Universität Bremen

Lokalisierung von Rocaille-Ornamenten in Kunstbildern mithilfe von Convolutional Neural Networks

Autor:
Lars Nieradzik

Prüfer:
Prof. Dr. Zachmann
Dr. Felix Putze

12. Februar 2019

Inhaltsverzeichnis

1 Einleitung

In der vorliegenden Arbeit geht es um die Lokalisierung von *Rocailles* in Kunstbildern. Ein Rocaille-Ornament ist ein muschelförmiges Dekorationselement, was u. a. auf Möbelstücken zu finden ist. Es ist im 18. Jahrhundert in Frankreich entstanden.

Wenn Kunsthistoriker solche Rocailles analysieren möchten, müssen im ersten Schritt manuell die Linien eingezeichnet werden, aus welchen diese Ornamente bestehen. Es ist daher von Interesse das Einzeichnen zu automatisieren, da sich der Kunsthistoriker dann stärker auf die Analyse der Rocailles selbst fokussieren kann. Die folgende Grafik zeigt, wie eine solche Einzeichnung in einem Kunstbild aussehen kann:

Abbildung 1: Rocaille in einem Kunstbild (blau eingefärbt)

Da dieses manuelle Einzeichnen von Linien zeitintensiv ist, wurde ein Programm entwickelt, welches mithilfe eines neuronalen Netzwerks automatisch die Rocailles findet. Außerdem wurde auch ein einfaches Verfahren konzipiert, um die Kunstbilder für das Netzwerk zu präparieren. Rocailles können auch auf wirklichen Gegenständen wie Schränken vorkommen. Der Fokus liegt hier aber auf Kunstbildern und nicht auf Fotos von Objekten, die Rocailles enthalten. Die Kunstbilder, für welches das Programm geschrieben wurde, haben eine Größe von 2000×2000 bis 3000×3000 Pixeln und enthalten immer mehrere Ornamente. Außerdem liegen die Bilder als RGBA vor und bestehen stets aus Grautönen.

Die Arbeit ist in vier Teile gegliedert. Zunächst wird in der Theorie die Funktionsweise neuronaler Netzwerke vorgestellt. Es folgt dann im darauffolgenden Kapitel die Anwendung eines solchen neuronalen Netzwerks auf die Rocailles. Schließlich kommt dann die Auswertung der Leistung des neuronalen Netzwerks im Bezug zu verschiedenen Konfigurationen. Im Fazit werden die Ergebnisse noch einmal zusammengefasst und es wird ein Ausblick gegeben.

2 Neuronale Netzwerke

2.1 Feedforward

Ein neuronales Netzwerk (NN) ist ein nicht-lineares Regressions- oder Klassifikationsmodell, das beim Trainieren zwei Phasen durchläuft: *forward propagation* und *backpropagation* [34]. Die *forward propagation* leitet eine Eingabe x durch das Netzwerk, um eine Ausgabe $\hat{y}$ zu erhalten. Außerdem wird die Verlustfunktion $\mathcal{L}(y, \hat{y})$ berechnet, die den Vorhersagefehler angibt. Bei der *backpropagation* wird der Fehler $\mathcal{L}(y, \hat{y})$ rückwärts durch das Netzwerk geleitet, um die Gradienten zu berechnen. Dabei ist in der Regel die Eingabe eine Matrix oder ein Tensor, da immer mehrere Eingaben gleichzeitig betrachtet werden, um stabilere Gradienten zu erhalten [14]. Aufgrund der Übersichtlichkeit wird hier stattdessen von einem Vektor **a** ausgegangen, also lediglich einer Eingabe mit mehreren Features.

Ein Beispiel soll die Funktionsweise eines NN genauer erläutern. Die folgende Grafik zeigt ein NN mit einem *hidden layer* bestehend aus zwei Neuronen und einer Ein- und Ausgabe.

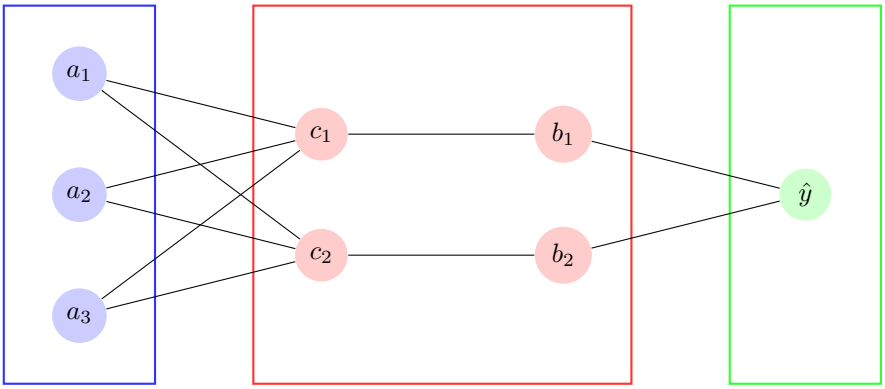


Abbildung 2: Beispiel eines NN (Eingabe blau, *hidden layer* rot, Ausgabe grün)

Bei der *forward propagation* wird zunächst die Eingabe **a**, die aus drei Features (Neuronen) besteht mit einer Gewichtsmatrix **C** multipliziert (Funktion l_1). Als Ergebnis erhält man den Vektor **c**, der dann durch eine Aktivierungsfunktion σ geleitet wird, um **b** zu produzieren. Üblicherweise wird als Aktivierungsfunktion die *rectified linear unit* $\sigma(\mathbf{x}) = \max(\mathbf{0}, \mathbf{x})$ gewählt. Die Aktivierungsfunktion wird u. a. genutzt um Nicht-Linearität zu erzeugen.

Zum Schluss wird **b** mit einem Gewichtsvektor **d** multipliziert, um die Vorhersage $\hat{y}$ zu erzeugen (Funktion l_2). Zur Übersichtlichkeit wurde im Beispiel auf das *bias*-Neuron verzichtet, welches den Abstand vom Ursprung angibt. Das *bias* wäre also zum Beispiel bei $a_1x_1 + a_2x_2 + b = 0$ das b.

Mathematisch lässt sich *forward propagation* als Funktionskomposition $(l_2 \circ \sigma \circ l_1)(\mathbf{a})$ darstellen, wobei $\sigma \circ l_1$ ein *hidden layer* und l_2 die Ausgabeschicht ist. Diese Funktionskomposition erzeugt $\hat{y}$. Wenn man auf σ verzichtet, ergibt sich ein lineares Modell, da lediglich eine lineare Kombination der Eingabe **a** erzeugt werden würde.

Bei binären Klassifikationsproblemen wird normalerweise vor der letzten Ausgabe noch die Sigmoidfunktion $\frac{1}{1+e^{-x}}$ auf l_2 angewendet, um die Zielmenge auf $[0, 1]$ einzuschränken. Bei dem vorliegenden Beispiel gehen wir von einem Regressionsproblem

aus, sodass l_2 bereits das Endergebnis $\hat{y}$ erzeugt.

Die Frage stellt sich nun wie die Gewichte **C** und **d** bestimmt werden können. Vor dem Trainieren werden sie zufällig initialisiert. Es wird häufig dazu der *Glorot uniform initializer* verwendet [5], der Samples aus einer Einheitsverteilung mit bestimmter Gewichtung zieht. Nach der ersten *forward propagation* muss zunächst dann die Verlustfunktion $\mathcal{L}(y, \hat{y})$ berechnet werden.

Sei diese in dem Beispiel definiert durch die mittlere quadratische Abweichung (*mean squared error*) $\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$. Um eine angenehmere Ableitung zu gewährleisten, wird häufig der Term $\frac{1}{2}$ hinzugefügt.

Im nächsten Schritt erfolgt die Berechnung der partiellen Ableitungen $\frac{\partial \mathcal{L}}{\partial d_j}$ und $\frac{\partial \mathcal{L}}{\partial C_{ik}}$. Hier kommt die *backpropagation* ins Spiel.

Da ein NN aus Funktionskompositionen besteht, muss häufig die Kettenregel angewendet werden. Bei der *backpropagation* werden daher die Ergebnisse der Ableitungen zwischengespeichert, um sie bei vorhergehenden Schichten wiederzuverwenden. Beispielsweise ergibt $\frac{\partial \mathcal{L}}{\partial d_j} = -(y - \hat{y})\frac{\partial \hat{y}}{\partial d_j}$, wobei $\hat{y}$ als eine Funktion mit Parameter d_j betrachtet wird. Der Term $-(y - \hat{y})$ wird auch wieder bei der Ableitung $\frac{\partial \mathcal{L}}{\partial C_{ik}}$ vorkommen.

Nach der Berechnung der Ableitungen müssen nur noch die Gewichte mittels *gradient descent* aktualisiert werden:

$$C_{ik}^{(n+1)} = C_{ik}^{(n)} - \gamma \frac{\partial \mathcal{L}}{\partial C_{ik}^{(n)}}$$
$$d_j^{(n+1)} = d_j^{(n)} - \gamma \frac{\partial \mathcal{L}}{\partial d_j^{(n)}}$$

γ gibt hier die Lernrate an und n ist die Iteration. Übliche Werte liegen zwischen 10^{-2} und 10^{-6}, wobei die Lernrate auch mit der Zeit verringert werden kann.

In dem Beispiel wird mit jeder einzelnen Eingabe **a** die *forward propagation* und *backpropagation* durchgeführt. Bei richtigen NN werden in der Regel mehrere Eingaben $\mathbf{a_1}, \ldots, \mathbf{a_n}$ gleichzeitig übergeben. Eine solche Ansammlung von Eingaben wird als *mini batch* bezeichnet. Werte können zwischen 10 und 512 liegen, wobei es auch bei Ausnahmen größere oder kleinere Batches sein können. Da vor der Eingabe ins neuronale Netzwerk die *mini batches* in eine zufällige Reihenfolge gebracht werden, spricht man auch von *stochastic gradient descent* (SGD).

Es gibt zahlreiche Erweiterungen von SGD, die in neuronalen Netzwerken verwendet werden, um das Trainieren zu beschleunigen. Einer der beliebtesten Erweiterungen ist *momentum* [33]. Bei dieser wird das nächste Update eines Gewichts basierend auf dem derzeitigen Gradienten und dem letzten Update durchführt. Es entsteht zusätzlich noch ein weiterer Parameter β, der die Gewichtung des letzten Updates angibt. Für das Gewicht **d** ergibt sich so z. B.

$$v_j^{(n+1)} = \beta v_j^{(n)} - \gamma \frac{\partial \mathcal{L}}{\partial d_j^{(n)}}$$
$$d_j^{(n+1)} = d_j^{(n)} + v_j^{(n+1)}$$

Normalerweise werden für $\beta \in [0, 1]$ Werte größer gleich 0.9 gewählt. Wenn $\beta = 0$, ergibt sich normaler *gradient descent*. Im Bezug zur Physik wäre $\beta \in \mathbb{R}$ die Masse eines Objekts und $\mathbf{v} \in \mathbb{R}^k$ der Geschwindigkeitsvektor. Es ergibt sich mit $\mathbf{p} = \beta\mathbf{v}$ der Impuls (*momentum*) eines Objekts. In diesem Fall erhält die Suche nach dem Minimum durch den Impuls eine Beschleunigung. Im übertragenen Sinne wäre SGD ohne Impuls eine Person, die beständig einen Hügel heruntergeht, während SGD mit Impuls ein Ball wäre, der einen Hügel herunterrollt [6]. Die Trägheit des Balles führt dazu, dass kleine Vertiefungen übersprungen werden können. Dieses wären lokale Minima, sodass Oszillationen auf dem Weg zum globalen Minimum abgedämpft werden.

Zusätzlich zum *momentum* wird auch häufig Gewichtsverfall (*weight decay*) oder L_2-Regularisierung verwendet. Diese Regularisierung verhindert ein zu starkes *Overfitting* auf den Trainingsdaten und kann auch allgemein die Leistung des NN erhöhen [40]. Bei Gewichtsverfall ändert sich der Geschwindigkeitsvektor $\mathbf{v}$ bei dem Gewicht $\mathbf{d}$ wie folgt [17]:

$$v_j^{(n+1)} = \beta v_j^{(n)} - \delta\gamma d_j^{(n)} - \gamma \frac{\partial \mathcal{L}}{\partial d_j^{(n)}}$$

Es wird also das jeweilige Gewicht zur Update-Regel von SGD hinzugefügt. 0.0005 und 0.0001 sind häufig genutzte Werte für δ.

L_2-Regularisierung ist ähnlich, fügt aber bei Gewicht d_j einen Term δd_j^2 zur Verlustfunktion $\mathcal{L}$ hinzu, sodass durch die Ableitung eine zusätzliche 2 entsteht. Daher müssen bei Deep-Learning Bibliotheken wie Keras, die nur über L_2 Regularisierung verfügen, der Term $\frac{\delta}{2}$ für Gewichtsverfall verwendet werden. Auf diese Weise sind L_2-Regularisierung und Gewichtsverfall für *gradient descent* mit *momentum* äquivalent. Teilweise wird je nach Definition auch noch die Lernrate aus der Regularisierung entfernt [22, 34].

