HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Neural Architecture Search zur Bildklassifikation

Bachelorarbeit

zur Erlangung des Akademischen Grades
Bachelor of Science (B.Sc.)

eingereicht von:	Darko Nedić
geboren am:	13. Mai 1996
geboren in:	Berlin, Deutschland
Gutachter/innen:	Prof. Dr. Verena V. Hafner Prof. Dr.-Ing. Peter Eisert

eingereicht am: .... verteidigt am: ....

Zusammenfassung

Neural Architecture Search (NAS) befasst sich mit Verfahren zur maschinellen Erstellung von Architekturen für neuronale Netze. Dazu wurden in den letzten Jahren einige vielversprechende Ansätze vorgestellt. Zentrale Fragen von NAS sind sind die Arbeitsweise des Suchverfahrens und wie groß der Suchraum ist. NAS bringt einen großen Vorteil mit sich, denn damit ist man in der Lage zu gegebenen Daten automatisiert ein gutes Netz zu finden, ohne großen Aufwand zu betreiben oder Experte auf dem Gebiet zu sein.

In dieser Bachelorarbeit wird zu Beginn die Arbeitsweise und die Entwicklung von *Convolutional Neural Networks* (CNNs) vorgestellt. Anschließend werden drei NAS Verfahren für CNNs betrachtet und dessen Qualität wird an einem für diese Arbeit erstellten Datensatz demonstriert.

Abstract

Neural Architecture Search (NAS) deals with the algorithmic creation of neural network architectures. For that some promising results have been presented in recent years. The key question is how does the search strategy work and how big is the search space. One big advantage NAS brings with it is that one must not be an expert in the area or invest big effort to algorithmically design a good architecture to a given dataset.

At the beginning this bachelor thesis presents the way of working and the evolution of *Convolutional Neural Networks* (CNNs). Afterwards three selected methods will be look at and their quality will be demonstrated on a dataset created for this thesis.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Bildklassifizierung weist einem gegebenen Bild ein Label zu. Das Ergebnis ist eine von einer oder mehreren definierten Klassen. Angenommen man hat ein Datensatz zu Tieren mit Hunden und Katzen als Labels. Jetzt möchte man zu einem Bild sagen können, welches der beiden Tieren darauf abgebildet ist. Für einen Menschen ist eine solche Aufgaben ohne viel Aufwand zu bewältigen. Will man aber ein Programm schreiben, welches diese Entscheidung treffen kann, steht man vor einer schwierigen Aufgabe. Der Bereich des Computer Vision setzt sich unter anderem mit genau solchen Aufgaben auseinander.

Als Benchmark für Bildklassifikationsalgorithmen wurde durch die Stanford University 2009 das Projekt *ImageNet* gestartet. Zugleich wurde damit die *ILSVRC* (ImageNet Large Scale Visual Recognition Challenge) ins Leben gerufen, an welcher Forscher ihre Klassifizierungsverfahren testen können. Dabei handelt es sich um ein seit 2010 jährlich stattfindenden Wettbewerb. Ein bedeutendes Ereignis fand im Jahr 2012 statt, als die Fehlerrate von 26% auf 15% fiel. [1, 2] Ermöglicht wurde diese enorme Verbesserung durch erstmalige Teilnahme eines Convolutional Neural Networks (kurz: *CNN*). Die dazu veröffentlichte Arbeit legte seit dem die Grundlage für die Benutzung von CNNs in allen Bereichen der Computer Vision.
Wie Arbeiten der letzten Jahre zeigten, ist für die Qualität eines CNNs dessen Architektur von hoher Bedeutung. Traditionellerweise wird die Architektur von einem Entwickler vorgegeben, bei der sich dieser beispielsweise um folgende Punkte Gedanken machen muss. Er muss sich kümmern um die Inputgröße, die Anzahl und Art der Layer und dessen Hyperparameter und die verwendete Fehlerfunktion. Es ist nicht klar wie eine optimale Architektur zu einem Datensatz auszusehen hat. Meist findet man eine passende Architektur durch Ausprobieren oder man greift zu bereits guten Architekturen, die für ähnliche Probleme entwickelt wurden. NAS versucht diesen Prozess des Architekturenfindens zu automatisieren.

Ziel dieser Bachelorarbeit ist es zu Untersuchen, wie geeignet die Methoden aktueller Forschungen sind, um die Aufgabe des automatischen Erstellens einer guten Architektur zu bewältigen. Dabei werden die NAS Verfahren analysiert und auf einem eigenen Datensatz durchgeführt.

2 Einführung und historische Entwicklung von CNNs

Convolutional Neural Networks werden zur Klassifikation von Bildern verwendet. Ein solches Netz besteht aus mehreren Schichten, sog. Layer, die nacheinander die Eingabe durch Transformationsoperationen verarbeiten und zum Schluss eine Wahrscheinlichkeit der Klassenzugehörigkeiten ausgeben. Durch einen Trainingsvorgang des Netzes wird dafür gesorgt, dass die Ausgabe dieser Aneinanderreihnung von Operationen qualitativ ausfällt.

Im Folgenden werden die meistgenutzten Layer, sowie der Trainingsvorgang vorgestellt.

Convolutional Layer

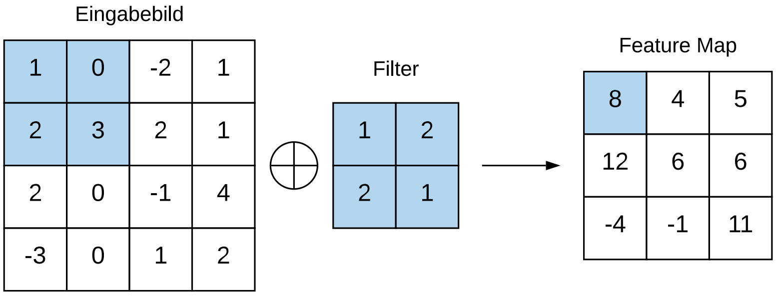
Die Convolution-Operation ist das eigentliche Herzstück, welches CNNs so besonders macht. Als Eingabe dient eine Matrix, auf die ein oder mehrere Filter angewendet werden. Als Ausgabe resultiert je Filter eine Matrix, die sog. Feature Map. Die Filter sind i.d.R. kleine quadratische Matrizen, die bereichsweise eine Faltungsoperation auf der Eingabe durchführen. Sei $\oplus$ der Faltungsoperator. Dann wird die einfache Convolutional-Operation durch folgende Formel beschrieben:

$$FeatureMap = Inputmatrix \oplus Filter + Bias \tag{1}$$

mit

$$FeatureMap[a][b] = \left(\sum_{y=0}^{Filter_{Spalten}} \left(\sum_{x=0}^{Filter_{Reihen}} Inputmatrix[x+a][y+b] \cdot Filter[x][y] \right) \right) + Bias \tag{2}$$

Eine Vorstellung dieser Berechnung ist in nachfolgender Abbildung zu sehen.



Eingabebild

1	0	-2	1
2	3	2	1
2	0	-1	4
-3	0	1	2

Filter

1	2
2	1

Feature Map

8	4	5
12	6	6
-4	-1	11

Abb. 1: Convolution Operation. Hierbei ist der Filter 2x2 Felder groß und hat einen Stride von 1. In blau ist beim Eingabebild die Überlagerung des Filters markiert und in der Feature Map die Ausgabe.

Diese Operation arbeitet so, dass die Summe der Produkte der jeweiligen überlagernden Felder des Filters und der Eingabematrix als Ausgabe in das entsprechende Feld der Feature Map gespeichert wird. Die Verschiebung des Filters, sog. Stride, hat hier den Wert 1, denn diese erfolgt um einen Pixelschritt. Je Verschiebung wird ein weiterer Wert für die Feature Map berechnet. Man stellt fest, wie die Feature Map eine geringere Auflösung hat als die Eingabe. Dies ist durch die Arbeitsweise der durchgeführten Operation gegeben. Die Dimension der Ausgabe eines Filters berechnet sich folgendermaßen. Sei E die Breite/Höhe der Eingabe, F entsprechend auch die Breite/Höhe des Filters, P das Padding und S der Stride, so gilt für die entsprechende Breite/Höhe der Ausgabe A:

$$A = \frac{E - F + 2P}{S} + 1 \tag{3}$$

Wie bei Randfällen umgegangen wird, wie wenn bei einem Stride von 3, der Filter nur an einer Pixelreihe über der Eingabe liegt, muss separat behandelt werden. Beispielsweise kann man die Eingabematrix an einem oder mehreren Rändern um eine Spalte/Zeile erweitern und mit Nullen auffüllen.
Während des Traingsvorgangs werden die Filterwerte gelernt. Bei der Parametriesierung ist festzulegen, wie groß ein Filter ist, und wieviele im Layer enthalten sind. Desweiteren mit welchem Stride der Filter bewegt werden soll und wie mit Randfällen umgegangen wird.

Activation Function
Die Funktion nimmt als Input eine Feature Map und gibt eine sog. Activation Map gleicher Dimension aus. Eine Aktivierungsfunktion ist dazu da, hohe Werte zu verstärken und schwache Werte gegen Null zu drücken. Diese Werte werden in dem Sinne aktiviert bzw. deaktiviert. Dadurch wird bestimmt, wie wichtig ein Wert ist, bzw. wie stark sein Einfluss auf die nachfolgenden Layer ist. Da Bilder i.d.R. nicht linearen Mustern folgen, so soll die Aktivierungsfuntkion auch nicht linear sein. Zu den bekannten Funktionen zählen die Sigmoidfunktion $f(x) = (1 + e^{-x})^{-1}$ oder die beliebte ReLU Funktion $\max(0, x)$, bei welcher alle negativen Werte auf Null gesetzt werden. Diese wird oft gewählt, weil die maschinelle Berechnung schnell abläuft.

Pooling Layer
Der pooling Layer führt eine Operation durch, bei der die Dimension von Feature Maps reduziert wird. Dabei wird beispielsweise ein 2x2 Feld über die Eingabematrix geschoben, sodass 4 Werte der Feature Map auf einen Wert abgebildet werden. Als Ausgabe resultiert eine kleinere Matrix. Es gibt verschiedene Variationen des Poolings. Beliebt ist das Average Pooling, bei dem auf den Mittelwert aller überlagernden Felder abgebildet wird. Zudem findet man auch ein Max Pooling, bei dem auf den größten Wert abgebildet wird. So kann man die Operation auf verschiedenste Weisen durchführen.
Die Hyperparameter, die hier zu setzen sind, sind der Stride, sowie die Größe des Filters. Außerdem muss entschieden werden, auf welche Weise die Pooling-Operation durchgeführt wird.

Fully-Connected Layer
Ein fully-connected Layer besteht aus einem Vektor, bei dem alle Felder jeweils mit allen Feldern der Ausgabe des vorigen Layers verbunden sind. Für diesen Vektor werden die Eingabematrizen ebenfalls als ein Vektor dargestellt. Beim Trainingsvorgang werden die Gewichte aller Verbindungen trainiert. Die Anzahl der Verbindungen hängt unter anderem von der Felderanzahl des Vektors vom fully-connected Layer ab, die durch einen Hyperparameter gesetzt werden. Die Anordnung dieses Layers in der Architektur erfolgt i.d.R. zum Schluss. Auf den letzten fully-connected Layer folgt dann nochmal ein weiterer als Output Layer. Dessen Größe entspricht der Anzahl der zu erkennenden Klassen. Die Ausgabewerte werden üblicherweise noch normalisiert, dass die Summe aller Ausgabefelder 1 ergibt und man dadurch eine Wahrscheinlichkeitsverteilung erhält.

Trainingsvorgang

Das Ziel beim Trainingsvorgang eines CNNs ist es, dass der Fehler in der Ausgabe minimiert wird. Angenommen man klassifiziert in drei mögliche Klassen. Dann wird die Ausgabe eines Netzes ein dreidimensionaler Vektor sein. Erwartet man, dass das Bild zur ersten Klasse gehört, erwartet man als Ausgabe den Vektor (1,0,0). Erhält man jedoch einen Vektor (0.2,0.5,0.3), so erhält man eine unsichere Ausgabe, und nicht die, die erwartet wird. Folglich ist das Netz schlecht trainiert. Für die Ausgabe und das erwartete Ergebnis lässt sich der Unterschied und damit die Qualität der Ausgabe mit Hilfe einer *Fehlerfunktion* berechnen. In der Regel wählt man dazu das Quadrat des Euklidischen Abstandes $\sum_{i=1}^{n}(x_{true\_i} - x_{out\_i})^2$. Damit das Netz eine verlässlichere Ausgabe machen kann, müssen die anfangs zufällig gewählten Gewichte, mit denen bei der Berechnung gearbeitet wird, in die richtige Richtung so justiert werden, dass die Fehlerfunktion minimiert wird. *Gradient Descent* ist ein Verfahren, welches zum Finden eines (lokalen) Minimums verwendet wird. Nachfolgende Abbildung soll die Idee des Verfahrens verdeutlichen. Dabei werden die Gewichte schrittweise stets so angepasst, dass man dem Minimum näher kommt.

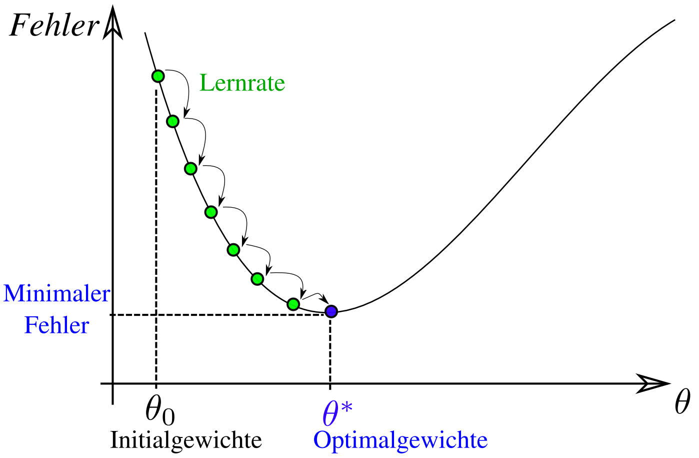


Abb. 2: Gradient Descent. Hierbei sind die grünen Punkte die schrittweisen Ergebnisse der Fehlerfunktion, nachdem die Gewichte justiert wurden. Der blaue Punkt signalisiert, dass es sich bei den dazu gewählten Gewichten um die lokal geeignetsten handelt.

Nun besteht die Herausforderung darin, die Gewichte so justieren zu können, dass die Fehlerfunktion minimiert wird. Dazu wurde in David E. Rumelhart et al. [3] der Backpropagation-Algorithmus vorgestellt. Dabei wurde gezeigt, dass man ihn zum Trainieren von neuronalen Netzen nutzen kann.
Dabei wird dem Netz ein Eingabebild gegeben. Dieses durchläuft den sog. *Forward Pass.* Zu gegebenen Gewichten wird hierbei die Ausgabe des Netzes und somit die Wahrscheinlichkeitsverteilung für die Klassen berechnet. Abbildung 3 soll ein kleines Netz sein, an dem die Berechnung beispielhaft, angepasst nach [4], durchgeführt wird.

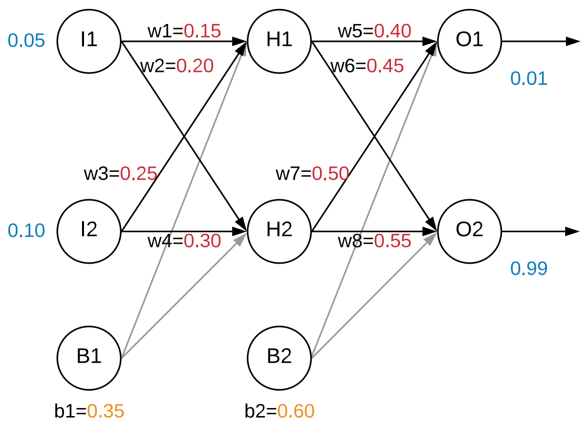


Abb. 3: Beispielnetz zur Erklärung des Trainingsvorgangs. In rot sind die zu trainierenden Gewichte. Links in blau sind die Eingabedaten und rechts in blau sind die tatsächlich erwarteten Ausgaben. In orange ist der zur Berechnung benötigte Bias. Die Knoten werden Neuronen genannt. H_i sind Hidden States, welche jeweils eine Operation durchführen und dessen Werte in eine Aktivierungsfunktion eingesetzt werden. Das Ergebnis ist die Ausgabe eines Hidden States.

Initial hat jedes Gewicht w_i einen zufälligen Wert $\in (0, 1)$. Zur Berechnung der Ausgabe des Netzes wird zunächst die Ausgabe der Hidden States berechnet. Sei net_i der ausgerechnete Eingabewert der in den Knoten i gelangt. Dazu wird die Eingabe für $H1$ wie folgt ermittelt:

$$\begin{aligned} net_{H_1} &= I1 * w_1 + I2 * w_3 + 1 * b1 \\ &= 0.05 * 0.15 + 0.10 * 0.25 + 1 * 0.35 \\ &= 0.3825 \end{aligned} \tag{4}$$

Die Berechnung für $H2$ erfolgt analog. Die Werte werden dann in die Aktivierungsfunktion eingesetzt. In diesem Beispiel rechnet man dafür mit der ReLU. Auf die gleiche Weise wie mit den Eingaben für $H1$, wird jetzt der Ausgabewert von $O1$ und $O2$ berechnet, jedoch mit den Werten aus $out_{H_i} := \max(0, net_{H_i})$ als Eingabe. Als Ergebnis erhält man $out_{O1} = 0.948$ und $out_{O2} = 0.986625$. Nun lassen sich die Werte als Vektoren $x_{true} := (0.1, 0.99)$, $x_{out} := (out_{O1}, out_{O2})$ in die Fehlerfunktion einsetzen und somit der gesamte Fehler errechnen:

$$\begin{aligned} E_{ges}(x_{true}, x_{out}) &= \frac{1}{2}\sum_{i=1}^{n}(x_{true\_i} - x_{out\_i})^2 \\ &\approx \frac{1}{2}((0.01 - 0.948)^2 + (0.99 - 0.986625)^2) \\ &\approx 0.4399276 \end{aligned} \tag{5}$$

Dies war der Forward Pass. Hier wurde bewusst die Summe mit $\frac{1}{2}$ multipliziert, der Grund ist nachfolgend zu sehen. Nun folgt der Backward Pass, bei dem die Gewichte angepasst werden. Dabei will man den Fehler für jedes Neuron und somit für das gesamte Netz minimieren. Um nun zu wissen, wie viel Einfluss eine Änderung von einem w_i auf den Fehler hat, muss man die Fehlerfunktion partiell ableiten nach w_i. Sei weiterhin out_i der ausgehende Wert von Knoten i, dann gilt für Beispiel w_5:

$$\frac{\partial E_{ges}}{\partial w_5} = \frac{\partial E_{ges}}{\partial out_{o_1}} \cdot \frac{\partial out_{o_1}}{\partial net_{o_1}} \cdot \frac{\partial net_{o_1}}{\partial w_5} \tag{6}$$

Dies ist der Gradient von w_5. Nun gilt es alle Faktoren zu berechnen. Man betrachte zunächst E_{ges}:

$$E_{ges} = \frac{1}{2}(x_{true\_o1} - x_{out\_o1})^2 + \frac{1}{2}(x_{true\_o2} - x_{out\_o2})^2 \tag{7}$$

Dann gilt:

$$\begin{aligned} \frac{\partial E_{ges}}{\partial out_{o_1}} &= 2 \cdot \frac{1}{2}(x_{true\_o1} - x_{out\_o1})^{2-1} \cdot (-1) + 0 \\ &= -(x_{true\_o1} - x_{out\_o1}) \\ &= -(0.01 - 0.948) = 0.938 \end{aligned} \tag{8}$$

Hier wird auch klar, warum die Fehlerfunktion mit $\frac{1}{2}$ multipliziert wurde. Nun muss der nächste Faktor berechnet werden. Dazu leitet man die Aktivierungsfunktion ab, und setzt out_{o1} ein. Für einen Wert ≤ 0 ergibt die Ableitung 0, sonst 1. Da für $out_{o1} > 0$ gilt, gilt weiterhin:

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = 1 \tag{9}$$

Jetzt noch zum letzten Faktor. Dazu betrachte zunächst net_{o1}:

$$net_{o1} = out_{h1} \cdot w_5 + out_{h2} \cdot 1 \cdot w_7 + b_2 \tag{10}$$

Daraus folgt:

$$\frac{\partial net_{o_1}}{\partial w_5} = 1 \cdot out_{h1} \cdot w_5^{1-1} + 0 + 0 = out_{h1} \approx 0.3825 \tag{11}$$

Fügt man nun die Zwischenergebnisse in Gleichung (6) ein, so erhält man:

$$\frac{\partial E_{ges}}{\partial w_5} \approx 0.938 \cdot 1 \cdot 0.3825 \approx 0.35878 \tag{12}$$

Um nun den Fehler zu minimieren, wird vom ursprünglichen Wert w_5 der Gradient abgezogen. Der Gradient wird oft nur zu einem Prozentteil genommen, damit beim Gradient Descent Verfahren nicht allzu große Schritte gemacht werden und das Minimum nicht verfehlt wird. Der Prozentteil wird als Gewicht angegeben und ist als Lernrate bekannt. So ergibt sich der neue Wert für w_5 mit einer Lernrate von 0.5:

$$w_{5neu} = w_5 - 0.5 \cdot 0.35878 = 0.22061 \tag{13}$$

Die Berechnung der Gewichte w_6, w_7, w_8 erfolgt analog. Die Berechnung der Gewichte für die Hidden States erfolgt ähnlich. Dazu wird der Gradient wie folgt am Beispiel für w_1 berechnet:

$$\frac{\partial E_{ges}}{\partial w_1} = \frac{\partial E_{ges}}{\partial out_{h_1}} \cdot \frac{\partial out_{h_1}}{\partial net_{h_1}} \cdot \frac{\partial net_{h_1}}{\partial w_1} \tag{14}$$

mit

$$\frac{\partial E_{ges}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial out_{h_1}} \tag{15}$$

Dies wird für alle Gewichte durchgeführt. Nach 15 Iterationen mit derselben Eingabe beträgt der Fehler $E_{ges} = 0.005$. Will man mit mehreren verschiedenen Bildern trainieren, so ist es üblich, dass der Trainingsdatensatz in kleine *Batches* aufgeteilt wird. Jeder Batch durchläuft einmal den Forward Pass. Dabei wird der durchschnittliche Fehler je Eingabe gemessen. Ist der Batch durch, so wird der Backward Pass durchgeführt. Danach folgt der nächste Batch. Sind alle Batches durch, ist eine *Epoche* durch. Eine Epoche ist ein Durchlauf aller Daten. Dem Trainingsvorgang wird neben der Batchgröße auch ein epoch-Wert gegeben, der aussagt, wie oft alle Daten durchgeschoben werden müssen.

