wenn Sie das Buch lesen, schon nicht mehr. Durch die Anwendung von `argmax` stellen wir fest, welche Klasse die höchste Wahrscheinlichkeit hat und entsprechend als Vorhersage gilt.

```
testGenerator.reset()
yP = CNN.predict_generator(testGenerator, steps=len(testGenerator), verbose=True)
yPClass = np.argmax(yP,axis=1)
```

`testGenerator.classes` enthält die Labels gemäß den Verzeichnissen. Hiermit finden wir heraus viele Hunde und Katzen denn genau in unserer Testmenge sind und abschließend schauen wir danach, wie oft es mit der Klassifikation nicht funktioniert hat.

```
cats = np.sum(testGenerator.classes == 0)
dogs = np.sum(testGenerator.classes == 1)
catsAsDogs = np.sum( np.abs(yPClass[testGenerator.classes == 0] -0) )
dogsAsCats = np.sum( np.abs(yPClass[testGenerator.classes == 1] -1) )
confMatrix = np.array([[(cats-catsAsDogs)/cats, catsAsDogs/cats],
                       [dogsAsCats/dogs, (dogs-dogsAsCats)/dogs]])
print(confMatrix)
```

Als Konfusionsmatrix erhalte ich:

	y_P Cat	y_P Dog
y_T Cat	85.08	14.92
y_T Dog	06.07	93.93

Es kann sein, dass Sie wegen der zufälligen Initialisierung eine andere Verteilung bekommen. Fall Sie genau mit einem Netz weiterarbeiten wollen, laden Sie sich das ZIP-File von einer Webseite. Das hd5 dieses Netzes ist ebenfalls enthalten.

Der Klassifikator ist also wesentlich besser, wenn es um Hunde geht als um Katzen. Oder um genauer zu sein: Im Zweifelsfall tendiert er dazu etwas als Hund zu klassifizieren. Im nächsten Abschnitt versuchen wir unseren Klassifikator noch etwas besser zu verstehen.

11.4 Class Activation Maps und Grad-CAM

Wenn man ein CNN verwendet, um zum Beispiel eine Klassifizierung vorzunehmen, geht schon mal was schief. Manchmal sind es Dinge, die man algorithmisch lösen kann. Also zum Beispiel, indem man die Architektur des Netzes verändert oder die Optimierer. Nicht selten sind es aber einfach die Daten, auf die man aufsetzt. Ein schönes Beispiel stammt aus dem Paper [RSG16]. Hier sollten u. a. Wölfe und Huskys unterschieden werden. Die Bilder der Wölfe hatten jedoch immer Schnee im Hintergrund, während die Bilder der Huskys keinen Schnee im Hintergrund hatten. Auf einem Testset, welches nach dem gleichen Prinzip aufgebaut ist, funktioniert die Klassifikation hervorragend. Wird jedoch ein Husky mit Schnee im Hintergrund gezeigt, so wird dieser, sogar mit einer hohen durch das Netz angegebenen Genauigkeit, als Wolf klassifiziert. Das Netz hat durch einen Bias in den Daten zwar hohe Genauigkeiten, ist aber für viele Einsätze sinnlos; ein Dackel im Schnee ist weitab von einem Wolf.

Während es sich bei [RSG16] um ein bewusstes Experiment handelte, kommt das Problem auch in wesentlich kritischeren Anwendungen vor. Das Problem wird klarer, wenn man sich

die Arbeiten um das Paper [EKN+17] vor Augen führt. Hier geht es darum, Flecken auf der Haut als Hautkrebs – sogar verschiedene Arten – und harmlose Artefakte zu identifizieren. Das Paper hat einen sehr großen Eindruck gemacht und wurde tausendfach zitiert. Die Aussage war vereinfacht, dass die KI bzw. AI besser war als die meisten Ärzte und sich bald vermutlich jeder einfach sein Smartphone schnappen könnte, um damit schnell herauszufinden, ob es ein harmloses Muttermal oder Hautkrebs ist. Die Begeisterung macht es schwer, im Netz bzw. den Medien die eher kritischen Stimmen ausfindig zu machen. Der Datensatz weist nämlich einige Probleme auf, wie u. a. in [NKS+18] thematisiert wird. Die Bilder des Trainingssets enthalten teilweise Markierungen, um die Größen der Abbildung besser einschätzen zu können. Das Problem, auf das auch in [NKS+18] hingewiesen wird, ist, dass in dem Trainingsdatensatz Bilder mit Größenmarkierungen eher bösartige Hautveränderungen beinhalten, sodass es nah liegt, dass der Algorithmus unbeabsichtigterweise lernt, dass die Markierungen für bösartige Veränderungen stehen.