Eine weitere Technik, die bei neuronalen Netzwerken genutzt wird, nennt sich *batch normalization* (BN) [14]. Diese spezielle Schicht, die vor der Aktivierungsfunktion gesetzt wird, verringert die mögliche Zahl der Veränderungen der Verlustfunktion $\mathcal{L}$. Formal gesagt verbessert sich die Lipschitz-Stetigkeit von $\mathcal{L}$ [31]. Durch BN variiert die Verlustfunktion weniger zwischen Trainingsschritten, sodass sich das Lernen der Gewichte stabilisiert.

Dieses wird erreicht, indem jedes Feature der Eingabe einer Schicht normalisiert wird:

$$x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j},$$

wobei der Mittelwert μ_j und die Standardabweichung σ_j basierend auf dem ganzen *mini batch* berechnet werden. Allerdings kann die Normalisierung beschränken, was eine nicht-lineare Aktivierung repräsentieren kann [14]. Zum Beispiel könnten von der Sigmoidfunktion die beiden Enden wegfallen. Daher werden zusätzlich noch zwei Parameter $\boldsymbol{\gamma}$ und $\boldsymbol{\beta}$ gelernt, sodass die komplette Batchnormalisierung wie folgt definiert ist:

$$\mathrm{BN}(x_j) = \gamma_j \left(\frac{x_j - \mu_j}{\sigma_j} \right) + \beta_j$$

Das neuronale Netzwerk könnte jetzt also lernen $\gamma_j = \sigma_j$ und $\beta_j = \mu_j$, dann wären wir wieder zur ursprünglichen Funktion zurückgekehrt.

Da β_j als *bias* fungiert, kann in der vorhergehenden Schicht der *bias*-Term entfernt werden. Die zuvor vorgestellte Regularisierung kann auch auf γ_j und β_j angewendet werden. Das bedeutet, so wie vorher der Term $d_j^{(n)}$ hinzugefügt wurde, würden jetzt auch die beiden trainierbaren BN-Terme hinzugefügt werden.

Üblicherweise wird Batchnormalisierung mithilfe des *internal covariance shift* (ICS) erklärt. Dieser Begriff beschreibt kurz gesagt, dass nachgehende Schichten nicht Veränderungen von vorhergehenden Schichten erhalten und dadurch die Wahrscheinlichkeitsverteilungen auseinander gehen. Da allerdings in [31] gezeigt wurde, dass sowohl empirisch als auch theoretisch diese Erklärung problematisch ist, wird sich hier nicht weiter auf ICS bezogen.

2.2 Convolutional

Convolutional neural networks (CNN) sind neuronale Netzwerke die Faltungen statt Matrixmultiplikationen verwenden [7], wobei die Basiseinheit immer noch Matrizen bzw. Tensoren sind. Wie die *feedforward* neuronalen Netzwerke sind es nicht-lineare Modelle, die für Regression oder Klassifikation eingesetzt werden. Sie können sowohl auf Text, Ton, Videos oder Bildern angewendet werden.

Im Kontext von Bildern werden sie in Verbindung mit zweidimensionalen Faltungen genutzt. Allerdings kommen zusätzlich noch zwei Dimensionen hinzu, da normalerweise mehrere Bilder gleichzeitig betrachtet werden und die Bilder noch Farbkanäle haben.

Bei normalen eindimensionalen Faltungen wie sie in der Mathematik bei Zufallsvariablen verwendet werden, kann nur der Parameter z variiert werden:

$$(p_X * p_Y)(z) = \sum_x p_X(x) p_Y(z - x)$$

Wenn man jetzt mehrdimensionale Wahrscheinlichkeitsfunktionen betrachtet, dann erhöhen sich die Parameter von p_X und p_Y. Also entstehen auch mehrere $z_1, \ldots, z_n$ statt nur ein z wie bei der eindimensionalen Faltung. Hier erhalten wir einen Unterschied zur Faltung in der Mathematik. Bei CNN betrachten wir vierdimensionale Tensoren mit Batchgröße, Höhe, Breite und Kanälen, aber nur Höhe z_2 und Breite z_3 können in der Faltung geändert werden.

Zum Beispiel nimmt die Architektur VGG [32] als Eingabe $b \times 224 \times 224 \times 3$, wobei b die Zahl der Bilder und 3 die Zahl der RGB-Kanäle ist. Jetzt können in der Höhe und Breite Verschiebungen durchgeführt werden, aber b und die Kanäle c sind fixiert.

Konkret wäre bei CNN die Funktion p_Y ein $h_K \times w_K \times c_K$ Tensor $\mathbf{K}$, genannt Kernel und p_X wäre ein $b \times h_I \times w_I \times c_I$ Eingabe-Tensor $\mathbf{I}$. VGG nutzt z. B. in der ersten Schicht $\mathbf{K}$ mit $3 \times 3 \times 3$ und $\mathbf{I}$ mit $b \times 224 \times 224 \times 3$. Das Ergebnis der speziellen Operation ist ein $b \times 224 \times 224 \times 1$ Tensor. Die Faltung hat $\mathbf{K}$ in Höhe und Breite auf $\mathbf{I}$ „umhergeschoben“, während ein Kanal und eine Eingabe fixiert wurde. Diese Operation wird für alle Kanäle und Eingaben durchgeführt. Zum Schluss werden alle Kanäle zusammenaddiert, sodass sich nur ein Ausgabekanal ergibt. Auf der nächsten Seite wird dieses mathematisch beschrieben.

Die Kanäle in $\mathbf{K}$ hängen immer von den Eingabekanälen c_I ab. Zum Beispiel bei Eingabe $\mathbf{I}$ mit Dimension $b \times 32 \times 32 \times 1$ hat $\mathbf{K}$ Dimension $h_K \times w_K \times 1$. Wenn $h_K = w_K = 1$, dann ist die Faltung in dem Spezialfall eine Multiplikation mit einem Skalar.

In der Realität werden mehrere $\mathbf{K}$ in einer Faltung verwendet. Deshalb fügt man eine weitere Dimension hinzu, sodass $\mathbf{K}$ mit $h_K \times w_K \times c_K \times f_K$ vierdimensional wird. Hier gibt f_K die Zahl der Kernels an. Ein alternativer Begriff für f_K ist Filter. Im vorherigen Beispiel haben wir bei VGG nur einen Kernel betrachtet. Wenn man alle Kernels betrachtet, dann sind es bei der ersten Schicht bei VGG $f_K = 64$, sodass dann die Ausgabe $b \times 224 \times 224 \times 64$ ist. Hier wird Konkatenation verwendet, um die Kernels zusammenzufügen.

Bisher wurde die Faltung im Kontext von CNN nur intuitiv mit Verschiebungen beschrieben. Es folgt eine Definition nach [35], die in Teilschritten vorgeht:

1. Single Channel Single Kernel (SCSK)
2. Multiple Channel Single Kernel (MCSK)
3. Multiple Channel Multiple Kernel (MCMK)

Bei SCSK ist eine $h \times w$ Matrix $\mathbf{I}$ und eine $k \times k$ Matrix $\mathbf{K}$ gegeben. Die Batches werden hier ignoriert und der Kernel hat gleiche Höhe und Breite.

$$\text{SCSK}_{x,y}(\mathbf{I}, \mathbf{K}) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \mathbf{I}_{x-\lfloor \frac{k}{2} \rfloor + i, y-\lfloor \frac{k}{2} \rfloor + j} \mathbf{K}_{i,j}$$

Wie man sieht, handelt es sich bei der CNN-Faltung eigentlich um eine Kreuzkorrelation. x und y geben die Position in der Ausgabe an.

Jetzt fügen wir zu $\mathbf{I}$ und $\mathbf{K}$ die Kanäle hinzu, sodass die Dimensionen $h \times w \times c$ und $k \times k \times c$ sind. Somit erhalten wir MCSK. Mit dem Index j wird der Kanal ausgewählt.

$$\text{MCSK}_{x,y}(\mathbf{I}, \mathbf{K}) = \sum_{j=0}^{c-1} \text{SCSK}_{x,y}(\mathbf{I}_j, \mathbf{K}_j)$$

Sowohl die Ausgabe von SCSK als auch von MCSK ist zweidimensional. Man vergleiche dieses Ergebnis mit dem vorherigen Beispiel von der VGG-Architektur. Das Zusammenaddieren wurde dort in Worten beschrieben.

Zum Schluss wird noch die Zahl der Kernel erhöht, sodass $\mathbf{K}$ jetzt $k \times k \times c \times f$ groß ist. Bei MCMK werden alle MCSK miteinander konkateniert, sodass die letzte Dimension der Ausgabe f ist. In diesem Fall ist das Ergebnis dreidimensional.

Nicht immer wird das Ergebnis der CNN-Faltung in Höhe und Breite so groß sein wie die Eingabe. In den meisten CNN-Architekturen wird deshalb mit *zero padding* gearbeitet. Dieser Begriff beschreibt das Auffüllen der Eingabe mit Nullen. Bei einer eindimensionalen Faltung auf einem Array mit 5 Elementen und einem Kernel der Größe 3, sind z. B. nur 3 Verschiebungen möglich. Daher wird auch die Ausgabegröße 3 sein. Wenn das Eingabearray durch 2 Nullen erhöht wird, dann hat man die Ausgabegröße 5.

Ein weiterer Aspekt ist *stride*. Dieser Begriff beschreibt die Schrittgröße eines Kernels d. h. wie weit das „Fenster“ mit jedem Schritt verschoben werden darf. In den meisten CNN wird aber 1 verwendet.

Im Vergleich zu normalen NN funktioniert bei CNN die *forward propagation* und *backpropagation* immer noch ähnlich. Am Anfang werden die Kernels $\mathbf{K}$ zufällig initialisiert. Dann wird die Funktionskomposition auf dem Eingabebild durchgeführt, um die Vorhersage zu erhalten. Bei der *backpropagation* werden die Gradienten mit Bezug zu $\mathbf{K}$ erzeugt und es findet schließlich das Updaten der Gewichte $\mathbf{K}$ statt.

Die Batchnormalisierung, die Aktivierungsfunktionen wie ReLU und andere Techniken von NN können auch hier verwendet werden. Allerdings gibt es auch noch weitere Schichten, die man nur bei CNN verwenden kann.

2.2.1 Spezielle Schichten

Abseits der Faltungen gibt es bei CNN noch viele weitere Arten von Schichten. In dieser Arbeit sind zusätzlich noch *Pooling*-Schichten, *Upsampling* und Konkatenation wichtig. Bei allen diesen drei Arten von Schichten müssen keine Parameter erlernt werden.

Pooling-Schichten reduzieren die Größe einer *feature map* und verringern damit die Zahl der notwendigen Parameter des CNN. Eine *feature map* ist die Ausgabe einer Faltung. Es kann entweder das Maximum (*max pooling*) oder der Durchschnitt (*avg pooling*) bei Pooling-Schichten berechnet werden. Diese Operation macht die Eingabe invariant zu kleinen Verschiebungen des Bildes [7]. Wenn z. B. bei der ursprünglichen Eingabe alle Werte um einige Schritte nach rechts verschoben werden, dann würde sich die Ausgabe von *max pooling* kaum verändern. Durch einen Kernel wird immer eine lokale Umgebung betrachtet und von dieser wird das Maximum oder der Durchschnitt gebildet.

Bei Pooling-Schichten fällt bei der Konfiguration die Zahl der Kernels weg und man benötigt bloß die Größe eines Kernels, die Schrittgröße und *zero padding*. Allerdings wird in der Realität fast immer der Kernel auf 2×2, die Schrittgröße auf 2×2 und *zero padding* auf 0 gesetzt.

Üblicherweise wird eine Pooling-Schicht nach jeden zwei oder drei Faltungen gesetzt, da ansonsten der Speicherverbrauch zu stark ansteigen würde. Bei VGG kommt beispielsweise nach den ersten beiden Faltungen, die als Ergebnis eine $224 \times 224 \times 64$ *feature map* ergeben haben, ein *max pooling*. Das Ergebnis ist dann eine $112 \times 112 \times 64$ *feature map*. Der Kernel und die Schrittgröße waren hier also 2×2 gewesen, da sich die *feature map* in Breite und Höhe um zwei reduziert hat. Wenn der *stride* lediglich 1×1 gewesen wäre, dann wäre die Ausgabe $223 \times 223 \times 64$. Die Formel ist also $\lfloor \frac{w_I - w_K}{S} \rfloor + 1$, wobei w_I die Breite des Bildes, w_K die Breite des Kernels und S *stride* ist. Für Höhe funktioniert es analog. Die Tiefe bleibt immer gleich.

Als Nächstes wird das Upsampling betrachtet, welches zur Vergrößerung eines Bildes dient. Hier wird meistens *nearest-neighbor interpolation* genutzt. Dabei wird jeder Pixel in der Ausgabe auf den nächstgelegenen der Eingabe gesetzt. Zum Beispiel bei der Vergrößerung von 3×3 auf 6×6 können in der Ausgabe zunächst die ersten 9 Pixel skaliert eingesetzt werden. Die restlichen Positionen, die noch 0 sind, werden mit den nächstgelegenen Pixeln aufgefüllt.

