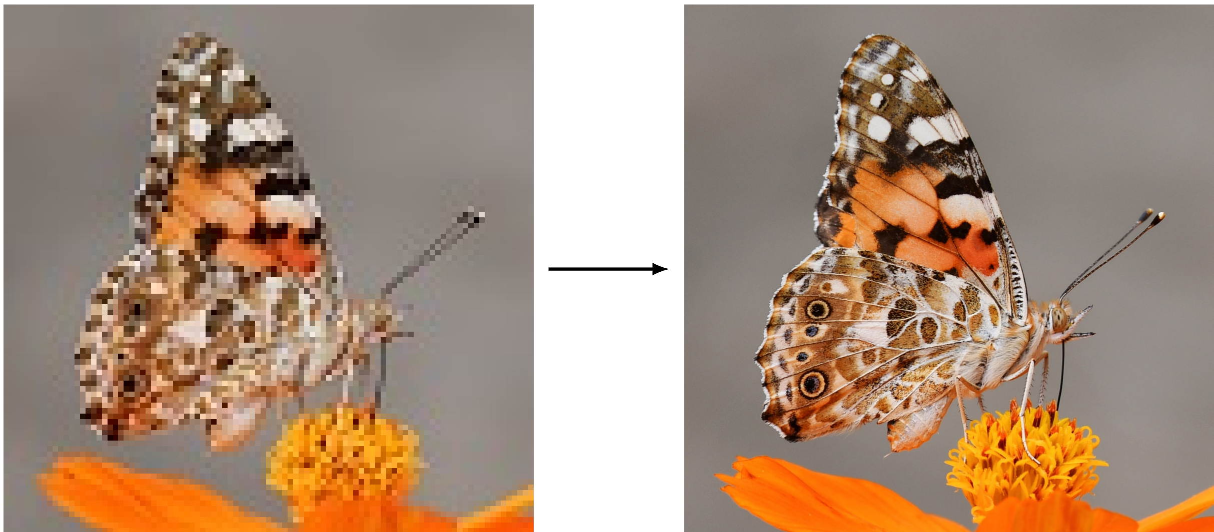
Entwicklung und Vergleich von Methoden zur Vergrösserung der Auflösung von digitalen Bildern (Super-Resolution)



Lukas Valentin Hüglin, M6e

Betreuer: Dr. Jörg Bader

Zürich, November 2022
Maturitätsarbeit an der Kantonsschule Zürich Nord

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Super-Resolution . . . 5
1.2 Problemstellung . . . 6

2 Methoden **7**
2.1 Super-Resolution mit Interpolationsmethoden . . . 7
2.2 Super-Resolution mit klassischen neuronalen Netzen . . . 8
2.3 Generative Adversarial Networks . . . 9
2.3.1 Schätzen der Parameter einer Normalverteilung . . . 10
2.3.2 Erzeugen einer Zufallsvariable nach vorgegebener Verteilung . . . 11
2.3.3 Maximum-Likelihood-Methode . . . 12
2.3.4 Die Aufgabe der Diskriminators . . . 13
2.3.5 Die Verlustfunktion des Diskriminators . . . 13
2.3.6 Die Verlustfunktion des Generators . . . 14
2.3.7 Der implementierte Algorithmus . . . 15
2.3.8 Ein anschauliches Beispiel . . . 15
2.4 SRGAN . . . 16
2.4.1 Architektur . . . 16
2.4.2 Convolutional Neural Networks . . . 17
2.4.3 Residual Networks . . . 18
2.4.4 Pixel-shuffle . . . 19
2.4.5 Verlustfunktion . . . 20
2.4.6 VGG19 . . . 21
2.5 Datensatz . . . 21
2.5.1 Unsplash . . . 22
2.5.2 Hierarchical Data Format . . . 22
2.5.3 Multiprocessing I . . . 22
2.6 Metriken . . . 23

3 Eigene Erweiterungen **23**
3.1 Aufbau des Programmes . . . 23
3.1.1 Modularität . . . 24
3.1.2 Multiprocessing II . . . 25
3.2 Problematik beim Training von GANs . . . 26
3.2.1 Bestrafungslösung . . . 27
3.3 Fourier-Verlustfunktion . . . 29
3.3.1 Fourier-Verlustfunktion für SRGAN . . . 30

4 Resultate und Diskussion **31**
4.1 Über den Trainingsvorgang . . . 31
4.2 Vergleich mit der originalen SRGAN Methode . . . 32
4.3 Die Trainingsprobleme von SRGAN . . . 33
4.4 SRGAN mit Fourier . . . 35
4.5 SRGAN mit limitiertem Diskriminator . . . 39
4.6 SRResNet . . . 42
4.7 Schlussfolgerung . . . 44

Zusammenfassung

In dieser Maturitätsarbeit wurde eine modulare Software entwickelt, mit welcher verschiedene Methoden zur Vergrösserung der Auflösung digitaler Bilder (Super-Resolution) implementiert und getestet werden können. Standardmethoden für Super-Resolution (SRResNet und SRGAN) beruhen auf Deep Learning, einem Teilgebiet des maschinellen Lernens, bei dem Zusammenhänge mit Hilfe von neuronalen Netzen modelliert werden. Beide Methoden, SRResNet und SRGAN, wurden in dieser Arbeit weiterentwickelt, indem eine Verlustfunktion eingebaut wurde, die auf einer zweidimensionalen Fourier-Transformation beruht (SRResNet Fourier und SRGAN Fourier). Ausserdem wurde der Trainingsprozess von SRGAN optimiert und eine Methode mit dem Namen SRGAN Limited entwickelt. Der Vergleich von meinen eigenen Methoden mit bestehenden Methoden für Super-Resolution zeigt, dass SRGAN Fourier und SRGAN Limited am besten sind. Das Training des normalen SRGAN ist zu instabil, um zuverlässig bei jedem Trainingsdurchgang ein gutes Resultat zu liefern. Welche Methode man zwischen SRGAN Fourier und SRGAN Limited bevorzugt ist situationsabhängig. SRGAN Fourier erzeugt leicht bessere Resultate, hat aber auch über die doppelte Trainingszeit. Hat man nur beschränkt Zeit, um die neuronalen Netze zu trainineren, dann ist SRGAN Limited sicherlich die bessere Lösung. Allerdings könnte man in zukünftigen Arbeiten die Trainingszeit von SRGAN Fourier nochmals stark reduzieren.

SRResNet Fourier lohnt sich nicht. Diese Methode liefert eher schlechtere Resultate als die anderen Methoden und benötigt auch eine lange Trainingszeit. Für Anwendungen, bei denen es wichtig ist keine, bzw. nur wenige Artefakte in einem Bild zu haben, ist SRResNet auch eine Option. Generell funktionieren alle Methoden für verschiedene Bilder unterschiedlich gut. Deshalb werden die besten Resultate erzielt, wenn das gewünschte Bild mit verschiedenen Methoden generiert und dann das Beste auswählt wird. Manchmal ist man überrascht, wie gut, aber auch wie schlecht, ein mit einer Super-Resolution Methode erzeugtes Bild aussehen kann.

1 Einleitung

In vielen Anwendungsbereichen werden Methoden der Bildbearbeitung eingesetzt, um die Auflösung von aufgenommenen Bildern zu vergrössern. Beispielsweise werden solche Methoden verwendet, um die Auflösung von Überwachungsbildern und -videos zu verbessern [37]. Wichtig sind solche Methoden auch in der Forschung, z.B. in der Mikrobiologie zur Verbesserung der Auflösung von Aufnahmen mit Mikroskopen [27, 29], oder in der Fernerkundung bei leistungsfähigen Verfahren der Objekterkennung [20]. Auch im medizinischen Bereich wird die Auflösung von MRI-Bildern mit Bildbearbeitungsmethoden erhöht, um die Interpretation der Bilder und so das Erstellen von Diagnosen zu erleichtern [12, 33, 15, 11]. Die Methode mittels Computerprogrammen die Auflösung eines Bildes zu vergrössern nennt man *Super-Resolution* (SR). In dieser Maturitätsarbeit werden gängige Super-Resolution Methoden vorgestellt und zwei der Methoden werden weiterentwickelt. Schliesslich wird die Leistungsfähigkeit der verschiedenen Methoden verglichen.

1.1 Super-Resolution

Super-Resolution ist ein Teilbereich der Bildbearbeitung und erzeugt aus Bildern mit tiefer Auflösung Bilder mit höherer Auflösung [23]. Dies bedeutet, dass jeder *Bildpunkt* (Pixel) in mehrere Bildpunkte unterteilt wird; aus einem Bildpunkt werden also mehrere Bildpunkte. In Abbildung 1 sind zwei Beispiele von Bildern gezeigt, die mit Super-Resolution erzeugt wurden. Um die Qualität des erzeugten Bildes (Super-Resolution) beurteilen zu können, sind in Abbildung 1 auch das Ausgangsbild mit tieferer Auflösung (Eingabe) sowie die „richtige Lösung“ (Re-

ferenz) dargestellt (mehr zur „richtigen Lösung“ in Abschnitt 1.2). Das Ausgangsbild (128×128 Pixel) wurde aus dem Referenzbild (512×512 Pixel) durch Herabsetzen der Bildauflösung erstellt. Der Super-Resolution Algorithmus vergrössert nun die Auflösung des Ausgangsbildes, das erzeugte Bild hat wie die Referenz eine Auflösung von 512×512 Pixel. [1]

Abbildung 1: Das Eingabebild (128×128) wird per Super-Resolution zum generierten Bild vergrössert (512×512).

