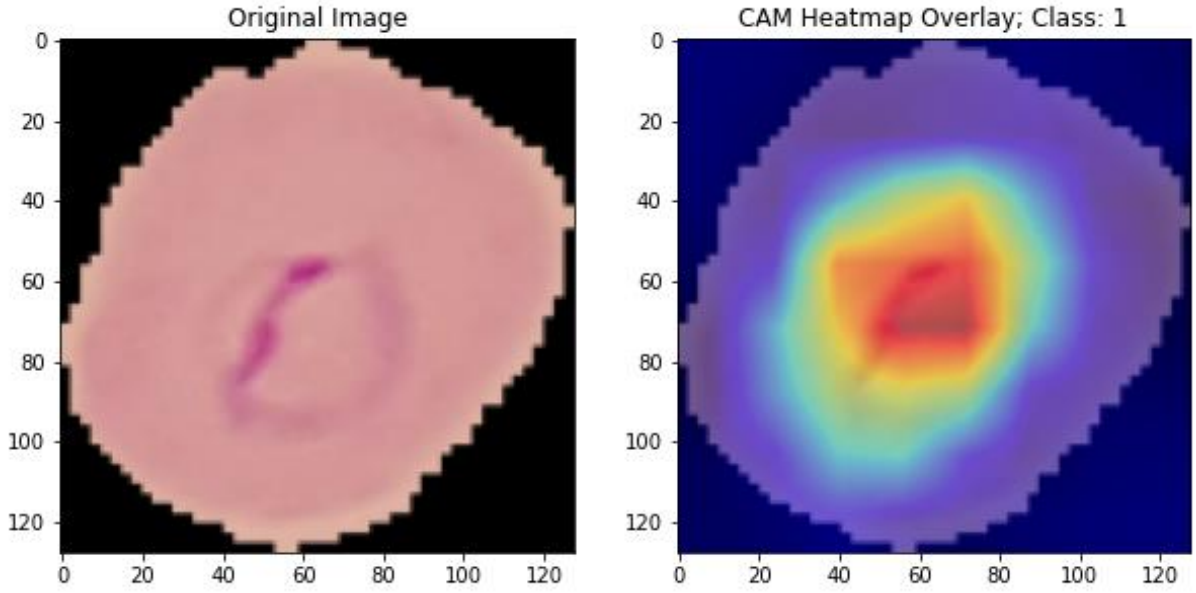
Die Klasse CheckpointTrainer leitet sich von der Klasse Trainer ab und dient dazu, wie der Name vermuten lässt, Checkpoints während des Trainings zu setzen, damit es zwischendurch – gewollt oder ungewollt – abgebrochen werden kann. Dazu wird überprüft, ob bereits ein Checkpoint vorhanden ist. Wenn es einer ist, werden seine Daten geladen und basierend darauf weitertrainiert. Wenn keine Daten vorhanden sind, wird das Modell von vorne trainiert. Nach jeder Epoche werden alle relevanten Daten in einer .pt Datei gespeichert.

3.1.4 model_main.py

Diese Datei besteht größtenteils aus if Abfragen, welche je nach Benutzerinput verschiedene Modelle trainieren lassen. Zu Beginn wird dabei der Benutzer gefragt, welchen Datensatz möchte. Da der Fokus des Projektes auf der Erkennung von infizierten Zellen liegt, gibt es für den Gender Datensatz keinen weiteren Input und das Training beginnt. Wenn er den Malaria Datensatz auswählt, kann er daraufhin entscheiden, ob er nur die Aktivierung des letzten Conv Blocks oder jedes Conv Blocks in der späteren Visualisierung sehen möchte. Je nach Antwort wird das passende Netzwerk ausgewählt. Als letzte Möglichkeit kann der Nutzer nach der Option jede Aktivierung sehen zu wollen, entscheiden, ob das Modell nach dem Loss des letzten Conv Blocks oder nach dem addierten Loss aller Conv Blöcke optimiert werden soll. Bevor ein Modell trainiert wird, wird außerdem ein Ordner für die Modelle erstellt, falls er noch nicht existiert.