Schließlich gibt es noch die Konkatenation. Wenn zwei Tensoren bis auf einer Dimension gleich groß sind, können sie zusammengefügt werden. Diese Technik wird häufig verwendet, da sie *skip connections* ermöglicht. Dieser Begriff beschreibt das Überspringen von bestimmten Schichten, um einen alternativen Datenfluss zu erzeugen. Zum Beispiel könnte die fünfte Schicht mit der 20ten Schicht verbunden werden, wenn die Dimensionen übereinstimmen. Wenn dies nicht der Fall ist, kann u. a. ein Upsampling verwendet werden.

Während des Durchlaufs durch das Netzwerk reduziert sich die Höhe und Breite der *feature map*. So ist nach der letzten Faltung die Ausgabe häufig nur noch 7×7 groß. Daher können Informationen verloren gegangen sein. Zum Beispiel würden Informationen von kleinen Objekten in der 7×7 Matrix verschwinden. Mit der Konkatenation werden kleine Objekte wieder in spätere Schichten gebracht [19].

2.2.2 Initialisierung der Gewichte

Zwar können die Gewichte – wie zuvor beschrieben – zufällig initialisiert werden, aber in der Praxis werden ImageNet-Gewichte verwendet [13, 11, 25, 3, 2, 38, 21].

Bei ImageNet [4] handelt es sich um eine große Bilderdatenbank, die derzeitig (2019) insgesamt 14.2 Millionen Bilder enthält. Basierend auf WordNet einer lexikalischen Datenbank für das Englische ist das Ziel jeder Kategorie (wie z. B. den unterschiedlichen Arten von Fischen, Sportarten, Pflanzen, Alltagsobjekten) $500 - 1000$ Bilder zur Verfügung zu stellen.

Wann immer eine neue CNN-Architektur erscheint [10, 30, 32], wird die Leistung anhand der *ImageNet Large Scale Visual Recognition Challenge* evaluiert [28]. Dieser Wettbewerb läuft seit 2010 und betrifft unterschiedliche Bereiche wie Bildklassifikation oder Objektlokalisierung. Es wird hierbei eine Teilmenge der ImageNet-Bilder genutzt.

Die Autoren von CNN-Architekturen stellen zumeist die trainierten ImageNet-Gewichte zur Verfügung, sodass der Nutzer einer *Deep Learning*-Bibliothek mit den ImageNet-Gewichten starten kann. Wenn der Nutzer z. B. Schachfiguren in Bildern finden möchte, aber ImageNet vielleicht noch nicht diese Kategorie enthält, dann würden trotzdem ImageNet-Gewichte die Leistung verbessern. Dies liegt daran, dass auf Wissen von anderen Bildern zurückgegriffen werden kann (Linien, Farben etc.). Dieser Prozess wird dann *transfer learning* genannt [39].

Falls das Datenset, was zur Verfügung steht, sehr groß ist, dann kann nicht so viel von den ImageNet-Gewichten profitiert werden. Gleiches gilt auch, wenn die verfügbaren Bilder sich zu stark von den ImageNet-Bildern unterscheiden. Ein weiterer Nachteil ist, dass nur begrenzt die Basisarchitektur geändert werden kann. Es ist z. B. nicht möglich von einer gegebenen Architektur die Kernelgröße zu verändern, da dann die Gewichte zurückgesetzt werden müssten. Ein Neutrainieren auf ImageNet ist aufgrund der Dauer und der Zahl der notwendigen GPU häufig nicht möglich.

2.3 Vergleich

Bevor zur Anwendung gekommen wird, sollen zuletzt noch die Unterschiede zwischen NN und CNN verdeutlicht werden. Bei NN gibt es Gewichte für jedes Neuron einer Schicht. Bei CNN wird mittels Faltung die Zahl der Verbindungen begrenzt, sodass der Speicherverbrauch sinkt und effizient Bilder verarbeitet werden können. Mit dem rezeptiven Feld wird die lokale Region beschrieben, mit welcher ein Neuron verbunden ist. Die folgende Grafik soll diesen Sachverhalt genauer erläutern.

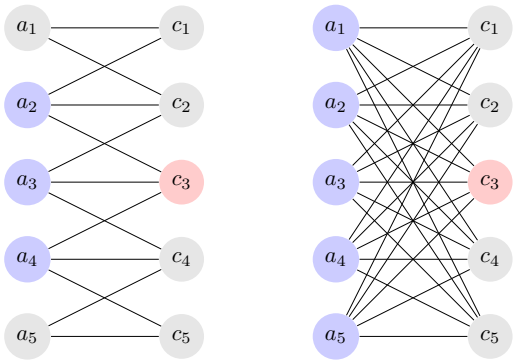


Abbildung 3: Verbindungen im Vergleich (links: CNN, rechts: NN)

Im Beispiel wird das rezeptive Feld von c_3 durch a_2 bis a_4 für ein CNN und a_1 bis a_5 für ein NN erzeugt, wobei *padding* durch versteckte Nullen berücksichtigt wird. Das rezeptive Feld ist die Größe des Kernels (in der Abb. links 3). In der Literatur wird häufig das rezeptive Feld von neuronalen Netzwerken über mehrere Schichten hinweg berechnet [36, 38, 21], da so herausgefunden werden kann, wie viel ein CNN „sieht“. Dieses wird dann als effektives rezeptives Feld (ERF) bezeichnet [18]. Zum Beispiel ergibt ein 9×9 Kernel mit *stride* 1, gefolgt von einem 2×2 Kernel mit *stride* 2, ein ERF von 10×10. Man fügt also die nicht-überlappte Region der nächsten Faltung hinzu: $9 + (2 - 1) = 10$. Für die Berechnungen im Detail, sei der Leser auf [18] verwiesen.

CNN haben im Vergleich zu NN wesentlich weniger Parameter. Dieses sieht man bereits an den Verbindungen in der Abbildung. Beim CNN bekommt man ausgehend von gleicher Größe und Höhe als Eingabe $h \times w \times c_I$ und als Ausgabe $h \times w \times c_O$. Der Kernel hat Dimension $h_K \times w_K \times c_I \times c_O$. Dann gibt es $h_K \cdot w_K \cdot c_I \cdot c_O$ Parameter (ohne *bias*). Wenn man beim NN das Bild als einen Vektor schreibt, ergeben sich durch die Matrixmultiplikation $m \cdot n$ Parameter. Die Eingabe ist in diesem Fall $m = h \cdot w \cdot c_I$ und n ist die Zahl der Neuronen. Da $m \gg h_K \cdot w_K \cdot c_I$, ist das CNN effizienter.

Bei einer Eingabe von $224 \times 224 \times 3$, einer Kernelgröße von 3×3 und Ausgabekanälen 64 ergeben sich ohne *bias* $3 \cdot 3 \cdot 64 \cdot 3 = 1728$ Parameter für das CNN. Im Vergleich benötigt das NN bei 64 Neuronen insgesamt $(224 \cdot 224 \cdot 3) \cdot 64 = 9633792$ Parameter. Wenn die Kernelgröße auf 224×224 erhöht wird, dann erhält man die gleiche Anzahl an Parametern bei NN und CNN. Deshalb werden stets kleine Kernelgrößen wie 3 gewählt.

3 Anwendung

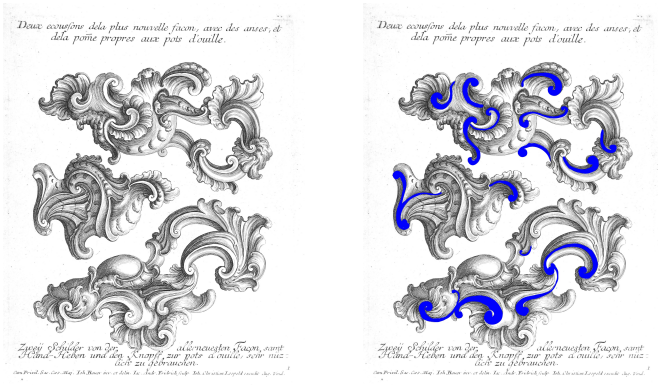
In diesem Kapitel wird ein CNN für die Kunstbilder konzipiert, welche in der Einleitung vorgestellt wurden. Das neuronale Netzwerk wird verwendet, um jedes Pixel in einer der Kategorien „Rocaille“ oder „Nicht-Rocaille“ einzuordnen. Es handelt sich also um Klassifikation. In der Bildverarbeitung wird dieses Problem auch Segmentierung genannt.

3.1 Präparierung der Daten

Zunächst müssen die Daten für das neuronale Netzwerk vorbereitet werden. Da das Annotieren eine gewisse Zeit in Anspruch nimmt, wurde nur eine kleine Menge von 17 Grafiken von der Universität Regensburg bereitgestellt. Es wurde hierbei wie folgt vorgegangen:

1. Grafik wird in einem Bildverarbeitungsprogramm geöffnet.
2. RGB-Werte $(0, 0, 255)$ werden auswählt (blau).
3. Gewünschte Region wird angemalt.
4. Bild wird als PNG abspeichert.

Es hätten auch andere Farben als blau verwendet werden können. Wichtig ist es bloß, dass diese Farbe nicht bereits im Bild vorkommt, da ansonsten nicht die richtigen Pixel in die Kategorie „Nicht-Rocaille“ kommen. Auch bei der Wahl des Bildformats ist man nicht zwingend an PNG gebunden. Lediglich das Format muss verlustfrei ist. JPG wäre somit nicht möglich, da die Pixel bei der Speicherung komprimiert werden.



(a) Originalbild (b) Annotiertes Bild

Abbildung 4: Resultat des ersten Vorverarbeitungsschrittes

Im nächsten Schritt muss aus dem angemalten PNG-Bild eine Matrix erzeugt werden, die nur aus Nullen und Einsen besteht. Dies entspricht den beiden Kategorien „Nicht-Rocaille“ und „Rocaille“. Dazu werden alle blauen Pixel in Grafik (b) auf 1 gesetzt und alle anderen Pixel auf 0.

Schließlich wird im letzten Schritt ein *sliding window* genutzt, um das Bild aufzuteilen. Dies ist notwendig, da die Bilder mit einer Größe von ca. 2000×2000 zu groß sind, um sie direkt dem neuronalen Netzwerk zu übergeben. Außerdem würde ohne Aufteilung die Zahl der Trainingsdaten dann nur 17 sein. Da in der Literatur eine Größe von 224×224 als Eingabe der CNN üblich ist [10, 30, 32], wird auch hier diese Größe für das *sliding window* verwendet.

Um die Zahl der Trainingsdaten zu erhöhen, wird das *sliding window* für die Startpunkte $30, 60, \ldots, 210$ wiederholt. Zum Schluss erhält man fast 100 mal so viele Bilder, als wenn bloß ein *sliding window* mit dem Startpunkt 0 genutzt werden würde. Alternativ zu dem *sliding window* wäre es auch möglich gewesen, zufällige Bildausschnitte zu generieren. Beide Verfahren liefern aber ähnliche Ergebnisse, sodass sich für das *sliding window* entschieden wurde. Außerdem wird später auch wieder ein solches Fenster bei der Rekonstruktion des Bildes genutzt.

Die folgende Grafik zeigt ein Bild aus diesem ersten Vorverarbeitungsschritt.

(a) Bildausschnitt (b) Maske

Abbildung 5: Bild (a) ist die Eingabe für das CNN und (b) ist die erwartete Ausgabe

Zur besseren Visualisierung wurden in der Abbildung 5 die Pixel der Kategorie „Rocaille“ auf 255 gesetzt (statt 1). In vielen Bildausschnitten überwiegen die schwarzen Pixel („Nicht-Rocaille“). Dieses zeigt auch das folgende Histogramm.

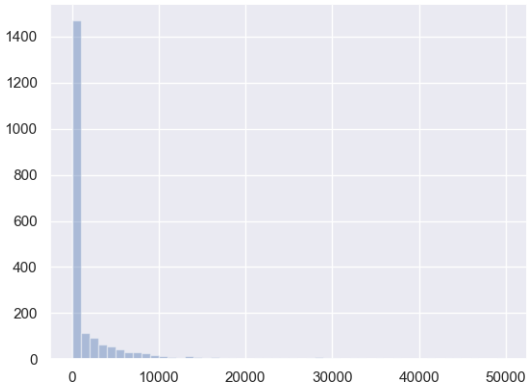


Abbildung 6: Zahl der weißen Pixel (Kategorie „Rocaille“) in Bildausschnitten

Die Mehrheit der Bildausschnitte enthält entweder keine weißen Pixel oder zwischen 1000 und 10000 weiße Pixel. Da ein Bild allerdings aus $224 \cdot 224 = 50176$ Pixeln besteht, gibt es überwiegend nur zwischen $\frac{1000}{50176} \approx 2\%$ und $\frac{10000}{50176} \approx 20\%$ weißen Pixeln.