Häufig können Artefakte in der Datenerfassung wie der Schnee oder die Markierungen auf medizinischen Bildern unerwünschte Korrelationen hervorrufen, die die Klassifikatoren während des Trainings aufgreifen und nicht im Sinne der späteren Anwendung nutzen.

Diese Probleme können allein durch die Betrachtung der Rohdaten und Vorhersagen sehr schwer zu identifizieren sein. Das gilt besonders, weil man es hier mit unstrukturierten Daten zu tun hat, bei denen der Klassifikator selber auch die Merkmale generiert. Die Forschung auf dem Gebiet ist sehr aktiv und es gibt vielversprechende Ansätze. Ich möchte hier ein Beispiel für eine Methode präsentieren, die in ihrer Leistungsfähigkeit zwar begrenzt ist, aber einen ersten Einblick liefert. Außerdem erlaubt uns die Beschäftigung noch einmal mehr über CNN zu lernen. Man geht bei CNN davon aus, dass die erzeugten Merkmale an Abstraktion zunehmen. Was bedeutet das? Man weiß, dass der erste Layer im Wesentlichen recht einfache Muster auf der Basis von Kanten oder Gradienten erkennt. Es sind eher einfache Filter wie in Abbildung 11.11 auf Seite 367, welche die Voraussetzung für das weitere Lernen abstrakter Dinge schaffen. Die Hoffnung ist, dass die nächsten Layer sich dann von den Pixeln wegbewegen und eben ein Konzept wie z. B. Auge oder Schwanz erkennen. Optimalerweise gibt es am Ende nach unserer letzten Faltung dann Merkmale, die quasi für *Katzenaugen* oder *Pfote ohne Krallen* stehen. Es ist aber keineswegs sichergestellt, dass diese abstrakten Konzepte irgendetwas mit dem zu tun haben, was wir Menschen als ein abstraktes Konzept für ein Tier oder ein Ding verstehen. Sie sind mit Sicherheit in dem Sinne abstrakter als sie nicht mehr direkt auf der Pixelebene basieren.

Nimmt man an, dass der eine Filter am Schluss primär für einen Hunde- oder Katzenschwanz bzw. Katzenaugen oder Pfoten ohne Klauen steht, dann kommt man schnell hinter die Grundidee aus dem Paper [ZKL+16]. In diesem Paper wird ein Ansatz mit dem Namen **Global Average Pooling**, kurz **GAP**, propagiert. Der Name ist Programm und der bedeutet nichts anderes als Mittelwertbildung jeder Feature-Map. Statt eines Übergangs mittels Flatten in ein nachgelagertes dichtes Netz wird ein **GAP-Layer** verwendet; eben der Ansatz, den wir schon als Alternative zum Max-Pooling kennengelernt haben. Nimmt man an, dass ein Netz 14 Feature Maps hat, besteht der GAP-Layer aus deren Mittelwerten. Das bedeutet aber auch, dass man, um diese Technik anwenden zu können, eine spezielle Netzarchitektur benötigt. Hinter der Merkmalsgenerierung durch Faltung und Pooling darf nur noch ein GAP-Layer und ein vollverbundener Output-Layer folgen. Es erfolgt kein Flattening. Das Netz muss mit der in Abbildung 11.17 dargestellten Architektur trainiert werden, bevor man diese nutzen kann, um zu visualisieren, welche Aspekte im ursprünglichen Bild am meisten zu einer Klassifizierung bei-

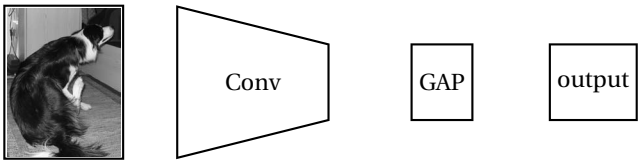

Abbildung 11.17 Netzwerkarchitektur mit GAP-Layer

getragen haben. Diese Darstellung, die wir erhalten wollen, nennt man **Class Activation Maps**. Es ist eine skalare Bitmap mit Werten, die den Grad der Aktivierung angeben.