3.2 WHERE_TO_LOOK.PY

Die where_to_look.py Datei ist das Kernstück der Arbeit. In ihr werden die Heatmaps zu den Eingabebildern generiert:



Die Datei besteht aus mehreren Funktionen zum Erstellen der Heatmaps, sowie einer main Funktion, welche ähnlich wie die model_main.py Datei je nach Benutzereingabe die verschiedenen Funktionen aufruft. Zu Beginn wird auf Wunsch ein Ordner, mit allen trainierten Modellen, von Google Drive heruntergeladen. Es Folgen nutzereingaben, um zu bestimmen, welche Funktion aufgerufen werden soll. Je nach Eingabe werden außerdem Dateinamen für Testbilder automatisch aus den zuvor erstellten JSON-Dateien oder einem angegebenen Datensatz geladen. Die Funktion ist so konstruiert, dass sie einfach vom Terminal aus bedient werden kann.

3.2.1 PredictSingle

Diese Funktion zeigt die Aktivierungen des letzten conv Lagers. Dazu wird zu Beginn das Input Bild, sowie die Modelle geladen. Das Bild wird daraufhin verarbeitet und das Modell in den evaluating Modus versetzt, was manche Funktionen, wie Dropout, deaktiviert. Weiter geht es mit folgender Zeile:

```
pred, last_conv_features = model(image, return_cam=True)
```

model ist unsere Network Klasse aus model_network.py, image ist unser, für das Modell vorverarbeitete, Eingabebild und return_cam gibt an, dass wir neben der Vorhersage auch die Features der letzten Faltungsebene zurückbekommen wollen. So wird pred unser Vorhersage Vektor mit der Wahrscheinlichkeit für beide Klassen, zum Beispiel
tensor([[-0.0685, 0.1490]], device='cuda:0', grad_fn=<AddmmBackward0>)
und last_conv_features unsere ganzen Featuremaps, die durch die Faltung entstehen. In diesem Fall hat last_conv_features die shape (1,256,14,14). Wir haben also 256 14x14 große Matrizen. Des Weiteren benötigen wir die Gewichtungen des fully connected layers, da nicht jede Featurmap den gleichen Einfluss hat. Die Gewichte bekommen wir mit

```
fc_weights = model.classifier[-1].weight.data
```

Dabei handelt es sich um Tensor der Form (2,256). Das bedeutet, dass wir für jede unserer 256 Featurmaps Ein Gewicht von jeder Klasse haben. Da wir uns aber die Aktivierung für die vorhergesagte Klasse anschauen wollen, müssen wir die vorhergesagte Klasse ermitteln, was sehr einfach ist, da wir nur den größten Wert aus unserem pred_vector benötigen:

```
 _, predicted_class = torch.max(pred, 1)
```

In unserem Beispiel ist -0.0685 < 0.1490, weshalb die Klasse der Vorhersage 1 wäre. Da, wie bereits gesagt, nur die Gewichte der richtigen Klasse benötigt werden, kürzen wir unsere fc_weights:

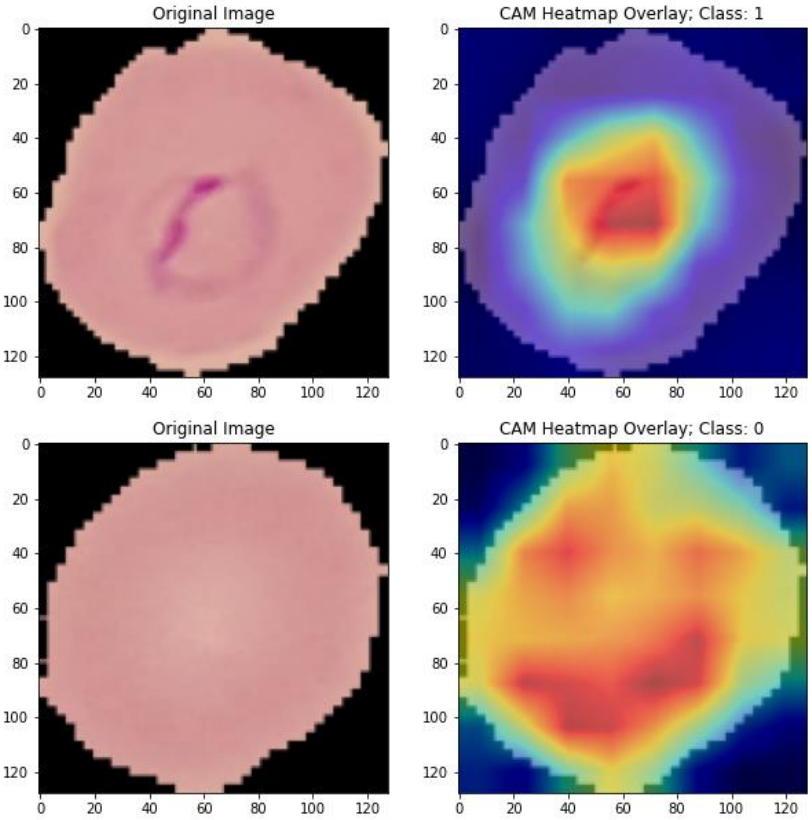
```
class_weights = fc_weights[predicted_class]
```