1.2 Problemstellung

Stellen Sie sich vor, Sie haben ein Bild von einer Landschaft aufgenommen; ganz klein am Horizont steht ein Aussichtsturm. Die Spitze des Turmes ist nur durch einen Bildpunkt dargestellt. Wir kennen daher nur die durchschnittliche Farbe der Turmspitze. Ob sich eine weisse Taube, ein roter Schirm, oder eine Person mit blauer Mütze auf dem Turm befindet lässt sich aus dem Bild *nicht* erschliessen. Egal wie gut ein Algorithmus für Super-Resolution ist, er kann nicht vorhersagen, wie der Turm zum Zeitpunkt der Aufnahme in Wirklichkeit aussah. Aber der SR-Algorithmus könnte eine weisse Taube, einen roten Schirm oder eine blaue Mütze erfinden. Das Bild mit höherer Auflösung ist also nicht *eindeutig*. Unsere Aufgabe besteht nun darin, *ein* solches passendes Bild von vielen passenden Bildern zu finden.

[1] In der gedruckten Version dieser Maturitätsarbeit sind die Unterschiede in den gezeigten Bildern nur schwer zu erkennen. Es wird empfohlen die Bilder in der elektronischen Version (PDF) zu betrachten.

2 Methoden

2.1 Super-Resolution mit Interpolationsmethoden

Wir wollen also von einem niedrig aufgelösten Eingabebild X ein hochaufgelöstes Ausgabebild Y erzeugen. Dazu werden sozusagen zwischen den Bildpunkten von X, wie in Abbildung 2, neue Bildpunkten eingeschoben.

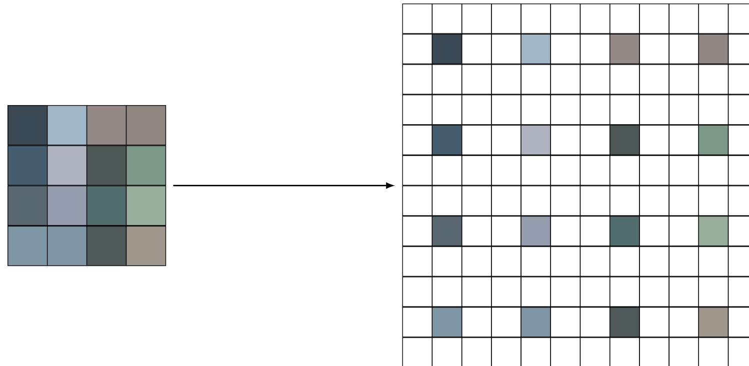


Abbildung 2: Zwischen die schon vorhandenen Pixel müssen weitere Pixel eingefügt werden.

Wir müssen also zwischen den Farbwerten der Bildpunkte neue Farbwerte erzeugen. Diesen Vorgang nennt man Interpolation. Ein Bild ist natürlich zweidimensional, das heisst man muss in zwei Dimensionen interpolieren. Als Vereinfachung betrachten wir aber eine Interpolation an einer Pixelreihe mit nur einem Farbkanal.

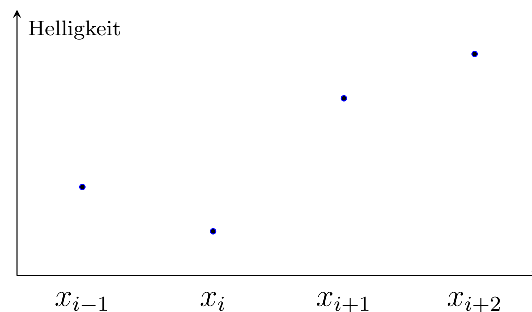


Abbildung 3: Zwischen dieser Reihe von Bildpunkten soll interpoliert werden.

Wir haben ja schon gesehen, dass Super-Resolution nicht eindeutig ist, aus einem schwarzen Pixel am Horizont könnten wir ein Auto oder einen Felsen zeichnen, es ist nicht eindeutig identifizierbar, was es eigentlich ist. Genau gleich ist es bei einer Interpolation. Wir können zwei benachbarte Punkte in Abbildung 3 unterschiedlich verbinden. Ein sinnvoller Start wäre z.B. eine Verbindung mit einer Geraden.

Dies nennt man eine *lineare* Interpolation [13]. Angewandt auf das zweidimensionale Bild wird sie dann zu einer *bilinearen* Interpolation. Wie man in der Abbildung 4 sieht, ist diese Interpolationsart etwas eckig und nicht glatt. Da eine Gerade bereits mit zwei Punkten definiert ist, werden für die lineare Interpolation auch nur die zwei Nachbarpunkte miteinbezogen. Alle anderen Bildpunkte haben keinen Einfluss. Dies ändert sich mit der *kubischen* bzw. *bikubischen* (engl. *bicubic*) Interpolation [5]:

Da ein kubisches Polynom erst durch 4 Punkte vollständig definiert ist, haben nicht nur zwei, sondern vier Punkte einen Einfluss. Wollen wir z.B. zwischen x_{i-1} und x_{i+2} kubisch interpolieren, dann müssen wir die gewünschte Kurve trotzdem in drei Polynome unterteilen, wie auch schon bei der linearen Interpolation. Anstelle einer Geraden (Polynom ersten Grades)

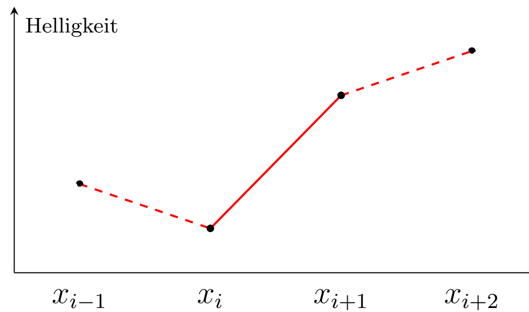


Abbildung 4: Eine lineare Interpolation zwischen x_i und x_{i+1}.

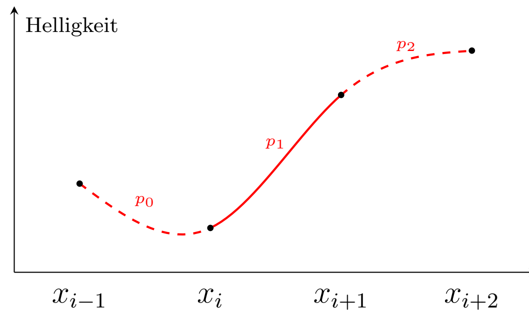


Abbildung 5: Eine kubische Interpolation zwischen x_i und x_{i+1}.

verwenden wir aber nun je ein kubisches Polynom (3. Grades). Betrachten wir nun das Teilpolynom p_1 im Intervall $[x_i, x_{i+1}]$. Dieses Polynom 3. Ordnung muss durch x_i und x_{i+1} gehen. Zusätzlich müssen die erste Ableitung dieses Polynoms p_1 an der Stelle x_i mit der ersten Ableitung des Interpolationspolynoms p_0 übereinstimmen [26]. Analog müssen auch die Ableitungen für die Grenze bei x_{i+1} der Polynome p_1 und p_2 übereinstimmen. Somit ist die ganze Kurve über dem Intervall $[x_{i-1}, x_{i+2}]$ stetig differenzierbar, bzw. die Kurve, sowie das Bild sind glatt.

Auch wenn die *bicubic*-Interpolation gar nicht so schlecht ist, geht es natürlich besser. In den weiteren Abschnitten werden wir uns Super-Resolution mit Hilfe von künstlichen neuronalen Netzen anschauen. Und dabei sind diese Interpolationsmethoden auch von Bedeutung, denn sie bieten einen guten Startpunkt für die Bildvergrösserung, und dies mit simplen Algorithmen, die wenig Rechenleistung erfordern.

2.2 Super-Resolution mit klassischen neuronalen Netzen

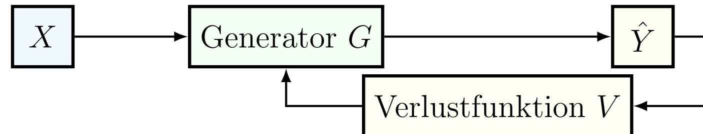
Deep Learning ist ein Teilgebiet des maschinellen Lernens und bezieht sich auf die Modellierung von Zusammenhängen mit Hilfe von neuronalen Netzen mit zahlreichen Zwischenschichten [18]. Mit einem solchen neuronalen Netz könnten wir den Zusammenhang bzw. eine Transformation von einem niedrig aufgelösten Eingabebild X in ein hochaufgelöstes Ausgabebild Y modellieren. Die niedrig aufgelösten Bilder haben eine Auflösung von 128×128 Bildpunkten und, da es farbige Bilder sind, 3 Farbkanäle für Rot, Grün und Blau. Das Eingabebild kann somit als Vektor $X \in \mathbb{R}^{128 \times 128 \times 3}$ dargestellt werden. Das hochaufgelöste Bild soll in beide Richtungen eine 4-mal (bzw. gesamt eine 16-mal) grössere Auflösung haben, somit betrachten wir es als hochdimensionalen Vektor, nämlich $Y \in \mathbb{R}^{512 \times 512 \times 3}$.