3.2 Architektur

Nachdem die Daten erzeugt wurden, muss jetzt die Architektur des CNN bestimmt werden. Bei der allgemeinen Segmentierung von Objekten sind Architekturen wie DeepLabv3+ oder Mask R-CNN beliebt [3, 9]. Diese Modelle werden auf einer großen Zahl von Bildern ($\geq$ 2 GB) trainiert und verwendet, um Alltagsobjekte wie Fahrräder oder Flaschen zu finden.

Das vorliegende Datenset ist jedoch mit 17 Bildern weder wesentlich groß, noch enthält es Objekte dieser Art. Die Bildausschnitte haben zwar die Zahl der Bilder stark erhöht, aber ein Großteil der Bilder gibt keine neuen „Informationen“. Die Verschiebungen entsprechen eher einer *data augmentation*. Mit diesem Begriff werden Verfahren bezeichnet die künstlich die Zahl der Daten erhöhen (Spiegelungen, Farbwechsel etc.), um die Leistung eines NN zu verbessern.

Aufgrund der wenigen Bilder wurde sich entschieden die U-Net Architektur zu verwenden [26]. Diese Architektur wurde ursprünglich für die biomedizinische Bildsegmentierung entwickelt, aber auch schon erfolgreich in anderen Domänen verwendet [11, 12, 13, 24]. Des Weiteren kommt sie mit geringen Datenmengen von ca. 30 Bildern zurecht.

U-Net besteht aus einem *contracting path* (Encoder), der Merkmale wie Linien enkodiert und einem *expansive path* (Decoder), der das Bild auf die ursprüngliche Größe zurückversetzt.

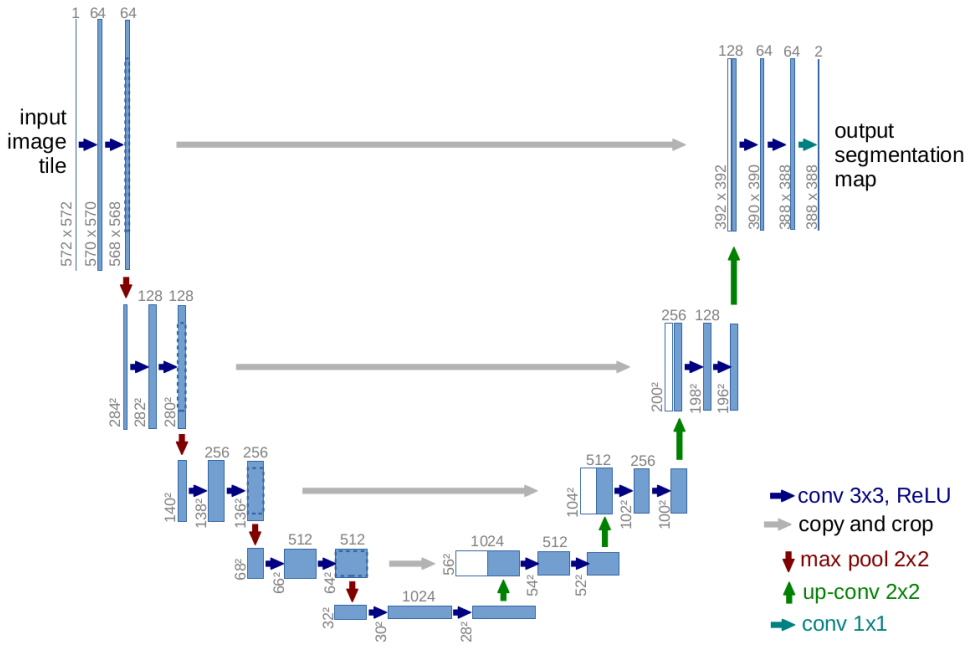


Abbildung 7: Originale U-Net Architektur von 2015 (Abbildung aus [26])

In der Abbildung wird durch die Anwendung von 3×3 Faltungen und 2×2 *max pooling* die Zahl der Filter erhöht, während die Auflösung des Bildes selbst sinkt. Zum Schluss nach dem letzten *max pooling* muss die ursprüngliche Auflösung wiederhergestellt werden. Dazu wird jeweils eine *nearest-neighbor interpolation* (up-conv 2×2) verwendet. Es folgt dann eine Konkatenation mit einer der Schichten aus dem Encoder

und zwei Faltungen (ggf. auch Abschneiden). Dann muss nur noch eine 1×1 Faltung mit der Zahl der Klassen auf die Ausgabe angewendet werden.

In dem ursprünglichen Paper von 2015 wurde U-Net noch mit zufälligen Gewichten trainiert. Neuere Arbeiten wie [11, 13] haben gezeigt, dass die Verwendung von Encodern, die auf ImageNet vortrainiert wurden, zu einer Verbesserung der Leistung führt.

Für die vorliegenden Kunstbilder wurden aus diesem Grund zwei verschiedene Architekturen basierend auf U-Net entwickelt: (a) Modell 1: U-Net mit zufälligen Gewichten und (b) Modell 2: U-Net mit DenseNet-121 als Encoder.

3.2.1 Modell 1

Dieses Modell entspricht der originalen U-Net Architektur und unterscheidet sich nur durch die Nutzung von Batchnormalisierung und einer anderen Zahl an Filtern.

Die folgende Grafik zeigt eine mögliche Architektur von Modell 1.

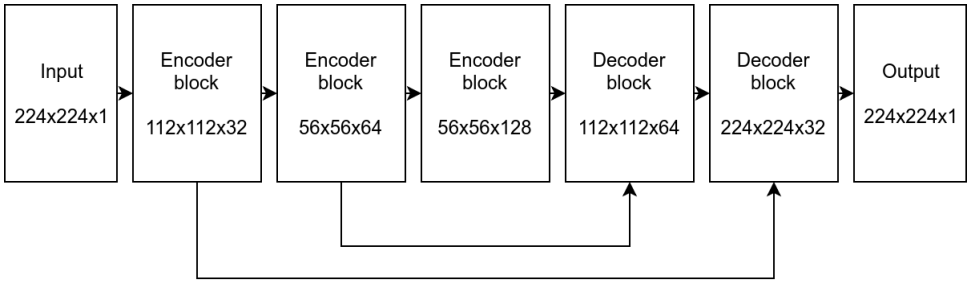


Abbildung 8: Modell 1 mit 32 Startfiltern, 128 Stoppfiltern und 472 225 Parametern

Ein Encoder-Block besteht aus zwei Faltungen mit einer Kernelgröße von 3, einer Batchnormalisierung und einer *max pooling*-Schicht. Lediglich bei dem letzten Encoder-Block wird auf das *max pooling* verzichtet. Die Decoder-Blöcke bestehen aus *nearest neighbor interpolation*, einer Konkatenation mit einer vorherigen Schicht und zwei Faltungen. Die beiden Pfeile unter den Blöcken bedeuten Konkatenation. Da nur zwei Klassen zu unterscheiden sind („Rocaille“ und „Nicht-Rocaille“), wird auf die Ausgabe eine 1×1 Faltung mit einer Sigmoid-Aktivierung angewendet. Bei allen anderen Faltungen wird die ReLU-Aktivierung und *zero padding* (Auffüllung mit Nullen) verwendet.

Modell 1 hat zwei Parameter: Startfilter und Stoppfilter. Beispielsweise wäre im ursprünglichen U-Net Paper (siehe Bild 7) die Zahl der Startfilter 64 und die Zahl der Stoppfilter 1024. Konkret würde in diesem Fall der erste Encoderblock 64 Filter haben. Mit jedem weiteren Block wird die Zahl der Filter um eine Zweierpotenz erhöht (d. h. $2^6 = 64, \ldots, 2^{10} = 1024$).

3.2.2 Modell 2

Das zweite Modell verfügt über eine höhere Zahl an Parametern und ist damit langsamer als das erste Modell. Nur wenn die Stoppfilter des ersten Modells auf 1024 gesetzt werden, hat das zweite Modell weniger Parameter. Die Architektur befolgt immer noch den generellen Aufbau von U-Net, unterscheidet sich aber durch die Nutzung eines DenseNet-121 Encoders [10].

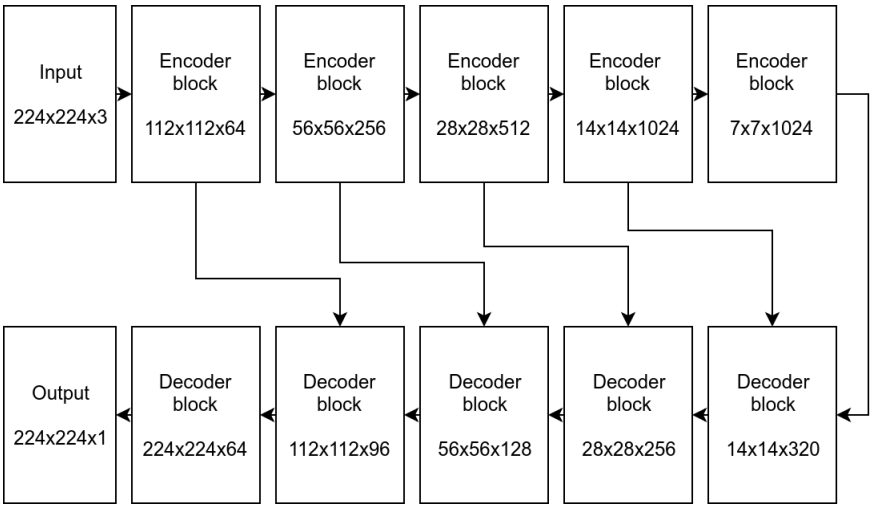


Abbildung 9: Modell 2 mit DenseNet-121 als Encoder, 17 451 073 Parameter

Statt DenseNet-121 hätten sich auch andere Encoder wie ResNet-34 oder Inception Resnet v2 angeboten, jedoch ist die Leistung zwischen den Modellen häufig ähnlich. Beispielsweise in [24] unterscheidet sich U-Net mit dem besten und schlechtesten Encoder nur um knapp 2% IoU (Intersection over Union). DenseNet-121 wurde gewählt, da es im Vergleich zu dem *residual neural network* weniger Parameter hat und auch in Wettbewerben verwendet wird [15].

Die Encoder-Blöcke bestehen bis auf dem letzten aus einem *dense block* und einem *transition layer*. Ein *dense block* besteht aus wiederholter 1×1 Faltung und 3×3 Faltung. Wie bei Modell 1 wird die ReLU-Aktivierung und *zero padding* verwendet. Ein *transition layer* besteht aus 1×1 Faltung und 2×2 *avg pooling*. Das *transition layer* fehlt nur beim letzten Encoder-Block, wegen des *avg pooling*.

Das Besondere an dem *dense block* ist, dass jede Faltung innerhalb eines solchen Blocks mit jeder folgenden Faltung mittels Konkatenation miteinander verbunden ist. Daher ergeben sich $\frac{L(L+1)}{2}$ Verbindungen statt L. Wenn zum Beispiel 5 Faltungen miteinander verbunden werden sollen, dann ergeben sich $1+2+3+4 = 10$ Verbindungen (eine Verbindung von Faltung 4 zu 5, zwei von Faltung 3 zu 5 etc.). Bei anderen neuronalen Netzwerken wie VGG gäbe es nur eine Verbindung zwischen Faltungen, also $1 + 1 + 1 + 1 = 4$.

Die Decoder-Blöcke sind wie die Decoder-Blöcke in Modell 1 aufgebaut. Wie bei Modell 1 werden mit jeder Konkatenation weniger Filter verwendet. Die letzten Schichten sind wichtiger, da sie mehr Informationen enthalten als frühere Schichten, die Gaborfiltern ähneln [39]. Gleichzeitig wird so die Zahl der notwendigen Parameter, die zum Trainieren notwendig sind, verringert.

3.3 Verlustfunktion

Bei Klassifikationsproblemen wird in der Regel die Kreuzentropie optimiert. Für den binären Fall ist diese mit zwei Erweiterungen wie folgt definiert:

$$H(p, \hat{p}) = -(1-\hat{p})^{\gamma} \alpha p \log(\hat{p}) - (1-\alpha)\hat{p}^{\gamma}(1-p)\log(1-\hat{p})$$

wobei $p \in \{0, 1\}$ die *ground truth* ist und $\hat{p} = \frac{1}{1+e^{-x}}$ die Vorhersage der letzten Schicht des CNN ist. Da bei den Daten aber ein Klassenungleichgewicht existiert (siehe Abbildung 6), kann eine bestimmte Gewichtung von der einen oder anderen Klasse zu einer Verbesserung der Leistung führen. Aus diesem Grund wurde wie in [38] noch der Term α hinzugefügt, der entweder auf einen festen Wert zwischen 0 und 1 gesetzt werden kann, oder einen dynamischen Wert bekommen kann wie

$$\alpha = \frac{\sum_i 1 - p_i}{\sum_i p_i + \sum_i 1 - p_i} \text{ oder } \alpha = \frac{\sum_i p_i}{\sum_i p_i + \sum_i 1 - p_i}.$$

Die positiven Beispiele werden somit mit der Zahl der negativen oder positiven Beispiele gewichtet. Alternativ können für α auch feste Werte eingesetzt werden.