Wodurch wir 256 Gewichte erhalten. Würde man die andere Klasse verwenden, könnte man zum Beispiel visualisieren, was an einem Mann ‚weiblich' aussieht, bzw. wo das Netz in einem Bild eines Mannes eine Frau erkennen könnte.

Nun haben wir 256 14x14 große Featurmaps und 256 Gewichte. Diese müssen verrechnet werden, indem die Featuremaps mit dem dazugehörigen Gewicht multipliziert werden und anschließend alle Featuremaps addiert werden:

```
# Eine 14x14 Matrix aus Nullen wird erstellt, welche danach befüllt wird
    cam = torch.zeros((last_conv_features.shape[2], last_conv_features.shape[3]),
dtype=torch.float32).to('cuda')
for i, w in enumerate(class_weights[0]):
        cam += w * last_conv_features[0, i]
```

So erhalten wir eine 14x14 Matrix mit allen gewichteten Featuremaps ‚übereinander'. Da die Featuremaps ihre räumliche Beziehung zum Bild beibehalten, kann die Matrix nun normiert und auf die Auflösung des Eingangsbildes hochskaliert werden und anschließend über Dieses gelegt werden. Im Folgenden ist diese Matrix als Heatmap über das originale Bild gelegt. Das erste Bild zeigt eine infizierte Zelle, das Zweite eine Gesunde.



Wie man erkennen kann, trifft die Lokalisierung bei der infizierten Zelle ziemlich genau zu. Die gesunde Zelle weist kein bestimmtes Merkmal auf, weshalb die gesamte Zelle erkannt wird.