Um ein neuronales Netz trainieren zu können, brauchen wir eine Verlustfunktion, welche angibt, wie gut die Schätzungen des Netzes sind und welche Gewichte es verändern muss, um besser zu werden. Zum Einstieg könnten wir eine weit verbreitete Verlustfunktion, die *mean squared error* (durchschnittlicher quadrierter Fehler) verwenden [34]. Sie vergleicht zwei Bilder

pixelweise,

$$V(Y, G(\theta, X)) = \frac{1}{whc} \sum_{i,j,k=1}^{w,h,c} (Y_{i,j,k} - G(\theta, X)_{i,j,k})^2$$

wobei die Funktion G die Auswertung des neuronalen Netzes mit seinen Gewichten θ und dem Eingabebild X beschreibt. Die Parameter w, h, c geben die Dimensionen (Breite, Höhe und Anzahl Farbkanäle) des Bildes mit hoher Auflösung an. In unserem Fall ist $w = h = 512$ und $c = 3$. Bezeichnen wir das erzeugte Bild mit $\hat{Y} = G(\theta, X) \in \mathbb{R}^{w \times h \times c}$, so lässt sich das neuronale Netz wie folgt darstellen. Dieses neuronale Netz mit der MSE Verlustfunktion nennen wir SRResNet.



Die *MSE* Verlustfunktion hat jedoch ein grosses Problem. Die generierten Super-Resolution Bilder haben keine *high frequency details*, sie wirken geglättet (siehe Abbildung 6). Dies liegt gemäss [19] daran, dass die MSE Verlustfunktion nur die Differenz der einzelnen Pixelwerte betrachtet und nicht die „Wirkung" des ganzen Bildes einbezieht. Wie in Abschnitt 1.2 erläutert, gibt es nämlich keine eindeutige Lösung. Es wird also nach einem von vielen guten Bildern gesucht. Das obige neuronale Netz entscheidet sich jedoch nicht für ein einzelnes Bild, sondern wählt eine Art Durchschnitt [9], was unbefriedigend ist.

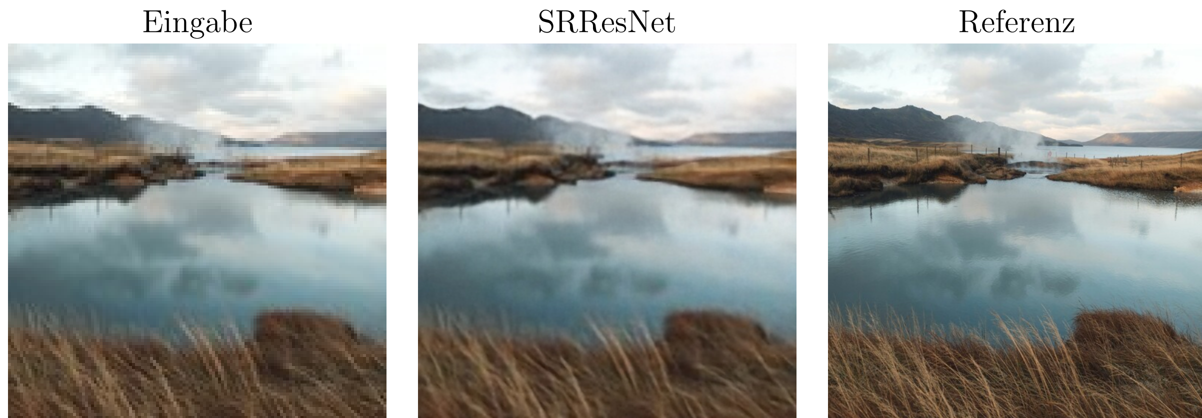


Abbildung 6: Das mittlere Bild (SRResNet) ist mit der Verlustfunktion MSE erzeugt und wirkt geglättet.

Einen Ausweg findet man mit folgender Idee: Da es keine scharfe Trennung zwischen guten und unrealistischen Bildern gibt, die Änderung weniger Pixel hat für den Betrachter schliesslich keinen wahrnehmbaren Effekt, kann man sich vorstellen, dass *jeder* Vektor $y \in \mathbb{R}^{w \times h \times c}$ mit einer „bestimmten Wahrscheinlichkeit" ein gutes Bild ist. Schön wäre es nun, wenn wir diese Wahrscheinlichkeitsverteilung kennen würden. Dann könnte man in einem zweiten Schritt versuchen, ein Bild gemäss dieser Verteilung zu erzeugen. In den folgenden Abschnitten wird eine Methode vorgestellt, welche mit einer solchen Wahrscheinlichkeitsverteilung arbeitet.

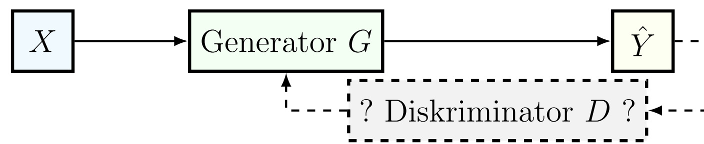
2.3 Generative Adversarial Networks

Generative Adversarial Networks (kurz GAN, auf Deutsch: „erzeugende gegnerische Netzwerke") sind eine spezielle Form von generativen Modellen.

Der Begriff *generativ* bedeutet hier, dass ein Bild aus der Menge der möglichen Bilder ausgewählt wird. Etwas genauer gesagt bedeutet dies, dass ein Sample aus der Verteilung der möglichen Bilder gezogen wird. Dabei wird die Verteilung selbst gar nicht explizit bestimmt, sondern direkt das Sample gezogen. Wie das genau funktioniert wird in Abschnitt 2.3.2 erklärt.

Das Spezielle an GANs ist, dass sie aus zwei neuronalen Netzen bestehen, welche „gegeneinander“ trainiert werden. Das eine neuronale Netz, der Generator, funktioniert analog wie das neuronale Netz aus Abschnitt 2.2. Es nimmt das Bild X als Eingabe und stellt mittels geeigneter Architektur und sinnvoller Verlustfunktion das Bild $\hat{Y}$ her. Neu hinzu kommt das zweite neuronale Netz, der Diskriminator. Der Diskriminator bewertet das Bild als Ganzes, indem er entscheidet, ob es echt oder erzeugt wirkt [10]. Der Diskriminator fokussiert sich also nicht so stark auf die einzelnen Pixel, wie ein Modell, welches den *MSE* minimiert. Wie der Diskriminator diese Aufgabe erfüllt, wird weiter unten in Abschnitt 2.3.4 erläutert.

Auch wenn wir die Details des Diskriminators noch nicht ganz verstehen, können wir das GAN so darstellen:



Wir haben ja erwähnt, dass der Generator aus der Wahrscheinlichkeitsverteilung der echten Bilder ein Sample erzeugt. Wir wollen in den nächsten Abschnitten erläutern, was die Idee dahinter ist. Zum einen geht es darum, wie man überhaupt eine Wahrscheinlichkeitsverteilung schätzen kann und dann auch darum, wie man ein Sample aus einer solchen Verteilung zieht. Zwar wird der Generator die Verteilung der echten Bilder nicht direkt schätzen, aber wir benötigen die Schätzung einer Wahrscheinlichkeitsverteilung auch für den Diskriminator. Deshalb lohnt es sich, diese Idee nachzuvollziehen.

2.3.1 Schätzen der Parameter einer Normalverteilung

Wir nehmen zur Vereinfachung an, dass das eigentlich hochdimensionale Bild Y nur eindimensional wäre und einer Normalverteilung $Y \sim \mathcal{N}(\mu, \sigma^2)$ und somit dieser Dichtefunktion folgen würde:

$$f(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

Wir müssten also nur geeignete Werte für μ und σ finden und wir würden diese Verteilung schon kennen. Wenn man n Realisationen von Y hätte, nennen wir sie $y_1, y_2, \dots, y_n$, so könnte man den Mittelwert $\hat{\mu} = \bar{y}$ als Schätzung für den Erwartungwert μ und die empirische Standardabweichung $\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{k=1}^{n}(y_k - \bar{y})^2}$ als Schätzung für die Standardabweichung σ verwenden [17] und wir hätten unser Problem schon fast gelöst. Wir würden nun die Dichtefunktion recht genau kennen und könnten uns ein mögliches Bild, welches einen *hohen* Dichtewert hat, auswählen:

Allerdings ist nicht klar, welches Y man auswählen soll. Im Beispiel von Abbildung 7 würden wir das beste Ergebnis erzielen, wenn wir das Bild an der Stelle $Y = 3$, also im Maximum auswählen würden. Dann ist aber *jedes* Bild immer gleich (da wir *immer* das Bild bei $Y = 3$ auswählen). Dies wollen wir natürlich nicht. Besser wäre es, ein hochaufgelöstes Bild Y in Abhängigkeit eines zufällig gezogenen tiefaufgelösten Bildes X auszuwählen. Wie man eine Zufallsvariable Y gemäss einer vorgebebenen Verteilung F aus einer anderen Zufallsvariablen X erzeugt, wird im folgenden Abschnitt gemäss [3] beschrieben.

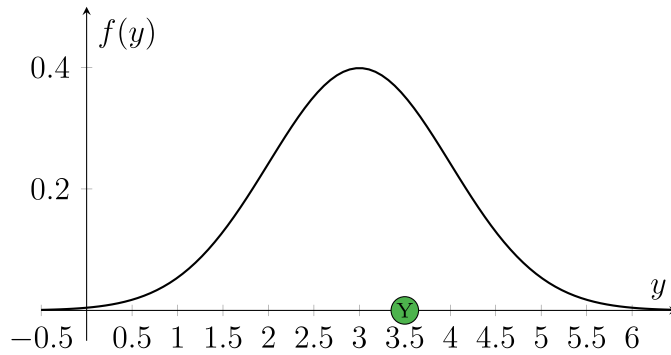


Abbildung 7: Illustration einer Wahrscheinlichkeitsverteilung am Beispiel der Normalverteilung.