In der Objekterkennung wird des Weiteren bei Klassenungleichgewichten der sogenannte *Focal Loss* verwendet [20]. Bei diesem erhalten leichtere Pixel eine geringere Gewichtung als schwerere Pixel. Dazu wurde zu $H(p, \hat{p})$ noch die Terme $(1 - \hat{p})^\gamma$ und $\hat{p}^\gamma$ hinzugefügt. Wenn beispielsweise $\gamma = 2$ ist und das neuronale Netzwerk vorhersagt, dass $\hat{p} = 0.9$, aber in Wahrheit $p = 0$ ist, dann wird der Verlust um $0.9^2 = 0.81$ gewichtet. Wenn allerdings $\hat{p} = 0.1$ gilt, dann fällt mit $0.1^2 = 0.01$ die Gewichtung geringer aus. Bei 0.1 hat sich das neuronale Netzwerk nicht so stark geirrt wie bei 0.9.

Um numerische Stabilität zu gewährleisten, wurde $H(p, \hat{p})$ wie folgt implementiert:

$$H(p, \hat{p}) = \left(\log\left(1 + e^{-|x|}\right) + \max(-x, 0)\right)(\alpha(1-\hat{p})^\gamma p + (1-\alpha)\hat{p}^\gamma(1-p)) + x(1-\alpha)\hat{p}^\gamma(1-p)$$

Wenn $\hat{p} < 10^{-7}$ ist, wird $\hat{p}$ auf 10^{-7} gesetzt, da $\log(x)$ bei $x \to 0^+$ minus unendlich ergibt. Dieses gewährleistet, dass während des Trainierens die Verlustfunktion nicht NaN ausgibt.

In der Literatur wird häufig Kreuzentropie mit dem Dice-Koeffizienten oder Jaccard-Index kombiniert [11, 12, 13, 24]. Allgemein lassen sich diese beiden Maße durch den Tversky loss (TL) ausdrücken [29], wobei $\beta = \frac{1}{2}$ den Dice-Koeffizienten ergibt und das Weglassen von β den Jaccard-Index.

$$\text{TL}(p, \hat{p}) = \frac{p\hat{p}}{p\hat{p} + \beta(1-p)\hat{p} + (1-\beta)p(1-\hat{p})}$$

β gibt hier die Gewichtung der *false positives* (FP) gegenüber *false negatives* (FN) an. Die gesamte Verlustfunktion für einen Pixel wird dann hier wie folgt definiert:

$$\mathcal{L}(p, \hat{p}) = \delta H(p, \hat{p}) - (1 - \delta)\log(\text{TL}(p, \hat{p}))$$

Auf das ganze Batch bezogen, muss die Funktion auf die Bildhöhe, Bildbreite und Batchgröße angewendet werden.

Diese kombinierte Formulierung ermöglicht es den Effekt verschiedener Parameter gleichzeitig zu überprüfen, ohne mehrere Verlustfunktionen zu erzeugen. Es gibt dafür vier zusätzliche Hyperparameter für das neuronale Netzwerk: δ, β, γ und α.

Wenn $\alpha = 0.5$, $\delta = 1$ und $\gamma = 0$ erhält man normale binäre Kreuzentropie, da $H(p, \hat{p})$ nur mit zwei multipliziert werden muss. Wenn $\delta = 1$ und α einen festen Wert annimmt, dann ergibt sich Focal Loss.

3.4 Trainieren

Die Bildausschnitte, die im Vorverarbeitungsschritt erzeugt wurden, müssen nun dem neuronalen Netzwerk übergeben werden. Aufgrund des Klassenungleichgewichtes werden drei verschiedene Batchstrategien überprüft:

1. Sliding: nur die Batches werden zufällig geladen. Bei einer Batchgröße von 10 würden also fast immer 10 Ausschnitte des gleichen Bildes im Batch enthalten sein.

2. Zufall: sowohl Batches als auch die Bildausschnitte in den Batches werden zufällig geladen.

3. Zyklisch: zunächst werden Bildausschnitte mit 10% weißen Pixeln gezogen und in ein Batch gelegt. Das nächste Batch wird dann mit 20% weißen Pixeln gefüllt. Dieser Vorgang wird wiederholt, bis 90% erreicht wurde, dann wird bis 10% wieder rückwärts heruntergezählt.

Das folgende Diagramm zeigt das Verhalten der drei Batchstrategien über 200 Batches, wobei die weißen Pixel solche Pixel sind, die Teile eines Ornamentes sind.

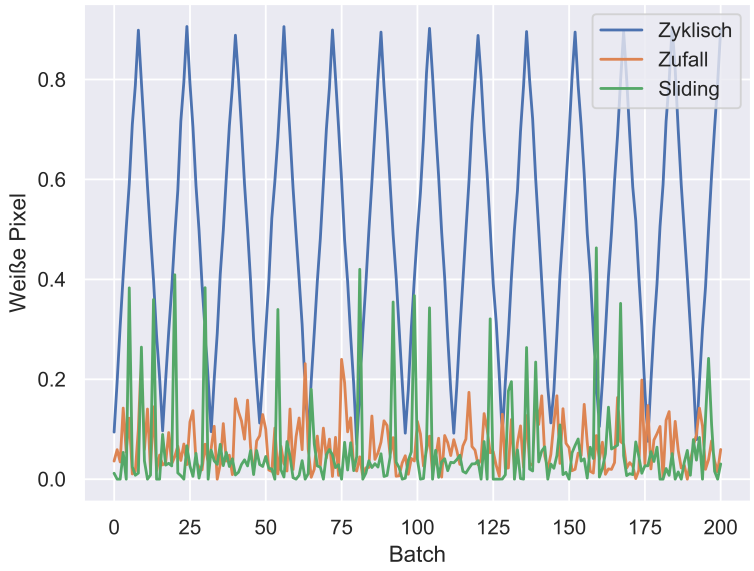


Abbildung 10: Verteilung der weißen Pixel in Bildausschnitten

Die Sliding-Batchstrategie hat mehr Ausreißer, da nebenliegende Bildausschnitte in einem Batch eher dazu tendieren entweder sehr wenige weiße Pixel oder sehr viele zu enthalten. Sobald man die Bildausschnitte zufällig wählt, verteilen sich gleichmäßiger die wenigen Bildausschnitte, die weiße Pixel enthalten. Daher ergeben sich bei der zweiten Batchstrategie weniger Ausreißer.

Bei der dritten Batchstrategie wird dem Klassenungleichgewicht direkt entgegengewirkt, da alle Arten von Bildausschnitten jede 16 Batches einmal vorkommen. Batches $0, 16, 32, \ldots$ enthalten so z. B. 10% weiße Pixel. Wie man im Diagramm sieht, kommt es bei den anderen Batchstrategien nie vor, dass mehr als 50% der Pixel in einem Batch weiß sind.

Nach der Auswahl der Bildausschnitte müssen die Daten noch normalisiert werden. Modell 1 verwendet $\frac{x_{i,j,r}}{128} - 1$ für alle drei RGB-Kanäle. Modell 2 subtrahiert den Mittelwert und dividiert mit der Standardabweichung von ImageNet.

Zur Evaluierung der Leistung während des Trainierens wird hauptsächlich der „harte“ Dice-Koeffizient (DC) verwendet:

$$\mathrm{DC}(p, \hat{p}) = \frac{2p\hat{p}}{p + \hat{p}},$$

wobei $\hat{p}$ auf 0 bzw. 1 gerundet wird. Statt nach jedem Trainingsschritt einen Validierungsschritt durchzuführen, wird DC nach jeder Epoche berechnet (d. h. nach einem Durchlauf durch allen Trainingsdaten). Mit „Schritt“ ist in dem Kontext ein einzelner Batch gemeint.

Bei Maßen wie Genauigkeit (engl. *accuracy*) kann der Durchschnitt der Ergebnisse aller Validierungsschritte gebildet werden, um die gesamte Validierungsgenauigkeit zu erhalten. Dieses ist bei DC nicht möglich, da $p+\hat{p}$ über die Batches hinweg variiert. Bei Genauigkeit wäre der Nenner des Bruches bei einer Batchgröße von 10 stets $10wh$, wobei w Bildbreite und h Bildhöhe ist. Daher würde bei Genauigkeit eine Approximierung durch die Batchgenauigkeit möglich sein.

Das Trainieren unterscheidet sich bei Modell 1 und Modell 2 nicht. Lediglich gibt es bei Modell 2 noch die Möglichkeit *fine tuning* zu betreiben. Ohne *fine tuning* trainiert man nur den Decoder, während man bei *fine tuning* alle Gewichte freischaltet. Da bei Modell 1 zufällige Gewichte verwendet werden, ergibt das „Einfrieren“ von bestimmten Schichten keinen Sinn. Daher fällt bei Modell 1 das *fine tuning* als Möglichkeit weg.

3.5 Vorhersage

Da das CNN lediglich Bildausschnitte der Größe 224×224 ausgibt, muss die Vorhersage für das gesamte Bild durch ein *sliding window* erzeugt werden. Dazu wird ein $h \times w \times i$ Tensor erzeugt, wobei h die Höhe, w die Breite und i die Zahl der *sliding windows* ist. Die *sliding windows* starten mit einem Abstand $k\lfloor \frac{224}{i} \rfloor$ für $0 \leq k \leq i-1$. Jede der Ausgaben des Fensters wird an das CNN übergeben und die Vorhersage wird in den Tensor eingesetzt.

Zum Schluss wird dann der Durchschnitt der i Bilder erzeugt, um ein einziges $h \times w$ Bild herauszubekommen. Durch dieses Vorgehen ist die Vorhersage nicht nur beschränkt auf den 224×224 Bildausschnitt, da eine Überlappung stattfindet. Zum Beispiel, wenn Fenster 1 den Bereich $[0, 224)$ abdeckt und Fenster 2 den Bereich $[20, 244)$, dann überlappen sich $[20, 224)$. Vom nächsten Bereich des Fensters 1 würde dann $[224, 448)$ kommen, sodass nur der Anfang $[0, 20)$ nicht überlappt sein würde. Wenn keine Überlappung stattfindet, werden Divisionen wie $\frac{x+0+0}{3}$ zu $\frac{x+0+0}{1}$ geändert.

Das Ergebnis des Durchschnitts wird dann auf das originale Bild gelegt, sodass wenn ein Pixel $\hat{p} \in [0, 1]$ größer als eine Schranke $T \in [0, 1]$ ist, dieser Bereich im originalen Bild blau gefärbt wird.

4 Auswertung

In diesem Kapitel wird die Leistung des konzipierten Programms ausgewertet. Alle folgenden Tests verwenden 5-fache Kreuzvalidierung und nehmen den Dice-Koeffizienten mit Schranke 0.5 als Metrik. Dabei steht das i-te *fold* für die erzeugte Validierungsmenge V_i mit $1 \leq i \leq 5$. Eine 10-fache oder sogar *Leave-one-out* Kreuzvalidierung hätte die Trainingszeit der neuronalen Netzwerke verdoppelt oder sogar verdreifacht. Aus diesem Grund wurde i auf ≤ 5 gesetzt. Welche Bilder in den jeweiligen *folds* enthalten sind, ist von geringer Bedeutung, da der Durchschnitt der Ergebnisse stets als Indikator für die Leistung genommen wird. Die Unterschiede zwischen den *folds* geben jedoch Aufschluss darüber, wie die Leistung variieren kann.

Für die Tests wird SGD genutzt, da adaptive Methoden wie Adam häufig zu suboptimalen Ergebnissen führen [16, 37, 22]. *Momentum* wird auf 0.9 und Gewichtsverfall auf 0.0005 gesetzt. Außerdem wird die Batchgröße auf $b = 10$ gesetzt. Zum Trainieren wird eine GeForce GTX 980 verwendet und das neuronale Netzwerk selbst wurde in TensorFlow und Keras implementiert.

Wie schon zuvor erwähnt wurde, wird ein Durchlauf durch alle Batches (d. h. allen Trainingsdaten) im Kontext von neuronalen Netzwerken als eine *Epoche* bezeichnet. Dabei wird üblicherweise ein Batch konstruiert, indem zufällig b Bilder ohne Zurücklegen aus den Trainingsdaten gezogen werden.

Im Kapitel zur Präparierung der Daten wurde davon gesprochen, dass ein *sliding window* genutzt wird, um die Bildausschnitte zu generieren. Durch das Nutzen von verschiedenen Startpunkten des *sliding window*, steigt die Zahl der Trainingsdaten auf knapp 100000. Bei weniger Startpunkten ist demnach auch die Trainingszeit geringer, aber es ergeben sich Einbußen in der Leistung.

