# Architektur Basis-Modell

## Generell (2 Klassen)

* Modelle immer mit Binärer Klassifikation ausprobiert und getestet:
  + 10'000 Bilder pro Klasse
  + Bird vs Dog
  + Zweites Datenset vorhanden für Transfer-Learning und ausprobieren wie es mit 4 Klassen funktioniert
    - Fish, Lizard, Monkey, Snake
* Basis-Architektur:
  + Input: Grösse von 224 (gewählt weil guter Mix von Bildqualität aber Trainingsgeschwindigkeit)
  + Dann Normalisierung zu 0-1
  + Dann 4x Blöcke
    - Grösser-werdende Filter-Anzahl
    - Convolution
    - Batch-Normalisierung
    - Pooling
    - Dropout
  + GlobalMaxPooling
  + Flatten
  + 2x Dense-Layer mit 256 und 512 Neuronen
  + Normalisierung
  + Dropout
  + Dense-layer mit N-Anzahl-Klassen Neuronen
* **Architekturen welche signifikant grösser waren auf unserer Hardware nicht trainierbar. (unmöglich)**
* Verwenden direkt die logits für die Berechnung der Metriken. Sonst hatten wir Fehler bei der Berechnung mit Pytorch
* Filter-Anzahl von über 128 (basis Wert) war Training zu langsam. Deshalb wurde diese Grösse gar nicht erst miteinbezogen.
* Batch-Grösse von 64 hat bei unserem System das schnellste Training ermöglicht.
* Augmentation direkt im Modell ist ebenfalls vorteilhaft, da dann jede Epoche neuen Augments verwendet werden.
* RandomRotation aus der Augmentation entfernt, da damit die Resultate immer schlechter waren.
* Early Stopping wird eingesetzt, um ein Overfitting zu verhindern. Zusätzlich wird ein learning-rate scheduler verwendet, der die lr heruntersetzt, wenn die accuracy nicht besser wird. Der lr-scheduler hat eine tiefere patience als das early-stopping. Dies bedeutet, dass wenn sich die accuracy nicht ändert zuerst die learning-rate anpasst und wenn dann immer noch nichts passiert, wird gestoppt.

## Tuning (2 Klassen)

Wir haben nicht alles gleichzeitig getuned sondern einzelne Teile der Architektur. Damit sind wir schrittweise vorgegangen und haben so für einzelne Teile der Architektur die besten Parameter auswählen können. Anschliessend haben wir von den verschiedenen Durchläufen die besten Werte weiterverwendeten. Die Reihenfolge ist dieselbe wie im Dokument hier. Als Basis wurde die oben beschriebene Architektur genommen. Diese wurde dann durch das Tuning weiter angepasst. Für das Tuning haben wir immer die Validation-Accuracy verwendet, da dies der wichtigste Wert für ein Modell ohne Overfitting ist. Es wurde jeweils pro Durchlauf für 4 Epochen trainiert. Aus Zeitgründen wurden nicht mehr verwendet, da sonst das Tuning zu lange gehen würde.

## Tuning von Basis-Filter, Learning-Rate und Pool-Type (2 Klassen)

Hier wird die Learning-Rate und zwei CNN-Spezifische Entscheidungen getroffen. Die Anzahl Filter in jedem Block und der Typ von Pool (max oder avg). Bei den anderen Sachen (wie zum Beispiel stride, pool-grösse, etc…) waren wir uns sicher, dass diese gut gewählt sind. Basis-Filter ist der tiefste Wert der 4 Convolution-Blöcke. Dieser wird dann \*1, \*2, \*4, \*8 gerechnet und das sind die Anzahl-Filter in jedem Block.

* Beste Werte:
  + Basis-Filter 8, pool\_type max und learning-rate 0.001. Mit Tuning
* Kleinste Modellgrösse also beste Resultate 🡪 wahrscheinlich am wenigsten Overfitting

Trial 05 summary

Hyperparameters:

filter\_size: 8

pool\_type: max

learning\_rate: 0.001

Score: 0.8165000081062317

Trial 12 summary

Hyperparameters:

filter\_size: 64

pool\_type: max

learning\_rate: 0.001

Score: 0.8159999847412109

Trial 15 summary

Hyperparameters:

filter\_size: 64

pool\_type: max

learning\_rate: 0.0001

Score: 0.812749981880188

Trial 13 summary

Hyperparameters:

filter\_size: 32

pool\_type: max

learning\_rate: 0.001

Score: 0.8004999756813049

Trial 19 summary

Hyperparameters:

filter\_size: 32

pool\_type: avg

learning\_rate: 0.0001

Score: 0.7637500166893005

## Tuning von Output-Dense-Layern (2 Klassen)

Mit diesem Tuning wurde geschaut was die beste Kombination für die Output Layer ist. Dabei wurde einerseits untersucht wie viele Dense-Schichten (1-3) und wie viele Neuronen pro Schicht (64, 128 oder 256).