Nun folgt ein Überblick zu der historischen Entwicklung von CNNs. Es wird chronologisch auf einzelne Netze eingegangen, die zur Entwicklung beigetragen haben.

2.1 Cognitron

Bereits in den 1970ern war die Forschung daran interessiert, das menschliche Gehirn in ein mathematisches Modell zu überführen. In Fukushima [5] wird ein Modell namens *Cognitron* vorgestellt und damit der erste Meilenstein in Richtung *CNNs* gesetzt. Das *Cognitron*, welches dem visuellen System von Säugetieren nachempfunden ist, ist durch mehrere Layer aufgebaut, bestehend aus sog. inhibitorischen und exzitatorischen Neuronen. Die exzitatorischen Neuronen bilden in einem Layer eine oder mehrere Nachbarschaften definierter Größe. Jede Nachbarschaft hat genau ein inhibitorisches Neuron. Eine Nachbarschaft bildet die sog. *connection competition region*, in welcher eine Menge benachbarter Neuronen (*connection region*) im vorhergehenden Layer, ein Neuron im Folgelayer bestimmen, das aktiviert werden soll (durch *competition*). Das hat das Ziel in jedem Layer eine „elitäre“ Menge an Neuronen zu bestimmen, die die Eingabe widerspiegelt. Diese Menge ist im letzten Layer ausschlaggebend für die Klassifizierung. Layerweise wird die Größe der „elitären“ Menge verringert.
Zur Verdeutlichung und weiteren Erklärung wird nun auf Abbildung 4 Bezug genommen.

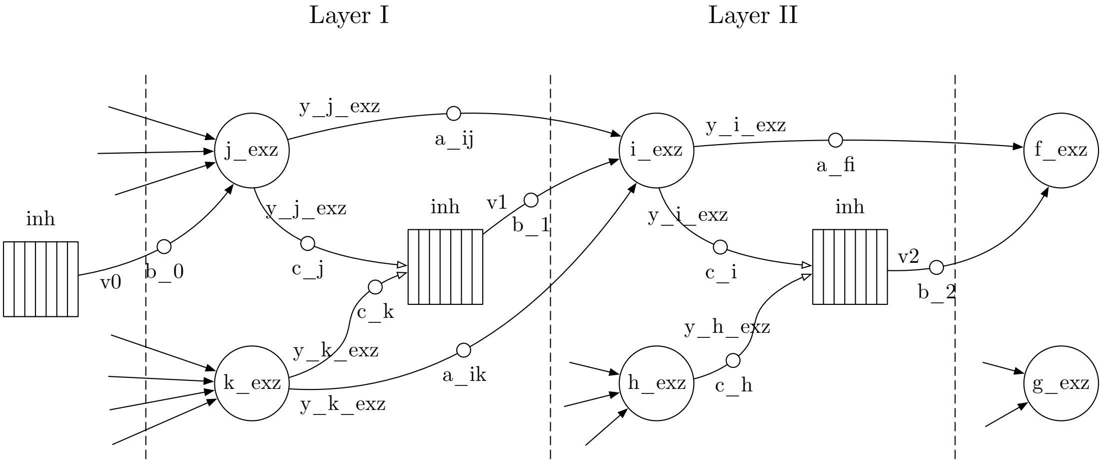


Abb. 4: Schematische Darstellung des Cognitron Netzes. So bilden z.B. die Neuronen j_{exz}, k_{exz} und i_{exz} eine Nachbarschaft, mit j_{exz} und k_{exz} als *connection region*.

Zur Berechnung des Outputs y_{i_exz} von Neuron i_{exz} werden zunächst die zwei Variablen x_i und z_i benötigt. Diese werden wie folgt berechnet:

$$x_i = \sum_j a_{ij} y_j \tag{16}$$

$$z_i = \sum_j b_{ij} v_j \tag{17}$$

mit y_j für den Output des vorhergehenden exzitatorischen Neuron und v_j für den Output des vorhergehenden inhibitorischen Neuron. a_{ij} und b_{ij} sind Gewichte, welche beim Trainieren angepasst werden. Für den Output y_{i_exz} gilt dann weiter folgende Rechnung:

$$y_{i\_exz} = f(N_i) \tag{18}$$

wobei gilt:

$$N_i = \frac{1 + x_i}{1 + z_i} - 1 = \frac{x_j - z_i}{1 + z_i} \tag{19}$$

und

$$f(N_i) = \begin{cases} N_i & , \quad \text{falls } N_i \geq 0 \\ 0 & , \quad \text{sonst} \end{cases} \tag{20}$$

Der Output v_1 von dem inhibitorischen Neuron ist gegeben durch:

$$v_1 = \sum_j c_j y_j \tag{21}$$

wobei gilt:

$$\sum_j c_j = 1 \tag{22}$$

mit y_i für den Output eines exzitatorischen Neuron und c_i ein vorderfiniertes Gewicht, welches sich beim Trainieren nicht verändert.

Wie zuvor erwähnt, bleibt jetzt noch zu klären wie die Gewichte in (16) und (17) bestimmt werden. Initial sind alle Gewichte in den Layern gleich 0 und alle Neuronen sind nicht aktiv. Für den Fall, dass keine Neuronen aktiv sind, werden die Gewichte a_{0j} und b_j für jedes exzitatorische Neuron j wie folgt bestimmt:

$$\partial a_{0j} = q' c_0 y_0 \tag{23}$$

$$\partial b_j = q' v_j \tag{24}$$

mit q' als Lernrate. Zusätzlich dient der Eingabevektor initial als y-Vektor für die y-Werte für den Folgelayer. Mit aktiven Neuronen in einer Nachbarschaft wird durch diese im aktuellen Layer bestimmt, ob das nächste exzitatorische Neuron im Folgelayer aktiv wird. Das geschieht, wenn gilt, dass sein Output größer oder gleich ist, als das all seiner vorhergehenden exzitatorischen Neuronen aus der Nachbarschaft.

Für diese berechnet man in dem Fall das neue a_{ji} und b_k wie folgt, wobei $q' < q$ gilt, mit q als Lernrate:

$$\partial a_{ji} = q c_j y_j \tag{25}$$

$$\partial b_i = \frac{q \sum\limits_{j} a_{ji} y_j}{2 v_j} \tag{26}$$

Ist dies nicht der Fall, wird das Neuron nicht aktiv und die Gewichte der Nachbarschaft werden nicht modifiziert. Dies wird darum auf diese Weise durchgeführt, damit für eine bessere Laufzeit nicht alle Neuronen trainiert werden müssen und damit die zuvor erwähnte „elitäre“ Menge an Neuronen entsteht. Sollte zwischendurch eine Nachbarschaft keine aktiven Neuronen enthalten, so wird wie in Gleichung (23) und (24) vorgegangen. Dieser Vorgang wird für alle Elemente aus dem Trainingsdatensatz durchgeführt und dabei werden die Gewichte nach und nach angepasst [6]. Eine abstrakte Darstellung für ein solches Netz ist in Abbildung 5 zu sehen.
Cognitron wurde auf handgeschriebenen Zahlen trainiert, als auch auf einer Reihe von Buchstaben. Die Ausgabe des Netzes erfolgt mithilfe eines Patterns (Abbildung 6), welches nach dem Trainieren an entsprechenden Feldern mit Labels gekennzeichnet ist. Das Pattern entsteht durch die letzten aktiven Neuronen - die „elitäre“ Menge. Bei der Benutzung des Patterns wird die Ausgabe des letzten Layers mit den aktiven Neuronen mit dem Pattern felderweise verglichen. Sobald eine ausreichende und überwiegende Anzahl eines Labels hervorsticht, wird die Klasse ausgegeben.

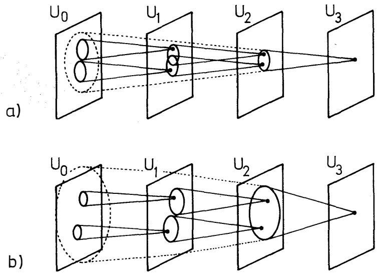
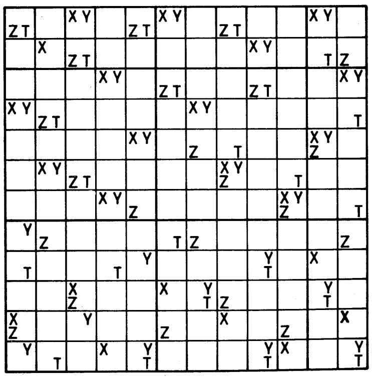


Abb. 5: Abstrakte Darstellung des Cognitron. a) zeigt eine Überlagerung zweier Nachbarschaften (geteilte exzitatorischen Neuronen). Es ist auch zu erkennen, wie eine Nachbarschaft auf ein Neuron abgebildet wird.



Z T		X Y		Z T	X Y		Z T			X Y	
	X	Z T						X Y		T	Z
			X Y		Z T			Z T			X Y
X Y	Z T					X Y					T
				X Y		Z	T			X Y Z	
	X Y	Z T					X Y Z		T		
			X Y	Z					X Y Z		T
Y	Z				T	Z					Z
T			T	Y				Y T		X	
		X Z			X	Y T	Z			Y T	
X Z		Y			Z		X		Z		X
Y	T		X	Y T				Y T	X		Y T

Abb. 6: Ein gelabeltes Pattern, mit welchem die Ausgabe des Netzes Feldweise verglichen wird.

2.2 Neocognitron

Es zeigte sich, dass das *Cognitron* sehr empfindlich bezüglich Skalierung, Rotation oder anderen Deformationen des Inputs war, da es sehr stark von den Trainigsmustern ähängt und schon kleine Abweichungen Schwierigkeiten machten. Infolgedessen entwickelte Kunihiko Fukushima in den Folgejahren das Cognitron weiter und schaffte so das *Neocognitron* [6, 7, 8]. Dabei wurden zwei Ansätze entwickelt, je eins für das überwachte und für das unüberwachte Lernen. Die ausschlaggebende Verbesserung gegenüber dem ursprünglichen Cognitron ist, dass das Neocognitron nun die Fähigkeit besitzt, nicht nur gelernte Muster wiederzuerkennen, sondern aus diesen weitere Muster zu erzeugen. Hierdurch soll dem Problem des Cognitrons vorgebeugt werden. Um nur die Funktionsweise zu erklären, reicht es lediglich auf die Variante mit unüberwachtem Lernen einzugehen.

Das Neocognitron funktioniert durch eine hierarchische Feature-Extraktion. Ein Feature ist hierbei ein charakteristisches Merkmal eines gegebenen Patterns. Wie in Abbildung 7 zu sehen ist, ist die Hierarchie dadurch gegeben, dass Stufenweise die Merkmale immer komplexere Muster bilden. So sind in der ersten Stufe nur einfache Striche zu sehen, während in der letzten Stufe das erkannte Muster — ein Kreis — zu sehen ist. Zu beachten ist, dass die Muster der aktuellen Stufe nur aus den Mustern der vorigen Stufe bestehen. Zwischen den Stufen sind in Abbildung 7 auch Kanten zu sehen. Diese zeigen auf, aus welchen Mustern einer vorherigen Stufe die in der darauf folgenden Stufe zusammengesetzt sind.
Bei weiteren, verschiedenen Eingabepattern werden nach und nach immer mehr Features extrahiert, mit denen dann bei der Klassifizierung gearbeitet wird.

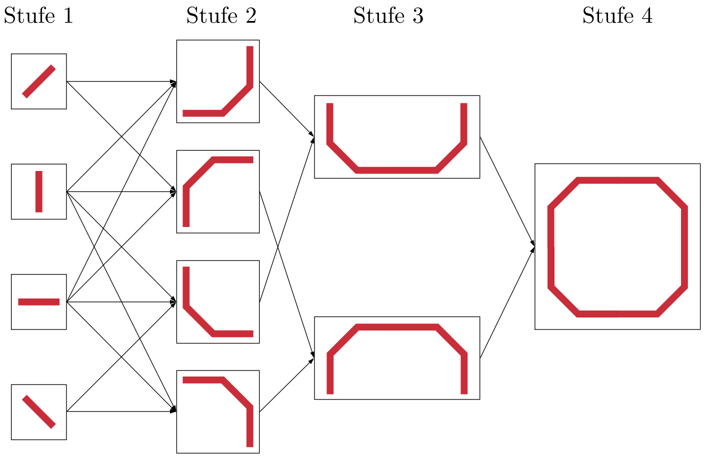


Abb. 7: Hierarchische Darstellung der Features eines Kreises. Stufenweise wächst die Komplexität der Features. Nach der letzten Stufe wird deutlich, dass ein Pattern in viele kleine, simple Features zerlegt werden kann.

Die gesamte Anzahl der Stufen hängt ab von der Komplexität der erkannten Muster. Je komplexer die Muster werden können, umso mehr Stufen enthält die Hierarchie. Weiterhin setzt sich jede einzelne Stufe aus drei Teilen zusammen. Diese werden Layer genannt. Es gibt den S-Layer, den V-Layer und den C-Layer. Jeder dieser Layer besteht nochmal aus sog. Zellflächen, die wiederum aus Zellen bestehen, welche die eigentliche Berechnung des Neocognitrons durchführen. Der S-Layer sorgt für die Feature-

Extraktion (z.B. Striche und deren Orientierung) und wird von dem V-Layer dabei unterstützt. Der V-Layer hat nämlich die Aufgabe, irrelevante Features zu erkennen und hat eine hemmende Wirkung auf die Aktivierung von Zellflächen im S-Layer, wenn irrelevante Features vorhanden sind. Nachdem ein Feature durch den S-Layer extrahiert ist, soll der C-Layer nochmal für eine Toleranz gegenüber Deformationen sorgen, sodass unterschiedliche (aber ähnliche) S-Zellflächen für einen gleichen Output der C-Zellfläche sorgen, wie in Abbildung 9 verdeutlicht wird. Hierbei reicht für die Aktivierung aus, wenn nur eine S-Zelle in der Nachbarschaft aktiv ist.

Eine S-Zellfläche entspricht einem Feature, somit ist die Anzahl der Zellflächen in einem S- und C-Layer gleich der Anzahl der extrahierten Features. V-Layer haben stets eine Zellfläche. Diese enthält Informationen über die durchschnittliche aktivität des C-Layers der vorigen Stufe. Die genaue Feature-Extraktion erfolgt mittels eines Patterns. So kann z.B. eine *connection region* einer C-Zellfläche eines vorigen C-Layers mit dem Pattern verrechnet werden, und es wird bestimmt, ob eine Zelle der aktuellen S-Zellfläche aktiv wird oder nicht. Das Pattern legt somit fest, wie das Feature ausschaut, nach welchem gesucht wird. Abbildung 8 veranschaulicht dieses Vorgehen.

Die Größe der Zellflächen (Anzahl der Zellen in einer Zellfläche) ist innerhalb eines Layers gleich, doch wird bei höheren Stufen immer kleiner. Die Klassifizierung erfolgt dann in der letzten Stufe. Der letzte C-Layer sollte dann nur so viele Zellflächen besitzen wie es Klassen gibt und jede Zellfläche hat nur eine Zelle. Je nachdem um welche Klasse es sich bei der Eingabe handelt, ist die Zelle in einer der C-Zellflächen aktiv. Zusätzlich zu allen Stufen gibt es noch die Stufe 0. Diese Stufe 0 ist das Eingabebild, also die Zahl oder der Buchstabe, mit dem gearbeitet wird. So haben der S-Layer und der V-Layer in Stufe 1 das Eingabebild als Eingabe.

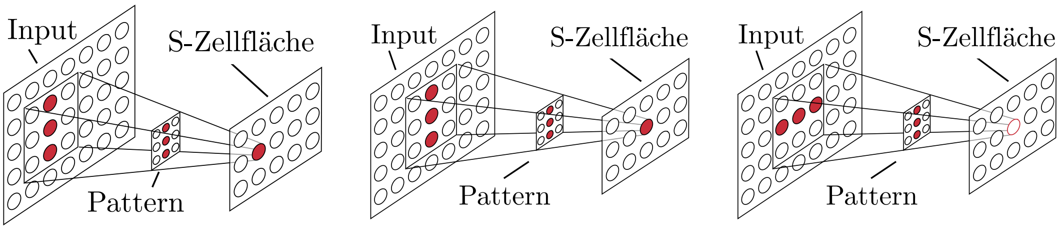


Abb. 8: Das Pattern dient als Filter für den Input. Es legt fest wie das Feature ausschaut, nach welchem gesucht wird. Wird es gefunden, wie in den ersten zwei Fällen, so wird die Zelle in der S-Zellfläche der nächsten Stufe aktiv, sonst wie im letzten Fall eben nicht.

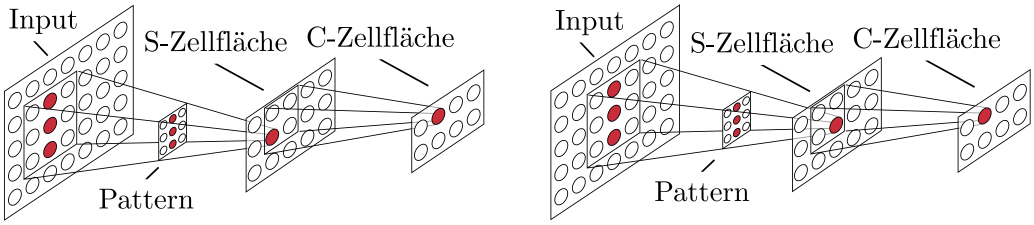


Abb. 9: Hier soll verdeutlicht werden, dass bei verschiedenen, aber ähnlichen Eingaben die C-Zellfläche ein gleiches Resultat liefert.

Das Neocognitron enthält vier verschiedene Arten von Gewichten. Zwei davon werden beim Lernen modifiziert und sind für die Feature-Extraktion von Bedeutung. Für die Arbeitsweise des Neocognitrons wurden über die Zeit mehrere Ansätze entwickelt. Ein Ansatz ist in Fukushima [7, 8] zu finden:

Es werden initial die Gewichte mit sehr geringen und zufällig gewählten Werten gesetzt, sodass fast jede S-Zelle in jeder S-Zellfläche in der ersten Stufe schwach von einem Eingabemuster beeinflusst wird. Einige Zellen werden jedoch etwas stärker beeinflusst, sie haben also einen größeren Output als die anderen. Infolgedessen wird angenommen, dass durch diese Zellen an ihrer Position ein Feature repräsentiert werden könnte. Nun wurde in dem Ansatz beschrieben, dass man zur Auswahl einer repräsentierenden Zelle aus den Kandidaten alle S-Zellflächen im S-Layer übereinander legt. Durch diese Überlagerung geht jetzt eine sog. Säule durch, wie es in Abbildung 10 dargestellt ist. Der S-Zellen Kandidat, der den größten Output innerhalb der Säule hat, bestimmt die S-Zellfläche, die ausschlaggebend ist für das Feature. Gibt es in der Säule nur eine Kandidatenzelle, wird ihre Zellfläche ohne weiteres für das Feature bestimmt. Dies beeinflusst die Gewichte zu entsprechender Eingabe. Gibt es keine Kandidaten, wird auch keine repräsentative Zellfläche gewählt. Dieser Vorgang wird Stufenweise fortgeführt und nach jedem Durchlauf wird ein neues Eingabepattern dem Netz gegeben. Es werden neue Features erkannt und die Gewichte werden weiter angepasst. Eine detailierte mathematische Beschreibung zur Neuberechnung der Gewichte ist ebenfalls in Fukushima [7, 8] angeführt.

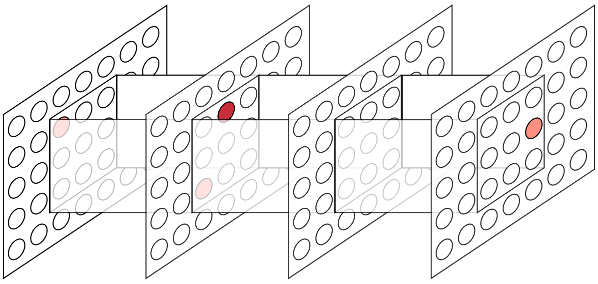


Abb. 10: Veranschaulichung des Verhältnisses zwischen S-Zellflächen und einer Säule innerhalb eines S-Layers. Je dunkler das Rot der Zelle, umso größer ihr Output. In der Abbildung würde die zweite Zellfläche für die Feature-Extraktion zuständig sein.