Da im Durchschnitt ein Trainingsvorgang $5 - 10$ Stunden dauert und wir 5 *folds* nutzen, benötigt ein Test max. 50 Stunden. Es müssen mindestens 25 Tests durchgeführt werden, um alle Aspekte zu betrachten. Das heißt insgesamt würde es $50 \cdot 25 = 1250 \approx 52$ Tage dauern bei einem GPU. Eine künstliche Verringerung der Trainingsdaten kann die Laufzeit halbieren oder vierteln, sodass vielleicht noch auf 10 Tage gekommen wird. Dieses betrachtet aber nicht die Probleme, die während des Trainierens auftreten können. Häufig muss manuell eingegriffen werden, da Ergebnisse die Annahmen beeinflussen und sich dadurch der Code auch ändern kann. Zum Beispiel wurde der Einfluss des Klassenungleichgewichts überschätzt, sodass die Lernrate verringert werden musste. Oder ein anderes Beispiel sind Fehler beim Threading, die dazu führten, dass viele Tests neu durchgeführt werden mussten. Ein Server könnte also nicht kontinuierlich laufen und müsste stetig unterbrochen werden, um die Korrekturen am Code vorzunehmen.

Dieses Problem der langen Laufzeit wurde zum Schluss durch Teilepochen und Ziehen mit Zurücklegen gelöst. Nach jeden 1000 Bildern wird eine komplette Validierung durchgeführt, wobei wenn sich 10 Mal in Folge keine Verbesserung des Dice-Koeffizienten ergibt, das Trainieren abgebrochen wird. Es werden nicht in einer Teilepoche alle Bilder betrachtet und es werden immer die Batches mit Zurücklegen generiert. Da allerdings eine Gleichverteilung vorliegt, hat jedes Bild zumindest die gleiche Wahrscheinlichkeit gezogen zu werden. Es hat sich gezeigt, dass diese Methode praktisch die gleichen Ergebnisse liefert, als wenn man die gesamte Epoche abwartet und ohne Zurücklegen zieht. Auf diese Weise dauert ein Test höchstens 1 Stunde und es muss nicht die Zahl der Bilder verringert werden. Alle Tests haben also insgesamt knapp $1 - 2$ Tage in Anspruch genommen. Die Verwendung von Ziehen ohne Zurück-

legen statt Ziehen mit Zurücklegen hat in Tests keinen Unterschied gemacht.

4.1 Modelle

Es wird mit einer Lernrate von 10^{-6} angefangen, da wie bei [38] ein hohes Klassenungleichgewicht besteht. Außerdem werden Kreuzentropie und zufällige Batches mit einer Batchgröße von 10 verwendet. Es stehen zwei Architekturen zur Verfügung: Modell 1 (U-Net mit zufälligen Gewichten) und Modell 2 (U-Net mit DenseNet).

Fold	1. Durchlauf	2. Durchlauf	3. Durchlauf
1	0.15	**0.24**	0.00
2	**0.07**	0.06	**0.07**
3	0.19	**0.30**	0.01
4	0.07	**0.12**	**0.12**
5	**0.29**	0.27	0.20
Durchschnitt	0.154	**0.198**	0.08

Tabelle 1: Modell 1 mit 32 Startfiltern, 128 Stoppfiltern, Graustufen, $\delta = 1, \alpha = 0.5, \gamma = 0$

Fold	1. Durchlauf	2. Durchlauf	3. Durchlauf
1	**0.15**	0.13	0.11
2	0.03	**0.06**	0.04
3	**0.22**	0.17	0.20
4	**0.08**	0.07	0.07
5	0.10	**0.17**	0.12
Durchschnitt	0.116	**0.12**	0.108

Tabelle 2: Modell 2 mit $\delta = 1, \alpha = 0.5, \gamma = 0$ und ImageNet-Gewichten

Es wurde dreimal jeweils das gleiche Modell trainiert. Wie man an den Tabellen sieht, hat Modell 1 eine höhere Varianz als Modell 2. Es ergeben sich starke Schwankungen auch bei anderen Einstellungen wie 256 bzw. 512 Stoppfiltern und verschiedenen Verlustfunktionen.

Eine Erhöhung der Batchgröße von 10 auf 16 und die Änderung der Batchstrategie hat bei Modell 1 keine Stabilisierung der Gradienten erbracht. Um zu überprüfen, ob die höhere Varianz durch die zufälligen Gewichte entsteht, wurde Modell 2 erneut ohne ImageNet-Gewichte trainiert.

Fold	1. Durchlauf	2. Durchlauf	3. Durchlauf
1	0.18	**0.26**	0.21
2	**0.04**	**0.04**	0.01
3	**0.18**	0.16	0.17
4	0.07	**0.09**	0.07
5	0.14	**0.17**	0.14
Durchschnitt	0.122	**0.144**	0.12

Tabelle 3: Modell 2 mit $\delta = 1, \alpha = 0.5, \gamma = 0$ und zufälligen Gewichten

Modell 2 mit und ohne ImageNet-Gewichte ergeben eine ähnliche Leistung. Daher sind es nicht die Gewichte, die zu Instabilität bei Modell 1 führen. Die Tiefe des Netzwerks kann auch ausgeschlossen werden, da die Nutzung von 1024 Stoppfiltern bei Modell 1 zu keiner Stabilisierung führt. Daher ist zu mutmaßen, dass die höhere Konnektivität innerhalb der Encoder-Blöcke von DenseNet eine stabilisierende Wirkung hat.

In einem weiteren Test wurde die Lernrate gesenkt und δ auf 0.5 gesetzt. Auf diese Weise wird auch der zweite Term in $\mathcal{L}(p, \hat{p})$ aktiviert.

Fold	Modell 2: ImageNet-Gewichte	Modell 2: zufällige Gewichte	Modell 1: 1024 Stoppfilter
1	**0.37**	0.18	0.31
2	**0.12**	0.04	0.02
3	**0.48**	0.24	0.01
4	**0.16**	0.14	0.12
5	**0.38**	0.14	0.28
Durchschnitt	**0.302**	0.148	0.148

Tabelle 4: Lernrate 10^{-4}, $\delta = 0.5$, $\alpha = 0.5$, $\gamma = 0$, $\beta = 0.5$

Es ist zu sehen, dass in diesem Fall die ImageNet-Gewichte bei Modell 2 einen höheren Dice-Koeffizienten ergeben. Im Vergleich ist bei Modell 1 durchschnittlich die Leistung gleichgeblieben. Durch die 1024 Stoppfilter hat Modell 1 jetzt über 31 Millionen Parameter.

Bis zu diesem Punkt ist also folgendes festzuhalten:

- Modell 1 ist relativ instabil und erreicht nie einen höheren Dice-Koeffizient als Modell 2 unter den besten Einstellungen.
- Modell 2 ist stabil und erreicht $\geq 30\%$ Dice-Koeffizient. Die ImageNet-Gewichte verbessern die Leistung bei richtiger Wahl der Lernrate und Verlustfunktion.

In den folgenden Unterkapiteln wird sich nur noch auf Modell 2 konzentriert, da Modell 1 schlechtere Ergebnisse liefert und zu instabil ist.

4.2 Batches

Die Lernrate wird erneut gesenkt, da die Gradienten bei Modell 2 nicht zu stark schwanken. Die restlichen Einstellungen sind wie in Tabelle 4. Hier werden die Batchstrategien ausgewertet.

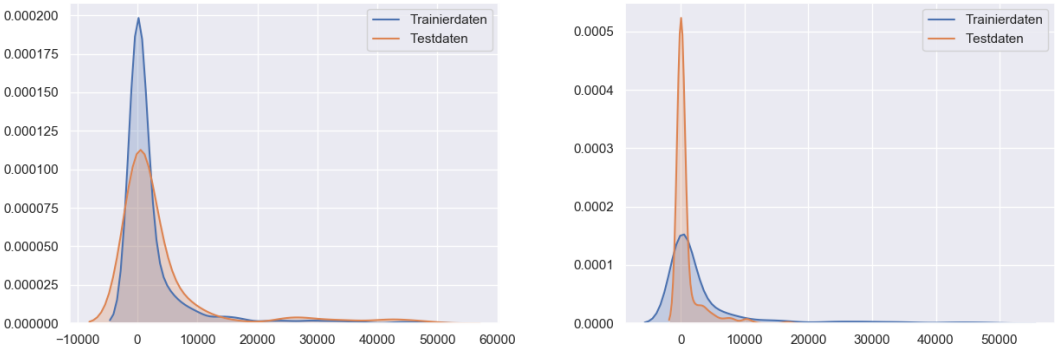
Fold	Zyklisch	Zufall	Sliding window
1	0.30	**0.43**	0.35
2	0.09	**0.17**	0.10
3	0.33	**0.47**	0.40
4	0.14	**0.18**	**0.18**
5	0.34	**0.44**	0.36
Durchschnitt	0.24	**0.338**	0.278

Tabelle 5: Lernrate 10^{-3}, $\delta = 0.5$, $\alpha = 0.5$, $\gamma = 0$, $\beta = 0.5$

Wie man aus der Tabelle ablesen kann, wird die beste Leistung erreicht, wenn man die Bilder zufällig anordnet. Ein Rückblick auf die Abbildung 10 zeigt, dass die beiden besten Einstellungen „Zufall“ und „Sliding window“ sich nur durch die Zahl der Ausreißer in den Batches unterscheiden. Allerdings gibt es immer noch ungefähr gleich viele schwarze und weiße Pixel bei beiden Methoden in den Batches.

Die bessere Leistung der zufälligen Methode kann anhand der unterschiedlichen Wahrscheinlichkeitsverteilungen erklärt werden. Häufig wird bei ML Algorithmen davon ausgegangen, dass die Trainier- und Testverteilungen i.i.d sind [1, 23]. Wenn die Trainingsdaten anfangs die gleiche Verteilung wie die Testdaten haben, dann führt die Gewichtung der Samples zu einer Verschlechterung der Leistung. Bei i.i.d würde die zyklische Strategie nicht das bewirken, was sie soll.

Allerdings sind die Verteilungen nicht unbedingt i.i.d. Dieses zeigen die folgenden Abbildungen:

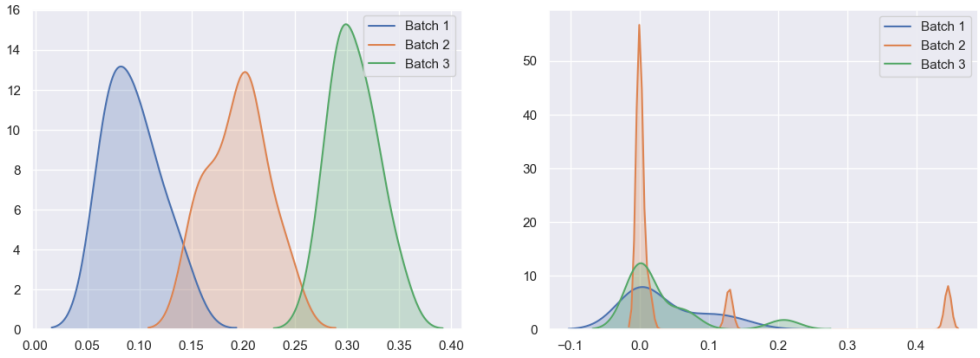


(a) Fold 1 (b) Fold 2

Abbildung 11: Kernel density estimation (KDE) von Fold 1 und 2 anhand der Zahl der weißen Pixel

Bei den beiden Abbildungen handelt es sich um die Verteilung der ganzen Daten. Wenn man von 0 absieht, haben in Abbildung a) die Trainingsdaten und Testdaten ungefähr die gleiche Verteilung. Dieses spiegelt sich auch in den Ergebnissen des CNN wider. In Abbildung b) sind die Testdaten weniger von Rocaille-Ornamenten bedeckt (mehr „leere“ Bildausschnitte). In a) ist die Situation genau anders herum. Außerdem ist der Übergang von 0 zu 10000 bei den Testdaten in b) unebener.

Eine Gewichtung der Trainiersamples kann also durchaus von Vorteil sein. Allerdings ist die Gewichtung so wie sie bei der zyklischen Variante durchgeführt wird, nicht zielführend. Jedes Batch enthält immer nur $10\%, \ldots, 90\%$ weiße Pixel (Rocaille). Die Batchverteilung entspricht weder den Trainingsdaten noch den Testdaten. Dieses zeigen auch die folgenden Abbildungen:



(a) Zyklisch (b) Zufällig

Abbildung 12: KDE von drei Batches

Bei (b) sind wie bei den ganzen Test-/Trainingsdaten die meisten Bildausschnitte bei 0 vertreten. Allerdings entspricht die Zahl der Bildausschnitte nicht vollständig den der richtigen Verteilungen. Mithilfe der Verlustfunktion wird versucht in dem nächsten Abschnitt die Verteilung der Batches im CNN zu korrigieren. Trotzdem kann man bereits feststellen, dass die zufällige Strategie besser die Trainings- bzw. Testverteilung annähert.

4.3 Verlustfunktion

Die Gewichtung der Samples kann mittels der Parameter der Verlustfunktion durchgeführt werden.