2.3.2 Erzeugen einer Zufallsvariable nach vorgegebener Verteilung

Sei $F(y) = P(Y \leq y)$ die Verteilungsfunktion von Y und nehmen wir eine auf dem Intervall $[0, 1]$ uniform-verteilte Zufallsgrösse X. Nun suchen wir die Umkehrung von F an der Stelle X. Die auf folgende Art erzeugte Zufallsgrösse $Y = F^{-1}(X)$ hat dann die gewünschte Verteilung F. Anhand von Abbildung 8 ist dieser Vorgang ersichtlich. Zusätzlich kann die Gleichheit in den folgenden Umformungen (übernommen aus [3]) nachvollzogen werden.

$$P(Y \leq y) = P(F^{-1}(X) \leq y)$$

Und da F monoton wachsend ist, wird die Ungleichung nicht geändert, wenn wir F auf beide Seiten der Ungleichung anwenden.

$$\begin{aligned} P(F^{-1}(X) \leq y) &= P(F(F^{-1}(X)) \leq F(y)) \\ &= P(X \leq F(y)) \\ &= F(y) \end{aligned}$$

Somit ist das Problem der Erzeugung eines Bildes Y gleichbedeutend mit dem Problem der Erzeugung einer neuen Zufallsvariable aus einer auf $[0, 1]$ uniform verteilten Zufallsvariable. Diese neue Zufallsvariable folgt der Wahrscheinlichkeitsverteilung $F(y)$ der „möglichen Bilder“ im 1-dimensionalen Raum. Unser Bild Y ist aber natürlich nicht normalverteilt. Auch X ist weder eindimensional noch uniformverteilt. Aber die Idee bleibt dieselbe. Wir suchen eine Abbildung F^{-1}, welche aus den zufällig gezogenen Bildern $X \in \mathbb{R}^{128\times128\times3}$ ein Bild $Y \in \mathbb{R}^{512\times512\times3}$ aus der Verteilung F der „guten Bilder “ erzeugt. Diese Abbildung F^{-1} nennen wir G. Das neuronale Netz wird diese Transformation beschreiben.

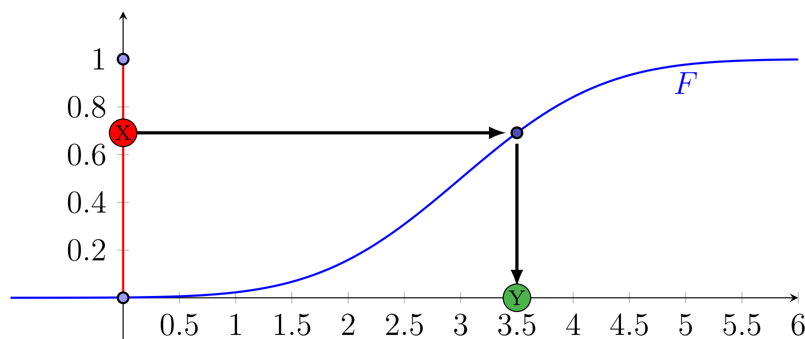


Abbildung 8: Die Zufallsvariable $Y = F^{-1}(X)$ soll der Verteilung F folgen.

2.3.3 Maximum-Likelihood-Methode

Ein viel allgemeinerer Ansatz als in Absatz 2.3.1 ist die Maximum-Likelihood-Methode [22]. Mit ihr kann man die Parameter einer Verteilung mit *vorgegebener Form* schätzen. Später wird es darum gehen, zu bestimmen, ob ein Bild aus dem Datensatz stammt, bzw. echt ist oder generiert wurde, weshalb wir die Maximum-Likelihood-Methode an diesem Beispiel erläutern. Ob ein Bild echt oder generiert ist beschreibt die Zufallsgrösse Z:

$$Z = \begin{cases} 0: & \text{ein zufällig ausgewähltes Bild ist } \textit{künstlich erzeugt} \\ 1: & \text{ein zufällig ausgewähltes Bild ist } \textit{echt} \end{cases}$$

Da Z nur den Wert 0 oder 1 annehmen kann, kann sie mit einer *Bernoulli-Verteilung* beschrieben werden. Es gilt also: $Z \sim Ber(p)$, d.h. $P(Z = 1) = p$ und $P(Z = 0) = 1 - p$. Wir können die Verteilung auch in einer einzigen Formel schreiben:

$$P(Z = z) = p^z \cdot (1 - p)^{1-z}$$

Jede Realisation von Z, welche den Wert 1 annimmt, spricht dafür, dass p gross ist, jede Realisation, welche den Wert 0 annimmt, spricht dafür, dass p klein ist. Haben wir nun n Realisationen $z_1, z_2, \ldots, z_n$, welche alle unabhängig voneinander sind, so ist die gemeinsame Wahrscheinlichkeit das Produkt der Einzelwahrscheinlichkeiten:

$$P(Z_1 = z_1, Z_2 = z_2, \ldots, Z_n = z_n) = \prod_{k=1}^{n} P(Z_k = z_k) = \prod_{k=1}^{n} p^{z_k} \cdot (1 - p)^{1-z_k}$$

Da wir diese Ereignisse wirklich beobachtet haben, muss die Wahrscheinlichkeit für die gegebenen Beobachtungen $z_1, \ldots, z_n$ eher gross sein. Wir wählen also p so, sodass die Wahrscheinlichkeit $P(Z_1 = z_1, \ldots, Z_n = z_n)$ möglichst gross wird. Dies ist die Kernidee der Maximum-Likelihood-Methode.

$$\begin{aligned} \hat{p} &= \underset{p}{\operatorname{argmax}} \prod_{k=1}^{n} P(Z_k = z_k) \\ &= \underset{p}{\operatorname{argmax}} \prod_{k=1}^{n} p^{z_k} \cdot (1 - p)^{1-z_k} \end{aligned}$$

Wenn man den streng monoton wachsenden Logarithmus auf das Produkt auf der rechten Seite anwendet, dann ändert sich die Stelle des Maximums nicht. Der Vorteil ist jedoch, dass man das Produkt durch eine Summe ersetzen kann

$$\begin{aligned} &= \underset{p}{\operatorname{argmax}} \log \left(\prod_{k=1}^{n} p^{z_k} \cdot (1 - p)^{1-z_k} \right) \\ &= \underset{p}{\operatorname{argmax}} \sum_{k=1}^{n} \log \left(p^{z_k} \cdot (1 - p)^{1-z_k} \right) \end{aligned}$$

Mit den Logarithmusgesetzen erhält man noch einfacher

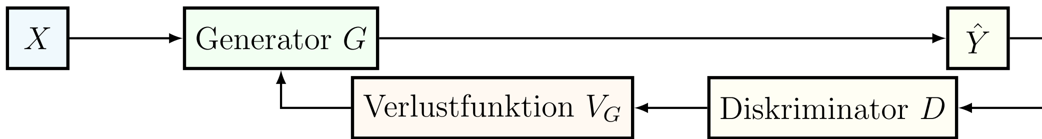
$$= \underset{p}{\operatorname{argmax}} \left(\sum_{k=1}^{n} z_k \cdot \log(p) + (1 - z_k) \cdot \log(1 - p) \right) \tag{1}$$

2.3.4 Die Aufgabe der Diskriminators

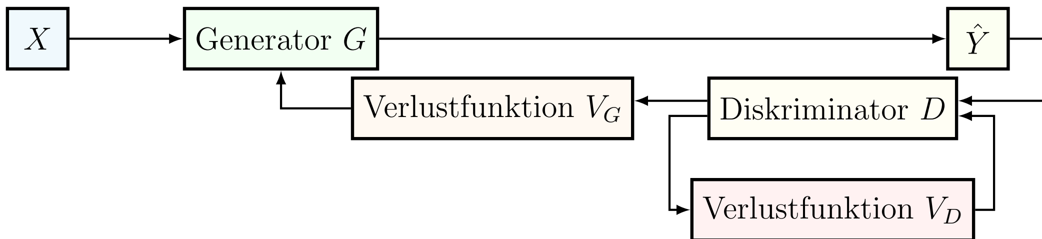
Im letzten Abschnitt haben wir ein Verfahren erarbeitet, mit dem wir die Parameter einer Verteilung mit *vorgegebenen* Form (d.h. welche durch einen expliziten Ausdruck gegeben ist) bestimmen können. Nochmals als Auffrischung: Wir wollen die *MSE* Verlustfunktion durch ein Verfahren ablösen, welches die Verteilung aller gutaussehenden hochaufgelösten Bilder bestimmt, respektive ein Sample aus ihr zieht. Die Form dieser Verteilung kennen wir aber nicht, wir können also die Maximum-Likelihood-Methode nicht direkt anwenden. Wir setzen also ein neues neuronales Netz, welches diese komplexe (nicht triviale) Verteilung auf eine Bernoulli-Verteilung reduziert. Wir betrachten dazu wieder die Zufallsgrösse Z, die folgendes beschreibt:

$$Z = \begin{cases} 0: & \text{ein zufällig ausgewähltes Bild ist } \textit{künstlich erzeugt} \\ 1: & \text{ein zufällig ausgewähltes Bild ist } \textit{echt} \end{cases}$$

Wir haben bisher angenommen, dass $Z \sim \text{Ber}(p)$ verteilt ist. Dies ist aber sehr unrealistisch, da dann ja jedes Bild mit derselben Wahrscheinlichkeit echt ist. Besser wäre es anzunehmen, dass p vom zufällig ausgewählten, hochaufgelösten Bild abhängt. Diese Abhängigkeit modelliert der Diskriminator D:



Die Verlustfunktion V_G soll dabei den Generator G belohnen, wenn der Diskriminator D denkt das generierte Bild wäre echt, bzw. wenn D eine *falsche* Aussage trifft. Bis jetzt trifft der Diskriminator D aber immer noch zufällig Entscheidungen. Es braucht auch für den Diskriminator noch eine Verlustfunktion V_D:



Diese belohnt den Diskriminator, wenn das Bild *richtig* eingeschätzt wird. Die neuronalen Netze G und D trainieren also auf unterschiedliche Ziele hin. Daher kommt das Attribut *adversarial* („gegnerisch") aus *GAN*.