2.3 LeNet-5

Das Neocognitron wurde jahrelang erweitert und verbessert und zudem entstanden weitere neuronale Netze, die durch das Neocognitron inspiriert wurden. Durch Yann LeCun et al., welcher den Begriff „Convolutional Neural Network“ prägte, entstand 1998 auch das *LeNet-5* [9], bei dem man in seiner Architektur Ähnlichkeiten zum Neocognitron wiederfindet. Das LeNet-5 besteht aus Layern mit verschiedenen Funktionen. Der convolutional Layer, welcher dem S-Layer entspricht, hat die Aufgabe, Features zu extrahieren, während der subsampling/pooling Layer Verzerrungen oder Verformungen toleriert, wie es der C-Layer macht. Einen Unterschied bilden die Layer zum Schluss des Netzes. Die Zellen in diesen Layern sind nicht mehr mit einer *connection region* verbunden, sondern mit allen Neuronen im vorgänger-Layer, sodass das LeNet-5 ein hybrides Netz aus dem Neocognitron und einem fully-connected ANN bildet. [10]

Außerdem handelt es sich bei dem *Cognitron*, sowie dem *Neocognitron* um reine feedforward Netze. Daten werden als Input-Layer eingegeben, durchlaufen die Hidden-Layer, die die Gewichte enthalten, die beim Durchlaufen neu angepasst werden, und resultieren als Klassen im Output-Layer. Dabei arbeitet jeder Layer nur mit den Informationen des vorhergeheneden Layers und kriegt keinerlei Informationen bzgl. seiner Auswirkung im nachfolgenden Layer. Für das LeNet-5 wurde der von David E. Rumelhart

et al. vorgestellte Backpropagation-Algorithmus [3] angepasst, damit es mit den geteilten Gewichten arbeiten kann. Man spricht von geteilten Gewichten, da zwischen zwei Layern mit ein und demselben Gewicht, für jedes Neuron einer Feature Map im Folgelayer, gearbeitet wird. Diese Anpassung war erforderlich, denn der Algorithmus von David E. Rumelhart et al. modifizert jedes Gewicht individuell, was für die Filter zwischen zwei Layern eben nicht das Ziel ist.
Mit Ausnahme des Input Layers, besteht das LeNet-5 aus sieben Layern. In Abbildung 11 ist C1 ein convolutional Layer mit sechs 5x5 Pixel großen Filtern mit einem Stride von 1. Dieser kriegt ein 32x32 Pixel großes Bild als Eingabe und gibt sechs Feature Maps der Größe 28x28 Pixel weiter an den nächsten Layer. Dieser ist ein subsampling Layer mit sechs Filtern der Größe 2x2 Pixel und einem Stride von 2. Dadurch entstehen sechs Feature Maps der Größe 14x14 Pixel. Der dritte Layer ist wieder ein convolutional Layer mit 16 Feature Maps der Größe 10x10 Pixel und 16 Filtern der Größe 5x5 Pixel mit einem Stride von 1. Anstatt aber, dass jeder Filter mit allen Feature Maps von dem vorigen subsampling Layer kombiniert wird, wird mit verschiedenen Mengen von Feature Maps des vorigen Layers eine neue Feature Map erzeugt. Letztlich wird jede Feature Map nur 10- statt 16-mal verwendet. Dies hat zwei Vorteile. Der offensichtliche ist, dass dadurch die Rechenzeit um einiges verringert wird, denn mit dem 10- statt 16-fachen Verwenden der Feature Maps gibt es anstelle von 2.400 Gewichten nur 1.516 Gewichte, die trainiert werden müssen und beim Berechnen der einzelnen Neuronen der neuen Feature Maps sind ebenfalls weniger Rechenschritte erforderlich, nämlich 151.600 anstelle von 240.000. Der zweite Vorteil ist jedoch viel wichtiger. Dadurch, dass ein Filter auf verschiedenen Eingaben trainiert wird, wird dieser auch verschiedene und neue Features erkennen. Die Kombination der Feature Maps ist Tabelle 1 zu entnehmen. Sie ist so entstanden, dass die ersten sechs Feature Maps des convolutional Layers als Eingabe, jeweils verschieden, eine Teilmenge von drei benachbarten Feature Maps des subsampling Layers bekommen, die mit den entsprechenden Filtern verrechnet werden. Weitere sechs kriegen das gleiche nur mit vier benachbarten Feature Maps. Weitere drei kriegen eine Teilemenge mit vier nicht benachbarten Feature Maps und der letzte kriegt alle.

		Feature Map in C3															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Feature Maps aus S2	0	X				X	X	X			X	X	X	X		X	X
	1	X	X				X	X	X			X	X	X	X		X
	2	X	X	X				X	X	X			X		X	X	X
	3		X	X	X			X	X	X	X			X		X	X
	4			X	X	X			X	X	X	X		X	X		X
	5				X	X	X			X	X	X	X		X	X	X

Tab. 1: Jede Spalte gibt an, welche Feature Maps aus dem subsampling Layer für die Feature Map im convolutional Layer kombiniert wurden.

Der vierte Layer ist erneut ein subsampling Layer mit 16 Feature Maps der Größe 5x5 Pixel. Die Filter haben eine Größe von 2x2 Pixel und haben einen Stride von 2. Als Ausgabe kommen 16 Feature Maps der Größe 5x5 Pixel raus. Der fünfte Layer wird erneut als convolutional Layer bezeichnet. Dieser hat 120 Feature Maps je mit einer Größe von 1x1 Pixel. Weil hierbei 120 Filter der Größe 5x5 Pixel auf jeden der 16 vorhergehenden Feature Maps angewendet werden, entspricht der Layer ebenfalls einem fully-connected Layer. Der sechste Layer enthält 84 Filter und Feature Maps und ist ebenfalls wie der fünfte Layer fully-connected zum vorherigen Layer. Der siebte Layer ist der Output Layer und damit der letze im Netz. Dieser Layer bildet einen Softmax Layer, welcher die Klassifizierung ermöglicht.

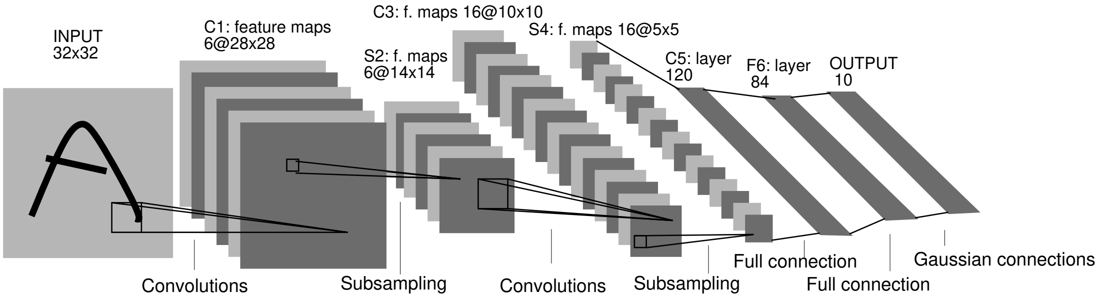


Abb. 11: Architektur des LeNet-5.

Wie das Neocognitron wurde das LeNet-5 grundsätzlich zur Klassifizierung handgeschriebener Ziffern und Buchstaben trainiert. Jedoch konnte Yann LeCun sein Netz populär machen, sodass es beispielsweise Anwendung in Banken der USA fand. Er hat an der Entwicklung eines Systems für einen Scanner mitgewirkt, welches bis heute zur maschinellen Erfassung von Daten auf Bankschecks verwendet wird. [11, 12]

Um die Performanceunterschiede zwischen dem Neocognitron und dem LeNet-5 fair messen und im Anschluss auch bewerten zu können, müssen beide Netze auf denselben Daten evaluiert werden. Dazu wurde 2014 in Fukushima [13] eine Durchführung einer Variante des Neocognitron auf dem MNIST Datensatz betrieben und eine Genauigkeit von 99,27% erreicht.

2.4 AlexNet

Nach dem LeNet-5 gab es eine Zeitspanne, in der sich recht wenig in dem Bereich von CNNs tat. Als das Interesse und die gegenwärtigen Möglichkeiten anstiegen, mehr als nur Ziffern und Zeichen zu erkennen, griff man das Thema CNN wieder auf. In der *ILSVRC-2012* sorgte das Ergebnis des Netzes *AlexNet* für eine Überraschung. Als einziges CNN erreichte es mit Abstand den ersten Platz der Challenge mit einer Fehlerrate von 15.4%. [1, 2]

In Krizhevsky et al. [14] wird der Aufbau des Netzes, sowie die Implementierung beschrieben. Dieses besteht aus 8 Layern, die ersten 5 sind convolutional Layer und die letzten 3 sind fully-connected Layer. Die Details zu den einzelnen Layern sind Tabelle 2, sowie Abbildung 12 zu entnehmen.

Viel wichtiger sind die weiteren Kniffe, die angewendet wurden. Da auf einem sehr großen Datensatz trainiert wurde, ergab sich beim Gradient Descent eine entsprechend sehr lange Trainingsdauer. Um dies zu beschleunigen wurde nicht, wie üblich, auf Aktivierungsfunktionen wie $f(x) = tanh(x)$ oder der logistischen Sigmoidfunktion $sig(x) = (1 + e^{-x})^{-1}$ zurückgegriffen, sondern auf die ReLU Funktion $f(x) = \max(0, x)$. Als weiteres Detail wurde eine Local Response Normalization (LRN) durchgeführt. Normalisiert wird, weil die ReLU Funktion für $x \to \infty$ unbeschränkt ist und LRN betrachtet dabei nur

die Nachbarschaft eines jeden Pixels, um so signifikante Pixel höher einzustufen. Zusätzlich wurde auch ein overlapping Max-Pooling verwendent, um Overfitting vorzubeugen.
Hinzu kommt noch eine weitere Besonderheit. Ausgehend von der Idee schnell auf GPUs zu trainieren stieß man auf ein Problem. Die Anzahl der Trainingsparameter bzw. das Netz wurde so groß, dass eine damals aktuelle GTX 580 3GB Grafikkarte alleine nicht gereicht hätte und man zu einer zweiten griff. Schon im Jahr 2012 konnten Grafikkarten untereinander lesen und schreiben ohne dafür extra mit dem Rechner zu kommunizieren. Dadurch konnte weiterhin eine relativ kurze Trainingsdauer (ca. 6 Tage) eingehalten werden. Also wurden die Filter bzw. Neuronen auf zwei Grafikkarten aufgeteilt. Als weiteren Kniff zur Beschleunigung der Trainingsdauer wurde das Netz so erstellt, dass die Grafikkarten nicht in jedem Layer miteinander kommunizieren. Die convolutional Layer 2, 4 und 5 erhalten als Eingabe nur die Ausgaben des vorigen Layers, welche sich auf derselben Grafikkarte befinden.

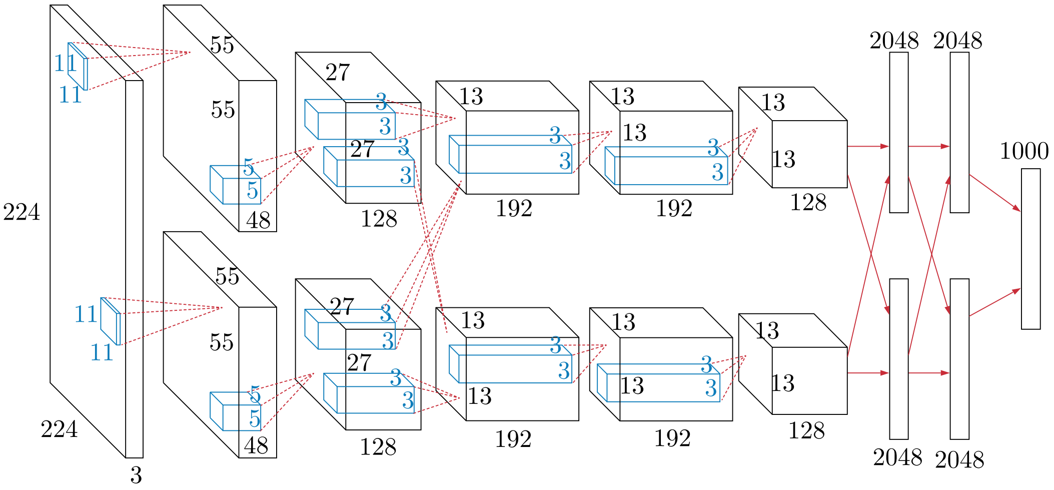


Abb. 12: Grobe Veranschaulichung der Architektur des AlexNet. Aufgeführt ist das Eingabebild, gefolgt von den 5 convolutional Layer und den 3 fully-connected Layer.

Layertyp	**Tensorgröße**	**Filtergröße & Anzahl**	**Parameter**
Eingabebild	224 x 224 x 3		
Convolution-1	55 x 55 x 96	11 x 11 x 3@96	34.994
ReLU + LRN			
Max-Pooling-1	27 x 27 x 96		
Convolution-2	27 x 27 x 256	5 x 5 x 48@256	614.656
ReLU + LRN			
Max-Pooling-2	13 x 13 x 256		
Convolution-3	13 x 13 x 384	3 x 3 x 256@384	885.120
ReLU			
Convolution-4	13 x 13 x 384	3 x 3 x 192@384	1.327.488
ReLU			
Convolution-5	13 x 13 x 256	3 x 3 x 192@256	884.992
ReLU			
Max-Pooling-3	6 x 6 x 256		
Fully-Connected-1	4.096 x 1		37.752.832
ReLU			
Dropout			
Fully-Connected-2	4.096 x 1		16.781.312
ReLU			
Dropout			
Fully-Connected-3	1.000 x 1		4.097.000
Softmax			
Klassifizierung	1.000 x 1		
			$\sum$ 62.378.344

Tab. 2: Detailierte Darstellung der Architektur des AlexNet. Etwas Verwirrung stiftet die Tatsache, dass in Krizhevsky et al. [14] von 8 Layern gesprochen wird. So wird in der Veröffentlichung Max-Pooling nicht als Layer gezählt, sondern als reine Operation nach einem convolutional Layer.

Die ImageNet Competition im Jahre 2013 wurde dann vom *ZF Net* mit einer Fehlerrate von 11.2% gewonnen. Dabei handelt es sich um eine Abwandlung des AlexNet. Die Autoren haben lediglich ein Fine-Tuning der Hyperparameter betrieben und dadurch ein besseres Ergebnis erzielt. [16, 15]

2.5 ResNet

Da man davon ausgeht mit tieferen Netzen komplexere Features erkennen zu können, liegt der Gedanke nahe, einfach ganz tiefe Netze zu erstellen, um so eine noch höhere Genauigkeit zu erhalten. Dies ist jedoch nicht ganz einfach zu realisieren, da man schnell auf ein Problem stößt. Beim Neujustieren der Gewichte durch Backpropagation, werden die Gradienten der immer näher zum Anfang liegenden Layer immer öfter mit den vorigen Gradienten multipliziert. Beim Multiplizieren zweier Zahlen $\in (0; 1)$ wird das Ergebnis kleiner als die kleinere der beiden Zahlen. Somit schwindet der Gradient für die am Anfang befindlichen Layer gegen Null und diese Layer werden viel langsamer trainiert, als die am Ende befindlichen. Dieses Problem ist als *Vanishing Gradient Problem* bekannt.
Dieses Problem wird geschickt durch das ResNet gelöst. Dabei steht ResNet für *Residual Neural Network* und beschreibt eher eine Funktionsweise als ein Netz. In Abbildung 13 sieht man wie eine Kante um die

gewichteten Layer hinzugefügt wurde. Diese Kante wird als *Skip Connection* bezeichnet. Mit solchen *Bausteinen* ist es dann möglich Architekturen zu erstellen, die sogar mehrere 100 Layer tief sein können.

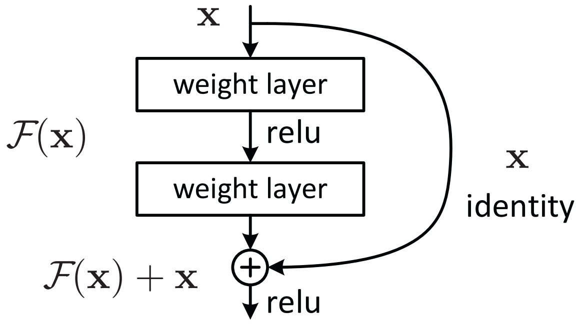


Abb. 13: Baustein zur Erstellung eines Residual Neural Networks.

Ausgehend von der Annahme, dass es eine optimale Funktion $H(x)$ gibt, die durch gewichtete Layer mit der Eingabe x durchgeführt werden sollte, um eine beste Ausgabe zu einer Eingabe x zu liefern, dann gibt es einen Unterschied $\mathcal{F}(x) = H(x) - x$ zwischen der Eingabe und der optimalen Funktion. Diese Gleichung lässt sich umstellen nach $H(x) = \mathcal{F}(x) + x$. Dabei wird $\mathcal{F}(x)$ als *Residual* bezeichnet. Nun soll in dem Layer das Residual gelernt werden, was laut den Autoren leichter zu lernen sei, als die optimale Funktion $H(x)$. Die Annahmen wurden von den Autoren experimentell auf tiefen Netzen getestet und durch die Ergebnisse bestätigt. Nun kann man bei Aneinanderreihung vieler convolutional Layer mit dem *Vanishing Gradient Problem* umgehen, denn bei einer Identitätsabbildung wird der Gradient mit 1 multipliziert und an den vorigen Layer übergeben und erreicht den ersten Layer ohne schwindend klein zu werden, um genug Einfluss auf die Neujustierung der Gewichte zu haben. Zahlreiche Experimentreihen mit tiefen Netzen haben gezeigt, dass diese Art der Erstellung tiefer Netze das Trainieren möglich macht (Architekturen mit 1202 Layern) und die Genauigkeiten durch die Tiefe besser sein können, als die von bisherigen Netzen. [17] Dies hat sogar dazu geführt, dass die Anwendung eines ResNet die *ILSVRC 2015* mit einer Fehlerrate von 3.6% gewann. [1, 18]

3 Neural Architecture Search

Anhand der in Kapitel 2 vorgestellten Netze ist zu sehen, wie sich der Umfang der Möglichkeiten ein Netz zu erstellen mit der Zeit immer weiter ausbreitete und komplexer wurde. Dieser Anstieg an Komplexität ist auf die Anwendungsgebiete von CNNs zurückzuführen. Anfangs war man interessiert gewesen, Zahlen und Buchstaben zu erkennen. Doch mit dem erfolgreichen und etablierten LeNet-5 (und Variationen davon) hatte weitere Forschung zu dem Thema vorerst keine hohe Relevanz mehr gehabt. Es wurden neue Anwendungsgebiete interessant, wie zum Beispiel die Gesichtserkennung auf digitalen Fotos, die mit dem Ablösen der Analogkamera durch die Digitalkamera ihren Aufschwung bekamen. Ein erkanntes Gesicht hilft beim Justieren des Fokus bei einer Kamera oder auch um bei Datenanalysen in der Lage zu sein, bestimmte Gesichter mit Datenbanken abzugleichen. Jedoch wurden dafür eher alternative Ansätze geliefert, wie es im Jahr 2001 beim Viola-Jones Algorithmus von Paul Viola und Michael Jeffrey Jones [19] als 38-fach kaskadierenden Klassifizierer der Fall war. Ein Mangel an Trainingsdatensätzen herrschte, sodass CNNs in den Hintergrund verschwanden. Erst als gezeigt wurde, dass komplizierte Netze wie AlexNet, ResNet und ZF Net deutlich bessere Ergebnisse liefern können, stieg das Interesse an der Forschung von Neuronalen Netzen wieder. Durch die derzeitig vielen Möglichkeiten eine Architektur zu gestalten, ist anhand der ILSVRC festzustellen, dass zu einer gegebenen Aufgabe jeder Entwickler eine Architektur erstellt, die sich in ihrem Aufbau und ihrer Genauigkeit von allen anderen unterscheidet. Wie eine Architektur im Ergebnis letztlich ausschaut, hängt dann von der menschlichen Expertise ab.

Neural Architecture Search soll hierbei in der Lage sein, zu einer Aufgabe automatisiert eine gute Architektur zu erstellen, sodass der Entwickler wenig Aufwand betreiben muss. Bestenfalls soll durch NAS die menschliche Expertise ersetzt werden.

3.1 Einführung in Neural Architecture Search

Elsken et al. [20] beschreiben eine allgemeine Vorgehensweise von *Neural Architecture Search.* Dabei wird der Vorgang in drei Stationen unterteilt: *Suchraum, Suchstrategie* und *Strategie zur Performancemessung.* Die Stationen sind wie folgt definiert und hängen wie in Abbildung 14 zusammen:

- **Suchraum.** Der Surchraum legt fest, welche Architekturen durch *NAS* generell erstellt werden können. Seine Größe kann unterschiedlich groß ausfallen. Dafür ist der Entwickler verantwortlich, denn er legt fest, wieviele und welche „Bausteine“ der Suchraum enthält, aus denen die Architektur gewonnen wird. Wenn man zusätzlich noch Wissen über die Aufgabe einbezieht, kann man den Suchraum auch einschränken und nur mit relevanten „Bausteinen“ arbeiten, und die Suche beschleunigen.
  Bei einer Architektur mit einfacher Kettenstruktur (ein sequentiell verlaufendes Netz, ohne Skip Connections) legt man im Suchraum die maximale Größe einer solcher Architektur fest. Als „Bausteine“ können einzelne Layer dienen. Man kann noch Hyperparameter, wie Strides oder Filtergrößen einbeziehen, damit man die Layer variieren kann.
- **Suchstrategie.** Durch die Suchstrategie wird festgelegt, wie man mit dem Suchraum arbeitet und so die Architektur aufstellt. Man hat hierbei das Ziel, schnell eine gute Architektur zu finden, jedoch soll man nicht zu schnell zu einem Ergebnis kommen, damit dabei nicht eine suboptimale Architektur herauskommt.

- **Strategie zur Performancemessung.** Das Ziel von *Neural Architecture Search* ist es, eine Architektur zu finden, die eine hohe Genauigkeit in ihrer Vorhersage hat. Bei der Strategie zur Performancemessung geht es darum, eine geeignete und möglichst schnelle Möglichkeit zu finden, die Performance eines Netztes messen zu können. Ein naiver Ansatz dafür wäre es, wenn man das Netz mit entsprechenden Daten trainiert und validiert. Dieser Ansatz ist jedoch sehr teuer, denn er erfordert viel Rechenleistung. Die aktuelle Forschung befasst sich damit, die Strategie zur Performancemessung zu optimieren.

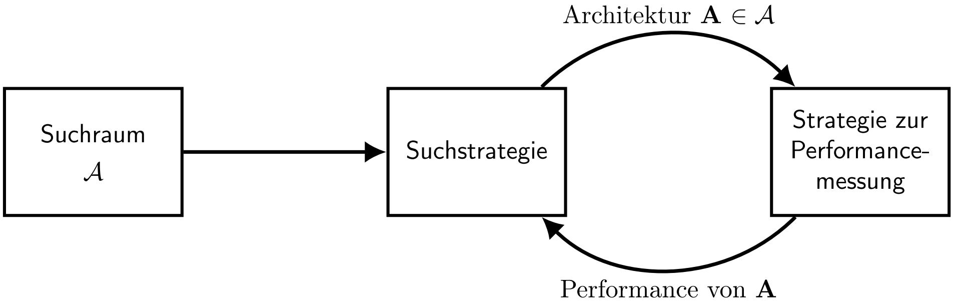


Abb. 14: Abstrakte Illustration der Vorgehensweise von *Neural Architecture Search*

Da die ersten beiden ausgewählten Forschungsarbeiten auf Methoden des Reinforcements Learning basieren, folgt vorerst ein Einschub zu diesem Thema.