3.2.2 PredictMulti

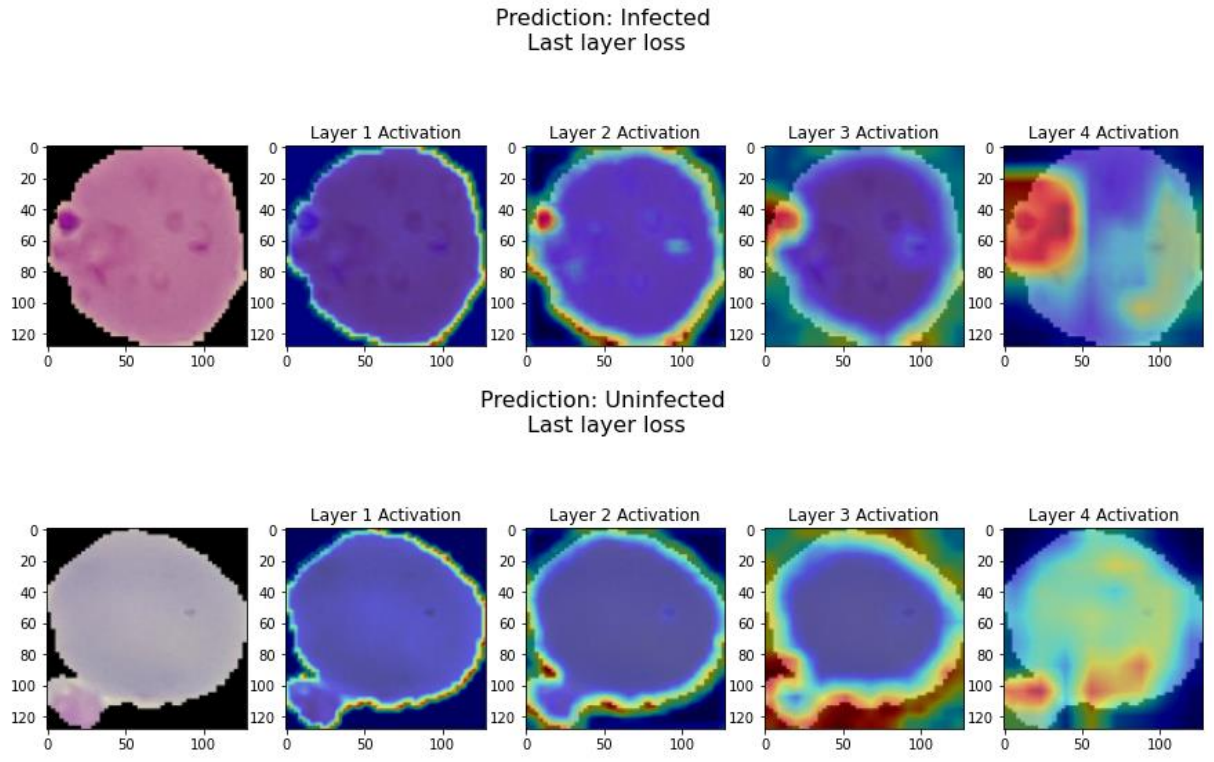
Die Funktion PredictMulti ist sehr ähnlich zu PredictSingle, nur, dass hier die Heatmaps nicht nur für die letzte Faltungsebene, sondern auch für die Zwischenebenen erstellt werden. Dazu wird da NetworkMultiLoss aus model_network.py genutzt. Wie auch in PredictSingle werden zu Beginn das Netzwerk, sowie das Bild geladen und wir lassen Das Bild durch das Netzwerk laufen. Anders als in der vorherigen Funktion bekommen wir jedoch bei der Vorhersage des Netzes:

```
feature_maps = model(image, return_features=True)
```

nicht nur einen Tensor mit der shape (1,256,14,14), sondern vier Tensoren mit folgender shape zurück:

Tensor[0] = [1, 32, 64, 64], Tensor[1] = [1, 64, 32, 32], Tensor[2] = [1, 128, 16, 16],
Tensor[3] = [1, 256, 8, 8]

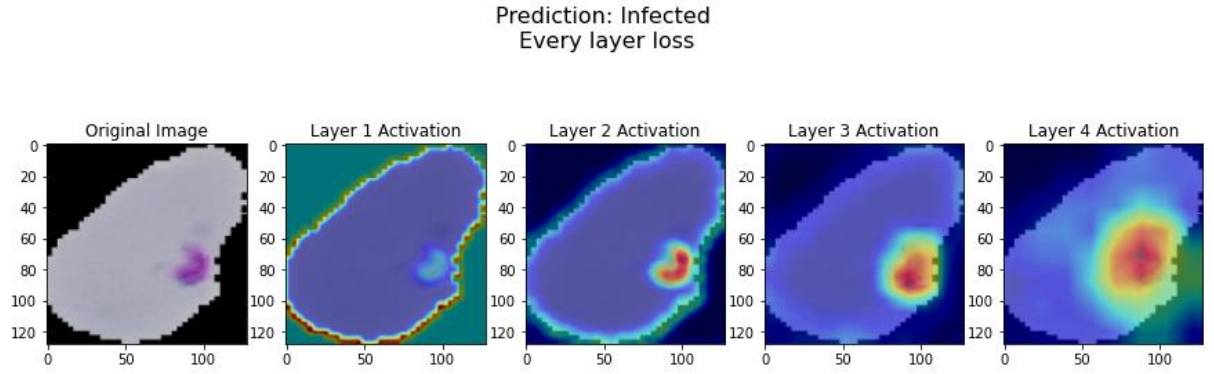
In der darauffolgenden Schleife wird über alle Tensoren iteriert und der Mittelwert über die Dimension an Stelle 1 errechnet. So haben wir vier Matrizen: Eine 64x64, eine 32x32, eine 16x16 und eine 8x8. Zum Schluss werden alle Matrizen normiert und auf die Größe des Eingabebildes skaliert, um über diesem geplottet zu werden.