Fold	$\alpha = 0.75$	$\alpha = \frac{\sum_i 1-p_i}{\sum_i p_i + \sum_i 1-p_i}$	$\alpha = 0.25$	$\alpha = \frac{\sum_i p_i}{\sum_i p_i + \sum_i 1-p_i}$
1	0.37	0.41	**0.44**	0.41
2	0.16	0.12	0.13	**0.17**
3	0.44	0.42	**0.47**	0.33
4	0.17	**0.18**	0.16	**0.18**
5	0.43	**0.47**	0.39	0.34
Durchschnitt	0.314	**0.32**	0.318	0.286

Tabelle 6: Lernrate 10^{-3}, $\delta = 0.5$, $\gamma = 0$, $\beta = 0.5$

Die α-Werte geben entweder den schwarzen oder weißen Pixeln eine höhere Gewichtung. Wie man sieht, hat die Gewichtung mit allen schwarzen Pixeln („Nicht-Rocaille“) in dieser Tabelle die beste Leistung. Das bedeutet, wenn ein Batch z. B. 90% schwarze Pixel enthält, dann werden die weißen Pixel („Rocaille“) um 90% gewichtet. Da zuvor α mit 0.5 in Tabelle 5 einen Dice-Koeffizienten von 0.338 gegeben hat, haben die verschiedenen α-Werte zu keiner Verbesserung geführt.

Als Nächstes werden verschiedene Werte für γ getestet. Diese Variable gibt leichteren Pixeln eine geringe Gewichtung als schwereren Pixeln. Dabei wird α auf 0.5 belassen.

Fold	$\gamma = 0.5$	$\gamma = 2$
1	**0.46**	**0.46**
2	**0.20**	0.16
3	0.34	**0.50**
4	0.17	**0.22**
5	0.42	**0.44**
Durchschnitt	0.318	**0.356**

Tabelle 7: Lernrate 10^{-3}, $\alpha = 0.5, \delta = 0.5, \beta = 0.5$

Man sieht, dass $\gamma = 2$ in diesem Fall am besten ist. Jetzt muss noch β in der Verlustfunktion optimiert werden. Hiermit wird die Gewichtung von false positives (FP) gegenüber false negatives (FN) angegeben.

Fold	$\beta = 0.75$	$\beta = 0.25$
1	0.36	**0.38**
2	**0.14**	0.12
3	**0.44**	0.43
4	0.19	**0.20**
5	**0.40**	0.35
Durchschnitt	**0.306**	0.296

Tabelle 8: Lernrate 10^{-3}, $\alpha = 0.5, \delta = 0.5, \gamma = 2$

β kann somit auf 0.5 bleiben. Das bedeutet also zusammenfassend, dass lediglich γ zu einer Verbesserung in der Leistung führt. Es wurden noch in weiteren Tests andere Werte für die Variablen α, β, γ und δ ausprobiert, jedoch konnte keine Verbesserung in der Leistung erzielt werden. Es ist durchaus möglich durch *Grid search* oder anderen Optimierungsalgorithmen noch kleine Erhöhungen ($\leq 1\%$) des Dice-Koeffizienten zu erreichen. Durch die hohe Rechenleistung und Laufzeit, die aber solche Algorithmen benötigen, wurde hier auf diese verzichtet.

Mit Bezug auf den letzten Abschnitt hat eine Gewichtung der weißen oder schwarzen Pixel innerhalb des neuronalen Netzwerks nicht zu dem gewünschten Effekt geführt. Es ist zu beobachten, dass höhere α-Werte tendenziell bessere Ergebnisse erzielen. Wenn $\alpha < 0.1$ ist (wie z. B. bei der letzten Spalte in Tabelle 6), werden weiße Pixel zu schwach gewichtet. Eine solche Gewichtung würde zwar eher den Wahrscheinlichkeitsverteilungen in Abbildung 11 entsprechen, aber das Finden des Minimums der Optimierungsfunktion gestaltet sich schwieriger. Die externe Gewichtung mittels Batches ist in diesem Fall von größerer Bedeutung als die interne Gewichtung mittels Verlustfunktion. Dies lässt auch darauf schließen, dass das Klassenungleichgewicht sich dank der zufälligen Batches nicht beim Netzwerk bemerkbar macht.

In Papers wie [38] konnte das Klassenungleichgewicht nur durch die Verlustfunktion gelöst werden, da jedes der Bilder durchschnittlich ein 10%-zu-90%-Klassenverhältnis hat. Hier wird die Optimierung erleichtert, dadurch dass ein Großteil der Bilder keine Rocaille-Ornamente enthält. Die *ground truth* sind häufiger komplett schwarze Bilder.

4.4 Fine tuning

Basierend auf dem Modell mit $\gamma = 2$ in Tabelle 8 wird *fine tuning* betrieben. Dazu werden alle Gewichte freigeschaltet und das Netzwerk wird erneut trainiert.

Fold	fine tuning
1	0.41
2	0.10
3	0.56
4	0.25
5	0.38
Durchschnitt	0.34

Tabelle 9: Lernrate 10^{-3}, $\alpha = 0.5, \delta = 0.5, \gamma = 2$

Überraschenderweise hat das *fine tuning* keine Verbesserung erbracht. Eine mögliche Erklärung ist, dass die Daten relativ schwierig für das CNN sind und die zusätzlichen Informationen nicht erlernt werden können. Wenn die ersten Schichten eingefroren werden, dann muss das CNN nicht versuchen diese Informationen zu erlernen und es kann sich nur auf das Hochsamplen der *feature map* konzentriert werden. Das Freischalten der Gewichte erschwert also das Lernen.

Es ist möglich, dass wenn die vortrainierten Gewichte näher an dem eigentlichen Datenset wären, dass dann das *fine tuning* zu besseren Ergebnissen führen würde. Die Bilder von ImageNet und die gegebenen Rocaille-Bilder sind im Vergleich sehr unterschiedlich. Zum einen bestehen die Rocaille-Bilder nur aus Graustufen, sodass die RGB-Kanäle keine zusätzlichen Informationen geben. Zum anderen sind die Bilder in einer einheitlichen Umgebung aufgenommen worden. Wenn man sich vergleichend die ImageNet-Bilder anschaut, dann findet man z. B. Tiere in allen möglichen Positionen und Größen. Das neuronale Netzwerk kann deshalb hier nur wenig von den ersten Schichten des Netzwerks mit ImageNet-Gewichten profitieren. Wichtiger sind die späteren Schichten, die spezifischer sind. Die ImageNet-Gewichte geben also einige Informationen über die Konturen, aber die Entfernung zwischen Kunstbildern und normalen Bildern ist zu groß.

So wie es die ImageNet-Datenbank gibt, so gibt es auch für Kunstbilder spezifische Datenbanken, die teils Millionen von Bildern enthalten. Es lässt sich vermuten, dass das Vortrainieren auf einer solchen großen Bilderdatenbank, die Leistung des Netzwerks steigern kann. Die ImageNet-Gewichte haben knapp $10\% - 15\%$ Verbesserung gebracht, daher könnte das Vortrainieren auf Kunstbildern eine noch größere Steigerung mit sich bringen. Auf ImageNet zu trainieren kann je nach Zahl an GPUs allerdings bis zu einem Monat dauern. Zum Beispiel gibt [8] 29 Stunden Trainierzeit auf 8 Tesla P100 GPUs an, während es bei [25] eine Woche gedauert hat. Aufgrund der notwendigen Rechenleistung wurde in dieser Arbeit auf ein Vortrainieren auf eine Kunstbilder-Datenbank verzichtet.

4.5 Bildgröße

Als Nächstes wurde die Eingabegröße des neuronalen Netzwerks auf 448x448 erhöht. Ansonsten wurden die besten Einstellungen von Tabelle 8 beibehalten.

Fold	448x448
1	0.40
2	0.18
3	0.42
4	0.16
5	0.48
Durchschnitt	0.328

Tabelle 10: Lernrate 10^{-3}, $\alpha = 0.5, \delta = 0.5, \gamma = 2$

Demnach führt eine Erhöhung der Bildgröße nicht zu besseren Ergebnissen. Da stets Verschiebungen beim Trainieren verwendet werden, „sieht“ das neuronale Netzwerk bereits über den 224x224 Bereich hinaus. Daher führt die Erhöhung auf 448x448 nur zu einer Verlangsamung des Trainierens. Außerdem werden bei der Vorhersage im übernächsten Abschnitt sowieso mehrere *sliding windows* verwendet. Aus diesem Grund kann die Bildgröße vernachlässigt werden, weil sie keinen positiven Einfluss auf die Leistung hat.

4.6 Einfluss der Datenmenge

Um den Effekt einer größeren Datenmenge zu überprüfen, wird dieses Mal keine Kreuzvalidierung verwendet. Stattdessen werden die 17 Bilder in zwei Teile eingeteilt: Bilder $10 - 17$ sind die Testmenge, die restlichen Bilder sind Trainingsdaten. Dabei wurden auch hier die Einstellungen aus Tabelle 8 entnommen.

Im ersten Test werden nur die ersten 4 Bilder zum Trainieren verwendet, während 6 Bilder nicht genutzt werden. Als Ergebnis erhält man 32.2%.

In einem zweiten Test wurde die Zahl der Trainingsbilder auf 8 verdoppelt, 2 Bilder werden nicht genutzt. Das Ergebnis ist 34.3%.

In einem dritten Test wurden alle 10 Bilder verwendet. Man erhält 33.22%.

Wenn man bedenkt, dass in der Regel bei neuronalen Netzwerken Schwankungen von $1\% - 2\%$ normal sind, dann hat eine Erhöhung der Daten keine Verbesserung gebracht. Allerdings sind vielleicht wesentlich größere Erhöhungen notwendig, um wirkliche Unterschiede zu sehen. Die Trainingsdaten sind relativ verschieden. Auf der einen Seite gibt es Kunstbilder mit sehr vielen Linien und auf der anderen Seite Bilder mit wenigen aber großen Linien. Außerdem unterscheiden sich die Strukturen stets stark. Das bedeutet also, dass so viele Bilder annotiert werden müssten, bis zumindest mehrere Bilder sind ähneln.

4.7 Vorhersage

Nach dem Trainieren werden jetzt die Vorhersagen der Bilder erzeugt. Dazu werden die Bilder mittels *sliding window* zusammengesetzt. Zuvor bezog sich der Dice-Koeffizient stets nur auf Bildausschnitte, jetzt werden die Bilder als Ganzes betrachtet. Zur Auswertung werden jeweils die Gewichte für jedes *fold* geladen und das CNN wird auf die jeweilige Validierungsmenge angewendet. Dann wird der Durchschnitt gebildet. Zunächst werden Schranken in 0.05 Abständen ausprobiert.

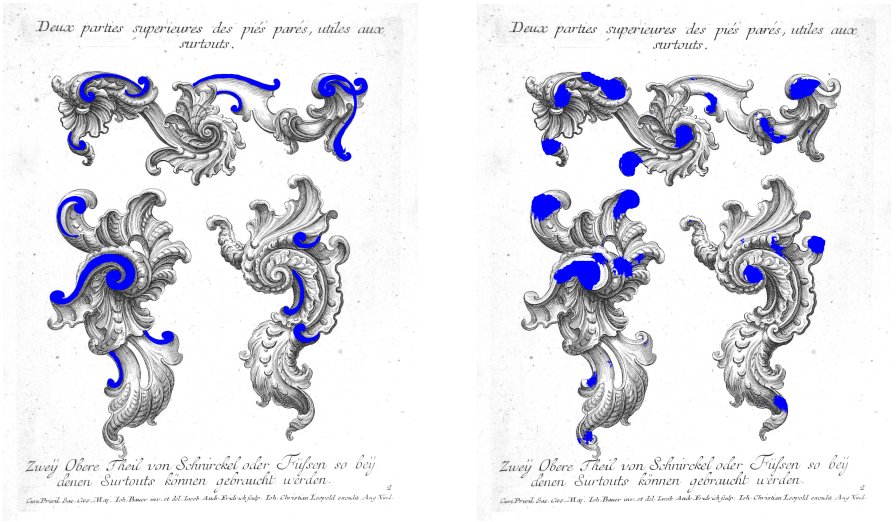
Schranke	Dice-Koeffizient
0.45	0.341
0.65	0.348
0.5	0.353
0.6	0.353
0.55	**0.355**

Wie man sieht, unterscheiden sich die Schranken nicht sehr stark. Da 0.55 die beste Leistung erreicht, wird dieser Wert für den nächsten Test genutzt. Bei diesem wird der Einfluss der Zahl der *sliding windows* ausgewertet (Iterationen).

Iterationen	Dice-Koeffizient
1	0.355
2	0.375
3	0.382
5	0.390
10	0.393
20	**0.395**

Umso höher die Zahl der Iterationen, umso höher ist auch der Dice-Koeffizient. Während die Verwendung von zwei *sliding windows* die Leistung um 2% erhöht, sind es bei drei Fenstern nur noch knapp 0.7%. Ab 5 oder 10 Iterationen lohnt sich eine weitere Erhöhung der Iterationen nicht mehr, da auch die notwendige Berechnungszeit ansteigt. Zum Beispiel ergeben sich bei durchschnittlich 120 Fenstern pro Iteration und 20 Iterationen insgesamt $120 \cdot 20 = 2400$ Vorhersagen pro Bild.