Reinforcement Learning

Maschinelle Lernalgorithmen werden hauptsächlich in zwei Kategorien eingeteilt: Überwachtes und unüberwachtes Lernen. Reinforcement Learning ist ein Bereich des maschinellen Lernens, der explizit weder zu überwachtem noch zu unüberwachtem Lernen zugeordnet werden kann. Überwachtes Lernen hat zum Ziel eine nicht im Trainingsdatensatz präsente Datenreihe richtig zu klassifizieren, während beim Reinforcement Learning durch Interaktion mit einer Umgebung gelernt werden soll, und nicht anhand eines Traininsdatensatzes. Die Struktur der Umgebung zu verstehen ist dafür ganz wichtig, aber unüberwachtes Lernen hat nicht das Ziel darin eine Belohnung zu maximieren, die man durch eine Interaktion mit der Umgebung bekommt. Deshalb ist Reinforcement Learning eine eigene Kategorie des maschinellen Lernens, und hat somit auch seinen eigenen Algorithmenzoo.

Mit Blick auf Abbildung 15 sieht man, dass Reinforcement Learning Algorithmen mit einem sog. Agenten arbeiten. Der Begriff „Agent“ ist ein schwer zu definierender Begriff. Hierfür wird die Definition von Michael Wooldridge [21] gewählt, welche in [22] übersetzt wurde: „Ein Agent ist ein Computersystem, das sich in einer bestimmten Umgebung befindet und welches fähig ist, eigenständige Aktionen in dieser Umgebung durchzuführen, um seine (vorgegebenen) Ziele zu erreichen.“ Eine Umgebung ist die Welt des Agenten, in der er Aktionen durchführen kann. Aktionen sind alle Möglichkeiten zur Interaktion mit der Umgebung. Vor Durchführen einer Aktion befindet sich der Agent in einem Zustand. Wird eine Aktion durchgeführt, wird die Umgebung verändert und der Agent muss nun von einer neuen Umgebung ausgehen, er befindet sich daher auch in einem neuen Zustand. Neben dem Zustandswechsel bekommt der Agent auch eine Auswertung der durchgeführten Aktion in der Umgebung. Diese Auswertung wird Belohnung genannt. Der Agent muss dann die künftige Aktion basierend auf dem erhaltenen Ergebnis auswählen. Dabei verfolgt er eine Policy, die bestimmt, auf welche Art der Agent die nächste Aktion auswählt. [23]

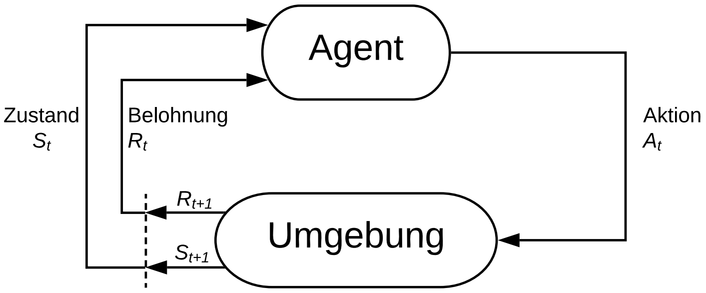


Abb. 15: Darstellung der Interaktion des Agenten mit der Umgebung.

Nachfolgend werden die drei ausgewählten Forschungsarbeiten zur Architektursuche vorgestellt. Basierend auf diesen werden die Arbeiten durch Experimente in Abschnitt 4 analysiert und ausgewertet.

3.2 Efficient Architecture Search

Cai et al. [24] setzen sich in erster Linie mit dem Problem bzgl. der Performancemessung auseinander. Der naive Ansatz die Genauigkeit zu messen ist zeit- und rechenintensiv, denn nach jeder noch so kleinen Modifikation der Architektur müsste man das gesamte Netz neu trainieren, um die Auswirkung der Modifikation bewerten zu können. Es wurde daher ein alternativer Weg eingeschlagen, sodass nicht nach jeder Modifikation das Netz neu trainiert werden muss. Ausgehend von der Annahme, dass bereits gute Netze sowohl von Menschen als auch maschinell für die gleichen Aufgaben erstellt wurden, basiert der Suchraum auf fertig trainierten Netzen. Ziel ist es, auf einem gewählten Netz Änderungen durchzuführen, die die Genauigkeit steigern sollen. Bei den Änderungen handelt es sich um Ausdehnung des Netzes, sowohl in Breite als auch in Tiefe. Änderungsoperationen an verschiedenen Stellen in der Netzarchitektur ergeben verschiedene Netze. Das Netz, welches nach seinen jeweiligen Operationen, die höchste Genauigkeit liefert, wird ausgegeben. Um diesen Vorgang zu realisieren, wurden zwei Ansätze von *Efficient Architecture Search* implementiert. Einer führt Operationen an zufälligen Stellen durch und der andere arbeitet mit einem Reinforcement Learning Agenten, sodass Operationen positiv, bzw. negativ belohnt werden und entschieden wird, wo welche Operation stattfindet.
Bei der Ausdehnung in Breite werden z.B. convolutional Layer mehr Filter und somit mehr Ausgabekanäle besitzen als ursprünglich. Dabei wird der Vorgang wie folgt erklärt. Ein Kernel K_l eines convolutional Layers l ist gegeben durch ein 4-Tupel der Form $(k_b^l, k_h^l, f_e^l, f_a^l)$ mit k_b^l und k_h^l für die Breite und Höhe der Filter und f_e^l und f_a^l für die Anzahl seiner Ein- und Ausgabekanäle. Um den Layer mit einem breiteren Layer zu ersetzen wird die Anzahl der Ausgabekanäle durch $\hat{f}_a^l(> f_a^l)$ vergrößert. Die Ausgabekanäle j mit $1 \leq j \leq \hat{f}_a^l$ sind gegeben durch die Mapping-Funktion

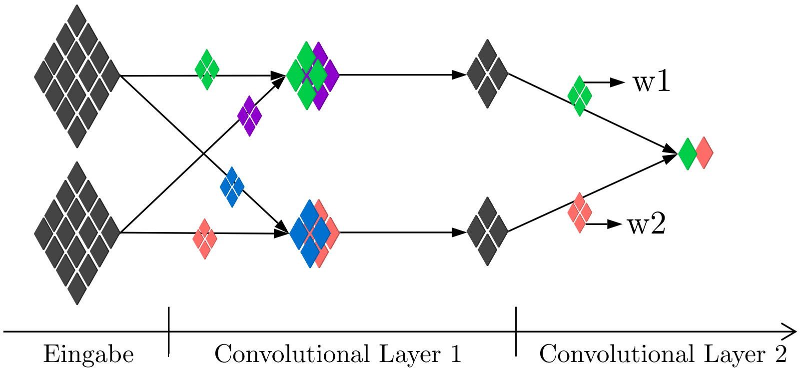
$$G_l(j) \coloneqq \begin{cases} j & \text{, falls } 1 \leq j \leq f_a^l \\ \text{zufällig aus } \{1, \ldots, f_a^l\} & \text{, falls } f_a^l < j \leq \hat{f}_a^l \end{cases}. \tag{27}$$

Diese Mapping-Funktion gibt einen neuen Kernel $\hat{K}_l \coloneqq (k_b^l, k_h^l, f_e^l, \hat{f}_a^l)$ mit $\hat{K}_l[x, y, i, j] = K_l[x, y, i, G_l(j)]$ zurück, sodass die ersten f_a^l Ausgabekanäle aus K_l übernommen werden, und die ergänzten $f_a^l + 1$ bis $\hat{f}_a^l$ werden zufällig durch G_l gewählt. Die Feature Maps werden entsprechend dem duplizierten Ausgabekanal übernommen. Somit ist die Ausgabe breiter in dem Sinne, dass mehr Filter zur Verfügung stehen. Damit das Netz weiterhin richtig funktionieren kann, muss auch der Kernel K_{l+1} des nachfolgenden Layers so angepasst werden, dass er die neue Anzahl der Ausgabekanäle von $\hat{K}_l$ als Eingabekanäle annimmt. Der nachfolgende gewichtete Kernel entspricht dann $\hat{K}_{l+1} \coloneqq (k_b^{l+1}, k_h^{l+1}, \hat{f}_a^l, f_a^{l+1})$

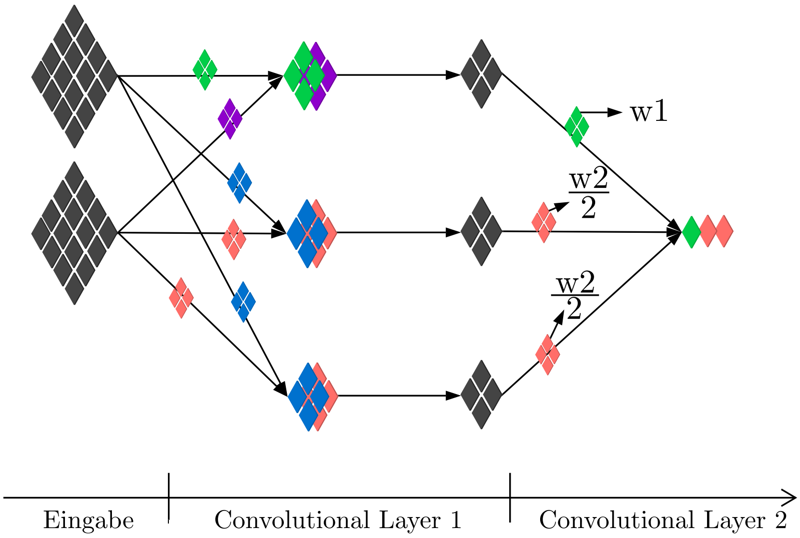
$$\hat{K}_{l+1}[x, y, j, k] = \frac{K_{l+1}[x, y, G_l(j), k]}{|\{z | G_l(z) = G_l(j)\}|}. \tag{28}$$

Dabei kann $|\{z|G_l(z) = G_l(j)\}|$ den Wert 1 haben oder größer. Der Betrag ergibt 1 für einen Ausgabekanal, falls dieser von $\hat{K}_l$ nicht durch Gleichung (27) dupliziert wurde. In dem Fall wird mit 1 Gewichtet, bzw. das Gewicht wird nicht verändert. Andernfalls, wurde beispielsweise ein Ausgabekanal aus $\hat{K}_l$ z.B. einmal dupliziert, ergibt der Betrag für diesen 2, da der Ausgabekanal zwei mal existiert. Würde der Betrag 1 ergeben, würden die zwei Filter von $\hat{K}_{l+1}$, die dieselbe Funktion erfüllen, doppelten Einfluss haben. Da die Ausbreitung initial vor dem Neutrainieren keinen Einfluss auf die Genauigkeit der Klassifizierung aufweisen soll, wird aus diesem Grund die Ausgabe einer Faltung durch das Gewicht im Betrag (s. Gleichung (28)) geteilt; im genannten Beispiel halbiert.

Im implementierten Ansatz von *Efficient Architecture Search* wird bei der Ausbreitung eines convolutional Layers immer die nächst höhere Anzahl der Kernel aus der Menge {16, 32, 64, 96, 128, 192, 256, 320, 384, 448, 512} gewählt. Waren es anfangs 32, werden es nach der Operation 64. Bei einem fully-connected Layer wird die Anzahl der Neuronen anhand von {64, 128, 256, 384, 512, 640, 768, 896, 1024} gewählt. Abbildung 16 veranschaulicht diesen Vorgang der Ausdehnung in Breite eines convolutional Layers, jedoch nicht nach Anordnung der definierten Menge, sondern von 2 zu 3 Kernels:



(a) In diesem Beispiel bekommt der erste convolutional Layer einen Eingabelayer mit zwei Eingabekanälen. Der convolutional Layer hat zwei Kernel mit je zwei Feature Maps wie der Anzahl der Eingabekanäle. Resultierend werden zwei Ausgabekanäle geliefert und kommen beim zweiten convolutional Layer als Eingabekanäle rein. Dieser führt eine weitere Faltung aus, diesmal mit einem Filter mit zwei Feature Maps.



(b) Der erste convolutional Layer hat diesmal drei Kernel mit je zwei Gewichten. Nach Gleichung 27 wurden die ersten beiden Kernel übernommen und das dritte zufällig gewählt. In diesem Fall wurde dafür der zweite Kernel (rot und blau) kopiert. Dadurch entstehen jetzt drei Ausgabekanäle, die an den zweiten convolutional Layer als Eingabe gegeben werden. Der zweite und dritte Eingabekanal sind dabei dieselben. So müssen die Ausgaben bei jeder Faltungsoperation auf diesen Eingabekanälen halbiert werden, damit die Qualität des Netzes vor dem Neutrainieren nicht verändert wird. Als Ausgabe des zweiten convolutional Layers würde in beiden Abbildungen weiterhin ein 1x1 Ausgabekanal folgen.

Abb. 16: Veranschaulichung der Ausbreitungsoperation anhand eines convolutional Layers. Abbildung 16a zeigt das Netz vor der Operation und Abbildung 16b das Netz aus Abbildung 16a nach der Operation.

Die Ausdehnung in die Tiefe geschieht durch Einschieben eines neuen Layers zwischen zwei Layern. Damit die Funktionalität weiterhin gegeben ist, wird dieser Layer initial eine Identitätsabbildung der Ausgabe des vorhergehenden Layers durchführen. Der neue Layer übernimmt auch die selbe Anzahl an Filtern wie sein vorhergehender Layer.

Der NAS Vorgang kann anhand folgendem Beispielcode beschrieben werden:

Alg. 1: Efficient Architecture Search

```
   Input  : Eine CNN Architektur A, tiefe=5, breite=4
   Output: Eine CNN Architektur A' $\neq$ A
1  Array cnnArray = new Array[300];
2  for ( $i = 0$; $i < 300$; $i++$ )
3      A_tmp = new CNNArchitecture(A.copy());
4      A_tmp = transformNet(A_tmp, tiefe, breite);
5      train(A_tmp, 20);
6      Reward r = calcReward(A_tmp);
7      updateAgent(r);
8      cnnArray[i] = A_tmp;
9  CNNArchitektur A_best = findBest(cnnArray);
10 A_best = train(A_best, 100);
11 cnnArray = new Array[150];
12 for ( $i = 0$; $i < 150$; $i++$ )
13     A_tmp = new CNNArchitecture(A_best.copy());
14     A_tmp = transformNet(A_tmp, tiefe, breite);
15     train(A_tmp, 20);
16     Reward r = calcReward(A_tmp);
17     updateAgent(r);
18     cnnArray[i] = A_tmp;
19 CNNArchitektur A_best = findBest(cnnArray);
20 A_best = train(A_best, 300);
21 return A_best;
```

Hierbei ist die Architektursuche in zwei Abschnitte gegliedert. Ausgehend von einer gegebenen Architektur **A**, erstellt der erste Abschnitt (Zeile 1-8) zunächst 300 Architekturen. Bei der Anzahl der Transformationsoperationen ist der Agent auf 5 Ausdehnungen in die Tiefe und von 4 Ausdehnungen in die Breite beschränkt. Diese 300 modifizierten Netze werden mit 20 Epochen trainiert. Die jeweils berechnete Belohnung wird an den Agenten weitergegeben. In einem Array werden die erstellten Architekturen gespeichert, damit in Zeile 9 daraus das beste Netz, anhand seines geringsten Fehlers auf einem Testdatensatz, gewählt wird. Dieses beste Netz wird anschließend mit einem noch höheren epoch Wert (=100) trainiert. Nun kann der zweite Abschnitt (Zeile 11-18) beginnen, welcher analog zum ersten arbeitet, jedoch als gegebene Architektur nun von der des als bestes ermittelten und trainierten Netzes ausgeht und daraus nun weitere 150 Netze erstellt. Aus diesen 150 Netzen wird erneut das beste Netz gewählt, mit 300 Epochen trainiert und dessen Architektur ausgegeben.

Die Aneinanderreihung von mehreren solchen Abschnitten zur Architektursuche kann beliebig wiederholt werden, jedoch wird das Netz mit jeder Hinzunahme eines solchen Abschnittes auch größer. Bei dem Ansatz mit Hilfe von Reinforcement Learning wird ein *LSTM* Network (Long short-term memory, dt: langes Kurzzeitgedächtnis) verwendet. Ein LSTM Network ist eine Weiterentwicklung des *Recurrent Neural Networks* (kurz: RNN). Ein RNN ist ein spezielles neuronales Netz, welches zur Eingabe einen Input-Vektor variabler Länge kriegen kann und entsprechend einen Outpput-Vektor variabler Länge zurückgibt. Ein RNN kriegt somit eine Serie an Eingaben. Dabei ist die Ausgabe an einem Index nicht von der jeweiligen Eingabe an einem Index abhängig, sondern von der ganzen Serie. Also beeinflussen alle vorigen Indizes des Vektors, die Ausgabe des aktuellen Index. Somit können bei gleicher Eingabe an einer Indexposition X, verschiedene Werte entstehen. Abbildung 17 zeigt, dass der Wert einer Position X des Ausgabevektors von dem Eingabewert an Position X und dem Hidden State der Position $X - 1$ abhängig ist. Der Hidden State wird zur Verrechnung mit dem Gewicht und der Eingabe herbeigezogen und arbeitet ähnlich wie ein Filter eines convolutional Layers in einem CNN.

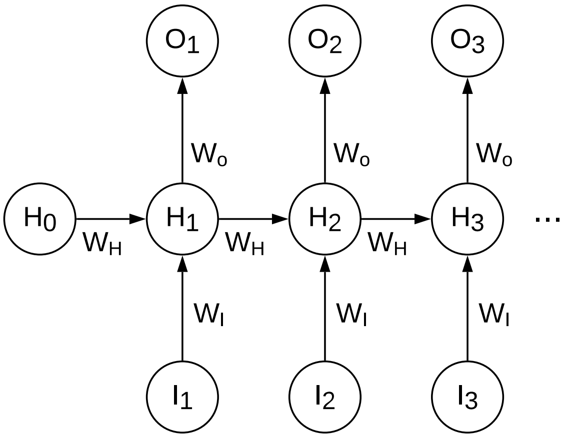


Abb. 17: Darstellung eines Recurrent Neural Networks. Mit $I_1 \dots$ ist der Eingabevektor gemeint, entsprechend mit $O_1 \dots$ der Ausgabevektor. $H_0 \dots$ stehen für die Hidden States.

Beim Trainieren eines RNN werden die Gewichte entsprechend angepasst. Für bestimmte Applikationen eines RNN (wie in EAS) besteht jedoch die Problematik des *Vanishing Gradient Problems* (s. Abschnitt 2.5), sobald größere Eingaben erfolgen. LSTMs beheben dieses Problem, indem sich die sog. LSTM-Zellen benötigte Informationen merken können. Dadurch wird ermöglicht, dass ein Gradient unverändert zurückpropagiert werden kann, sodass ein Schwinden gegen Null verzögert wird. [25]

Die Ausgaben der Hidden Layer des LSTM Netzes werden zur Entscheidung genutzt, ob an einer Stelle eine Operation durchgeführt werden soll. Zur Bestimmung ob eine Ausdehnung in Breite stattfindet, wird ein Sigmoid Klassifizierer genutzt, zur Entscheidung über Ausdehnung in Tiefe wird ein weiteres RNN verwendet.

Der gradientenbasierte policy Algorithmus *REINFORCE* [26] wurde zur Berechnung der Gewichte des LSTM Netzes implementiert. Damit soll die erwartete Belohnung einer Operation an einer bestimmten Position durch ein Gradientenabstiegverfahren maximiert werden. Die Trainingsumgebung des Agenten ist das gegebene vortrainierte Startnetz. Als Aktion erstellt der Agent ein neues Netz durch verschiedene Transformationsoperationen am gegebenen Netz. Das neue Netz wird für eine festgelegte Anzahl von Epochen trainiert und dessen Genauigkeit mit einem Datensatz validiert. Aus der Genauigkeit wird die Belohnung berechnet und das LSTM Netz angepasst. Dabei wird auch beachtet, dass die Belohnung höher ausfällt, wenn ein Anstieg der Genauigkeit von z.B. 90% auf 91% ansteigt, als von 60% auf 61%. Somit lernt der Agent welche Operationen eher sinnvoll sind und kann seine Strategie danach anpassen, um die Belohnung zu maximieren. Im Gegensatz zum zufälligen Ansatz führt dies dazu, dass der Fokus auf dem Erstellen eines besseren Netzes liegt und somit das Netz schrittweise besser wird.

Insbesondere ermöglicht diese Art von NAS ein Verfahren mit vergleichsweise wenig Rechenaufwand, denn es werden nicht neue Architekturen erstellt, die komplett neu trainiert werden müssen.

3.3 MetaQNN

In Baker et al. [27] wird ebenfalls ein mit Hilfe von Reinforcement Learning arbeitender Ansatz vorgestellt. Die Suchstrategie basiert auf dem Q-Learning Algorithmus. Dazu werden Q-Values bezogen mit Hilfe welcher die Architektur layerweise aufgebaut wird. Zur Exploration kommt der ϵ-Greedy Algorithmus zum Einsatz.

Q-Learning:
Q-Larning ist eine weitere Möglichkeit Reinforcement Learning anzuwenden. Beim Algorithmus *REINFORCE* [26] wird eine Belohnung erhalten, nachdem man eine Aktion durchführt und einen neuen Zustand erreicht hat. Im Gegensatz zu *REINFORCE*, wird beim Q-Learning eine Aktion schrittweise aufgebaut. Das Q-Learning wird daher der sog. Algrithmengruppe *Temporal-Difference Learning* zugewiesen. Jeder gewählte Schritt führt zu einem Zustand aus dem ein weiterer Schritt durchgeführt werden kann. Dabei kann sich ein Schritt positiv wie negativ auf die Belohnung der Aktion auswirken. Damit man später einen Schritt richtig wählen kann, pflegt der Agent in einer sog. Q-Tabelle die Qualität der Schritte als Q-Values. Der Agent führt nun in mehreren Aktionen verschiedene Schritte aus, und passt mit jedem Zustandswechsel die Q-Values mit einer neuen Schätzung an. Damit wird die Q-Tabelle trainiert. Nach genug durchlaufenen Aktionen konvergiert der Trainingsvorgang der Q-Tabelle. Anschließend kann man nach den besten Q-Values die Aktionen wählen. [28]
Darüber hinaus kann es nicht nur eine Belohnung für die Durchführung einer Aktion geben. Üblicherweise gibt es schon für die Durchführung eines Schrittes eine Belohnung. Nachfolgendes Beispiel [29] wird zur Hand genommen, um das Prinzip des Q-Learnings zu verdeutlichen.

Gegeben sei ein Graph wie er in Abbildung 18 abgebildet ist. Es wird angenommen, dass ein Agent in einem zufälligen Zustand startet. Das Ziel besteht darin, dass der Agent durch Kantenübergänge in Zustand 5 ankommt.