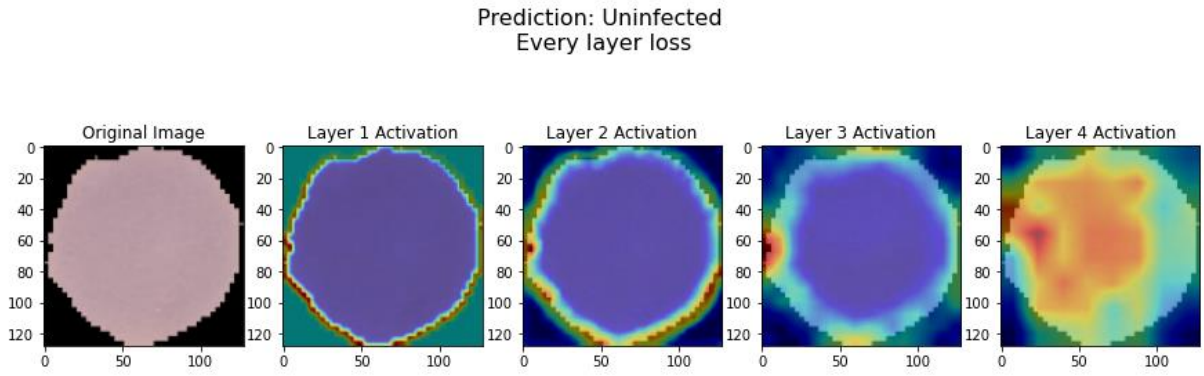
Spannend ist hierbei zu sehen, wie sehr die Features voneinander abweichen. Nach der ersten Faltung sind die Features noch sehr einfach gehalten und zeigen in diesem Fall Kanten an. Je tiefer es ins Netz geht, desto komplexere Muster werden erkannt.

3.2.2.1 Multi Aktivierung mit gemeinsamem Loss

Die obigen Bilder wurden mit einem Modell vorhergesagt, welches darauf trainiert wurde, in der letzten Faltungsebene den möglichst geringsten Loss zu haben. Wenn man nun jedoch den Loss jeder Ebene addiert, wird das Netzwerk gezwungen die klasse schon früher richtig zu klassifizieren. Dadurch kommen bereits ab der ersten Schicht bessere Ergebnisse:

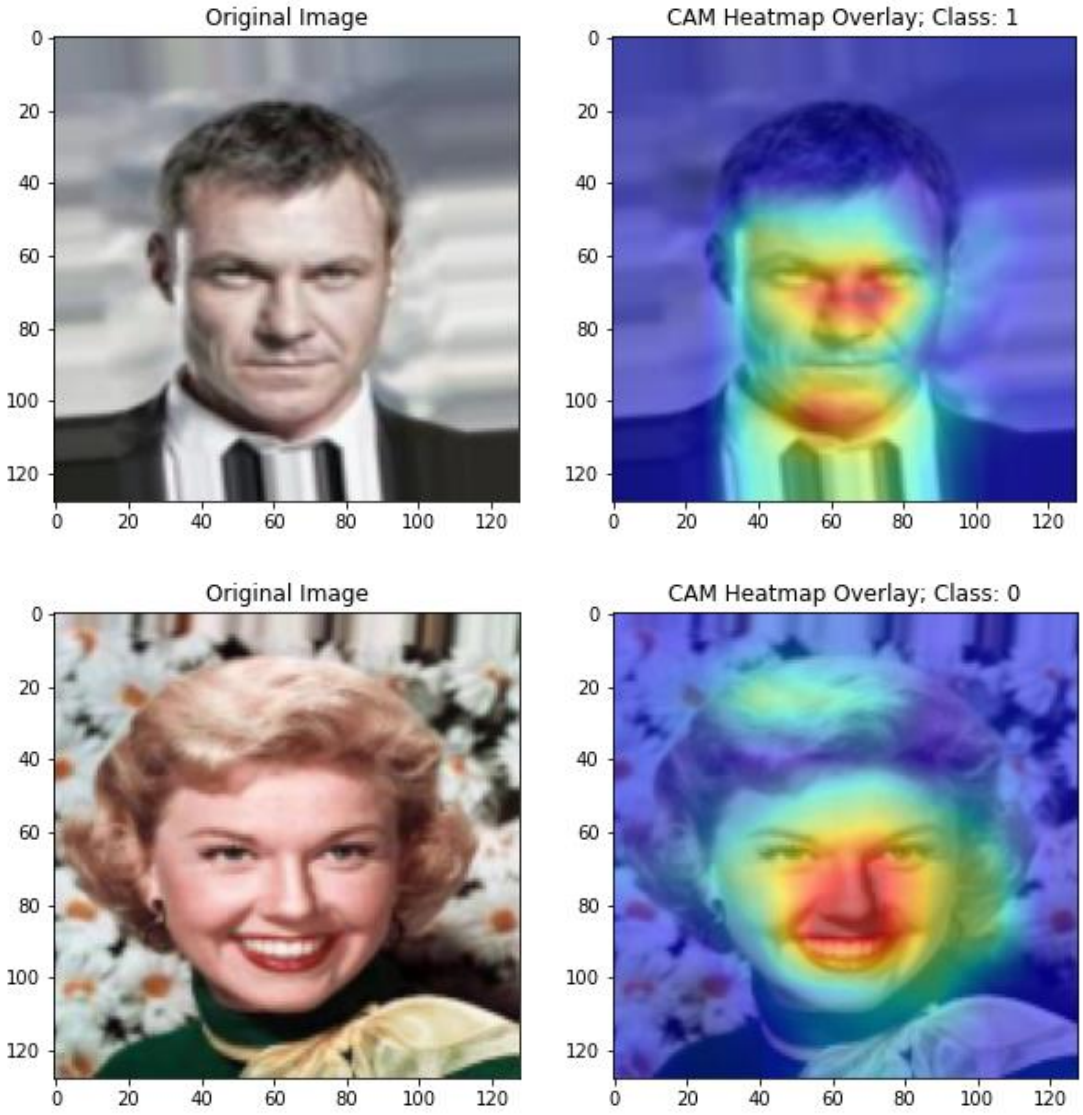


Hieran lässt sich gut erkennen, dass das Netzwerk gar nicht so groß hätte sein müssen, wie es ist. Bereits ab der zweiten Ebene kann man einen eindeutigen Hotspot erkennen und die dritte Ebene ist die eindeutigste. Bei den Gesunden Zellen kann man keinen Unterschied zur ‚Last layer loss' Version feststellen:



3.2.3 Gender

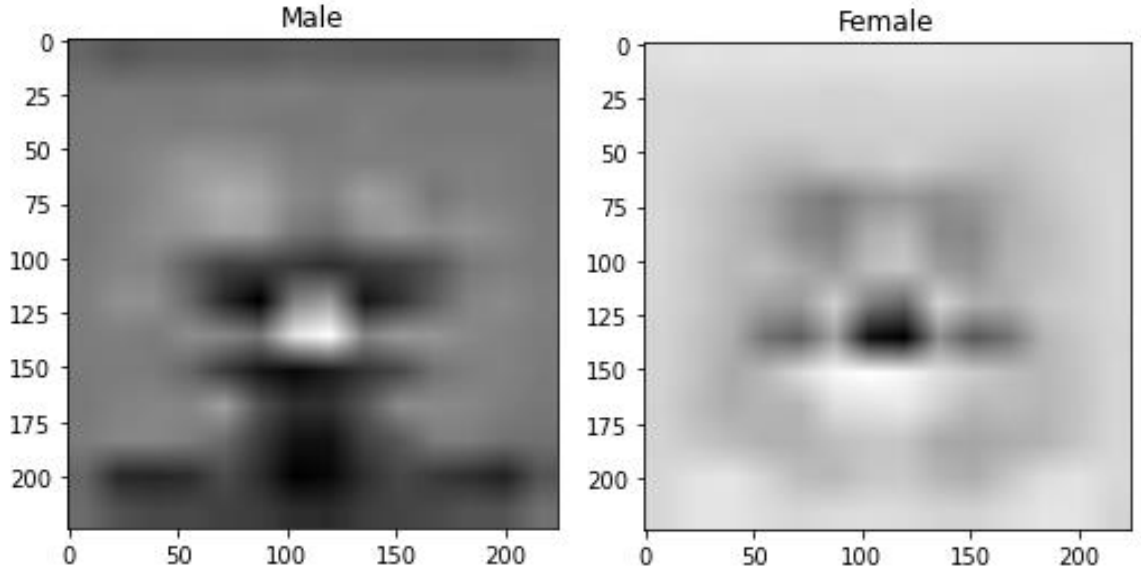
Tumore in einer Zelle zu erkennen ist eine Klassifikationsaufgabe, welche sehr einfach ist und wir im Alltag nur selten gebrauchen. Daher kam die Idee zusätzlich zu den Malaria Zellen das Netzwerk mit einer Genderklassifikation zu trainieren, um zu sehen, woran das Netz das Geschlecht ausgemacht hat, oder anders gesagt, was, dem Netzwerk nach, „männlich“ oder „weiblich“ in einem Gesicht ist. Dazu wurde das Netz mit dem Gender Datensatz trainiert, wobei folgende Ergebnisse herauskamen:



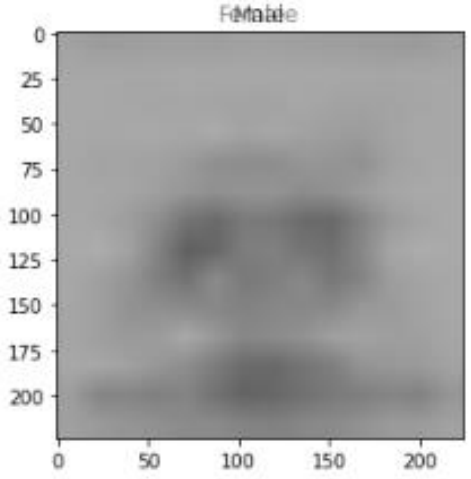
Hier ist zwar ein Unterschied zu erkennen: Der Mann wird eher am oberen Teil der Nase und am Kinn erkannt, die Frau an der Mund- Nasen Partie, jedoch kommt das sehr stark auf das trainierte Modell an.

3.2.3.1 Gender100

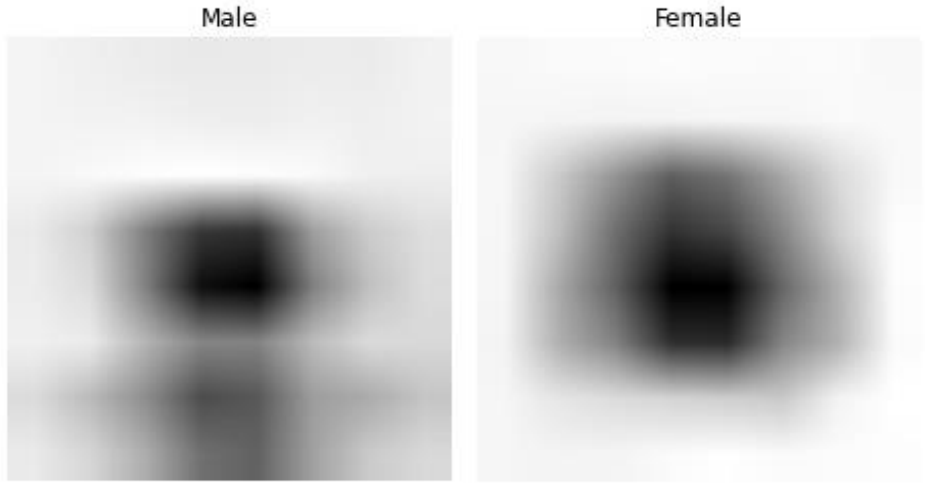
Um nicht hunderte von Bildern manuell sichtigen zu müssen, um zu erkennen, welche Bereiche besonders oft als männlich oder weiblich angesehen werden, generiert die PredictGender100Heatmaps Funktion nach dem System der PredictSingle Funktion eine Heatmap für ein Bild und gibt dieses zurück, anstatt es zu plotten. Die PredictGender100 Funktion ruft diese dann 100-mal auf, berechnet für jeden 128x128 Pixel den Mittelwert aus allen Heatmaps und stellt diesen als einzelne Heatmap dar. Dies funktioniert nur, weil die Gesichter in diesem Datensatz in der Regel zentriert und auf die Kamera gerichtet sind. In folgendem Beispiel ist gut zu erkennen, dass der Mann besonders an seinen Wangenknochen und seinem Kinn, und die Frau an ihrer Nase und der Augenpartie erkannt wird.