Das folgende Bild gibt ein Beispiel, wie die Ausgabe zum Schluss aussieht:



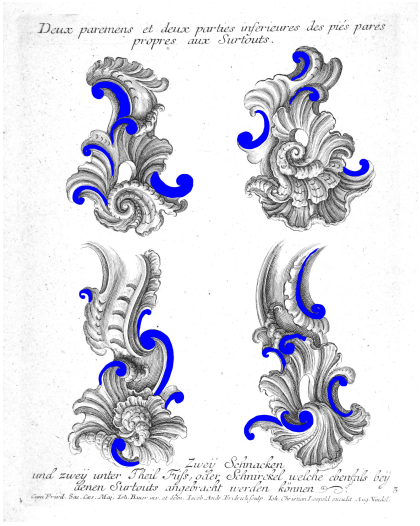
(a) Ground truth (b) Vorhersage

Abbildung 13: Bild (a) ist die erwartete Ausgabe und (b) ist die Vorhersage

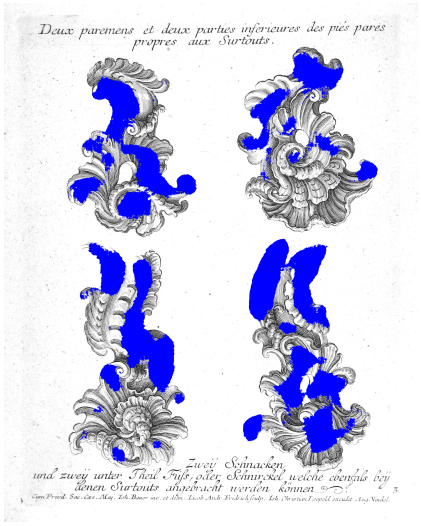
In vielen Bereichen sind noch Fehler vorhanden, aber zumindest ist das neuronale Netzwerk imstande die ungefähre Position der Ornamente zu lokalisieren. Statt exakte

Striche wie in (a), ergeben sich in (b) vom neuronalen Netzwerk eher Farbkleckse. Es ist davon auszugehen, dass eine größere Zahl an Bildern notwendig ist, um noch exaktere Vorhersagen machen zu können.

Auch bei anderen Bildern zeigt das neuronale Netzwerk ein ähnliches Verhalten auf.



(a) Ground truth



(b) Vorhersage

Abbildung 14: Bild (a) ist die erwartete Ausgabe und (b) ist die Vorhersage

Wieder entstehen Farbkleckse in den Bereichen, wo Linien eigentlich stehen müssten. Des Weiteren werden bei diesem Bild noch mehr *false positives* gefunden.

5 Fazit

In dieser Arbeit wurde ein neuronales Netzwerk konzipiert, welches imstande ist Rocaille-Ornamente in Kunstbildern zu finden. In der Theorie wurden zunächst Techniken wie u. a. Gewichtsverfall, Batchnormalisierung, Konkatenation und Faltung definiert, die für das Netzwerk benötigt wurden.

Dann kam im nächsten Kapitel die Präparierung der Daten, die Struktur der Architektur, die Verlustfunktion und die Nachbereitung der Daten. Da im Voraus nicht zu wissen war, welche Konfiguration, die beste Leistung erzeugen würde, wurden mehrere Möglichkeiten vorgestellt.

In der Auswertung hat sich dann gezeigt, dass die folgende Konfiguration, die beste Leistung ergibt:

- Architektur: U-Net mit DenseNet als Encoder, ImageNet-Gewichte und Eingabegröße $224 \times 224 \times 3$
- Verlustfunktion: Mischung aus Focal Loss mit $\gamma = 2, \alpha = 0.5$ und Dice-Koeffizient
- Lernrate: 10^{-3}
- Batches: zufällig
- Nachbearbeitung: ≥ 5 Iterationen

Das Hinzufügen von Focal Loss hat zwar eine kleine Verbesserung gebracht, aber der Dice-Koeffizient innerhalb der Verlustfunktion war dennoch relevanter. Wichtig war auch insbesondere die Architektur, die Lernrate und die Nachbearbeitung. Interessanterweise haben die ImageNet-Gewichte, trotz der Verwendung von Kunstbildern, die Leistung verbessert. Besondere Batchstrategien und Tversky-Loss haben hier nur die Leistung verschlechtert.

Das Netzwerk erreicht auf 5 Folds betrachtet ungefähr einen Dice-Koeffizienten von 40%, wobei die Prozentzahl um 2% bis 3% variieren kann. Die Bilder, die in der Auswertung gezeigt wurden, machen deutlich, dass die Leistung zum bisherigen Zeitpunkt noch nicht ausreicht. Das Netzwerk ist lediglich imstande ungefähr die Position der Linien zu lokalisieren, macht aber auch hier Fehler.

5.1 Ausblick

Es ist im Laufe der Arbeit deutlich geworden, dass die geringe Datenmenge von 17 Bildern nicht ausreicht, um die Ornamente exakt zu bestimmen. Außerdem eignet sich die Nutzung von ImageNet nur begrenzt für Kunstbilder. Es gab zwar merklich Verbesserungen in der Leistung, aber trotzdem konnte nie ein Dice-Koeffizient von über 35% auf den Bildausschnitten erreicht werden.

Wenn man ein Blick auf andere Bereiche des Machine Learning wie *Natural Language Processing* (NLP) wirft, dann zeigt sich auch hier die Bedeutung von vortrainierten Gewichten. So machen Artikel wie „NLP's ImageNet moment has arrived“ [27] darauf aufmerksam, dass auch abseits der Bildverarbeitung die Wichtigkeit von guten Startpunkten bei den Gewichten erkannt wurde. Beim Trainieren eines neuronalen Netzwerks stehen fast nie große Datenmengen zur Verfügung, daher wird zwangsläufig auf *transfer learning* gesetzt.

In einer weiteren Arbeit müssten daher basierend auf einer großen Kunstbilder-Datenbanken vortrainierte Gewichte für eine Architektur erzeugt werden. Diese Gewichte könnten dann nicht nur für spezifische Aufgaben wie der Lokalisierung von Rocaille-Ornamenten genutzt werden, sondern auch allgemein kann die Segmentierung oder Objekterkennung in der Kunst betrachtet werden. Zum Zeitpunkt des Schreibens waren solche Gewichte noch nicht im Internet zu finden, wodurch lediglich auf ImageNet gesetzt werden konnte.

Weitere Möglichkeiten zur Verbesserung der Leistung liegen in der Erhöhung der verfügbaren Rocaille-Bilder und dem Testen von anderen Konfigurationen. Jedoch hat die Auswertung ergeben, dass Aspekte wie Verlustfunktionen oder Batchstrategien nur kleine Unterschiede in der Leistung machen. Der Hauptfokus sollte daher auf die Zahl der Bilder gesetzt werden.

Zusammenfassend eignen sich CNN auch zum Finden von Rocaille-Ornamenten. Die bisherigen Ergebnisse sind noch unzureichend für die tatsächliche Anwendung in der Kunst, aber durch mehr Daten und der Verwendung von anderen vortrainierten Gewichten sollten gute Leistungen machbar sein.

Literatur

[1] Steffen Bickel, Michael Brückner und Tobias Scheffer. „Discriminative Learning Under Covariate Shift“. In: *The Journal of Machine Learning Research* 10 (Dez. 2009), S. 2137–2155.

[2] Liang-Chieh Chen u. a. „Attention to Scale: Scale-aware Semantic Image Segmentation“. In: *CoRR* abs/1511.03339 (2015).

[3] Liang-Chieh Chen u. a. „Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation“. In: *CoRR* abs/1802.02611 (2018).

[4] J. Deng u. a. „ImageNet: A large-scale hierarchical image database“. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Juni 2009, S. 248–255.

[5] Xavier Glorot und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Hrsg. von Yee Whye Teh und Mike Titterington. Bd. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, Mai 2010, S. 249–256.

[6] Gabriel Goh. „Why Momentum Really Works“. In: *Distill* (2017). URL: http://distill.pub/2017/momentum.

[7] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[8] Priya Goyal u. a. „Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour“. In: *CoRR* abs/1706.02677 (2017).

[9] Kaiming He u. a. „Mask R-CNN“. In: *CoRR* abs/1703.06870 (2017).

[10] Gao Huang, Zhuang Liu und Kilian Q. Weinberger. „Densely Connected Convolutional Networks“. In: *CoRR* abs/1608.06993 (2016).

[11] Vladimir I. Iglovikov u. a. „TernausNetV2: Fully Convolutional Network for Instance Segmentation“. In: *CoRR* abs/1806.00844 (2018).

[12] Vladimir Iglovikov, Sergey Mushinskiy und Vladimir Osin. „Satellite Imagery Feature Detection using Deep Convolutional Neural Network: A Kaggle Competition“. In: *CoRR* abs/1706.06169 (2017).

[13] Vladimir Iglovikov und Alexey Shvets. „TernausNet: U-Net with VGG11 Encoder Pre-Trained on ImageNet for Image Segmentation“. In: *CoRR* abs/1801.05746 (2018).

[14] Sergey Ioffe und Christian Szegedy. „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: *CoRR* abs/1502.03167 (2015).

[15] kaggle. *2018 Data Science Bowl*. 2019. URL: https://www.kaggle.com/c/data-science-bowl-2018 (besucht am 06. 01. 2019).

[16] Nitish Shirish Keskar und Richard Socher. „Improving Generalization Performance by Switching from Adam to SGD“. In: *CoRR* abs/1712.07628 (2017).

[17] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Advances in Neural Information Processing Systems 25*. Hrsg. von F. Pereira u. a. Curran Associates, Inc., 2012, S. 1097–1105.

[18] Hung Le und Ali Borji. „What are the Receptive, Effective Receptive, and Projective Fields of Neurons in Convolutional Neural Networks?“ In: *CoRR* abs/1705.07049 (2017).

[19] Tsung-Yi Lin u. a. „Feature Pyramid Networks for Object Detection“. In: *CoRR* abs/1612.03144 (2016).

[20] Tsung-Yi Lin u. a. „Focal Loss for Dense Object Detection“. In: *CoRR* abs/1708.02002 (2017).

[21] Yun Liu u. a. „Richer Convolutional Features for Edge Detection“. In: *CoRR* abs/1612.02103 (2016).

[22] Ilya Loshchilov und Frank Hutter. „Decoupled Weight Decay Regularization“. In: *CoRR* abs/1711.05101 (2019).

[23] Georgia McGaughey, W. Patrick Walters und Brian Goldman. „Understanding covariate shift in model performance“. In: *F1000Research* 5 (Okt. 2016), S. 597.

[24] Alexander Rakhlin, Alex Davydow und Sergey Nikolenko. „Land Cover Classification from Satellite Imagery with U-Net and Lovász-Softmax Loss“. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, Juni 2018.

[25] Joseph Redmon u. a. „You Only Look Once: Unified, Real-Time Object Detection“. In: *CoRR* abs/1506.02640 (2015).

[26] Olaf Ronneberger, Philipp Fischer und Thomas Brox. „U-Net: Convolutional Networks for Biomedical Image Segmentation“. In: *CoRR* abs/1505.04597 (2015).

[27] Sebastian Ruder. *NLP's ImageNet moment has arrived*. 2018. URL: http://ruder.io/nlp-imagenet/.

[28] Olga Russakovsky u. a. „ImageNet Large Scale Visual Recognition Challenge“. In: *CoRR* abs/1409.0575 (2014).

[29] Seyed Sadegh Mohseni Salehi, Deniz Erdogmus und Ali Gholipour. „Tversky loss function for image segmentation using 3D fully convolutional deep networks“. In: *CoRR* abs/1706.05721 (2017).

[30] Mark Sandler u. a. „Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation“. In: *CoRR* abs/1801.04381 (2018).

[31] Shibani Santurkar u. a. „How Does Batch Normalization Help Optimization?“ In: *CoRR* abs/1805.11604 (2018).

[32] Karen Simonyan und Andrew Zisserman. „Very Deep Convolutional Networks for Large-Scale Image Recognition“. In: *CoRR* abs/1409.1556 (2014).

[33] Ilya Sutskever u. a. „On the Importance of Initialization and Momentum in Deep Learning“. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, S. III-1139–III-1147.

[34] Jerome Friedman Trevor Hastie Robert Tibshirani. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. Second. Springer, 2009.

[35] Aravind Vasudevan, Andrew Anderson und David Gregg. „Parallel Multi Channel Convolution using General Matrix Multiplication“. In: *CoRR* abs/1704.04428 (2017).

[36] Shih-En Wei u. a. „Convolutional Pose Machines“. In: *CoRR* abs/1602.00134 (2016).

[37] Ashia C. Wilson u. a. „The Marginal Value of Adaptive Gradient Methods in Machine Learning“. In: *arXiv e-prints* (Mai 2017).

[38] Saining Xie und Zhuowen Tu. „Holistically-Nested Edge Detection“. In: *CoRR* abs/1504.06375 (2015).

[39] Jason Yosinski u. a. „How transferable are features in deep neural networks?“ In: *CoRR* abs/1411.1792 (2014).

[40] Guodong Zhang u. a. „Three Mechanisms of Weight Decay Regularization“. In: *CoRR* abs/1810.12281 (2018).