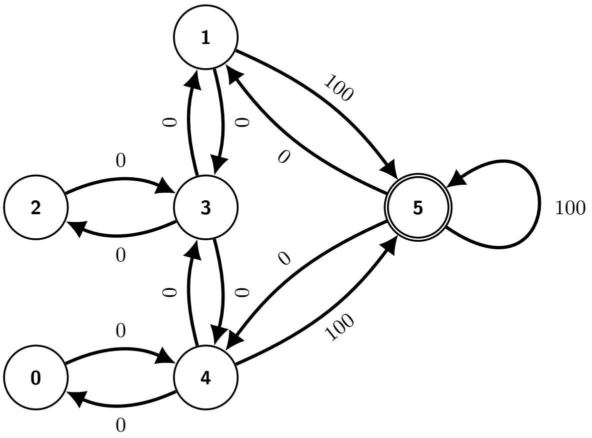


Abb. 18: Gegebener Graph als Trainingsumgebung für den Agenten. Die Kantenbezeichnungen stehen für die Belohnungen, die man bei dem Kantenübergang erhält.

Für das Q-Learning wird für diese Aufgabe zunächst eine Q-Tabelle aufgebaut. Die Q-Values für diese Tabelle werden nach mehrfachem Absolvieren der Aufgabe nach und nach trainiert. Da 6 Zustände existieren, erstellt der Agent eine Tabelle der Größe 6x6, die mit 0 initialisiert wird. Somit startet man mit der Q-Tabelle aus Tabelle 3a.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(a) Initialzustand der Q-Tabelle.

Für den ersten Durchgang wird angenommen, dass der Agent in Zustand 1 startet. Für seine Aktion hat er die Wahl zwischen dem Übergang zu 5 oder zu 3. Diese Auswahl wird zufällig getroffen. Nach zufälliger Auswahl, wählt der Agent den Übergang nach Zustand 5. Nun kann er den Q-Value für die Q-Tabelle berechnen. Dazu wird folgende Formel [28] herbeigezogen:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{29}$$

Der neue Q-Value wird durch den aktuellen Zustand S_t und einer Aktion A_t, die der Übergang in einen nächsten Zustand von S_t ist, berechnet. Dieser Übergang wird ausgeführt. Der neue Wert ergibt sich aus der Summe der alten Schätzung des Q-Values und aus einen Anpassungsterm. In dem Anpassungsterm ist die Belohnung R_{t+1} enthalten, welche durch Ausführen von A_t aus dem Zustand S_t erhalten wird. Auf diesen wird der maximal mögliche Q-Value addiert, der aus dem neuen Zustand durch einen weiteren Übergang theoretisch erreicht werden könnte. Dieser wird mit dem Discount Faktor $\gamma \in [0;1]$ gewichtet. Hierbei sagt γ, wie wichtig zukünftige hohe Belohnungen (Belohnungen nach mehreren Zustandswechseln) für den Agenten sind. Für $\gamma = 0$, wird nur der nächste Schritt, bzw. dessen Belohnung als wichtig empfunden. Mit steigendem γ dagegen, sind künftige hohe Belohnungen wichtiger, als vorige. Der gesamte Anpassungsterm ist mit α gewichtet, womit die Lernrate festgelegt wird. Von der Summe im Anpassungsterm wird zum Schluss noch die alte Schätzung des Q-Values abgezogen. Damit wird dafür gesorgt, dass die Q-Values nicht ins Unendliche steigen. Für das Beispiel ergibt sich also der neue Q-Value für Zustand 1 und die Aktion den Übergang zu 5 mit $\alpha = 0.1$ und $\gamma = 0.8$ wie folgt:

$$\begin{aligned} Q(1,5) &= Q(1,5) + 0.1 \cdot \left[100 + 0.8 \max_a Q(5,a) - Q(1,5)\right] \\ &= 0 + 0.1 \cdot [100 + 0.8 \cdot 0 - 0] = 10 \end{aligned} \tag{30}$$

Mit dem neuen Q-Value wird der entsprechende Eintrag in der Q-Tabelle aktualisiert (Tabelle 3b). Der Agent stellt dann fest, er hat Zustand 5 und damit sein Ziel erreicht und terminiert. Jetzt führt der Agent den zweiten Durchgang aus. Es wird angenommen, er startet in Zustand 3. Als nächsten Zustand wählt er zufällig 1. Der Agent berechnet nun für den Q-Value $Q(3,1)$ den Wert 0.8 und aktualisiert die Q-Tabelle (Tabelle 3c). Der Übergang zum Zustand 1 wird durchgeführt. Es handelt sich nicht um den Endzustand. Darauf wählt er erneut einen zufälligen Folgezustand, diesmal von 1 aus und entscheidet sich für 5. Der Q-Value $Q(1,5) = 19$ ersetzt nun den alten Wert in der Q-Tabelle (Tabelle 3d).

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	**10**
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(b) Q-Tabelle nach der Berechnung $Q(1,5)$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	**10**
2	0	0	0	0	0	0
3	0	**0.8**	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(c) Q-Tabelle nach der Berechnung $Q(3,1)$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	**19**
2	0	0	0	0	0	0
3	0	**0.8**	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(d) Q-Tabelle nach der Berechnung $Q(1,5)$ als Folge, dass nach Übergang von 3 zu 1 kein Endzustand erreicht wurde.

Führt man den Prozess weiter fort, konvergiert die Q-Tabelle. Das bedeutet, dass sich die Q-Values in der Q-Tabelle nicht weiter verändern. Nach wie vielen Schritten dieser Zustand erreicht wird, kann nicht genau bestimmt werden, da die Startposition zufällig gewählt wird. Abhängig davon können je Durchlauf verschieden viele Aktualisierungen der Q-Tabelle erfolgen. Tabelle 3e zeigt die konvergierte Q-Tabelle. Diese lässt sich anschließend zu Tabelle 3f normalisieren.

	0	1	2	3	4	5
0	0	0	0	0	400	0
1	0	0	0	320	0	500
2	0	0	0	320	0	0
3	0	400	256	0	400	0
4	320	0	0	320	0	500
5	0	400	0	0	400	500

(e) Q-Tabelle nach der Berechnung $Q(1,5)$.

	0	1	2	3	4	5
0	0	0	0	0	0.8	0
1	0	0	0	0.64	0	1.0
2	0	0	0	0.64	0	0
3	0	0.8	0.51	0	0.8	0
4	0.64	0	0	0.64	0	1.0
5	0	0.8	0	0	0.8	1.0

(f) Q-Tabelle nach der Berechnung $Q(3,1)$.

Tab. 3: Q-Tabellen in ihren Zwischenschritten zur Abbildung 18

ε-Greedy:

ε-Greedy ist eine Policy, die mit Q-Learning oft kombiniert wird. Diese Policy kann beeinflussen, wie sehr der Agent eher eine Exploration durchführt, statt einer Exploitation. Beim Q-Learning findet man im Anpassungsterm den Faktor $\max_a Q(S_{t+1}, a)$. Hier wählt der Agent stets den Q-Value aus, der den größten Wert hat. Damit kommt nur der Wert in Betracht, der dann zur höchsten Belohnung führt. Das ist auch alles gut, nur gibt es Aufgabenbereiche, in denen es sein kann, dass mal eine nicht als optimal angesehene Aktion sinnvoller sein kann. Bei der Erstellung von CNNs kommt es nämlich vor, dass eine andere Anordnung der Layer, aus unbestimmten Gründen, zu einer besseren Genauigkeit führt. Somit sollte der Agent nicht immer den Fokus auf der besten Aktion, die er kennt, haben, sondern ab und an auch unbekannte Aktionen wagen, die nicht wegen dem Maximalwert als optimal angenommen werden. Hier kommt ε-Greedy zum Einsatz. Dieser bestimmt die Wahrscheinlichkeit, zu welcher eine Exploration stattfindet, bzw. die Gegenwahrscheinlichkeit dazu, dass exploitet, also ausgenutzt, bzw. nach Regeln der Q-tabelle gearbeitet wird. Dazu wird der maximale Q-Value gewählt. [23]

Anders ausgedrückt, sei $\mathcal{A}(S)$ als die Menge aller Aktionen, die man aus Zustand S erreichen kann definiert und sei $\xi \sim U(0,1)$ eine Zufallszahl, die nach jedem Schritt neu generiert wird. Dann wird der Q-Value für ein gewähltes $\varepsilon \in [0,1]$ wie folgt berechnet:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma\pi(S_{t+1}) - Q(S_t, A_t)\right] \tag{31}$$

mit

$$\pi(S) \coloneqq \begin{cases} \text{zufällig aus } \mathcal{A}(S) & \text{, falls } \xi \leq \varepsilon \\ \max\limits_{a \in \mathcal{A}(S)} Q(S, a) & \text{, falls } \xi > \varepsilon \end{cases} . \tag{32}$$

Die Arbeit von Baker et al. kann anhand des Beispiels in Abbildung 18 erklärt werden. Dabei ist ein Zustand ein Layer samt seinen Parametern und ein Übergang führt zur Auswahl des darauf folgenden Layers. Der Architekturaufbau erfolgt so durch sequentielle, gewählte Aneinanderreihung von Layern. Zur Layerauswahl stehen convolutional Layer und pooling Layer zur Verfügung. Ein dropout Layer folgt nach jedem zweiten Layer. Wie sich nachher zeigt, ist der pooling Layer jedoch nur eingeschränkt einsatzbereit. Nachdem eine Architektur gefertigt wurde, wird das Netz trainiert und anhand eines Validierungsdatensatzes wird die Genauigkeit berechnet. Diese wird als Belohnung angesehen und damit werden die Q-Values für die gewählten Layer rückwirkend angepasst. Das Netz und seine Genauigkeit wird zwischengespeichert, da mit ε-Greedy die Layer gewählt werden, und es vorkommen kann, dass eine Architektur mehrmals erstellt wird. Deswegen kann einfach auf die bereits berechnete Genauigkeit zugegriffen werden ohne nochmals zu trainieren.

Der Suchraum bei diesem NAS Vorgang beschränkt sich auf die Anzahl verschiedener Layermöglichkeiten, also auf die Layertypen und deren Parameter, wie Filtergrößen oder Filteranzahl eines convolutional Layers. Die Architektursuche erfolgt zunächst mit komplettem Zufall bei der Auswahl der Layer und seiner Parameter. Dies wird durch Setzen von $\varepsilon = 1$ ermöglicht. Danach sinkt ε schrittweise um 0.1 bis $\varepsilon = 0.1$, sodass der Agent mit sinkendem ε immer mehr exploitet. Dabei wird vor der Architektursuche festgelegt, wie viele neue Architekturen bei den einzelnen ε erstellt werden sollen und wie tief ein erstelltes Netz sein darf. Erst dann, wenn die Anzahl neu entdeckter Architekturen erreicht wurde, sinkt ε um 0.1. Hierbei zählt eine Architektur als neu, wenn sie auch wirklich nicht in einem vorigen oder aktuellem ε gefunden wurde.

3.4 Genetic CNN

Lingxi Xie and Alan L. Yuille [30] stellen ein evolutionäres Verfahren zu Neural Architecture Search für CNN vor. Dazu wurde ein genetischer Algorithmus verwendet. Diesem Verfahren werden zu Beginn sog. *Stages*, sowie die Anzahl der Knoten pro Stage gegeben. Eine Stage ist ein convolutional Block, in welchem mehrere convolutional Layer, dargestellt durch Knoten, enthalten sind. Diese Knoten können auf unterschiedlichste Weise miteinander verbunden sein, sodass bekannte Architekturen wie das des VGGNet, ResNet oder DenseNet dargestellt werden können. Das Verfahren soll optimale Verbindungen der convolutional Layer je Stage finden.

Nachfolgend wird erklärt, wie ein genetischer Algorithmus funktioniert, und wie er zur Umsetzung von *Genetic CNN* genutzt werden kann.

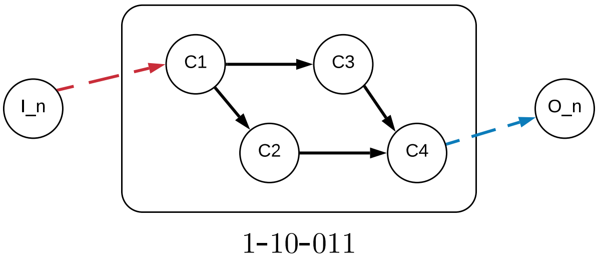
Genetischer Algorithmus:

Bei genetischen Algorithmen handelt es sich um aus der biologischen Evolution nachempfundenen Verfahren zur Optimierung eines Problems. Sie spiegeln dabei den Prozess der natürlichen Selektion wider. Dabei wird zunächst das Problem durch eine Bitfolge dargestellt. Jede mögliche Bitfolge stellt als *Individuum* eine Lösung für das Problem dar. Um den Prozess zu starten wird eine Menge von zufällig gewählten Bitfolgen als *Startpopulation* generiert. Durch eine Auswertung jedes Individuums mit Hilfe einer sog. *Fitness Funktion* soll *selektiert* werden, ob das Individuum zur Reproduktion geeignet ist und somit überlebt oder nicht. Nachdem man durch die Selektion eine Menge von überlebenden Individuen erhält, findet für je zwei Individuen mit einer bestimmten Wahrscheinlichkiet eine *Crossover*-Operation statt. Dies führt eine Rekombination von *Genen* (Bitsequenzen) durch. So entstehen neue Individuen. Darüberhinaus kann ein Individiuum auch mit einer Wahrscheinlichkeit mutieren. Eine *Mutation* hat ein Flippen eines oder mehrerer Bits zur Folge. Ist die neue Population wieder ausreichend groß, so spricht man, dass eine neue Generation entstanden ist. Dieser Vorgang wird so lange wiederholt, bis eine *Terminierung* eintrifft. Terminiert werden kann nach einer festgelegten Anzahl an Generationen oder wenn sich die Qualität einer Generation nicht signifikant von der vorigen unterscheidet. Da gute Individuen wahrscheinlicher überleben als schlechte, erwartet man mit jeder Generation eine immer qualitativ bessere Menge an möglichen Lösungen für das gegebene Problem.

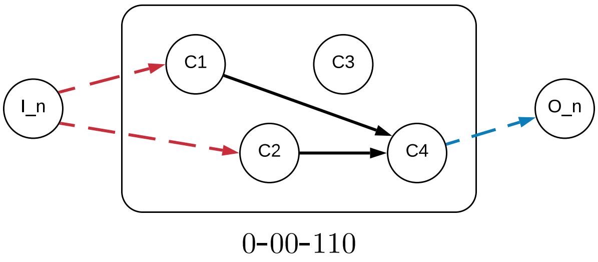
Nun folgt die Umsetzung des genetischen Algorithmus. Angefangen wird mit der Kodierung. Abbildung 19 veranschaulicht, wie durch eine Bitfolge eine Stage dargestellt wird. Zu beachten ist, dass die Knoten in einer Stage geordnet sind und nur Kanten von niederen zu höheren Knoten in der Stage erlaubt sind. Daraus erschließt sich, dass der erste Knoten keinen Vorgänger haben kann, der zweite Knoten nur den ersten, der dritte den ersten und den zweiten usw. Die Länge des Bitstrings für eine Stage kann also durch die Anzahl der Knoten berechnet werden:

$$\textit{Bitlänge}\,(n) = \sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \tag{33}$$

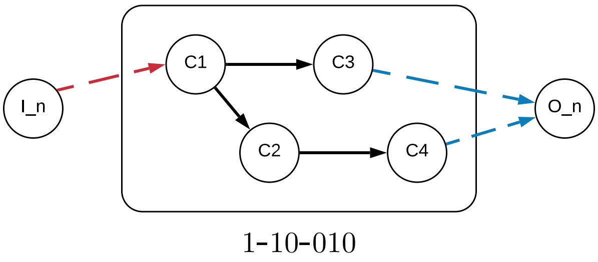
Dabei kann man den Bitstring in Bitgruppen/Gene unterteilen. Beginnend an Position 1 wird das erste Gen durch eine Sequenzlänge von 1 dargestellt. Das nächste Gen beginnt mit der nächsten Position und ist stets um eine Stelle länger als das davor. Das n-te Gen beschreibt aus welchen Vorgängern eingehende Kanten zum $n{+}1$-ten Knoten zeigen. Das dabei n-te Bit im Gen repräsentiert den n-ten Knoten aus der geordneten Menge seiner Vorgänger. So hat die Stage in Abbildung 19a bspw. 3 Gene für 4 Knoten. Das erste Bit im ersten Gen sagt aus, dass der Knoten *C2* als eingehenden Knoten *C1* hat. Beim zweiten Gen kriegt *C3* nur von *C1* eine eingehende Kante, aber nicht von *C2*. Das dritte Gen beschreibt den Knoten *C4*. Dieser kriegt von *C1* keine Kante, aber von *C2* und *C3* jeweils eine.
Darüber hinaus werden in Abbildung 19 alle möglichen Fälle dargestellt die im einzelnen auftreten können, die aber beliebig ausbaubar sind.



(a) Ein Bitstring, bei dem der Input in nur einen Knoten übergeben wird und der Output nur von einem Knoten kommt.



(b) Eine Kombination des Bitstrings, bei dem mehrere Knoten den Input erhalten. Ein Knoten kriegt dann ein Input, wenn dieser keine eingehende Kante durch die Kodierung kriegt, aber eine Kante von diesem durch die Kodierung ausgehen sollte. Man beachte, dass der Knoten *C3* keine eingehende Kante kriegt, weil kein weiterer Knoten einen Input von ihm erwartet.

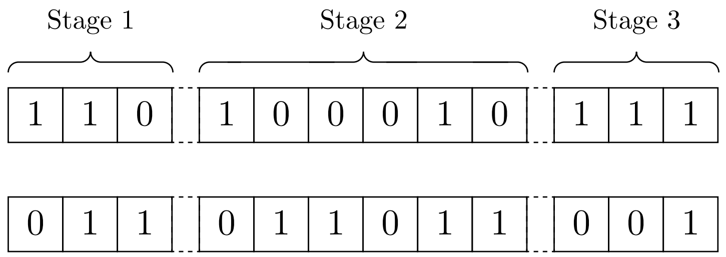


(c) Eine Kombination, bei der der Output durch mehrere Knoten entsteht. Dies ist dann möglich, sofern mehrere Knoten exisiteren, die eingehende Kanten haben, aber keine ausgehenden. Dies erklärt warum in Abbildung 19b der Knoten *C3* keinen Output liefert.

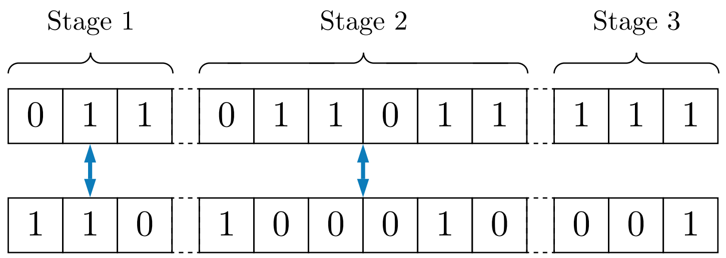
Abb. 19: Drei Fälle von Stages, die durch die Kodierung in Bitstrings entstehen können. Zudem die Kodierung zu jeder Stage, sowie die Zusammenstellung mit dem Input und dem Output.

Die Länge der kompletten Bitfolge ist somit die Summe der Länge der Bitstrings aller Stages. Nun kann eine Population festgelegter Größe erzeugt werden. Dabei werden n zufällige Bitfolgen erzeugt. Daraus wird jeweils ein Netz mit Stages konstruiert. Die Fitness Funktion berechnet den Fitness Score s für ein

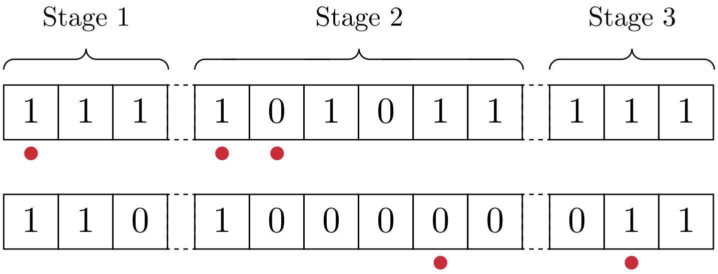
Individuum. Für diesen wurde die Genauigkeit des Netzes gewählt, nachdem es trainiert und mit einem Testdatensatz evaluiert wurde. Mit Hilfe eines Roulette Verfahrens findet die Selektion statt. Dabei wird zu einer Wahrscheinlichkeit von $(1-s)$ entschieden, ob ein Individiuum stirbt. Die Crossover-Operation wird dann für je zwei Individuen mit einer Wahrscheinlichkeit p_c durchgeführt. Dabei wird für zwei übereinanderliegende Gene/Stages mit einer Wahrscheinlichkeit q_c ein Austausch durchgeführt. Danach wird für jedes Individuum entschieden ob eine Mutation stattfinden soll. Eine Mutation findet zu einer weiteren Wahrscheinlichkeit p_m statt. Dabei wird jedes Bit mit einer Wahrscheinlichkeit q_m geflippt. Die Crossover-Operation sowie die Mutation wird in Abbildung 20 anhand eines Beispiels im Detail gezeigt. Als Input in eine Stage dient dann entweder das initiale Eingabebild oder eine oder mehrere Feature Maps. Die Knoten in einer Stage führen jeweils eine Convolution-Operation durch. Damit die Aneinanderreihung mehrerer convolutional Layer in einer Stage nicht dazu führt, dass die Dimension der Eingabe kleiner wird, wird wie beim ResNet (Abschnitt 2.5) ein Padding hinzugefügt. Nach einer Stage kann dann entweder ein weiterer convolutional Layer folgen, ohne Padding, oder ein pooling Layer. Anschließend kann eine weitere Stage folgen. Jeder Layer innerhalb der Stage hat entsprechend auch gleich viele Filter, sowie Ausgabegrößen, sodass eine Konsistenz in Größe und Anzahl der Feature Maps gegeben ist. An das Ende des Netzes wird ein fully-connected Layer und ein soft-max Layer angefügt. So lässt sich ein Netz optimieren, bei dem jeder convolutional Layer durch eine Stage ersetzt wird, deren optimalen Hyperparameter – die Kanten – durch den genetischen Algorithmus gefunden werden können.



(a) Hier wurden zwei Individuen gewählt. Mit einer Wahrscheinlichkeit von p_c soll für diese ein Crossover stattfinden.