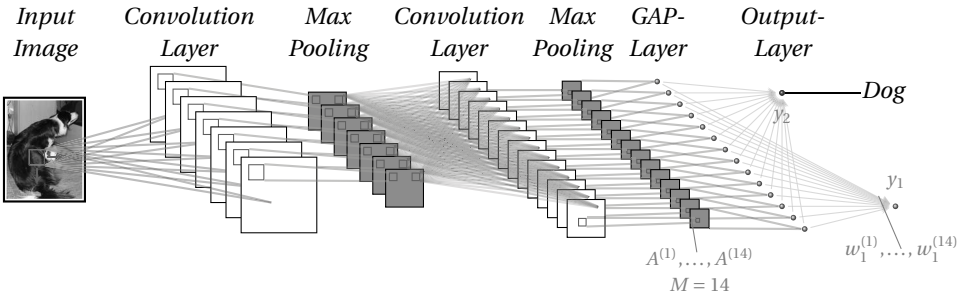

Abbildung 11.18 Detail-Netzwerkarchitektur mit GAP-Layer

Der erste Schritt ist das Bilden des Mittelwertes über eine Feature Map $A^{(m)}$:

$$\text{Mittelwert} = \sum_{i,j}^{I,J} A_{i,j}^{(m)} \tag{11.9}$$

Anschließend werden diese mit den Gewichten $w_c^{(m)}$ multipliziert. c ist dabei der Index für die Verbundene Klasse, während m für die Nummer der Feature Map steht. Die Entscheidung für eine Klasse geschieht dann gemäß der Gleichung (11.10).

$$y_c = \sum_{m=1}^{M} \left(\frac{1}{I \cdot J} \sum_{i,j}^{I,J} A_{i,j}^{(m)} \right) \cdot w_c^{(m)} \tag{11.10}$$

Das ist bis jetzt nur eine etwas andere Art, die Klassifikation durchzuführen. Wie hängt dies nun mit der Visualisierung der Aktivierung zusammen. Dazu stellen wir die Gleichung (11.10) ein wenig um.

$$y_c = \frac{1}{I \cdot J} \sum_{i,j}^{I,J} \underbrace{\left(\sum_{m=1}^{M} A_{i,j}^{(m)} \cdot w_c^{(m)} \right)}_{\text{CAM}} = \frac{1}{I \cdot J} \sum_{i,j}^{I,J} \underbrace{\left(\sum_{m=1}^{M} A^{(m)} \cdot w_c^{(m)} \right)}_{\text{CAM}} \frac{1}{I \cdot J} \sum_{i,j}^{I,J} \text{CAM}_{i,j} \tag{11.11}$$

Hierbei vertauschen wir die Summen über die Dimensionen I und J der Feature Map und die Summe über alle Feature Maps. In der inneren Klammer von Gleichung (11.11) erhalten wir nun eine Matrix, welche an den Stellen – über alle Feature Maps hinweg gemittelt – große Einträge hat, wo eine starke Aktivierung für die Entscheidung *Klasse c* vorliegt. Der Ausdruck *gemittelt* ergibt sich wegen der Division durch $I \cdot J$, also der Größe der Matrix. Die Darstellung

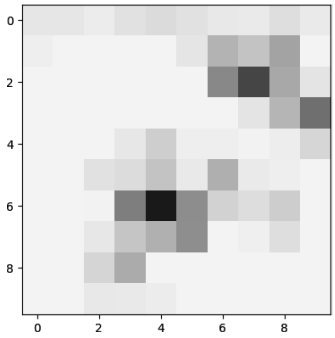

Abbildung 11.19 Beispielhafte Darstellung einer Class Activation Map (CAM)

$A^{(m)}$ soll deutlich machen, dass es eben Matrizen sind, die mit $w_c^{(m)}$ gewichtet und anschließend durch die Summation überlagert werden.

Diese CAM-Matrix können wir nun verwenden, um im ursprünglichen Bild diese Stellen für einen Menschen sichtbar zu machen. Dazu muss man sich aber vor Augen führen, dass die Feature Map durch die Pooling-Schritte und den Verschnitt am Rand – es sei denn, es wurde durchgängig ein Padding angewendet – deutlich kleiner geworden ist. Daher ist es üblich, diese eine Class Activation Map, wie sie beispielhaft in Abbildung 11.19 gezeigt wird, hochzuskalieren und diese skalierten Werte mit dem ursprünglichen Bild zu überlagern.

Das Problem ist, dass man mit dem GAP-Ansatz in der Regel schlechtere Ergebnisse erzielt als mit einem dichten Netz. Wir würden also bei gleich komplexen Modellen mehr Transparenz gegen weniger Genauigkeit tauschen. Zum Glück wurde ein Jahr nach dem GAP-Ansatz noch ein weiteres Paper publiziert. In [SCD+17] wird ein Ansatz vorgestellt, mit dessen Hilfe man versucht, die alten Architekturen weiterverwenden zu können und gleichzeitig doch mehr Transparenz zu erhalten. Das Ergebnis ist insofern sehr angenehm, als es auf viele Netze angewendet werden kann. Die Zuordnung zu den Aktivierungen für die Klassenentscheidung ist aber weniger gradlinig und an einigen Stellen eher heuristisch motiviert.