2.3.5 Die Verlustfunktion des Diskriminators

Der Generator kann als Funktion G beschrieben werden mit den Gewichten θ_G. Er bildet das Eingabebild X auf das Super-Resolution Bild $\hat{Y}$ ab.

$$\begin{aligned} G: \mathbb{R}^{128\times128\times3} &\to \mathbb{R}^{512\times512\times3} \\ X &\mapsto G(\theta_G, X) \end{aligned}$$

Auch der Diskriminator wird als Funktion D mit den Gewichten θ_D beschrieben.

$$\begin{aligned} D: \mathbb{R}^{512\times512\times3} &\to [0,1] \\ Y &\mapsto D(\theta_D, Y) \end{aligned}$$

Im ersten Paper über GAN von Goodfellow et al. [10] ist die Verlustfunktion des Diskriminators folgermassen definiert:

$$V_D\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = -\frac{1}{m}\sum_{k=1}^{m}\log\left(D\left(\theta_D, y_k\right)\right) - \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

wobei:

$y_1, \ldots, y_m \in \mathbb{R}^{512\times512\times3}$ eine Stichprobe aus der Verteilung der *hoch* aufgelösten Bilder ist.

$x_1, \ldots, x_m \in \mathbb{R}^{128\times128\times3}$ eine Stichprobe aus der Verteilung der *niedrig* aufgelösten Bilder ist.

Dabei kann $D\left(\theta_D, y_k\right)$ als Wahrscheinlichkeit (ein Wert zwischen 0 und 1) dafür interpretiert werden, dass das echte hochaufgelöste Bild vom Diskriminator auch als echtes hochaufgelöstes Bild erkannt wird. Ist der Diskriminator gut, dann sollte dieser Wert nahe bei 1 liegen. Mit dem Logarithmus wird der Ursprüngliche Wertebereich $[0, 1]$ von $D\left(\theta_D, y_k\right)$ auf das Intervall $]-\infty, 0]$ transformiert. Eine starke Falscheinschätzung wird also auch stärker bestraft. Ohne das Vorzeichen beschreibt der linke Term ein mittlerer Logarithmus der Wahrscheinlichkeitsvorhersage, ob der Diskriminator das hochaufgelöste Bild richtig zuordnet. Durch das Vorzeichen wird daraus eine Verlustfunktion, welche für eine gute Schätzung ($D\left(\theta_D, y_k\right) \simeq 1$) einen tiefen Wert ($-\frac{1}{m}\sum_{k=1}^{m}\log\left(D\left(\theta_D, y_k\right)\right) \simeq 0$) und für eine schlechte Schätzung einen hohen Wert zurückgibt. Den rechten Term kann man eigentlich analog betrachten. $D\left(\theta_D, G\left(\theta_G, x_k\right)\right)$ beschreibt auch hier die Wahrscheinlichkeit, dass der Diskriminator ein echtes hochaufgelöstes Bild als echtes hochaufgelöstes Bild erkennt. Für einen guten Diskriminator wäre $D\left(\theta_D, G\left(\theta_G, x_k\right)\right)$ deshalb auch nahe bei 0. Daher stammt das $1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)$.

Diese Verlustfunktion sieht äusserst kompliziert aus. Dabei fragt man sich, woher die Funktion stammt und wie sie mit der Rivalität der beiden neuronalen Netze zusammenhängt.

Vergleichen wir diese Verlustfunktion mit der Maximum-Likelihood-Methode (1), so erkennen wir, dass sie eine ähnliche Struktur hat. Aus [9] und [1] kann man schliessen, dass das Minimieren der Verlustfunktion für den Diskriminator bezüglich θ_D (bei fixem θ_G) gleichbedeutend mit dem Maximieren des Likelihood bezüglich der Verteilungsfunktion von Z ist.

2.3.6 Die Verlustfunktion des Generators

Wie in Abschnitt 2.3.4 beschrieben, soll der Generator G belohnt werden, wenn der Diskriminator D eine Fehlentscheidung trifft. Anstatt die Verlustfunktion des Diskriminators V_D zu minimieren, soll er sie also maximieren. Daher kommt diese Schreibweise:

$$\min_{\theta_G}\max_{\theta_D}\frac{1}{m}\sum_{k=1}^{m}\log\left(D\left(\theta_D, y_k\right)\right) + \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

Sie beschreibt die Rivalität der neuronalen Netze, die versuchen, in unterschiedliche Richtungen zu Lernen. Da der Generator aber sowieso nur einen Einfluss auf den hinteren Term hat, benutzt man eine seperate Verlustfunktion, welche minimiert werden soll:

$$V_G\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

Mit dieser Verlustfunktion haben wir das Problem umgangen, die Verteilungsfunktion für Y schätzen zu müssen. Denn diese wäre erstens sehr kompliziert und für die Maximierung des Likelihoods müssten wir auch die Form der Verteilungsfunktion kennen. Der Generator G erzeugt aus X zwar die Bilder $\hat{X}$, welche der Verteilung von Y folgen sollten, die Verlustfunktion von G maximiert $\hat{X}$ aber nicht mit dem Likelihood gegenüber der Verteilung von Y, sondern

mit der Verteilung von Z, welche aber von Y abhängt. Diese Abhängigkeit sorgt dafür, dass auch $\hat{X}$ langsam der Verteilung von Y folgt. Und da wir wissen, dass Z Bernoulli-verteilt ist, ist diese Verteilung viel einfacher zu modelieren.

2.3.7 Der implementierte Algorithmus

Der Trainingsalgorithmus, den ich verwende, sieht folgermassen aus:

Algorithm 1 Training eines GANs für Super-Resolution

1: **for** Anzahl Trainingsschritte = Anzahl Mini Batches × Epochen **do**
2: Wähle Mini-Batch $\{x_1, \ldots, x_m\}$ aus dem Datensatz
3: Wähle Mini-Batch $\{y_1, \ldots, y_m\}$ aus dem Datensatz

4: Aktualisieren des Generators, d.h. minimiernen von:
$\frac{1}{m}\sum_{k=1}^{m} \log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$
5: Aktualisieren des Diskriminators, d.h. minimiernen von:
$\frac{1}{m}\sum_{k=1}^{m} \log\left(D\left(\theta_D, y_k\right)\right) + \frac{1}{m}\sum_{k=1}^{m} \log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$

2.3.8 Ein anschauliches Beispiel

Nun wurde viel über den theoretischen hintergrund von GANs erzählt. Ich möchte nun noch mit einem Beispiel aus [31] eine intuitivere Vorstellung der Funktionsweise von GANs im Allgemeinen und vor allem den Aspekt der „gegnerischen Modelle“ hervorrufen. Dieses Beispiel ist auch in Abbildung 9 dargestellt.

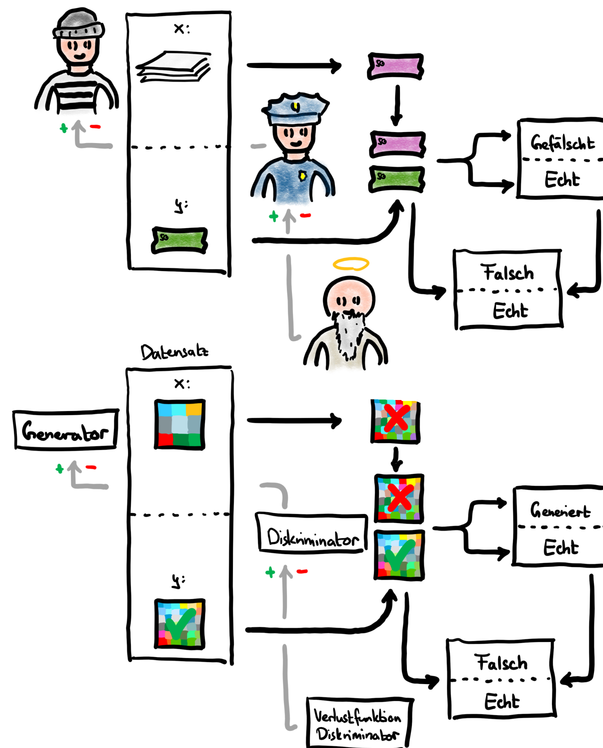


Abbildung 9: Die Analogie des Geldfälschens verglichen mit dem GAN-Algorithmus angewandt auf Super-Resolution.