(b) Es findet der Crossover statt. Je zwei übereinanderliegende Stages werden nun zu der Wahrscheinlichkeit q_c getauscht. In diesem Beispiel hat es Stage 1 und Stage 2 getroffen.



(c) Der Crossover wurde nun für das eine Paar durchgeführt. Nachdem dies für alle Individuenpaare durchgeführt wurde, wird für jedes Individuum geschaut, ob eine Mutation stattfinden soll. Dies geschieht mit der Wahrscheinlichkeit p_m. Die hier mit rotem Punkt gekennzeichneten Bits wurden anschließend mit der Wahrscheinlichkeit q_m geflippt.

Abb. 20: Crossover und Mutation am Beispiel erklärt.

Hier ist auch leicht zu erkennen, dass es vorkommen kann, dass eine Generation weniger neue Individuen haben kann, als die zu Beginn n große, erzeugte Startpopulation. Denn wegen den Wahrscheinlichkeiten für Crossover und Mutation bleiben manche Individuen unverändert. So werden nur die neuen Individuen trainiert und ausgewertet. Die übrigen übernimmt man inklusive ihrer alten Fitness Scores in die neue Generation.

4 Experimente

Dieser Abschnitt beinhaltet den praktischen Teil dieser Arbeit. Zunächst werden bekannte Datensätze vorgestellt sowie ein für diese Arbeit erstellter Datensatz. Anschließend erfolgt die Durchführung der Experimente. Dabei werden die NAS Verfahren auf einem für diese Arbeit zusammengestellten Datensatz angewendet. Zum Ende wird dann die Auswertung durchgeführt.

4.1 Datensätze

Im Bereich der CNNs gibt es eine Reihe von populären Standarddatensätzen, die zum Trainieren verwendet werden. Die mit ihnen erreichten Genauigkeiten werden genutzt, um einen Vergleich verschiedener CNNs zu haben, damit man Schlüsse über die Qualität des Netzes ziehen kann. Abhängig vom Anwendungsgebiet benötigt man spezielle Datensätze, von denen viele nicht einfach öffentlich abrufbar sind. Zu den bekannten gehören unter anderem MNIST [31], ImageNet [32], CIFAR10 [33] und SVHN [34], die entweder öffentlich oder auf Anfrage zu Forschungszwecken zur Verfügung gestellt werden:

- **MNIST.** Dieser Datensatz ist einer der bekanntesten. Er wurde von Yann LeCun erstellt, um sein LeNet-5 (2.3) und weitere Netze zu trainieren. Dieser besteht aus handgeschriebenen Zahlen. Der Trainingsdatensatz beinhaltet 60.000 Bilder und der Testdatensatz nochmal weitere 10.000. Mit MNIST wurde ein Dateiformat *IDX* eingeführt. IDX ist ein einfach gehaltenes Format zur Speicherung von Vektoren und multidimensionalen Matrizen.
- **ImageNet.** ImageNet ist eine Bilderdatenbank mit rund 14 Millionen Bildern. Jedes Bild wird einer Synonymmenge mit Substantiven zugeordnet. Jeder Synonymmenge gehören im Schnitt über 500 Bilder. Insgesamt gibt es rund 20.000 solcher Synonymmengen, die zusammen über 80.000 Substantive besitzen. Die Kategorien sind sehr breit gefächert, sodass es Bilder von Kissen oder Tieren oder verschiedenem Besteck gibt. ImageNet zählt als annotierte Bilderdatenbank zu einer der größten.

  Zudem findet seit 2010 jährlich die *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* statt, bei der das am besten arbeitende Netz auf dem ImageNet Datensatz gesucht wird.
- **CIFAR10/CIFAR100.** CIFAR10 ist ein Datensatz mit 60.000 kleinen (32x32 Pixel) Bildern. 6.000 Bilder gehören jeweils zu einer von 10 Klassen. Der Trainigsdatensatz besteht aus 50.000 Bildern, der Testdatensatz entsprechend aus 10.000 Bildern. Zudem gibt es noch den CIFAR100 Datensatz, der sich nur darin unterscheidet, dass er 100 Klassen hat, mit 600 Bildern pro Klasse (500 Trainingsbilder und 100 Testbilder pro Klasse).
- **SVHN.** Hierbei handelt es sich um einen ähnlichen Datensatz wie bei MNIST. SVHN ist kurz für *Street View House Numbers*. Dabei handelt es sich um Bilder mit Zahlen, die aus Hausnummern aus Google Street View extrahiert wurden. Gestellt wird ein Trainingsdatensatz mit 73.257 Ziffern. Weitere 26.032 Ziffern hat man zum testen. Zusätzlich gibt es noch weitere 531.131 eher leicht zu erkennende Ziffern.

Die vorgestellten Arbeiten zu den NAS Verfahren in Abschnitt 3.1 haben ihre Ergebnissen anhand einem oder mehrerer dieser Datensätze vorgestellt. Dies wird in der Forschung häufig so gehandhabt, da es sich zu einem Standard entwickelt hat, mit den Ergebnissen auf diesen Datensätzen die Qualität eines Netzes zu zeigen.

Damit die Arbeiten zu NAS auch hinsichtlich ihrer Anwendung auf andere Datensätze ausgewertet werden können, wurde im Rahmen dieser Arbeit ein Datensatz zusammengestellt:

- **NaoTH Balldatensatz 2016.** Dieser Datennsatz besteht aus rund 170.000 Bildausschnitten von Bildern, die mit dem Nao v5 im Jahr 2016 aufgenommen wurden. Dabei handelt es sich um Grauwertbilder, die eine Größe von 16x16 Pixeln haben. Jeder dieser Bildausschnitte hat entweder einen Ball abgebildet oder nicht. Die entsprechenden Klassen wurden händisch zugeordnet. Für diese Arbeit wurden je Klasse 3.833 Bilder nochmals händisch selektiert. Die Bildausschnitte wurden vom RoboCup Team der Humboldt-Universität zu Berlin bereitgestellt. In Abbildung 21 sind je Klasse 9 Beispiele abgebildet.

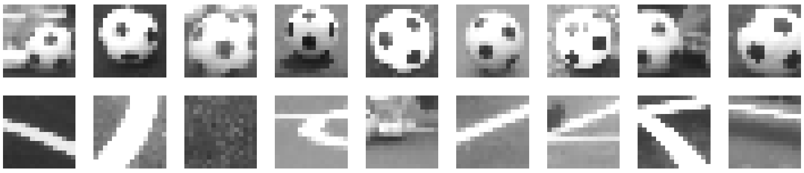


Abb. 21: Beispiele des NaoTH Balldatensatz 2016. Oberhalb sind Bilder der Klasse mit dem Fußball, unterhalb die ohne. Der hierfür verwendete Datensatz ist unter dem angelegten GitHub Respository [35] zu finden.

In den folgenden Teilabschnitten wird beschrieben, wie die Experimente durchgeführt wurden. Dazu gehören die Rechnerarchitekturen auf denen die Experimente liefen, die eingesetzte Programmiersprache, die Laufzeiten, die Ausgabe und darüber hinaus, welche Vorkehrungen bzw. Vorbereitungen eines Nutzers zu unternehmen sind, damit ein Experiment gestartet werden kann. Zum Schluss folgt immer ein Zwischenfazit zur jeweiligen Arbeit.

4.2 EAS

Zu EAS wurde in der Publikation auf ein von den Autoren erstelltes GitHub Repository [36] verwiesen. Es handelt sich um den original verwendeten Code.

Mit dem Code der Autoren ließen sich auch nach Anpassungen nach stundenlangen Training keine Ausgaben erzielen. Nichtsdestotrotz werden nachfolgend die Ergebnisse der Autoren [24] vorgestellt, um eine Vorstellung zu haben, wozu EAS in der Lage ist. Gestartet wurde mit folgender Architektur:

```
C(16,3,1), P(2,2,MAX), C(32,3,1), P(2,2,MAX), C(64,3,1), P(2,2,MAX),
C(128,3,1), P(4,4,AVG), FC(256), SM(10)
```

Dabei steht `C(16,3,1)` für einen convolutional Layer mit 16 Filtern der größe 3x3 und einem Stride von 1. Bei `P(2,2,MAX)` handelt es sich um ein Max Pooling mit der Filtergröße 2x2 und einem Stride von 2, entsprechend der Reihenfolge der Eingabe dieser Werte. `FC(256)` ist ein Fully-Connected Layer der Länge 256 und `SM(10)` beschreibt einen Soft-Max Layer mit einem Ausgabevektor mit 10 Feldern. Gearbeitet wird auf dem CIFAR10 Datensatz und dort wurde zunächst auf oben genannter Architektur eine Genauigkeit von 87.07% erreicht. Dann wurde EAS darauf ausgeführt. Die eingesetzte Konfiguration bestand aus 2 Phasen. Die erste Phase erlaubte 4 Operationen in die Breite und 5 Operationen in die Tiefe. Es wurden 300 Netze generiert und trainiert, wovon nochmals das beste mit 100 Epochen trainiert wurde. Die zweite Phase unterschied sich darin, dass 150 Netze ausgehend vom besten der ersten Phase generiert wurden. Eine Auswahl der besten Netze nach der zweiten Phase wurde nochmal mit 300 Epochen auf dem vollständigen Datensatz trainiert. Das daraus resultierende beste Netz hatte eine Genauigkeit von 95.11%. Diese Architektur wurde dann nochmals auf dem vollständigen SVHN Datensatz mit 40 Epochen trainiert und erreichte eine Genauigkeit von 98.17%. So wurde das Ergebnis von EAS mit dem Ergebnis des NAS Verfahrens, MetaQNN, verglichen, sowie mit dem Ergebnis weiterer von Menschen erstellten Netzen. In nachfolgender Tabelle sind die Ergebnisse zu sehen:

Typ	Name	CIFAR10+	SVHN
menschlich erstellt	Maxout [37]	9.38	2.47
	NIN [38]	8.81	2.35
	All-CNN [39]	7.25	-
	VGGnet [40]	7.25	-
maschinell erstellt	MetaQNN (Tiefe=7) [27]	6.92	-
	MetaQNN(Tiefe=7, neu trainiert)	-	2.06
	EAS(Tiefe=16)	4.89	1.83
	EAS(Tiefe=20)	**4.23**	**1.73**

Tab. 4: Vergleich der Fehlerrate von verschiedenen CNNs, sowohl von Menschen erstellt, als auch durch NAS. In Fett ist das beste Ergebnis bei EAS zu sehen.

Zwischenfazit

Den Autoren war es wichtig, den Punkt der Strategie zur Performancemessung bei der Vorgehensweise von NAS (s. Abbildung 14) zu optimieren. Dabei wird ein fertig trainiertes Netz verwendet und nachoptimiert. Die Zeit zum trainieren mit den vorher genannten Einstellungen dauerte weniger als 2 Tage auf 5 GeForce GTX 1080 GPUs. Es ist schwer einen Vergleich zu anderen Verfahren zu machen, wenn die Verfahren nicht auf den gleichen Rechnerarchitekturen ausgeführt werden. Daher bleibt offen, ob EAS wirklich so Zeitsparend ist, wenn man mit vortrainierten Netzen arbeitet. Darauf ist in der Veröffentlichung leider nicht weiter eingegangen worden.
In Abbildung 22 kann man erkennen, wie das Verfahren zu einem steigenden Wachstum in der ersten Phase führt. So verfolgt EAS erfolgreich das Ziel, eine bessere Architektur zu finden. Auch weitere Experimente bestätigen diese These und zeigen, dass der Reinforcement Learning Ansatz seine Arbeit richtig macht und in der Lage ist noch weitere bestehende Netze in der Genauigkeit zu übertreffen.

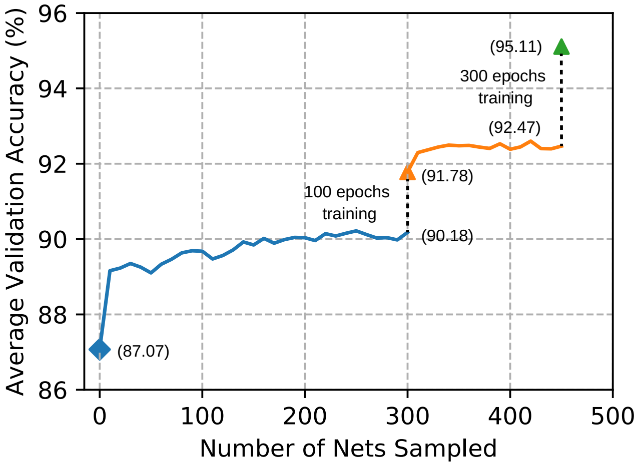


Abb. 22: Architektursuche der beiden Phasen auf dem vollständigen CIFAR10 Datensatz.

Die resultierende Architektur nach allen Anpassungen wird in der Veröffentlichung weiter nicht gezeigt. Jedoch ist diese Herangehensweise eine Architektur zu erweitern mit dem Nachoptimieren durch menschliche Expertise gleichzusetzen. Im Falle von EAS muss nur der Mensch selbst nicht anhand der Zwischenresultate seiner Veränderungen die nächsten Veränderungen durchführen. Dies wird durch den Agenten von EAS übernommen. Dies führt zum Schluss, dass EAS die menschliche Expertise in der Nachoptimierung sehr gut und teils besser ersetzen kann. Als Entwickler hat man nur die Aufgabe eine bereits erstellte und vortrainierte Architektur als Eingabe zu geben, wenn man nicht den Zufallsansatz von EAS nutzt. Die Parametrisierung dafür scheint nicht außergewöhnlich komplex zu sein. Weitere Experimente haben gezeigt, dass der Zufallsansatz eher schlechter abschneidet, weshalb zum Reinforcement Learning Ansatz geraten werden kann.

4.3 MetaQNN

Zu MetaQNN wurde in der Publikation auf ein von den Autoren erstelltes GitHub Repository [41] verwiesen. Es handelt sich um den original verwendeten Code.

Einrichtung

MetaQNN wurde in der Programmiersprache *Python* geschrieben. Dazu wurde die Version 2.7 verwendet. Zusätzlich wurde mit dem Framework *Caffe* gearbeitet. Caffe wurde im Rahmen einer PhD arbeit entwickelt und dient dazu CNNs zu erstellen. Es wurde Wert darauf gelegt, dass ein Netz schnell trainiert werden kann. Laut den Entwicklern wird beschrieben, dass Caffe mitunter zu den am schnellsten arbeitenden Implementationen für CNNs gehört. [42]
Zur Einrichtung gehört das Aufstellen seiner Python Umgebung. In dem erstellten GitHub Repository findet sich eine Anleitung zur Einrichtung. Dazu gehört zum einen die Installation benötigter Bibliotheken für Python, zum anderen die Installation von Caffe selbst. Da es sich bei Caffe empfiehlt auf einer Grafikkarte zu arbeiten, muss auch der Rechner entsprechende Voraussetzungen erfüllen. Auf diesem müssen auch passende Treiber installiert werden.

Ist alles eingerichtet und der Datensatz entsprechend aufbereitet, so können die Hyperparameter gesetzt werden. Besonders wichtig ist der Parameter, der die maximale Anzahl an Layern festlegt. Besteht die Architektur aus vielen Layern, so wird auch mehr Grafikspeicher zum Trainieren benötigt. Sollte der Speicher nicht ausreichen, wird der Trainingsvorgang für das Netz schlichtweg abgebrochen und es wird mit einem neuen weitergearbeitet. Weiterhin sollte man darauf achten, wie viel Speicher die Festplatte hat, da jedes CNN mit seinen Gewichten abgespeichert wird. Durch den ε-Greedy Ansatz wird pro ε eine Menge an neuen Architekturen generiert. Diese Menge wird vorher festgelegt, sodass man von Anfang an weiß, wie viele verschiedene Architekturen gefunden werden. Dabei sollte man die Größe des Suchraumes beachten. Diese ist abhängig von der Anzahl der maximalen Layer in der Architektur, sowie von weiteren Parametern. Man kann nämlich festlegen, wie viele Filter ein convolutional Layer haben kann, wie groß diese sind, oder auch wie groß ein pooling Layer ist und welchen Stride er haben kann. Diese Parameter beeinflussen die Größe des Suchraumes. Je mehr verschiedene Varianten eines Layers es geben kann und je höher die Zahl der maximal erlaubten Layer ist, umso größer ist der Suchraum, in welchem MetaQNN Architekturen sucht.

Eine Auswahl an wichtigen Parametern ist in der Durchführung (Tabelle 5 & 6) zu finden. Sind nun auch alle Parameter gesetzt, so kann man den Vorgang starten. Dabei startet man eine Serverumgebung und einen oder mehrere Clienten. Denn MetaQNN ist auch dafür ausgelegt, auf mehreren Rechnern parallel zu arbeiten, sodass der NAS Vorgang auf diese Weise beschleunigt werden kann, indem mehrere Netze gleichzeitig trainiert werden. Die Serverumgebung kümmert sich um die Architektursuche und stellt den Clienten jeweils die neue Architektur zum Trainieren bereit. Die Clienten können mit dem Server über ein Netzwerk kommunizieren. Ist ein Client frei, wird ihm eine neue Architektur zugeordnet. Das Netz wird beim Clienten trainiert und anschließend wird dem Server die Genauigkeit zurückgeliefert. Dies wird so lange durchgeführt, bis die festgelegte Anzahl an verschiedenen Architekturen gefunden wurde. Es ist keine Netzwerkeinrichtung erforderlich, sofern man auf einem Rechner sowohl Server als auch Client laufen lässt. Beim Starten des Client ist nur die Angabe über die IP des Servers notwendig. In dem Fall ist das einfach der lokale Rechner, also `127.0.0.1`.

Kompatibilität mit eigenen Datensätzen

In dem GitHub Repository wird eine Anleitung und ein Skript bereitgestellt, einen Datensatz in ein *LMDB* Format zu bringen. Das LMDB Format ist das Format, welches üblicherweise für Caffe verwendet wird und steht für Lightning Memory – Mapped Database. Die Daten werden dabei als key-value Datenbank organisiert und sollen dadurch zu einer schnellen I/O Performanz sorgen.
Problematisch am gestellten Skript ist, dass dieses nur für die von den Autoren ausgewählten Datensätze wie MNIST, SVHN oder CIFAR10/CIFAR100 funktioniert. Dabei wird, falls nicht schon geschehen, der beim Starten des Skriptes gewählte Datensatz heruntergeladen und in das LMDB Format gebracht, ansonsten wird der dynamische Pfad für den Datensatz genutzt, worin dieser heruntergeladen wurde. Das Skript lässt sich an dieser Stelle leicht erweitern, sodass ein neuer, eigener Datensatz beim Starten des Skriptes gewählt werden kann und so in das LMDB Format gebracht wird. Ein von den Autoren gewählter Datensatz ist so aufgebaut, dass die Bilder, die zu einer Klasse gehören in einem Ordner abgespeichert sind. Somit gibt die Ordneranzahl die Anzahl der Klassen an. Die Implementierung zur Überführung des Datensatzes in das LMDB Format wurde abgewandelt, sodass man seinen eigenen

Datensatz in solch eine Ordnerstruktur bringt. Das Skript wird dann zwei Mal ausgeführt, einmal für die Trainingsdaten und einmal für die Testdaten.
Der hierfür verwendete NaoTh Balldatensatz 2016, wurde für den Trainingsdatensatz auf 3333 Bilder je Klasse unterteilt, sowie auf 500 Bilder je Klasse für den Testdatensatz.

Durchführung

Die Hyperparameter werden durch zwei Dateien bestimmt. Zum einen gibt es eine Datei *hyper_ parameters.py*, in der die Hyperparameter zum Datensatz zu setzen sind, zum anderen müssen in der *state_ space_ parameters.py* die Hyperparameter für den NAS Vorgang gesetzt werden. Die Parameter für den Datensatz wurden von den Paramtern für MNIST übernommen und angepasst. Nachfolgend in Tabelle 5 ist eine Auflistung der relevanten Hyperparameter für den Datensatz zu sehen:

Parameter	Wert
Klassen	2
Bildauflösung	16
Anzahl an Trainingsdaten	2000
Batchgröße Trainingsdaten	32
Anzahl an Testdaten	1000
Batchgröße Testdaten	32
Epochen	10

Tab. 5: Hyperparameter zu den Trainingsdaten auf dem NaoTH Balldatensatz 2016 zur Durchführung von MetaQNN. Es fällt auf, dass nicht alle Trainingsdaten verwendet werden. Diese Entscheidung wird in der Auswertung begründet.

Zudem muss man noch auf die Pfade des Datensatzes verweisen, sowie zum Pfad vom Caffe Root-Verzeichnis. Dann werden noch die Parameter für das NAS Verfahren wie in Tabelle 6 gesetzt:

Parameter	Wert
Klassen	2
Bildauflösung	16
Max. Anzahl an Layern	6
Mögliche Filteranzahl in conv. Layern	[8, 16, 32, 64]
Mögliche Filtergrößen in conv. Layern	[1,3,5]
Mögliche pooling Größen und Strides	[(5,3), (3,2), (2,2)]
Maximale Anzahl an Fully-Connected Layern	2 (+1 am Ende)
Mögliche Fully-Connected Neuronen	[128, 256, 512]
Initiales Pooling	✗
Aufeinanderfolgendes Pooling	✗
Convolutional Layer Padding	✓
ε-Greedy:	
ε-Werte: Exemplare jeweils	1.0: 1000
	0.9, 0,8 0,7: 100
	0.6, 0.5, 0.4, 0.3, 0.2, 0.1: 150
Q-Learning:	
Lernrate α	0.01
Discount Faktor γ	1.0

Tab. 6: Hyperparameter zum NAS Verfahren von MetaQNN auf dem NaoTH Balldatensatz 2016.