Als Erstes konzentrieren wir uns auf den Term $A^{(m)} \cdot w_c^{(m)}$ in Gleichung (11.11). Die Feature Maps existieren natürlich auch bei der klassischen CNN-Architektur, aber statt $w_c^{(m)}$ haben wir einen wesentlich komplexeren Zusammenhang, der durch ein neuronales Netz gegeben ist. Hier nimmt man eine Anleihe bei der Backpropagation, bei der man auch für die Gewichtsänderung versucht, die Einflüsse durch die Gradienten zu motivieren. Allerdings ist in diesem Fall nicht die ganze Feature Map mit dem Output-Neuron y_c verknüpft, sondern einzeln Einträge der Feature Map, welche in einen Vektor durch Flattening umformatiert werden.

$$A^{(m)} \cdot w_c^{(m)} \rightsquigarrow A^{(m)} \cdot \underbrace{\left(\sum_{i,j}^{I,J} \frac{\partial y_c}{\partial A_{i,j}^{(m)}} \right)}_{\text{Summierter Gradient}}$$

Nimmt man an, dass ein Netz vorliegt, in dem die Zusammenhänge linear sind und zuvor ein Average Pooling durchgeführt wurde, ist damit der Ansatz über einen GAP-Layer ein Spezialfall dieser Art, Gewichte zu berechnen.

Die gesamte Formel aus [SCD+17] lautet:

$$\text{ReLU}\left(\sum_{m=1}^{M}\left(\underbrace{\frac{1}{I\cdot J}\sum_{i,j}^{I,J}\frac{\partial y_c}{\partial A_{i,j}^{(m)}}}_{\text{Gemittelter Gradient}}\right)\cdot A^{(m)}\right)=\text{ReLU}\left(\sum_{m=1}^{M} w_c^{(m)}\cdot A^{(m)}\right) \tag{11.12}$$

In (11.12) tritt der gemittelte Gradient an die Stelle des Gewichtes für die Feature Map beim GAP-Ansatz. Anschließend verhindert die ReLU-Funktion, dass negative Werte auftreten können. Das ist einer der Aspekte, den ich mit *heuristisch* vorab meinte. Es liegt die Idee zugrunde, dass negative Gewichte beim ursprünglichen GAP-Verfahren eher ausdrücken wollen; *gehört nicht zu der Klasse* und positive *gehört zu der Klasse*. Daneben haben die Autoren des Grad-CAM rumprobiert – und was positiv ist diese Tests auch in die Veröffentlichung aufgenommen – und bei Ihnen klappte die Darstellung mit ReLU besser. Das ist aber kein mathematischer Beweis, nicht einmal für einen Spezialfall.

Diese Formel werden wir nun einmal versuchen umzusetzen. Dazu müssen wir Gradienten berechnen, und das wollen wir gerne automatisch berechnen lassen. Hierzu bietet TensorFlow – nicht Keras – eine geeignete Funktionalität. Entsprechend müssen wir TensorFlow selbst einbinden. Darüber hinaus soll der Gradient nicht über das ganze Modell berechnet werden, sondern nur von den Feature Maps bis zu den Output-Neuronen. Hierzu müssen wir quasi ein Teilmodell bilden. Um dieses Teilmodell zu erzeugen, nutzen wir ausnahmsweise einen Aspekt der **Model Class API** von Keras statt der Sequential. Entsprechend importieren wir die Klasse **Model**.

```
80
81  import tensorflow as tf
82  from tensorflow.keras.models import Model
83  def heatmap(img, model):
84      for l in model.layers:
85          if isinstance(l, Conv2D): lastConvLayer = l
86      calcFeaturesAndPred = Model([model.input], [lastConvLayer.output, model.output])
```

Als Erstes Suchen wir in Zeile 85 den letzten Layer vom Typ `Conv2D`, also die letzte Faltung. Liegt dahinter noch ein Pooling, wird dies in die Gradientenbildung einbezogen. In Zeile 86 bauen wir uns mit der Model-API ein neues Modell auf der Basis unseres bereits trainierten Modells. Der Unterschied ist hier lediglich, dass wir zwei Outputs festlegen, nämlich die bekannten Output des Softmax und zusätzlich den Output des letzten Faltungslayers `lastConvLayer.output`.