Dabei wurde auch hier wieder beispielsweise in der ersten Schicht 64\*1, 64\*2, 64\*4 gerechnet.

* Beste Werte: 3 Schichten mit 64 als Basis-Wert
* Also auch hier wieder sind kleinere Werte besser
* Resultiert in den folgenden Dense-Schichten:
  + Dense 64
  + Dense 128
  + Dense 256

Trial 03 summary

Hyperparameters:

dense\_layer\_amount: 3

dense\_layer\_base: 64

Score: 0.843999981880188

Trial 01 summary

Hyperparameters:

dense\_layer\_amount: 2

dense\_layer\_base: 128

Score: 0.8370000123977661

Trial 07 summary

Hyperparameters:

dense\_layer\_amount: 1

dense\_layer\_base: 128

Score: 0.8332499861717224

Trial 08 summary

Hyperparameters:

dense\_layer\_amount: 2

dense\_layer\_base: 64

Score: 0.8322499990463257

Trial 04 summary

Hyperparameters:

dense\_layer\_amount: 3

dense\_layer\_base: 256

Score: 0.8299999833106995

## Tuning von Regularisierung (Batch-Normalisierung und Dropout) (2 Klassen)

Damit wollten wir herausfinden was am besten hilft gegen Overfitting. Ob man in den einzelnen Blöcken schon Regularisierung soll und/oder erst nachher.

* Beste Werte konnten ohne jegliches DropOut und mit einer Batch-Normalisierung in den Schichten und in der Output-Schicht erreicht werden.

Trial 04 summary

Hyperparameters:

inter\_dropout: False

inter\_batch\_norm: True

output\_dropout: False

output\_batch\_norm: True

Score: 0.827750027179718

Trial 00 summary

Hyperparameters:

inter\_dropout: True

inter\_batch\_norm: True

output\_dropout: False

output\_batch\_norm: True

Score: 0.8180000185966492

Trial 05 summary

Hyperparameters:

inter\_dropout: False

inter\_batch\_norm: False

output\_dropout: True

output\_batch\_norm: False

Score: 0.815500020980835

Trial 13 summary

Hyperparameters:

inter\_dropout: False

inter\_batch\_norm: True

output\_dropout: True

output\_batch\_norm: False

Score: 0.8142499923706055

Trial 06 summary

Hyperparameters:

inter\_dropout: True

inter\_batch\_norm: True

output\_dropout: True

output\_batch\_norm: False

Score: 0.812250018119812

## Training Basis-Modell (2 Klassen)

Insgesamt 13 Epochen dann hat der Callback das Training gestoppt. Die Weights wurden von Epoche 7 übernommen, da dort der val\_accuracy am besten war. Training hat viele Spikes, also wahrscheinlich Overfitting.

* **Parameter: 71793**

A screenshot of a graph

Description automatically generated

## Transfer-Learning Basis-Modell (4 Klassen)

* Basis-Modell wurde geladen
* Dann wurde der letzte Layer entfernt und durch ein Dense mit 4 Units ersetzt
  + 4 Units weil nun mit Datenset mit 4 Klassen getestet wurde
* Dann wurden die Weights aller Layer ausser dem neu dazugefügten Dense gefreezed (layer.trainable = false)
* Dann wurde erneut ein Training mit fast gleicher Herangehensweise durchgeführt wie auch schon beim Basis-Modell
* Resultate schlecht. Stoppt bei 50% auf besser zu werden
  + Entweder Fehler von uns oder Modell einfach zu klein für transfer
* Insgesamt 17 Epochen trainiert. Aber beste Weights von Epoche 9 wurden gespeichert

A screenshot of a computer

Description automatically generated

# Grösseres Residual Modell (2 Klassen)

## Generell Architektur (2 Klassen)

* Ohne Tuning Parameterauswahl. Sondern von Tuning-Resultate des Basis-Modells inspiereren lassen, dann durch trial und error.
* «Residual Connections». Mit diesen erhofft man sich in früheren Schichten erkannte Features wieder aufgreifen zu können.
* **358’353 Parameter**