Ein Verbrecher will Falschgeld drucken, um seinen Lebensunterhalt zu sichern. Würde niemand sein gedrucktes Geld untersuchen, müsste er sich keine Mühe geben - schlecht gefälschtes

Geld würde reichen. Doch genau so ist es nicht, denn die Polizei untersucht Geld aus dem Umlauf stichprobenhaft und prüft, ob es sich um Falschgeld handelt. Um also nicht erwischt zu werden, muss sich der Verbrecher verbessern. Angenommen ein Allwissender steckt der Polizei immer die Information zu, ob sie sich richtig entschieden hat oder nicht. So wird auch die Polizei besser. Um ihre Ziele zu erreichen, verbessern sich also die Polizei und der Verbrecher.

Ein GAN funktioniert eigentlich genau gleich. Das neuronale Netz, welches die Bilder zeichnet (bzw. das Geld fälscht) nennt man *Generator*. Dieser wertet die Eingabebilder des Datensatzes aus und stellt unsere generierten Super-Resolution-Bilder her. Dem *Diskriminator* (bzw. der Polizei) wird dieses Bild und auch irgendein zufällig ausgewähltes, echtes hochaufgelöstes Bild aus dem Datensatz vorgesetzt. Der Diskriminator bestimmt, ob die Bilder Originale oder vergrösserte Bilder sind. Mit dieser Ausgabe des Diskriminators wird schlussendlich der Verlustwert des Generators berechnet. Dem Diskriminator sagen wir dabei natürlich nicht, welches welches Bild ist, aber wir selbst hingegen wissen natürlich, welches das generierte ist. Somit könnten wir ganz leicht eine Verlustfunktion für den Diskriminator bestimmen und diesen trainieren.

2.4 SRGAN

SRGAN ist eine Methode von Ledig et al. [19], welche höchaufgelöste Bilder mittels GANs erzeugt. Damit wird das Problem der *MSE* Verlustfunktion gelöst. Denn wie schon in Abschnitt 2.2 erklärt, wirken Bilder, die mit dieser MSE Verlustfunktion trainiert wurden geglättet und haben keine sogenannten *high frequency details*. Mit GANs funktioniert das schon besser. Trotzdem braucht es zusätzlich ein paar Feinheiten, damit gute Bilder erzeugt werden. Diese werden nun in den folgenden Abschnitten vorgestellt.

2.4.1 Architektur

Mit der Architektur ist der Aufbau und die Struktur des neuronalen Netzes gemeint. Dabei ist sie ein limitierender Faktor. Ist der Aufbau des neuronalen Netzes zu einfach, dann *kann* es, im Beispiel von Super-Resolution, den Zusammenhang zwischen tief- und hochaufgelösten Bildern gar nicht lernen. Das neuronale Netz braucht also genügend Gewichte, die angepasst werden können, um alle Feinheiten lernen zu können. Denn zu einem gewissen Grad schneiden grössere Architekturen besser ab. Ist dass Modell aber sehr gross, dann braucht man sehr viel Rechenleistung um das Modell zu trainieren. Man muss versuchen eine möglichst kleine Architektur zu entwickeln, die immer noch gut die Zusammenhänge modellieren kann. Dabei kommen viele verschiedene spezielle *Layer* zum Einsatz. Der Generator und der Diskriminator der SRGAN Methode sehen überwältigend kompliziert aus. Sie sind hier in Abbildung 10 und 11 dargestellt.

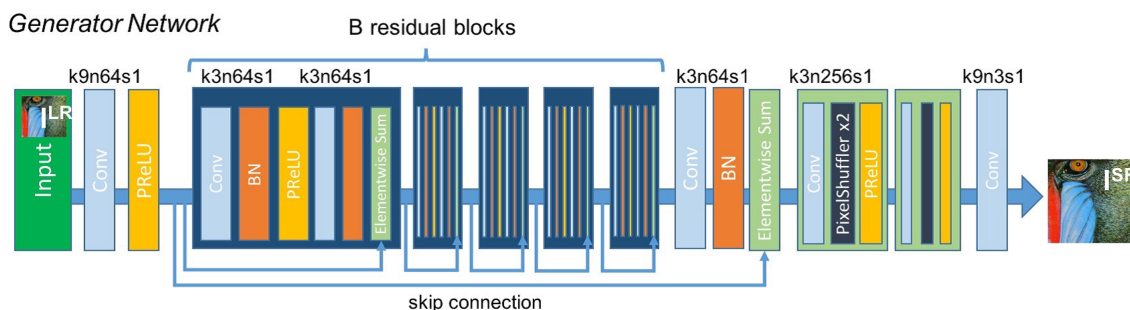


Abbildung 10: Die Architektur des Generators der SRGAN Methode aus [19].

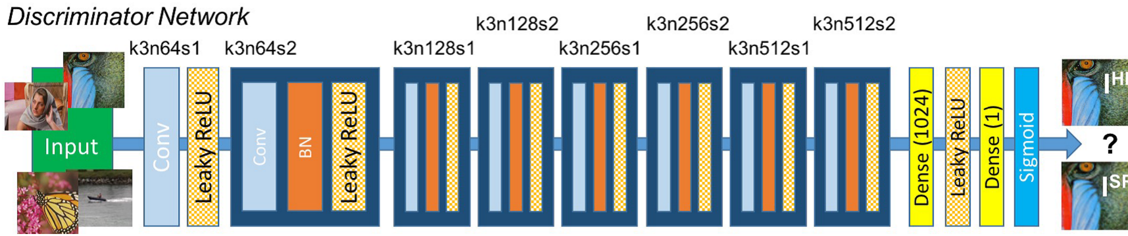


Abbildung 11: Die Architektur des Diskriminators der SRGAN Methode aus [19].

Die wichtigsten Aspekte des Generators sind die sogenannten *convolutional layer*, hier mit „Conv“ bezeichnet, und die Struktur mit den vielen *residual blocks*.

2.4.2 Convolutional Neural Networks

Convolutional Neural Networks sind spezielle neuronale Netze mit einer bestimmten Architektur [6]. Sie bestehen nämlich aus *Convolutional*, *Pooling* und *Fully-Connected* bzw. *Dense* Layers. Wir werden aber nur die Convolutional Layer besprechen, da die anderen nicht für *SR-GAN* verwendet werden. Heuzutage sind Convolutional Neural Networks essentiell, vor allem in der Mustererkennung (engl. *pattern recognition*). Sie können nämlich Merkmale aus Bildern unabhängig von der Position im Bild erkennen. Dies ist mit herkömlichen künstlichen neuronalen Netzen nicht möglich. Zudem verringern sie stark die Anzahl der Trainingsgewichte, weil nicht alle Farbwerte des Eingabebildes mit allen Neuronen des nächsten Layers verbunden werden. Daher kann man grössere Modelle trainieren.

Die Idee eines *Convolutional Layer* ist es Strukturen zu erkennen. Dabei werden meistens nicht nur ein Convolutional Layer, sondern gleich mehrere hintereinander eingesetzt. Das Erste erkennt Linien- und Kantensegmente, das zweite schon längere Linien. Mit weiteren Convolutional Layer kann das neuronale Netz Formen bzw. Flächen und sogar Objekte wie ein Hund oder ein Auto erkennen. Dies kann ein Convolutional Layer lernen, indem es ein Bild „abtastet“. Es bewegt einen sogenannten Filter über das ganze Bild (Stichwort *Sliding Window*), wie in Abbildung 12. Ein Filter ist ein kleines Bild (z.B. 3×3 px), auf dem eine Linie oder Kurve gezeichnet ist. Dabei gibt es ganz viele verschiedene Filter. Um diese muss man sich aber nicht wirklich kümmern, da diese immer schon von der verwendeten Bibliothek implementiert sind.

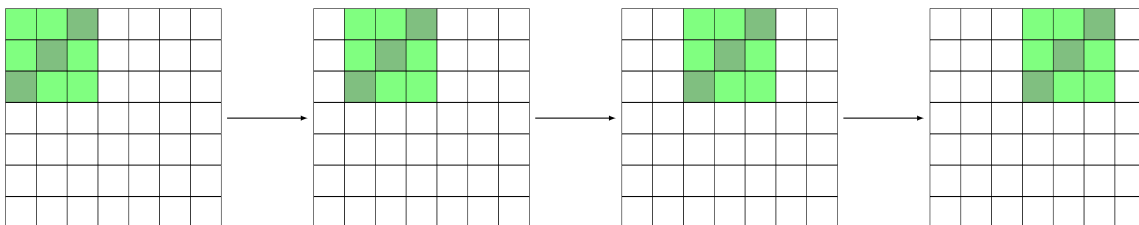


Abbildung 12: Ein Filter (grün) wandert mit vorgegebener Grösse (3×3 px) und Schrittweite (1 px) über das ganze Bild.