Nun benötigen wir nach langer Zeit das in Abschnitt 3.5 erwähnte `with`-Statement. Es ist die in TensorFlow vorgesehene Methode, um bei der parallelen Ausführung mit Ressourcenkonflikten und der Bereinigung umzugehen. Mit diesem `with`-Statement verknüpft ist hier `tf.GradientTape`. Es ist Teil der TensorFlow API zur automatischen Differenzierung, also der Berechung des Gradienten einer Funktion in Bezug auf die Eingabevariablen. Die Bezeichnung `tape` kommt daher, dass alle Operationen, die im Kontext eines `tf.GradientTape` ausgeführt werden quasi auf ein *Band* (eng. Tape) aufzeichnet. TensorFlow verwendet anschließend dieses Band bzw. Tape, um die Gradienten, die mit jeder aufgezeichneten Operation verbunden sind zu berechnen. Das funktioniert aber lediglich mit TensorFlow-Variablen. Entsprechend wandeln wir in Zeile 87 unser Bild in eine solche Variable um. Mit diesem Typ kann man fast genauso arbeiten wie mit NumPy-Arrays. In Zeile 89 müssen wir wieder eine leere Dimension hinzufügen, da wir eben nur ein Bild durch die Verarbeitung verfolgen wollen. Die Rückga-

bewerte unserer konstruierten Funktion sind jetzt wie gewünscht Feature Maps und die Vorhersagen. Wir sind nur daran interessiert, warum sich das Netz für die vorhergesagte Klasse entschiedenen hat, daher ermitteln wir deren Index (Zeile 90) und nutzen diesen anschließend, um den zugehörigen Outputwert zu ermitteln. Die Feature Maps haben vier Achsen. Die erste ist unsere künstliche Dimension, die wir hinzugefügt haben, da wir nur ein Bild haben. Die nächsten beiden stehen für die Größe der Map und die letzte für die Anzahl der Feature Maps.

```
87      img = tf.Variable(img);
88      with tf.GradientTape() as tape:
89          featureMaps, predictions = calcFeaturesAndPred(img[np.newaxis,...])
90          maxActivation = np.argmax(predictions[0])
91          predictedClassEntry = predictions[:, maxActivation]
```

Nachdem die Berechnungen ausgeführt sind, nutzen wir nun in Zeile 93 die auf dem Tape aufgezeichneten Daten, um die Gradienten aus Gleichung (11.12) zu berechnen. Die Syntax ist so, dass das erste Argument die Variable ist, die differenziert werden soll, und das zweite di,e nach der differenziert wird. In Zeile 94 bilden wir den Mittelwert für jede Feature Map. Hierbei ist es hilfreich, das `axis`-Argument gleich einem Tuple zu setzen. Die Werte $(0,1,2)$ bedeuten, dass der Mittelwert über eben diese Achsen gebildet wird. Der Gradient hat die Dimensionen: *(Anzahl von Datensätzen, x-Auflösung der Feature Map, y-Auflösung der Feature Map, Anzahl der Feature Maps)*. Die erste Dimension ist bekanntlich unsere künstliche, da wir nur ein Bild haben. Wird also über die ersten drei Achsen gemittelt, erhalten wir so viele Werte, wie wir Feature Maps haben, bzw. so viele Werte, wie die letzte Dimension angibt.

```
92
93      grads = tape.gradient(predictedClassEntry, featureMaps)
94      pooledGrads = np.mean(grads, axis=(0, 1, 2))
```

Nun multiplizieren wir gemäß Gleichung (11.12) die Feature Maps mit den Gewichten in Zeile 95 und erhalten $w_c^{(m)} \cdot A^{(m)}$. Anschließend bilden wir in Zeile 96 die Summe über alle Feature Maps.

```
95      weightedFeatures = featureMaps * pooledGrads
96      heatmapImg = np.sum(weightedFeatures, axis=-1).squeeze()
```

Am Schluss wenden wir ReLU an, was faktisch durch `np.maximum` geschieht. In den letzten zwei Zeilen der Funktionen geben wir die Werte zurück. Da wir aber die Heatmap als Bild weiterverarbeiten wollen, normieren wir diese zuvor vor. Sinnvolle Ansätze sind da auf den Bereich 0 bis 1 oder bis 255. Durch den Einsatz von Tape ist `predictions` eine TensorFlow-Variable. Mit der Methode `numpy` wandeln wir diese wieder um.

```
98      heatmapImg = np.maximum(heatmapImg, 0)
99      if np.max(heatmapImg) > 0 : heatmapImg /= np.max(heatmapImg)
100     return heatmapImg, predictions.numpy()
```