Kontrast / Brightness Augmentation

* Rescaling auf 0-1
* Eine Input-Convolution + Batch-Normalisierung für anfangs-residual
* Dann Vier Convolution Schichten (16, 32, 64, 128)
  + Pro Schicht 2x Convolution
  + 2x Batch-Nprm
  + Max-Pooling
  + Residual berechnen und addieren
* Dann Klassifikation
  + GlobalMax
  + 2x Dense (128, 256)
  + BatchNorm
  + Dropout
  + Dense (Anzahl Klassen)

## Resultate Training (2 Klassen)

* 16 Epochen trainiert
* Weights von Epoche 10 wiederhergestellt
* Allgemein gut. Besser aber nicht viel besser als Basis-Modell aber hat viel mehr Parameter.
* Callback für early stopping und beste weights auswählen war auch hier wieder gute Entscheidung
* Training etwas «ruhiger»

A screenshot of a computer screen

Description automatically generated

## Resultate Training (4 Klassen)

* Selbes Modell anschliessend trainiert mit dem Datensatz der 4 Klassen beinhaltet
* Genau gleicher Code und Trainings-Loop
  + Einziger Unterschied ist die Loss-Function und das letzte Dense-Layer
* Training hat mit 19 Epochen länger gedauert und wurde von Callback gestoppt
* Finales Modell hat Weights von Epoche 13 übernommen
* Stärkeres Overfitting ist sichtbar
  + Vielleicht hätte hier im Vergleich zum Basis-Model mehr Regularisierung etwas gebracht. Also mehr Dropouts

A screenshot of a computer screen

Description automatically generated

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 189ms/step - accuracy: 0.7788 - f1: 0.7777 - loss: 0.5431 - precision: 0.5753 - recall: 0.9110

Epoch 10: val\_accuracy did not improve from 0.69737

500/500 ━━━━━━━━━━━━━━━━━━━━ 101s 203ms/step - accuracy: 0.7788 - f1: 0.7777 - loss: 0.5431 - precision: 0.5753 - recall: 0.9110 - val\_accuracy: 0.6379 - val\_f1: 0.6332 - val\_loss: 0.9341 - val\_precision: 0.5031 - val\_recall: 0.8630 - learning\_rate: 0.0010

Epoch 11/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 191ms/step - accuracy: 0.7898 - f1: 0.7890 - loss: 0.5138 - precision: 0.5790 - recall: 0.9176

Epoch 11: val\_accuracy did not improve from 0.69737

Epoch 11: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.

500/500 ━━━━━━━━━━━━━━━━━━━━ 102s 204ms/step - accuracy: 0.7898 - f1: 0.7890 - loss: 0.5138 - precision: 0.5790 - recall: 0.9176 - val\_accuracy: 0.6620 - val\_f1: 0.6560 - val\_loss: 0.9566 - val\_precision: 0.4786 - val\_recall: 0.8662 - learning\_rate: 0.0010

Epoch 12/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 191ms/step - accuracy: 0.8284 - f1: 0.8276 - loss: 0.4309 - precision: 0.5908 - recall: 0.9387

Epoch 12: val\_accuracy improved from 0.69737 to 0.72075, saving model to fish\_lizard\_monkey\_snake\_residual\_classifier.keras

500/500 ━━━━━━━━━━━━━━━━━━━━ 102s 205ms/step - accuracy: 0.8284 - f1: 0.8276 - loss: 0.4309 - precision: 0.5908 - recall: 0.9387 - val\_accuracy: 0.7207 - val\_f1: 0.7222 - val\_loss: 0.7843 - val\_precision: 0.5151 - val\_recall: 0.9070 - learning\_rate: 5.0000e-04

Epoch 13/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 191ms/step - accuracy: 0.8527 - f1: 0.8522 - loss: 0.3727 - precision: 0.5922 - recall: 0.9552

Epoch 13: val\_accuracy improved from 0.72075 to 0.75400, saving model to fish\_lizard\_monkey\_snake\_residual\_classifier.keras