Während der Filter über das Bild wandert, wird zwischen ihm und dem Bildteil, über dem er sich gerade befindet, eintragsweise das Produkt gebildet und aufsummiert (siehe Abbildung 13). Betrachten wir den Filter und den Bildteil als Vektoren in $\mathbb{R}^9$ (da sie eine Grösse von 3×3 px haben), so ist diese Operation das Skalarprodukt. Das Skalarprodukt ist ja 0, wenn zwei Vektoren orthogonal aufeinander stehen und grösser, wenn sie in eine ähnliche Richtung zeigen. Das Skalarprodukt zwischen dem Filter und dem Bild ist also gross, wenn sie sich ähneln, und klein, wenn sie unterschiedlich sind. Ist das Bild nicht nur schwarz-weiss, sondern hat noch eine 3. Dimension, dann arbeitet der Filter alle „Farben“ getrennt ab und bildet anschliessend

die Summe über die Farbkanäle. Anschliessend wird diese Summe für jeden Schritt in ein sogenanntes *Feature Map* eingetragen. Meist benützt man aber nicht nur einen Filter sondern ganz viele. Dadurch entstehen auch ganz viele Feature Maps. Diese werden in der 3. Dimension des Bildes, sozusagen im Farbkanal abgelegt. Bei z.B. 16 verschiedenen Filtern gäbe es dann auch 16 Feature Maps. Da diese in der 3. Dimension gespeichert werden, gibt es dann anstelle der 3 Farben für Rot, Grün und Blau 16 Feature Maps. Deshalb kann man nach einem Convolutional Layer nicht mehr von Farbe sprechen.

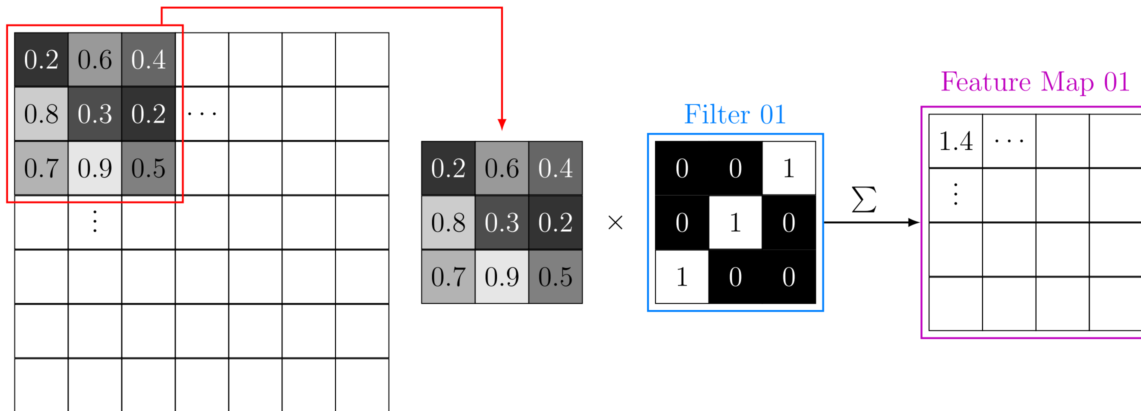


Abbildung 13: Ein Convolutional Layer erkennt Linien bzw. Kanten und Formen, indem stückweise das Bild mit einem Filter, welcher Strukturen im Bild via Skalarprodukt entdecken kann, „abgetastet“ wird.

2.4.3 Residual Networks

Residual Networks sind eine der grössten Durchbrüche im Deep Learning und lösen ein wichtiges Problem. In Abschnitt 2.4.1 wurde erwähnt, dass je grösser bzw. tiefer ein neuronales Netz ist, desto besser ist es. He et al. [14] haben herausgefunden, dass dies nur zu einem bestimmten Grad stimmt. In Abbildung 14 wird gezeigt, dass kleinere neuronale Netz besser als Grössere abschneiden können. Das Kleinere ist in diesem Beispiel allerdings auch schon sehr gross.

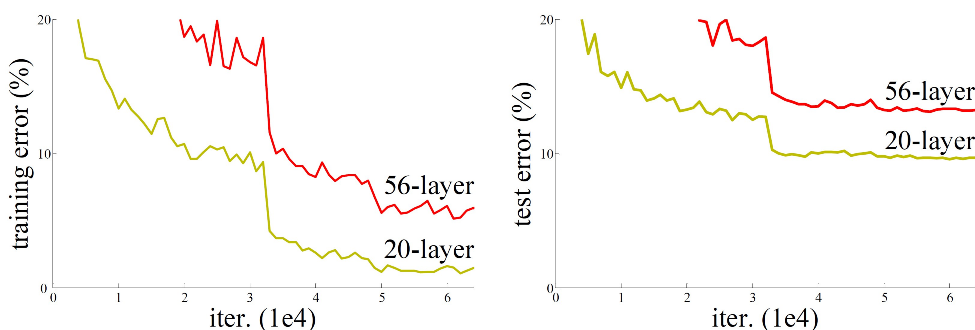


Abbildung 14: Vergleich von zwei neuronalen Netzen mit unterschiedlicher Anzahl Schichten (Layer). Auf den x-Achsen sind die Anzahl Trainingsschritte abgebildet. Links bezeichnet die y-Achse den Trainingsfehler und rechts den Testfehler. In beiden Abbildungen ist zu erkennen, dass das 20-schichtige Netz besser abschneidet als das Grössere. Abbildung aus [14].

Intuitiv macht dies keinen Sinn, da ein grösseres Modell sicherlich gleich gut sein müsste wie das kleinere. Es müsste die Layer des kleineren nur kopieren und die restlichen Layer müssten die Identitätsfunktion beschreiben, bzw. den Eingabewert wieder ausgeben. Es stellt sich heraus,

dass dies gleich schwer ist wie jede andere Funktion zu erlernen. Dies hängt damit zusammen, dass alle Gewichte des neuronalen Netzes um 0 herum initialisiert werden. Das Modell muss den ganzen Zusammenhang neu erlernen. Einfacher wäre es, anstelle des ganzen Zusammenhangs *zusätzlich* zum schon Erlernten, neue Details dazu zu lernen. Es entsteht ein *Residual Block*:

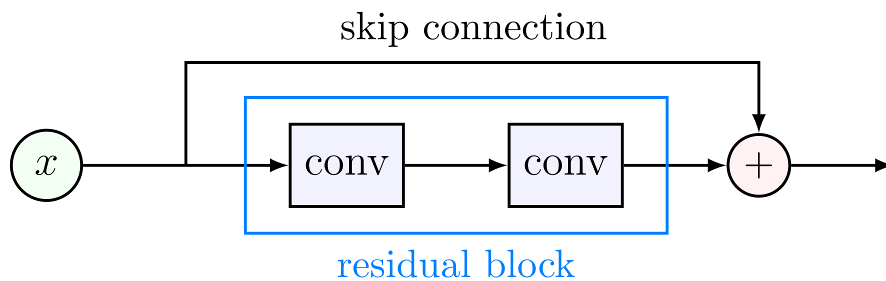


Abbildung 15: Schematische Darstellung eines Residual Blocks der aus zwei Convolutional Layer bestehtDurch die skip connection wird der Eingabewert wieder hinzuaddiert. Dadurch müssen nur die Differenzen gelernt werden.

Ein Residual Block fasst mehrere beliebige Layer zu einer Einheit zusammen. Die zwei Convolutional Layer, in der Abbildung 15, sind nur Beispiele. Der Eingabewert wird über eine sogenannte *skip connection* zum Ergebnis des Residual Blocks wieder hinzuaddiert. Deshalb braucht das Modell nur noch die Differenz zu lernen, was natürlich viel einfacher ist. Um nun trotzdem ein genügend grosses neuronales Netz zu erhalten, werden viele Residual Blocks aneinander gehängt. In SRGAN sind es 16 Residual Blocks.

2.4.4 Pixel-shuffle

Betrachten wir nochmals die Architektur des Generators der SRGAN Methode, wie in Abbildung 10 dargestellt. Fast am Schluss kommen zwei grüne Blöcke mit sogenannten *Pixel-shuffle* Layer vor. Nur diese Pixel-shuffle Layer alleine sorgen dafür, dass das Bild vergrössert wird. Diese Methode wird *Post Upsampling* genannt und wie der Name sagt, wird das Bild im allgemeinen *nach*, bzw. hier gegen Ende des neuronalen Netzes vergrössert. Dies mag merkwürdig erscheinen, ein Bild zuerst lange und rechenaufwendig zu bearbeiten und erst dann zu vergrössern. Nun ja, zum einen spart man sich dadurch viel Rechenaufwand, denn man muss nur ein 4-mal kleineres Bild (also 128×128 px) bearbeiten. Zweitens liegt es an der speziellen Vergrösserungsmethode, denn das Modell generiert zuerst die Information die das Bild ausmacht. Erst zum Schluss wird diese Information (ein Pixel Array) zu einem Bild angeordnet. Ein Pixel-shuffle Layer funktioniert nämlich so:

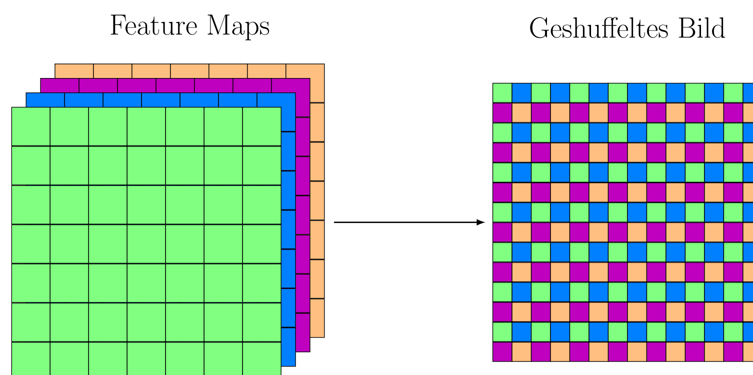


Abbildung 16: Die Feature Maps werden aneinandergereiht, wodurch das Bild grösser wird.