Jetzt haben wir eine Class Activation Map, müssen diese aber noch über unser ursprüngliches Bild legen. Hierzu müssen wir das Bild skalieren und ein paar weitere Anpassungen vornehmen. Da dies jetzt eher technisch ist, versuche ich kurz alles zusammenzufassen. `cm.Reds` wandelt als Klasse skalare Daten in eine RGBA-Darstellung gemäß der Colormap um. Rottöne passen hier schön zu dem Namen Heatmap, aber natürlich geht alles, was sequenziell ist,

und Ihren Geschmack trifft. Den Alpha-Kanal brauchen wir nicht, deshalb unterdrücken wir diesen mithilfe von `[...,:3]`. In PIL gibt es eine optisch schöne Methode zum Skalieren. Um diese nutzen zu können konvertieren wir etwas hin und her. Die Methoden dazu kommen aus der **image**-Klasse von Keras.

```
from matplotlib import cm
from PIL import Image as PILImage
from tensorflow.keras.preprocessing import image

def overlayHeatmap(img, heatmapImg):
    heatmapImg = cm.Reds(heatmapImg)[..., :3]
    heatmapImg = image.array_to_img(heatmapImg)
    heatmapImg = heatmapImg.resize(img.shape[:-1], resample=PILImage.BICUBIC)
    heatmapImg = image.img_to_array(heatmapImg)
```

Die nächsten drei Zeilen sind nur da, falls Sie die gleiche Ausgabe haben wollen wie im Buch. Eigentlich ist es in bunt natürlich etwas hübscher.

```

    imGray = 0.2989*img[:,:,0] + 0.5870*img[:,:,1] + 0.1140*img[:,:,2]
    imGray = cm.gray(imGray)[..., :3]
```

Nun geht es daran, die Heat bzw. Class Activation Map mit dem Bild zu überlagern. Das passiert einfach per Addition. Die Gewichte sorgen dafür, dass die Class Activation Map dominant ist und das Bild nur leicht im Hintergrund sichtbar ist. Wenn Sie es gerne anders haben wollen, ändern Sie es halt. Die beiden Zeilen bewirken, dass beide Bilder gleich skaliert sind, auch wenn sich das Verhalten der Konvertierungsmethoden mal ändert oder nicht gleich ist.

```

    if np.max(imGray) <= 1: imGray = 255*imGray
    if np.max(heatmapImg) <= 1 : heatmapImg = 255*heatmapImg
    superimposedImg = np.minimum(heatmapImg * 0.6 + 0.2*imGray, 255).astype(np.uint8)
    return superimposedImg
```

Nun wollen wir das alles auch mal ausprobieren. Dazu greifen wir auf ein paar Bilder aus meinem privaten Fundus zurück. Diese sind im ZIP-File auf der Webseite enthalten. Sie können aber auch Bilder aus der Testmenge auswählen. Diese enthält aber noch eine kleine Gemeinheit, die ich mit Ihnen besprechen möchte. Ich nehme nämlich auch drei Bilder, die weder Katzen noch Hunde enthalten. Darüber hinaus noch eine freigestellte Katze, auf die wir später kommen.

```

import matplotlib.pyplot as plt
plt.rcParams.update({'figure.max_open_warning': 0})
imageList = ['hund1.jpg', 'hund2.jpg','hund3.jpg', 'hund4.jpg',
             'katze1.jpg', 'katze2.jpg', 'katze3.jpg', 'katze4.jpg',
             'ratte.jpg', 'luchs.jpg', 'wolf.jpg', 'katze1freigestellt.jpg']
```

Zunächst laden wir in einer Schleife alle Bilder und skalieren diese direkt beim Laden auf die Größe, die das CNN verarbeiten kann. Unser Netz ist trainiert, Voraussagen für Daten zu treffen, die auf den Bereich 0 bis 1 normiert wurden, entsprechend normieren wir das Bild bzgl. der Wert für die weiteren Schritte.