Wenn man die beiden Bilder mit der übereinander legt, ist bei diesem Modell leicht zu erkennen, dass alle Aktivierungen innerhalb einer Silhouette eines Menschen liegen, was die Korrektheit des CAMs bestätigt. Zum Überlagern zweier Bilder ist die Funktion OverlayMaleFemale da. Bei der Überlagerung der beiden ‚Gender100-Bilder' kommt folgendes Bild heraus:



Dies weicht jedoch sehr stark vom trainierten Modell ab. Ein weiteres Modell, das gleich trainiert wurde, hat zum Beispiel eine viel uneindeutiger Aktivierung:



4 RÜCKBLICK

4.1 WAS LIEF GUT / WO GAB ES SCHWIERIGKEITEN?

Die Arbeit lief insgesamt sehr gut. Ich bin sehr zufrieden mit den Ergebnissen. Insbesondere der Unterschied zwischen dem Modell, das auf den Loss der letzten Ebene trainiert wurde und dem Modell, das auf den gemeinsamen Loss trainiert wurde, ist deutlich zu erkennen.

Mein größtes Problem war im Nachhinein betrachtet der Umfang des Projektes, welchen ich um die Geschlechter Klassifikation erweitert habe. Nichtsdestotrotz bin ich froh auch dies gemacht zu haben, auch wenn dabei noch das Problem aufkam, dass die Modelle sehr stark voneinander abwichen und so nicht auf eine allgemeine Aussage geschlossen werden kann, welche Bereiche in einem menschlichen Gesicht männlich, bzw. weiblich aussehen.

4.2 WIE HÄTTE MAN DAS PROJEKT NOCH FORTFÜHREN KÖNNEN?

Einige Funktionen und Klassen, wie z.B. die beiden Netzwerke, könnte man zusammenfügen und allgemein den Code besser strukturieren. Dazu wäre es spannend auch ein Netz mit mehreren Klassen zu trainieren. Zur Optimierung könnte man außerdem sicherlich noch verschiedene Parameter des Netzes anpassen und testen. Eine weitere Idee, welche ich auch verfolgen wollte, ist die Aktivierungen der falschen Klasse zu plotten und so zu sehen, was z.B. an einem Mann ‚weiblich' aussieht. Dies ist jedoch mit dem aktuellen Netz kaum aussagekräftig, da es sehr stark vom trainierten Modell abhängt, wo die Klasse erkannt wird.

5 FAZIT

Wie im Rückblick schon angesprochen bin ich sehr zufrieden mit dem Projekt. Die Erforschung der Aktivierungen von zwischenebenen in einem Netzwerk war ein voller Erfolg. Der Code ist definitiv ausbaufähig, aber hat mir die Visualisierungen erbracht, die ich mir erhofft habe. Würde ich das Projekt nochmal neu starten, würde ich mich jedoch zuerst darauf konzentrieren, sauberen Code für eine bestimmte Funktion zu schreiben und diese dann ggf. erweitern, anstatt zu versuchen möglichst viele Features zu implementieren.

6 QUELLEN

6.1 DATENSÄTZE:

Gender: https://www.kaggle.com/datasets/ashishjangra27/gender-recognition-200k-images-celeba?resource=download

Malaria: https://www.kaggle.com/datasets/iarunava/cell-images-for-detecting-malaria?resource=download&select=cell_images

6.2 FÜR DAS PROJEKT IM ALLGEMEINEN GENUTZTE QUELLEN:

https://www.statworx.com/content-hub/blog/car-model-classification-3-erklarbarkeit-von-deep-learning-modellen-mit-grad-cam/#:~:text=Die%20Class%20Activation%20Map%20weist,Australian%20Terrier%E2%80%9C)%20%E2%80%93%20berechnet.

https://www.heise.de/select/ix/2018/12/1543309186455433

https://github.com/alexisbcook/ResNetCAM-keras/blob/master/ResNet_CAM.py

https://arxiv.org/pdf/1512.04150.pdf

https://www.youtube.com/watch?v=P9NdQG_vIvo

https://www.youtube.com/watch?v=gNRVTCf6lvY