Der Rechner, auf welchem MetaQNN ausgeführt wurde, hat folgende Spezifikationen:

• Rechnertyp:	Custom
• Betriebssystem:	Linux Ubuntu 16.04 LTS
• Prozessor:	AMD Ryzen 5 1600 mit 12 Threads auf 6 Kernen à 3,2 GHz
• Grafikkarte:	Nvidia GeForce GTX 1060 6GB
• Arbeitsspeicher:	16 GB

Leider ist der pooling Layer nur nach dem letzten convolutional Layer einsatzbereit. Also erfolgt vor der eigentlichen Klassifikation noch ein Pooling oder eben nicht.
Mit den gewählten Layer Parametern beinhaltet der Suchraum bei maximal 6 Layern und der Option am Ende zusätzlich noch einen von drei pooling Layer zu haben grob $4 \cdot \sum_{i=1}^{6} 12^i \sim 13M$ verschiedene Architekturen. Durch die Installation von Caffe ist es möglich, die Netze auf der Grafikkarte zu trainieren. Da nur eine Grafikkarte vorhanden ist, wird hier auch nur ein Netz gleichzeitig trainiert. Dazu wird die Serverdatei ausgeführt und ein Client, an welchen die Netze zum Trainieren geschickt werden. Im Hintergrund werden in einem ausgewählten Verzeichnis die trainierten Netze und dazugehörige Log Dateien gespeichert. Zudem wird noch eine Datei über die Q-Values angelegt – die Q-Tabelle. Darin ist je Layer neben den Q-Values enthalten welcher der vorhergehende Layer war, dazu wie viele Feature Maps eingegangen sind, wie groß die Filter waren und wie groß das entsprechende Bild als Input für den aktuellen Layer war. Darüber hinaus auch entsprechend die Angaben zum Output des aktuellen Layers, sowie die Information, ob es sich bei dem Layer um einen terminierenden Layer handelt. Gleichzeitig wird wie vorher schon beschrieben ein Gedächtnis aufgebaut. Eine Datenbank in der bereits generierte Architekturen mitsamt ihrer Genauigkeit gesammelt werden, sodass zum aktualisieren der Q-Values das gleiche Netz nicht neu trainiert werden muss.
Durch die festgelegten Parameter für ε werden letztendlich 2200 verschiedene Architekturen erzeugt. Zunächst wird mit einem ε-Wert von 1.0 begonnen, um damit eine rein explorative Phase einzuleiten.

Dabei werden bereits 1000 unterschiedliche Architekturen erzeugt. Somit wird die Q-Tabelle gefüttert und mit dem nächsten ε-Wert werden dann zu einer Wahrscheinlichkeit von 90% die Regeln der Q-Tabelle befolgt, und zu 10% wird davon abgewichen. Dies wird so lange fortgesetzt, bis die 150. neue Architektur bei $\varepsilon = 0.1$ gefunden wurde. Anschließend terminiert das Programm und der Server erzeugt keine weiteren Architekturen. Ein $\varepsilon = 0$ würde nur dazu führen, dass die Regeln der Q-Tabelle befolgt werden und kein stochastischer Einfluss auf die Q-Tabelle einwirkt.
Auf Serverseite wird parallel protokolliert, welches Netz erstellt wurde. Dieses wird durch einen String wie den nachfolgenden getan:

```
[C(64,3,1), C(16,5,1), D(1,2), C(16,1,1), C(8,5,1), D(2,2), GAP(2), SM(2)]
```

Dabei steht z.B. `C(64,3,1)` für einen convolutional Layer mit 64 Filtern der größe 3 und einem Stride von 1, `D(1,2)` für den ersten Dropout von 2, `GAP(2)` für ein Global Average Pooling und `SM(2)` für einen Soft-Max Layer mit 2 Klassen. Die Bedeutung des Wertes beim Pooling ist leider nicht klar, jedoch ist aufgefallen, dass der Wert der Anzahl der Klassen entspricht. Die erstellte Architektur wird dann an einen laufenden Clienten gegeben. Serverseitig wird dazu ausgegeben, an welchen Client die Architektur geschickt wurde. Bei dem Client erscheint dann die Nachricht, dass die entsprechende Architektur vom Server empfangen wurde. Nachdem sie trainiert und mit dem Testdatensatz ausgewertet wurde, wird die ermittelte Genauigkeit an den Server zurückgeschickt. Dieser zeigt nochmals die Architektur als String an und dazu die empfangene Genauigkeit. Dann wird die nächste Architektur erzeugt.

Auswertung & Zwischenfazit

Wie in der Durchführung angesprochen, wurde nicht der komplette Datensatz verwendet. Dies liegt an der Art und Weise wie MetaQNN arbeitet. Da das Caffe Framework verwendet wird, wird relativ viel Grafikspeicher beim Trainingsvorgang benötigt. Sollte beim Trainingsvorgang festgestellt werden, dass nicht genug Grafikspeicher zur Verfügung steht, so wird das Netz verworfen und auf ein neues gewartet. Dies schränkt natürlich die Tiefe der Architektur ein, sowie die Größe des Trainingsdatensatzes und die verwendeten Batchgrößen, da diese Daten ebenfalls in den Grafikspeicher geladen werden. So handelt es sich hier um einen Trade-Off zwischen der Architekturtiefe und der Größe der Trainingsdaten. Sollte ein tiefes Netz vom Server generiert werden, so wird beim Trainieren mehr Speicher benötigt und entsprechend gibt es weniger Speicher für die Daten. Andersherum, sollte man viele Daten nehmen, so muss man auf z.B. 1 bis 2 Layer verzichten und hätte nach oben hin weniger tiefe Netze oder man muss die Anzahl der Filter in einem convolutional Layer gering halten. Diese Einschränkung würde zu einem kleineren Suchraum resultieren, welcher wiederum groß sein sollte. Vor allem wegen der Implementierung des ε-Greey Ansatzes mit der Bedingung, dass bei jedem ε nur neue Architekturen gefunden werden sollen, ist es notwendig, dass der Suchraum groß ist. Wird ε nachher klein, so dauert es länger eine neue Architektur zu finden, da mehr exploitet wird, und es daher selten auftritt, dass man abweicht. Und falls doch, so soll die Abweichung zu einer neuen Architektur resultieren, die auch nicht bei den vorigen ε gefunden wurde. Dies wird bei geringem Suchraum und bei 2200 gewünschten Architekturen nur langsam geschehen, denn der Server würde in den meisten Fällen Architekturen generieren, die bereits bekannt sind. Wird die Anzahl der verschiedenen Architekturen gering gehalten, besteht die Gefahr, dass man nicht die guten Architekturen findet. So steht man bei MetaQNN vor einer wichtigen Herausforderung, wenn man die Hyperparameter setzt. Nichtsdestotrotz ist dies nicht unmöglich, denn nach mehreren Testdurchläufen von einigen Architekturen und dem Beobachten des Grafikspeichers, kann man die Hyperparameter von Hand optimieren, bis es selten vorkommt, dass mal zu wenig Speicher besteht und der Suchraum dennoch groß bleibt.

Die Einrichtung des Caffe Frameworks stellt auch eine Herausforderung dar. Die Installation der Treiber erscheint zunächst trivial, jedoch funktioniert die verwendete Caffe Version aus der Anleitung der Autoren nicht mit jeder Version der Treiber. Man gelangt in einen Durchlauf von Neuinstallationen dieser und ständigen Reboots des Rechners. Zudem gehört die Einbindung von Caffe in den Pythonpath, damit Python Caffe als Bibliothek erkennt und damit arbeiten kann, sofern die Treiber korrekt installiert wurden. Die Installation weiterer Python Bibliotheken stellt hingegen kein Problem dar, jedoch ist die alleinige Einrichtung von Caffe mühevoll.
Die Erstellung des Datensatzes erfolgt nach Anleitung. An dieser Stelle verlief alles unkompliziert ohne weitere Schwierigkeiten.

Der komplette Durchlauf beanspruchte einen Zeitraum von 6d11h6m42s ohne Unterbrechung. Dies unterstreicht nochmals wie wichtig die Auswahl der Hyperparameter ist. Stellt man nämlich später einen Fehler fest, so kostet es wieder viel Zeit, diesen Vorgang von vorn laufen zu lassen. In diesem Zeitraum wurden Serverseitig insgesamt 35503 Architekturen generiert, bis das Ziel von 2200 individuellen erreicht wurde. Dabei wurden ~150GB der Festplatte in Anspruch genommen. Der hohe Speicherbedarf kommt dadurch zustande, dass die trainierten Netze samt Gewichten zur Wiederverwendung abgespeichert werden. Es fand weiter kein Ranking aller Architekturen statt, sodass dies manuell ausgewertet werden muss. Dazu sind alle Architekturen in Abbildung 23 sortiert nach ihrem Ranking zu sehen. Dabei haben etwa 24% der Architekturen eine Genauigkeit von $\geq 90\%$. Auffällig ist, dass über ein Drittel der Architekturen eine Genauigkeit $\leq 60\%$ haben. Die besten 5 Architekturen sind nachfolgend aufgelistet. Dabei hat sich bei Anwenden von MetaQNN ergeben, dass es am sinnvollsten ist, wenn die Architektur aus zwei aufeinander folgenden convolutional Layern besteht:

- Genauigkeit: 99.5968%, Netz: `[C(8,1,1), C(64,3,1), D(1,1), SM(2)]`
- Genauigkeit: 99.3952%, Netz: `[C(64,5,1), C(64,1,1), D(1,1), SM(2)]`
- Genauigkeit: 99.2944%, Netz: `[C(16,5,1), C(32,3,1), D(1,1), SM(2)]`
- Genauigkeit: 99.1935%, Netz: `[C(16,5,1), C(32,3,1), D(1,1), SM(2)]`
- Genauigkeit: 99.1935%, Netz: `[C(16,3,1), C(64,1,1), D(1,1), SM(2)]`

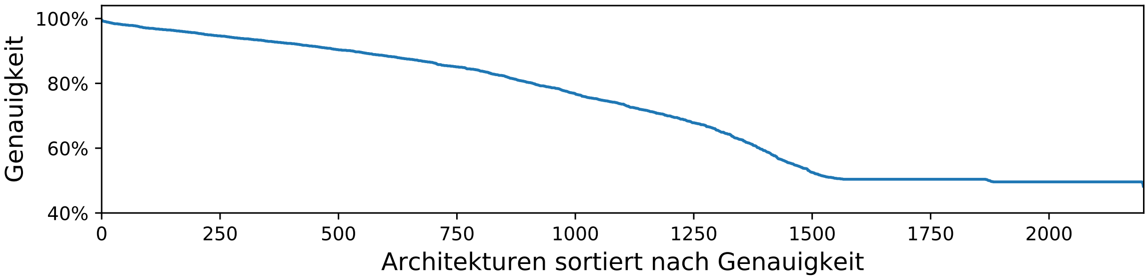


Abb. 23: Alle Architekturen sortiert nach ihrer Genauigkeit.

In Abbildung 24 hat man einen Überblick, wie bei kleiner werdendem ε die Anzahl der generierten Architekturen steigt, da immer schwieriger eine neue gefunden wird, denn es kommt seltener zu einer Abweichung in der Q-Tabelle. Abbildung 25 soll zeigen, wie Q-Learning in Kombination mit dem ε-Greedy Ansatz sich bemüht immer bessere Architekturen zu finden (orange und grüne Linie) und außerdem ist zu sehen, wie in der Gesamtheit der generierten Architekturen je ε (also mit Duplikaten) der Durchschnitt der Genauigkeiten steigt (blaue Linie). Da durch Q-Learning beim Exploiten die beste Wahl zur Steigerung der Genauigkeit getroffen wird, ist mit Betrachtung der Duplikate zu sehen, dass der Q-Learning Ansatz richtig arbeitet und stets gute Architekturen aus der Vergangenheit in Betracht zieht und diese durch Exploration ggf. abwandelt, um ggf. eine noch bessere zu finden. Bei $\varepsilon = 1.0$ sind die orange und grüne Linie gleich, da zu Beginn genauso viele verschiedene wie neue Architekturen generiert werden. Einige werden schon dabei mehrmals generiert, weshalb die blaue Linie um 5% höher liegt – dabei handelt es sich um Architekturen, die wenige Layer besitzen, die jedoch in ihrer Genauigkeit scheinbar besser liegen, als der Durchschnitt aller verschiedenen.

Die Sprünge nach unten bei $\varepsilon = 0.7$ und $\varepsilon = 0.2$ sind nicht ganz klar. Da diese Sprünge beim Durchschnitt aller generierten Architekturen nicht auftreten, ist anzunehmen, dass die Q-Tabelle bis dahin richtig geführt wurde. Es kann schlicht am Zufall liegen, dass die Architekturen sich einfach nicht eignen. Bei beiden ε hatten etwa 30% der neuen Architekturen eine Genauigkeit von unter 60%. Bei allen verschiedenen sind es bei $\varepsilon = 0.7$ ebenfalls etwa 30%, bei $\varepsilon = 0.7$ lediglich etwa 20% der Architekturen, die eine Genauigkeit von unter 60% haben. Diese Tatsache hat den Schnitt stark nach unten gezogen, jedoch findet sonst bei allen anderen ε stets eine Verbesserung statt, bis auf $\varepsilon = 0.5$. Dort verläuft der Durchschnitt nahe dem des Vorgängers.

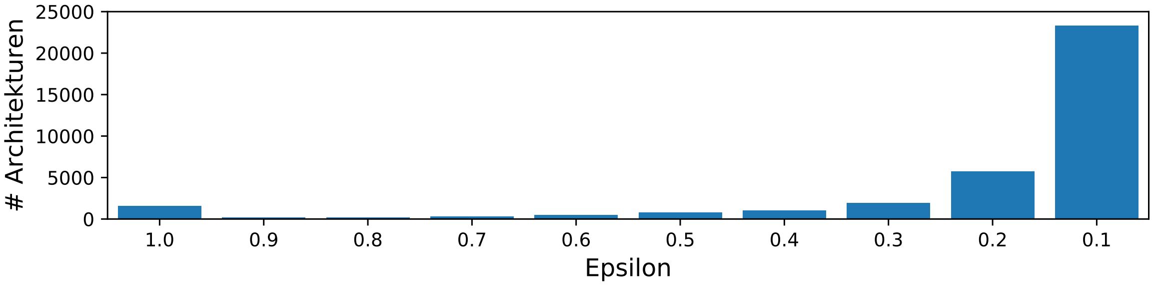


Abb. 24: Generierte Architekturen je ε bis alle neu zu entdeckenden Architekturen gefunden wurden.

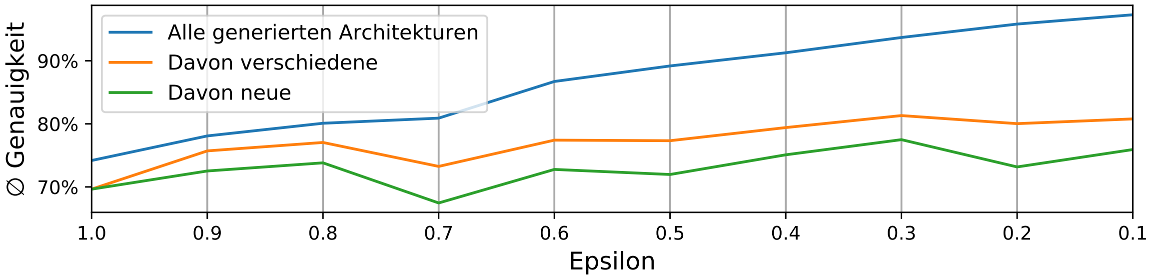


Abb. 25: Qualität des Verfahrens. In blau wird die Gesamtheit aller erstellten Architekturen je ε, also inkl. der generierten Duplikate, betrachtet. Die darin enthaltenen Unikate sind in orange dargestellt und in grün sind die neuen Architekturen, die in vorigen ε-Schritten nicht entdeckt wurden.

Letztendlich ist die einzige erwähnenswerte Problematik bei MetaQNN die, dass die Einrichtung mit sehr viel Zeit und Geduld verbunden ist. Abgesehen davon hat dieses Verfahren aber gezeigt, dass sich die Anwendung von Q-Learning gut für NAS eignet, denn dabei wird gezielt versucht die Architektur so aufzubauen, dass die Genauigkeit steigt. Fraglich ist die Kombination mit der Art und Weise wie ε-Greedy genutzt wird. Natürlich ist eine zufällige Abweichung von der Q-Tabelle notwendig, um neue Architekturen zu finden, jedoch findet sich keine weitere Forschung dafür, wie viele neue Architekturen je ε mindestens erforderlich oder zu viel sind. Man sieht in Abbildung 24 wie für $\varepsilon = 0.1$ erst knapp 23.000 Architekturen generiert werden mussten, damit 150 neue Architekturen gefunden werden. Dabei kommt man je neue Architektur auf etwa 150 bereits gefundene Architekturen, was im Vergleich zu den anderen ε sehr viel erscheint.
Zusammenfassend lässt sich sagen, dass MetaQNN in der Lage ist, die menschliche Expertise zu einem großen Teil zu übernehmen. Der Suchraum ist ausreichend groß, sodass die Möglichkeit geboten ist, zu Beginn viele verschiedene Architekturen zu finden. Aus diesen wird eine Erfahrung gewonnen. Das Verfahren ist dann weiter in der Lage diese Erfahrung zu nutzen, um mit Hilfe dieser weitere Architekturen zu erstellen und diese zu variieren, sodass ggf. eine bessere Architektur herauskommt.

4.4 Genetic CNN

Neben der Publikation wurde kein Code von den Forschern veröffentlicht. Jedoch findet sich auf GitHub ein Repository, welches diesen Ansatz speziell für den MNIST Datensatz versucht hat zu implementieren. [43] Es fanden sich einige Fehler sowie fehlende Funktionalitäten. Nachfolgend werden die bereits implementierten Funktionalitäten sowie Änderungen/Erweiterungen tabellarisch festgehalten:

	Beschreibung	Vorher	Nachher
	Datensatz:		
1.	Mögliche/r Datensatz/-sätze	MNIST	Variabel
	Architektur:		
2.	Convolutional Layer	Fehlerhaft	✓
3.	Pooling Layer	Fehlerhaft	✓
4.	Fully-connected Layer	✗	✓
5.	Softmax Layer am Ende	✓	✓
6.	Filteranzahl der convolutional Layer	1	Variabel
6.1	Gleiche Filteranzahl der convolutional Layer pro Stage	(✓)	✓
7	Filtergröße der convolutional Layer pro Stage	5x5	Variabel
8.	Korrekte Ein- und Ausgabedimension der convolutional Layer pro Stage	✗	✓
	Gentischer Algorithmus:		
9.	Korrekte Indizierung der Gene	✗	✓
10.	Korrektes Crossover	✗	✓
11.	Korrekte Mutation	✗	✓
12.	Korrekte Fitnessfunktion	✓	✓
13.	Korrekte Selektion	✓	✓
14.	Korrekte Wahrscheinlichkeiten für p_c, q_c, p_m, q_m	Falsch/ Fehlend	✓

Tab. 7: Übersicht über Funktionalitäten des Programms zu GeneticCNN, sowie eine Gegenüberstellung zum Zustand, wie er vor und nach der Fehlerbereinigung und Hinzufügen von Erweiterungen war.

Nach Fehlerbeseitigung und einigen Erweiterungen war es dann möglich mit dem Ansatz zu arbeiten.

Einrichtung

Da der Ansatz ebenfalls in der Programmiersprache *Python* geschrieben wurde, ist der Nutzer angewiesen seine Umgebung dafür einzurichten. Dafür kann mit der Python Version 3.6.5 gearbeitet werden. Nachdem die benötigten Bibliotheken installiert wurden, müssen einige Hyperparameter eingestellt sowie der Datensatz in das IDX Format gebracht werden. Die Hyperparameter werden aus einer separaten Datei eingelesen und zur Erstellung des Datensatzes in das IDX Format wird ein Skript bereitgestellt.

Die Architektur wird so aufgebaut, dass jeweils die Stages aufeinanderfolgen und zwischen den Stages eine Pooling-Operation stattfindet und auf die letzte Stage ein Fully-Connected Layer folgt. In den Hyperparametern wird unter anderem die Anzahl der Stages gesetzt. Dabei gilt es zu beachten, wie oft ein Downsampling in Frage kommt, da zwischen zwei Stages eine Pooling-Operation stattfindet, die ein Downsampling durchführt, sodass die Ausgabeauflösung halb so groß ist, wie die bei der Eingabe. Ausgehend von dem *NaoTH Balldatensatz 2016* kommt man bei einer Bilderauflösung von 16x16 Pixeln nach fünf Stages zu einer Auflösung von 1x1 Pixeln. Daher wären vier Stages sinnvoll, denn man erhält so eine Auflösung von 2x2 Pixeln, die der fully-connected Layer übernimmt. Die Anzahl der Knoten pro Stage und die Filteranzahl der convolutional Layer je Stage bleibt dann dem Nutzer überlassen. Die Trai-

ningsdauer eines Netzes würde sich entsprechend den Werten verändern, wenn dadurch mehr/weniger Parameter trainiert werden müssen.
Man muss weiterhin festlegen über wie viele Generationen der genetische Algorithmus arbeiten soll. Laut der Veröffentlichung zu Genetic CNN [30] wird am Ergebnis des Durchlaufs des *CIFAR10* Datensatzes erklärt, wie bei einer Startpopulation der Größe 20 ab etwa 30 Generationen keine große Veränderung in der Qualität der Netze besteht. So kann man diese Werte als Referenz nehmen. Entsprechend genauso kann man mit den Wahrscheinlichkeiten zum Crossover und zur Mutation umgehen und diese auch übernehmen.

Kompatibilität mit eigenen Datensätzen

Der implementierte Ansatz von Genetic CNN arbeitet mit dem von LeCun eingeführten IDX Dateiformat für die Trainings- und Testdaten und deren Labels. Es handelt sich dabei um ein einfaches Format, welches zur Handhabung von Vektoren und Matrizen entworfen wurde. [31] Um mit einem eigenen Datensatz arbeiten zu können, muss der Bilderdatensatz in Trainings- und Testdaten geteilt werden und wird anschließend durch das gestellte Skript in das IDX Format gebracht. Der umgesetzte Ansatz von Genetic CNN funktioniert nur mit quadratischen Bildern. Die Auflösung der Bilder, sowie die Anzahl der Klassen muss dann noch in der separaten Datei mit den Hyperparametern gesetzt werden.

Der genetische Algorithmus mit Auflistung der benötigten Bibliotheken, sowie das Skript zur Erstellung des Datensatzes im IDX Format werden im hierfür angelegtem GitHub Repository bereitgestellt. [35]

Durchführug

Sind die Hyperparameter gesetzt, wie auch der Datensatz erstellt, bedarf es nur des Ausführens des Python Skriptes in der Kommandozeile. Weitere Eingaben sind nicht erforderlich. Es wird auf einem Institutsrechner der Humboldt-Universität zu Berlin (Rechnername: greuanu5 – Stand Juli/2019) gearbeitet. Dieser ist wie folgt aufgebaut:

• Rechnertyp:	Dell R920
• Betriebssystem:	Linux SuSE Leap 15
• Prozessor:	4x Intel Xeon E7-4880 v2 mit je Prozessor 30 Threads auf 15 Kernen à 2,5 GHz
• Arbeitsspeicher:	1024 GB

Dieser Rechner arbeitet ohne Grafikkarten, sodass die Netze nacheinander stets auf den CPUs trainiert werden.