500/500 ━━━━━━━━━━━━━━━━━━━━ 103s 205ms/step - accuracy: 0.8527 - f1: 0.8522 - loss: 0.3726 - precision: 0.5922 - recall: 0.9552 - val\_accuracy: 0.7540 - val\_f1: 0.7524 - val\_loss: 0.7049 - val\_precision: 0.5476 - val\_recall: 0.9043 - learning\_rate: 5.0000e-04

Epoch 14/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 193ms/step - accuracy: 0.8676 - f1: 0.8672 - loss: 0.3370 - precision: 0.5970 - recall: 0.9636

Epoch 14: val\_accuracy did not improve from 0.75400

500/500 ━━━━━━━━━━━━━━━━━━━━ 103s 207ms/step - accuracy: 0.8676 - f1: 0.8673 - loss: 0.3370 - precision: 0.5970 - recall: 0.9636 - val\_accuracy: 0.7368 - val\_f1: 0.7340 - val\_loss: 0.7862 - val\_precision: 0.5529 - val\_recall: 0.8995 - learning\_rate: 5.0000e-04

Epoch 15/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 192ms/step - accuracy: 0.8800 - f1: 0.8798 - loss: 0.3075 - precision: 0.6036 - recall: 0.9704

Epoch 15: val\_accuracy did not improve from 0.75400

500/500 ━━━━━━━━━━━━━━━━━━━━ 103s 206ms/step - accuracy: 0.8800 - f1: 0.8798 - loss: 0.3075 - precision: 0.6035 - recall: 0.9704 - val\_accuracy: 0.7434 - val\_f1: 0.7419 - val\_loss: 0.8071 - val\_precision: 0.5474 - val\_recall: 0.9028 - learning\_rate: 5.0000e-04

Epoch 16/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 192ms/step - accuracy: 0.8936 - f1: 0.8935 - loss: 0.2757 - precision: 0.6050 - recall: 0.9731

Epoch 16: val\_accuracy did not improve from 0.75400

500/500 ━━━━━━━━━━━━━━━━━━━━ 103s 205ms/step - accuracy: 0.8936 - f1: 0.8935 - loss: 0.2757 - precision: 0.6051 - recall: 0.9731 - val\_accuracy: 0.7448 - val\_f1: 0.7343 - val\_loss: 0.8836 - val\_precision: 0.5512 - val\_recall: 0.8901 - learning\_rate: 5.0000e-04

Epoch 17/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 193ms/step - accuracy: 0.9002 - f1: 0.9001 - loss: 0.2590 - precision: 0.6045 - recall: 0.9770

Epoch 17: val\_accuracy did not improve from 0.75400

Epoch 17: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.

500/500 ━━━━━━━━━━━━━━━━━━━━ 104s 208ms/step - accuracy: 0.9002 - f1: 0.9001 - loss: 0.2590 - precision: 0.6045 - recall: 0.9770 - val\_accuracy: 0.7281 - val\_f1: 0.7250 - val\_loss: 0.9392 - val\_precision: 0.5234 - val\_recall: 0.9004 - learning\_rate: 5.0000e-04

Epoch 18/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 194ms/step - accuracy: 0.9250 - f1: 0.9249 - loss: 0.2040 - precision: 0.6200 - recall: 0.9840

Epoch 18: val\_accuracy did not improve from 0.75400

500/500 ━━━━━━━━━━━━━━━━━━━━ 104s 208ms/step - accuracy: 0.9251 - f1: 0.9249 - loss: 0.2039 - precision: 0.6200 - recall: 0.9840 - val\_accuracy: 0.7400 - val\_f1: 0.7419 - val\_loss: 0.9462 - val\_precision: 0.5299 - val\_recall: 0.9086 - learning\_rate: 2.5000e-04

Epoch 19/50

500/500 ━━━━━━━━━━━━━━━━━━━━ 0s 194ms/step - accuracy: 0.9424 - f1: 0.9424 - loss: 0.1601 - precision: 0.6306 - recall: 0.9887

Epoch 19: val\_accuracy did not improve from 0.75400

500/500 ━━━━━━━━━━━━━━━━━━━━ 104s 208ms/step - accuracy: 0.9424 - f1: 0.9424 - loss: 0.1600 - precision: 0.6306 - recall: 0.9887 - val\_accuracy: 0.7495 - val\_f1: 0.7501 - val\_loss: 0.9589 - val\_precision: 0.5394 - val\_recall: 0.9144 - learning\_rate: 2.5000e-04

Epoch 19: early stopping

Restoring model weights from the end of the best epoch: 13.