Die Vorhersage muss immer als Pipeline gedacht werden. Daten werden geladen, vorverarbeitet und dann die Prognose durchgeführt. Wenn die Quelle wechselt ist das eine ganz unangenehme Fehlerquelle. Vielleicht waren vorher Daten immer schon auf [0,1] normiert und es erfolgte daher keine Normierung im Code. Wechselt die Quelle, bekommt man auf einmal z. B. Daten im Bereich [0,255]. Da die Bilddimensionen stimmen, gibt es keine Fehlermeldung, aber völlig sinnlose Vorhersagen. Solche Fehler zu finden, kostet manchmal Stunden!

```
125 for imageFile in imageList:
126     imgSize = CNN.input_shape[1:-1]
127     img = image.load_img(imageFile, target_size=imgSize)
128     img = image.img_to_array(img)/255.0
129     plt.figure(); plt.imshow(img)
```

Nun erstellen wir mit unserer Funktion die Heatmap und lassen uns gleichzeitig über die Vorhersage informieren.

```
130     hm, predictions = heatmap(img, CNN)
131     plt.figure(); plt.imshow(hm, cmap=cm.Reds)
132     if np.argmax(predictions[0]) == 0: classstring = 'cat'
133     else: classstring = 'dog'
134     print(imageFile,' predicted as ',classstring, ' with ', predictions)
```

Anschließend überlagern wir die Heatmap mit dem Original und speichern das Ergebnis als PNG, um es z. B. in einem Buch abdrucken zu können.

```
135     fusion = overlayHeatmap(img, hm)
136     plt.figure(); plt.imshow(fusion)
137     name = imageFile.split('.')[0]
138     plt.title(name+str(predictions))
139     name = 'heat'+name+'.png'
140     image.array_to_img(fusion).save(name, 'PNG')
```

Den Code oben kann man schlecht wiederverwenden. Kopieren Sie die beiden Funktionen `heatmap` und `overlayHeatmap` inklusive aller benötigten Einbindungen einmal in eine Datei `gradCAM.py`. Dann können wir diese später noch benutzen.

Abbildung 11.20 Heatmaps und Vorhersagen für Bilder mit Hunden und Katzen

Nun schauen wir mal, was wir rausbekommen haben. Wir wussten aus Abschnitt 11.3 schon, dass unser Netz ein Hundefreund ist und im Zweifel eher dazu tendiert, eine Katze zum Hund zu machen. Abbildung 11.20 zeigt die Prognosen und die Heatmaps für drei Hundebilder. Alle drei sind richtig als Hund klassifiziert und das auch mit einer hohen Sicherheit von Seiten des CNN. Man sieht, dass sich das Netz auf Schnauzen und Schwänze zu konzentrieren scheint. Das dritte Bild ist aber verdächtig. Es wurden viele Bereiche im Bild berücksichtigt, auf denen kein Hund zu sehen ist. Entweder sind mehr Hundebilder in freier Wildbahn im Archiv oder das Netz lässt sich von Strukturen im Hintergrund leicht ablenken. Solchen Klassifizierungen sollten wir mit etwas Vorsicht begegnen. Mal sehen, wie es bei den Katzen weitergeht.

Wie man aus Abschnitt 11.3 vermuten konnte, treten die Fehler eher bei den Katzen auf. Die erste Katze wurde mit beinahe 80% als Hund klassifiziert. Wir sehen aber auch, dass ähnlich wie bei der dritten Katze für die Klassifizierung eher der Hintergrund den Ausschlag gegeben hat.

Abbildung 11.21 Heatmap & Vorhersage für freigestellte Katze

Das kann legitim sein, weil man z. B. ein Kamel eher nicht in der Antarktis vermutet, aber hier erscheint es weniger verständlich. Der Hintergrund lenkt scheinbar ab. Wie wäre es wohl ausgegangen, wenn die erste Katze ohne Hintergrund hätte eingeordnet werden sollen? Für den Test habe ich diese einmal unfachmännisch freigestellt und erneut klassifizieren lassen. Abbildung 11.21 zeigt eindrucksvoll, dass sich das Netz nun wieder auf Ohren und Fellaspekte konzentriert. Die Katze wird nun auch zu 90% als Katze erkannt.

Wie sieht es mit den eingebrachten Bildern einer Farbratte, eines Luchses und eines Wolfes aus? Biologisch wäre es plausibel, wenn sich das Netz beim Luchs für Katze, beim Wolf für Hund und bzgl. der Ratte sich das Netz als knappes Ergebnis für Katze oder Hund entscheiden würde. Wie man in Abbildung 11.22 deutlich sieht, geht unser Netz aber nicht nach der Biologie, sondern nach Mustern. Beim Wolf ist noch alles, wie wir es erwarten. Es schaut auf Gesicht und Körper und sagt *Hund*. Beim Luchs lässt es sich wieder vom Hintergrund ablenken und klassifiziert ihn ebenfalls als *Hund*. Entgegen der generellen Tendenz unseres Netzes wird die Farbratte mit großer Sicherheit zur Katze. Vermutlich weil der Spielplatz der beiden Haustiere sich ähnelt. Die Ratte jedenfalls kommt in der Heatmap kaum vor.