Zum *NaoTH Balldatensatz 2016* wurden folgende Hyperparameter gesetzt:

Parameter	Wert
Klassen	2
Bildauflösung	16
Validierungsdatensatz	0
Größe Trainingsdatensatz	4000
Größe Testdatensatz	2000
Fully-Connected Größe	1000
Stages	[s1,s2,s3,s4]
Knoten je Stage	[3,3,4,5]
Filter der Nodes je Stage	[16,16,32,32]
Filtergrößen der Nodes je Stage	[5,3,3,2]
Epochen	20
Batchgröße	50
Populationsgröße	20
Generationen	30
p_c, q_c, p_m, q_m	0.2, 0.3, 0.8, 0.1

Tab. 8: Hyperparameter zur Durchführung von Genetic CNN auf dem NaoTH Balldatensatz 2016

Diese Konfiguration bietet $2^3 \cdot 2^3 \cdot 2^6 \cdot 2^{10} = 2^{22} =\sim 4M$ mögliche Bitstrings zur Kodierung der Individuen.

Wird das NAS-Verfahren gestartet, so wird für jedes Netz nacheinander die Kodierung der Stages als eine Sequenz in der Konsole ausgegeben. Im Hintergrund wird das Netz entsprechend dieser Kodierung erstellt und trainiert. Danach erfolgt die Berechnung des Fitness Scores, wobei es sich hier um die Genauigkeit des Netztes auf den Testdaten handelt. Dieser wird dann auch ausgegeben. Darauf folgt die nächste Kodierung und somit das nächste Netz.

Dies wird so lange fortgesetzt, bis alle Individuen der Generation bewertet wurden. Nach dem Durchgang einer Generation, wird auch die Größe der darin neu evaluierten Individuen ausgegeben. Darauf wird die Selektion durchgeführt, gefolgt vom Crossover und der Mutation.

Sind 31 Generationen (inklusive Startpopulation) durchgelaufen, erfolgt ein Ranking aller Individuen über ihre Fitness Scores.

Als Resultat erhält man also eine Auflistung aller Individuen aller Generationen, sortiert nach Genauigkeit. Wie auch zu erwarten ist, sind einige Individuen mehrmals vertreten. Die Gesamtzahl aller Individuen bei den in Tabelle 8 gesetzten Hyperparametern beträgt 532. Die besten fünf Architekturen, die durch Anwenden von Genetic CNN erstellt wurden, sind nachfolgend aufgelistet:

- Fitness Score: 99.65%
  `[0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1]`
- Fitness Score: 99.15%
  `[0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1]`
- Fitness Score: 99.15%
  `[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0]`
- Fitness Score: 99.15%
  `[1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1]`
- Fitness Score: 99.1%
  `[0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0]`

Die beste Architektur wird nachfolgend in Abbildung 26 dargestellt:

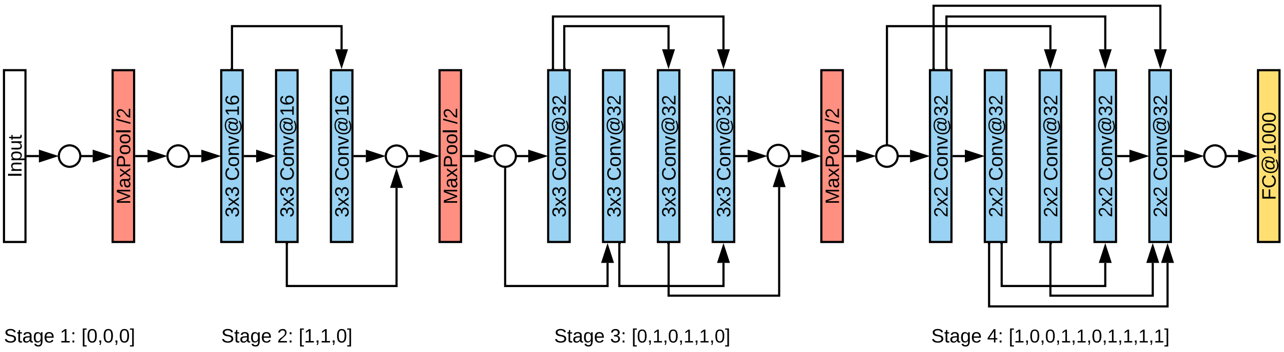


Abb. 26: Die beste Architektur, die auf dem *NaoTH Balldatensatz 2016* durch Genetic CNN erstellt wurde. Da die erste Stage keine Kanten hat, gibt es sogar ein initiales Pooling.

Auswertung & Zwischenfazit

Nachdem der nachgeahmte Ansatz nachgebessert wurde, ist kein größerer Einsatz erforderlich, um das Programm laufen zu lassen. Wie sich auch gezeigt hat, ist es ein Leichtes mit einem eigenen Datensatz Architekturen zu trainieren. Aus den ausgegebenen Daten in der Konsole ließ sich bei den 532 Architekturen eine Gesamttrainingsdauer von 8h59m17s berechnen, was einem Mittel von fast genau 1 Minute entspricht. Unter den 532 Architekturen befinden sich 477 Unikate, sodass es sich anbietet für die anderen mehrfach vorkommenden Individuen zu schauen, wie sehr das Ranking zwischen den Duplikaten variiert, und wie sehr sich Duplikate in ihrer Genauigkeit unterscheiden, um zu untersuchen, wie aussagekräftig ein Ergebnis ist. Aus den Unikaten gab es 49 Individuen, die mehrfach vorkamen. 45 davon kamen 2 mal vor, 2 kamen 3 mal vor und weitere 2 kamen 4 mal vor. Abbildung 28 zeigt ein Streudiagramm. Es zeigt das Verhältnis zwei gleicher Individuen bzgl. der Differenz ihrer Genauigkeit und der Differenz ihres Rankings. Man erkennt, wie ein linearer Zusammenhang zwischen den Duplikaten besteht, jedoch sieht man, dass auch Differenzen im Ranking von $>$100 existieren. Dies lässt schließen, dass für zwei Duplikate die Fitness Scores stärker voneinander abweichen können, jedoch liegt diese Abweichung im Durchschnitt bei 2.79%. Ein großer Unterschied im Ranking lässt sich mit Hilfe von Abbildung 27 erklären. Dort ist nämlich zu sehen, dass ein Großteil aller Individuen einen Fitness Score um 96% hat. Da würde eine geringe Abweichung zu einem großen Unterschied im Ranking führen. Eine Abweichung von 2.79% liefert im schlimmsten Fall eine Differenz im Ranking von 233. Dies wird wohl auch der Grund gewesen sein, dass zur Selektion ein Roulette Verfahren genutzt wurde, und nicht beispielsweise dass nur die top-n Individuen überleben, da sonst wegen der Abweichung evtl. bessere Individuen verworfen werden würden, falls sie zufällig um 2% abweichen und somit fälschlicher Weise schlechter gerankt wurden.

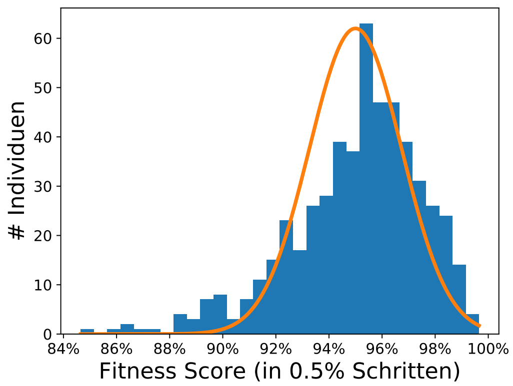


Abb. 27: Übersicht der Fitness Scores aller Individuen. Diese wurden Buckets zugeordnet, die jeweils einen Bereich von 0.5% umfassen.

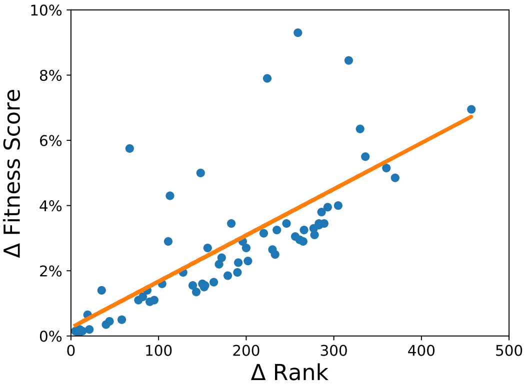


Abb. 28: Verhältnis von Duplikaten bzgl. der Differenz ihrer Fitness Scores und der Differenz ihres Rankings.

Ein Kritikpunkt an diesem NAS Vorgang ist, dass die Architektur dennoch teils von selbst entworfen werden muss. Dies kann man darin sehen, dass z.B. bei der Implementierung gezwungenermaßen ein pooling Layer auf eine Stage folgt. Es wird auch nicht die Hyperparameterzahl der Layer je Stage variiert (Filteranzahl, Filtergrößen, etc.), diese werden als Hyperparameter vor Beginn einmal festgelegt und so behalten, sodass beispielsweise eine Stage stets die gleiche Filteranzahl hat. Ein Entwickler einer Architektur muss also im Vornherein die Architektur vorgeben, der Beitrag ist einzig und allein die Möglichkeit convolutional Layer in seiner Architektur durch Stages zu ersetzen und damit ein tieferes Netz zu erstellen.

Natürlich ist diese Einschränkung damit zu begründen, dass der genetische Algorithmus durch Variation der vorher festgelegten Hyperparameter komplexer wird und der Suchraum mit allen möglichen Architekturen dadurch sehr viel größer wäre. So müsste man die Kodierung erweitern und den Vorgang für mehrere Generationen und größere Populationen durchlaufen lassen, um ein gutes Ergebnis über möglichst viele verschiedene Architekturen zu bekommen. Dies ist mit mehr Rechenaufwand und längerer Dauer verbunden. Einem Entwickler bleiben hier nur die Verbindungen unter den convolutional Layern einer Stage erspart, was reine Optimierungsarbeit durch Probieren ist.

Eine weitere kleine positive Auffälligkeit gibt es aber in der Implementierung. Durch den genetischen Algorithmus, bzw. der Kodierung kann es vorkommen, dass z.B. eine Stage komplett durch Null-Bits definiert wird, somit also keine Convolution-Operationen darin vorkommen und so zwei pooling Layer aufeinander folgen bzw. ein initiales Pooling stattfinden kann. Durch einen solchen Zufall kann ein wenig aus der Norm ausgebrochen werden, sodass die Architektur nicht ein reiner Wechsel von Stages und pooling Layern ist. Dies sieht man sogar deutlich an den besten fünf Ergebnissen in der Durchführung. Darunter befinden sich vier, die ein initiales Pooling haben.

Insgesamt ist die Kodierung der Architektur ein gelungener Beitrag der Veröffentlichung [30] und bietet damit Anregungen zum Ausbau von Genetic CNN oder zur Implementierung in andere NAS Verfahren. Genetic CNN ist von selbst aus nicht in der Lage, die komplette Expertise eines Menschen zu übernehmen, um eine Architektur von alleine zu erstellen – vielmehr wird ein Muster der Architektur vorgegeben und es werden die convolutional Layer durch Stages ersetzt, die wiederum optimiert werden. Damit ist das Verfahren eine enorme Stütze, wenn es um die Erstellung von solchen tieferen Netzen geht, da die Stages allesamt einen großen Suchraum bereitstellen. Es wäre sonst eine mühevolle Arbeit sich selbst mit der Kantensetzung zwischen den convolutional Layern auseinander zu setzen, um die beste Anordnung zu finden. Verglichen mit MetaQNN entfällt hier außerdem die mühevolle Einrichtung des Programms, sowie die Dauer, bis das Verfahren beendet ist.

5 Diskussion & Fazit

Die Ergebnisse in den Veröffentlichungen der Autoren haben anhand verschiedenen Datensätzen gezeigt, dass die maschinell erstellten Architekturen konkurrenzfähig sind. Sie erzielen ggf. sogar bessere Ergebnisse, als von Menschen erstellte Architekturen. Die Ergebnisse der Experimente in dieser Bachelorarbeit deuten auch darauf hin, dass NAS eine gute Herangehensweise ist, wenn es um die Erstellung eines guten Netzes zu gegebenen Daten geht.

Zwar wird bei großen Datensätzen dafür sehr viel Zeit in Anspruch genommen, da viele Netze trainiert werden, jedoch kann dem Entwickler die Arbeit des Architekturdesigns und dessen Optimierung erleichtert oder gar übernommen werden. Dabei ist vor allem die Wahl des NAS Verfahrens wichtig, denn sie alle arbeiten mit verschiedenen Methoden. Letztlich ist es auch erfreulich, dass viele verschiedene gute Architekturen erstellt werden, denn der Entwickler kann an ihnen durch seine Expertise auch weitere Änderungen und ggf. Optimierungen vornehmen, die nicht durch das NAS Verfahren durchgeführt wurden.

Für unerfahrene Entwickler bieten aktuelle NAS Verfahren die Möglichkeit ein gutes Netz zu erstellen, ohne sich tief mit der Materie befassen zu müssen. So kann die erforderliche menschliche Expertise ersetzt werden. Will man jedoch das Beste aus dem Verfahren kriegen, so muss sich ein Entwickler in das Verfahren einarbeiten, um die Parameter geeignet setzen zu können.

In Abschnitt 2 wird klar, dass die weitere Entwicklung von CNNs immer noch durch die Expertise eines Menschen erfolgt. So ist nachvollziehbar, dass die NAS Verfahren nicht in der Lage sind, neue Arbeitsweisen von CNNs entwickeln zu können, da sie stets höchstens auf Basis des aktuellen Standes arbeiten können. Mit jedem Entwicklungsschritt sollten die NAS Verfahren in Zukunft aber auch immer bessere Ergebnisse liefern können.

Somit ist zu sagen, dass die NAS Methoden zur Erstellung guter Architekturen geeignet sind und die menschliche Expertise dadurch zu einem gewissen Grad ersetzt werden kann.

Literaturverzeichnis

[1] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[2] *ImageNet Large Scale Visual Recognition Competition 2012 (ILSVRC2012).* URL: `http://image-net.org/challenges/LSVRC/2012/results.html` (Besucht am 8.03.2019).

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back Propagating Errors”. In: *Nature* 323 (Oct. 1986), pp. 533–536. DOI: `10.1038/323533a0`. URL: `https://doi.org/10.1038/323533a0`.

[4] Matt Mazur. *A Step by Step Backpropagation Example.* URL: `https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/` (Besucht am 4.09.2019).

[5] Kunihiko Fukushima. “Cognitron: A self-organizing multilayered neural network”. In: *Biological Cybernetics* 20.3 (Sept. 1975), pp. 121–136. ISSN: 1432-0770. DOI: `10.1007/BF00342633`. URL: `https://doi.org/10.1007/BF00342633`.

[6] Daniel Graupe. *Principles of Artificial Neural Networks.* Advanced Series in Circuits and Systems, Volume 7. World Scientific Publishing Co., 2013. ISBN: 978-981-4522-73-1.

[7] Kunihiko Fukushima. “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In: *Biological Cybernetics* 36 (1980), pp. 193–202.

[8] Kunihiko Fukushima. “Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition”. In: *Neural Networks* 1 (1988), pp. 119–130.

[9] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[10] Hayaru Shouno. “Recent Studies Around the Neocognitron”. In: *Neural Information Processing.* Ed. by Masumi Ishikawa et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1061–1070. ISBN: 978-3-540-69158-7.

[11] Yann LeCun, Léon Bottou, and Yoshua Bengio. “Reading checks with multilayer graph transformer networks”. In: *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing.* Vol. 1. Apr. 1997, 151–154 vol.1. DOI: `10.1109/ICASSP.1997.599580`.

[12] *Yann LeCun’s Home Page.* URL: `http://yann.lecun.com/` (Besucht am 8.03.2019).

[13] Kunihiko Fukushima. “One-Shot Learning with Feedback for Multi-layered Convolutional Network”. In: *Artificial Neural Networks and Machine Learning – ICANN 2014.* Ed. by Stefan Wermter et al. Cham: Springer International Publishing, 2014, pp. 291–298. ISBN: 978-3-319-11179-7.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[15] *ImageNet Large Scale Visual Recognition Competition 2013 (ILSVRC2013).* URL: `http://www.image-net.org/challenges/LSVRC/2013/results.php` (Besucht am 8.03.2019).

[16] Matthew D. Zeiler and Rob Fergus. “Visualizing and Understanding Convolutional Networks”. In: *CoRR* abs/1311.2901 (2013). arXiv: `1311.2901`. URL: `http://arxiv.org/abs/1311.2901`.

[17] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: `1512.03385`. URL: `http://arxiv.org/abs/1512.03385`.

[18] *ImageNet Large Scale Visual Recognition Competition 2015 (ILSVRC2015).* URL: `http://image-net.org/challenges/LSVRC/2015/results` (Besucht am 31.08.2019).

[19] Paul Viola and Michael Jones. “Rapid Object Detection using a Boosted Cascade of Simple Features”. In: vol. 1. Feb. 2001, pp. I–511. ISBN: 0-7695-1272-0. DOI: `10.1109/CVPR.2001.990517`.

[20] Thomas Elsken, J. Hendrik Metzen, and F. Hutter. “Neural Architecture Search: A Survey”. In: *ArXiv e-prints* (Aug. 2018). arXiv: `1808.05377 [stat.ML]`. URL: `http://arxiv.org/abs/1808.05377`.

[21] Michael Wooldridge. “Intelligent Agents: The Key Concepts”. In: *Multi-Agent Systems and Applications II*. Ed. by Vladimír Mařík et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. Chap. 2, pp. 3–43. ISBN: 978-3-540-45982-8.

[22] Till A. Heilmann. “Es gibt keine Software. Noch immer nicht oder nicht mehr”. In: *Smartphone-Ästhetik. Zur Philosophie und Gestaltung mobiler Medien*. Ed. by Oliver Ruf. Bielefeld: Transcript, 2018, pp. 159–178. ISBN: 978-3-8376-3529-4.

[23] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 2nd. Cambridge, MA, USA: MIT Press, 2018. ISBN: 9780262039246.

[24] Han Cai et al. “Efficient Architecture Search by Network Transformation”. In: *CoRR* abs/1707.04873 (2017). arXiv: `1707.04873`. URL: `http://arxiv.org/abs/1707.04873`.

[25] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “Understanding the exploding gradient problem”. In: *CoRR* abs/1211.5063 (2012). arXiv: `1211.5063`. URL: `http://arxiv.org/abs/1211.5063`.

[26] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8.3 (May 1992), pp. 229–256. ISSN: 1573-0565. DOI: `10.1007/BF00992696`. URL: `https://doi.org/10.1007/BF00992696`.

[27] Bowen Baker et al. “Designing Neural Network Architectures using Reinforcement Learning”. In: *CoRR* abs/1611.02167 (2016). arXiv: `1611.02167`. URL: `http://arxiv.org/abs/1611.02167`.

[28] Christopher John Cornish Hellaby Watkins. “Learning from Delayed Rewards”. PhD thesis. Cambridge, UK: King’s College, May 1989. Chap. 7.3 & Appendix 1, pp. 95-96 & 220-228. URL: `https://www.cs.rhul.ac.uk/home/chrisw/new_thesis.pdf`.

[29] *Q-Learning Example*. URL: `http://mnemstudio.org/path-finding-q-learning-tutorial.htm` (Besucht am 7.09.2019).

[30] Lingxi Xie et al. “Genetic CNN”. In: *CoRR* abs/1703.01513 (2017). arXiv: `1703.01513`. URL: `http://arxiv.org/abs/1703.01513`.

[31] *THE MNIST DATABASE of handwritten digits*. URL: `http://yann.lecun.com/exdb/mnist/` (Besucht am 17.07.2019).

[32] *ImageNet Dataset*. URL: `http://image-net.org/` (Besucht am 17.07.2019).

[33] *The CIFAR-10 dataset*. URL: `https://www.cs.toronto.edu/~kriz/cifar.html` (Besucht am 17.07.2019).

[34] *The Street View House Numbers (SVHN) Dataset*. URL: `http://ufldl.stanford.edu/housenumbers/` (Besucht am 17.07.2019).

[35] aqibsaeed and DarkoNedic. *Genetic-CNN/Darko*. URL: `https://github.com/DarkoNedic/Genetic-CNN/tree/modified_for_BachelorThesis` (Besucht am 7.09.2019).

[36] han-cai. *EAS*. URL: `https://github.com/han-cai/EAS/tree/master/code` (Besucht am 4.09.2019).

[37] Ian J. Goodfellow et al. “Maxout Networks”. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML’13. Atlanta, GA, USA: JMLR.org, 2013, pp. III-1319–III-1327.

[38] Min Lin, Qiang Chen, and Shuicheng Yan. “Network In Network”. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014. URL: `http://arxiv.org/abs/1312.4400`.

[39] Jost Tobias Springenberg et al. “Striving for Simplicity: The All Convolutional Net”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. 2015. URL: `http://arxiv.org/abs/1412.6806`.

[40] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: `http://arxiv.org/abs/1409.1556`.

[41] bowenbaker. *metaqnn*. URL: `https://github.com/bowenbaker/metaqnn` (Besucht am 4.09.2019).

[42] *Caffe | Deep Learning Framework*. URL: `https://caffe.berkeleyvision.org/` (Besucht am 4.09.2019).

[43] aqibsaeed. *Genetic-CNN*. URL: `https://github.com/aqibsaeed/Genetic-CNN` (Besucht am 4.09.2019).

[44] Anibal Pedraza et al. “Automated Diatom Classification (Part B): A Deep Learning Approach”. In: *Applied Sciences* 7 (May 2017), p. 460. DOI: `10.3390/app7050460`.

[45] Bildquelle. *Gradient Descent*. URL: `https://slides.com/abdellahchkifa/indabaxmorocco19/#/2` (Besucht am 4.09.2019).

[46] Bildquelle. *Beispielnetz für Backpropagation*. URL: `https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/` (Besucht am 4.09.2019).

[47] Bildquelle. *Hierarchische Feature-Extraktion des Neocognitron*. URL: `https://de.slideshare.net/mentelibre/neocognitron` (Besucht am 4.09.2019).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 23. September 2019 ....