Machen Sie sich doch mal einen Spaß daraus und schicken Sie Bilder von Tieren mit und ohne Hintergrund durch das Netz und sehen Sie sich die Ausgaben genau an. Vielleicht bekommen Sie so ein besseres Gefühl für das Verhalten des CNN.

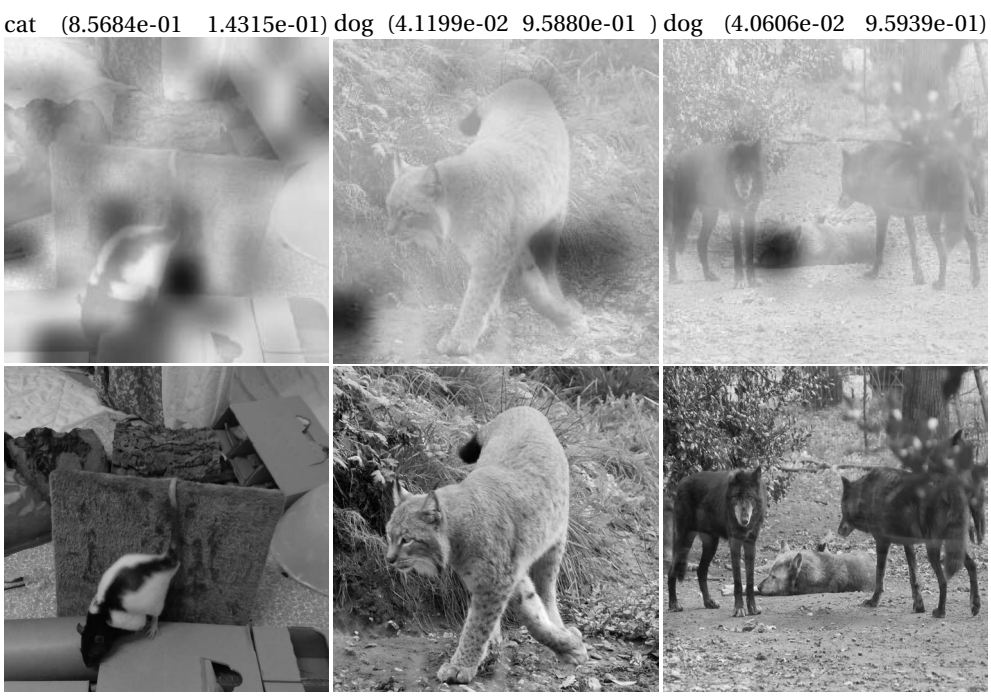

Abbildung 11.22 Heatmaps und Vorhersagen für Bilder mit nicht berücksichtigten Tieren

Nachdem die Idee einmal in der Welt war, wurde natürlich an unterschiedlichen Orten geschaut, ob man da nicht noch mehr rausholen kann. Ein Problem ist, dass die Größe der Feature Maps am Schluss automatisch die Genauigkeit der Lokalisation beeinflusst. Hätten wir noch ein paar mehr Poolings eingebaut, wäre noch weniger über gewesen, um herauszufinden, was interessant war. Ein anderes ist, dass wir händisch vorgehen müssen. Man muss Bilder als Mensch stichpunktartig analysieren und schauen, ob es einem plausibel vorkommt. Varianten wie Grad-CAM++, siehe [CSHB18], aus dem Jahr 2018 lösen diese Probleme noch nicht. Es geht eher um punktuelle Verbesserungen, um den Preis von mehr Heuristik bzw. weniger leicht nachvollziehbaren Formeln. Ich hoffe, dass Sie durch das Grad-CAM- bzw. GAP-Verfahren einen Einblick in die Möglichkeiten und Grenzen bekommen haben. Nun werden wir im nächsten Kapitel sehen, ob wir bzgl. unserer Hunde und Katzen nicht noch besser werden können und ggf. auch die starke Sensitivität für den Hintergrund etwas verringern können.

11.5 Transfer Learning

Wir haben im Abschnitt 11.3 unseren Datenbestand durch Data Augmentation aufgewertet. Dies kann auch ein wenig helfen, wenn man nur auf vergleichsweise wenig Daten zum Training zurückgreifen kann bzw. wenn es darum geht, die Erkennung unempfindlicher gegenüber Rotationen zu machen. Wie wir schon diskutiert haben, kann man hiermit jedoch nur beschränkt neue Informationen dem Netz zuführen. Ein anderer Ansatz, um ggf. mit vergleichsweise ge-