Die Convolutional Layer erzeugen viele Feature Maps der Grösse 128×128 Pixel, welche die eigentliche Bildinformation beinhalten. Diese Feature Maps werden nebeneinander aufgereiht, wodurch eine Art Bild mit doppelter Auflösung entsteht [28]. In Abbildung 16 werden 4 Feature Maps benutzt. Würde man z.B. 8 Feature Maps einsetzten, dann würden einfach zwei geshuffelte Bilder erzeugt werden. Da wir in SRGAN eine 4-fache Vergrösserung anstreben, wird zweimal ein Pixel-shuffle Layer eingesetzt. Anschliessend wird aus diesen geshuffelten Bildern über ein letztes Convolutional Layer das Outputbild mit drei Farbkanälen generiert.

2.4.5 Verlustfunktion

Wie erwähnt basiert *SRGAN* auf der *GAN* Methode und GANs bestehen aus einem *Generator* und einem *Diskriminator*, welche beide eigene Verlustfunktionen haben. Für den Diskriminator bleibt die Verlustfunktion gleich wie im GAN Paper von Goodfellow et al. [10]:

$$V_D\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = -\frac{1}{m}\sum_{k=1}^{m}\log\left(D\left(\theta_D, y_k\right)\right) - \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

Für den Generator ist es ein wenig anders. Die Generator Verlustfunktion aus dem GAN kommt aber trotzdem vor. Nennen wir sie aber nun V_{adv}, wegen dem Attribut *Adversarial*:

$$V_{adv}\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

In GANs werden Generator und Diskriminator ja gegeneinander trainiert. Wird der Diskriminator zu stark, so erkennt er bei allen erzeugten Bildern $G\left(\theta_G, x_k\right)$ des Generators, dass sie falsch sind. $D\left(\theta_D, G\left(\theta_G, x_k\right)\right)$ ist also immer nahe bei 0. Folglich ist auch der Verlustwert des Generators V_{adv} immer nahezu 0, also beim Maximum. Der Generator weiss nicht mehr, in welche Richtung er seine Gewichte verändern soll, um besser zu werden, da der Gradient beim Maximum der Nullvektor ist. Dieses Problem wird auch *Vanishing Gradient* genannt. Daraus folgt, dass der Generator keine Bilder mehr generiert, sondern nur noch zufällige farbige Flecken. Um dieses Problem zu umgehen schlägt Ledig et al. [19] vor, die folgende Verlustfunktion zu benutzen:

$$V_G\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = V_{con}\left(Y, G\left(\theta_G, X\right)\right) + 10^{-3}V_{adv}\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right)$$

V_{con} ist dabei eine weitere Verlustfunktion und steht für *content loss*, welche wie z.B. die *MSE* Verlustfunktion, im „Pixelraum" arbeitet. Die gegnerische Verlustfunktion V_{adv} trainiert den Generator darauf, dass die „Wirkung" des Bildes stimmt. Nur mit dem Feedback des Diskriminators durch V_{adv} braucht der Generator zu lange, um ein gutes Bild zu erzeugen. Der Diskriminator ist schneller und gewinnt gleich die Oberhand. Eine Verlustfunktion im Pixelraum bringt den Generator schnell auf den richtigen Weg, es fehlen einfach die Details. Die Schwierigkeit liegt darin, beide Verlustfunktionen richtig zu gewichten, sodass der Generator mit dem Diskriminator mithalten kann, aber trotzdem gute Resultate erziehlt werden. In Abbildung 17 sind Bilder, die während des Trainings mit unterschiedlichen Gewichtungen erzeugt wurden, dargestellt. Ganz Links ist V_{con} zu schwach gewichtet, der Generator wird zu stark und es kommt zum Effekt der durch Vanishing Gradient verursacht wird. Ganz rechts wird V_{con} zu stark gewichtet. Der Generator beachtet nur noch diese Verlustfunktion und nicht mehr den Diskriminator.

Wie diese Verlustfunktion V_{con} genau aussieht ist etwas speziell, denn sie besteht aus einem Convolutional Neural Network. Warum man nun noch ein weiteres neuronales Netz braucht, und was für eine Wirkung es erziehlt, wird im folgenden Abschnitt erklärt.

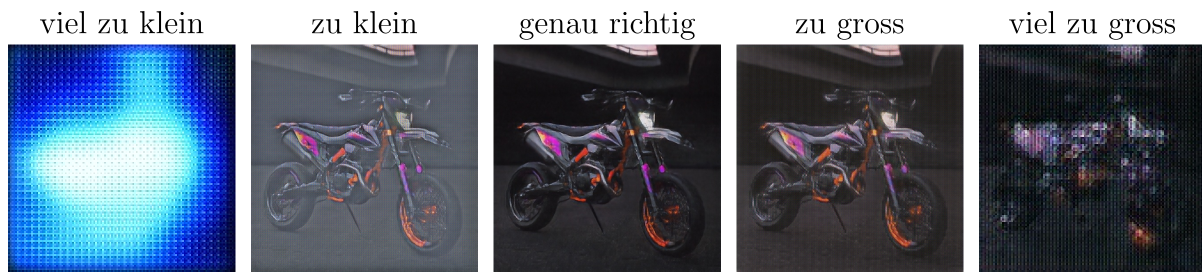


Abbildung 17: Bilder mit unterschiedlichen Gewichtungen der Verlustfunktionen V_{con} und V_{adv}. Links ist das Gewicht von V_{con} viel zu klein, rechts viel zu gross.

2.4.6 VGG19

VGG19 ist ein sehr tiefes Convolutional Neural Network, das von Simonyan et al. [30] entwickelt wurde. Ursprünglich wurde es verwendet, um Bilder zu klassifizieren. Wir können also auf ein *schon trainiertes Modell* zurückgreifen. Dieses Modell wird also nicht weiter trainiert, sondern ist einfach eine sehr komplexe Funktion, welche dem Generator hilft rasch auf Kurs zu kommen. Der ganze Klassifizierungsteil interessiert uns nicht, aber die Convolutional Layer sind hilfreich für uns. In Abschnitt 2.4.2 haben wir ja gesehen, dass Convolutional Layer Kanten, Formen und ganze Objekte erkennen können und diese in Feature Maps speichern. Solche Feature Maps des VGG19 Netzwerkes sind in Abbildung 18 dargestellt.

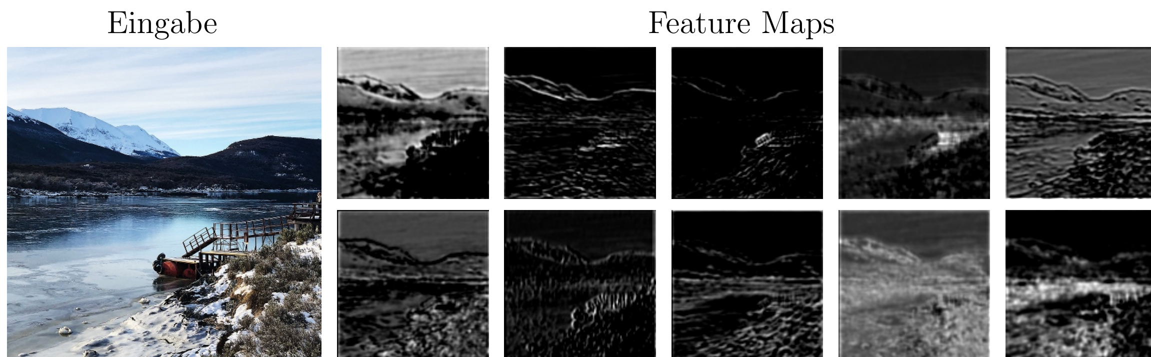


Abbildung 18: Links das Eingabebild, rechts 10 von 256 Feature Maps.

Geben wir also beide Bilder, das hochaufgelöste Bild Y und das zugehörige generierte Bild $G(\theta_G, x_k)$ durch das VGG19 Netzwerk, so erhalten wir für beide die Feature Maps. Vergleichen wir diese mit einer MSE Verlustfunktion, so können wir überprüfen, wie gut Kanten und kleinere Details übereinstimmen. Diese Verlustfunktion arbeitet also noch im Pixelbereich, konzentriert sich aber schon auf Strukturen, die für die natürliche Wahrnehmung wichtig sind. Dies ist sicherlich eine Verbesserung gegenüber einer normalen MSE Verlustfunktion, welche den ganzen Bildbereich gleich gewichtet.

2.5 Datensatz

Für das Trainieren eines neuronalen Netzes braucht man einen grossen Datensatz. Je mehr Bilder man hat, desto weniger *Epochen* muss man trainieren, um gleich viele Trainingsschritte zu erreichen. *Epochen* geben an, wie oft der ganze Datensatz durchtrainiert wird. Trainiert man die gleichen Bilder zu viele Male, kann es zu *Overfitting* kommen. Dabei merkt sich das Netz die einzelnen Bilder so gut, sodass schlechtere Aussagen über neue Bilder getroffen werden. Ein beliebter Datensatz für Bild-Klassifizierung ist z.B. **ImageNet** [7]. Ledig et al. [19] haben diesen Datensatz auch für die originale SRGAN Methode verwendet. Das Problem dabei ist,

dass die Bilder nur eine Durschschnittsgrösse von 469×387 haben. Ledig et al. [19] verwendeten daher auch nur eine Vergrösserung von $96 \times 69 \mapsto 384 \times 384$. Dies war mir ursprünglich nicht bewusst, ich dachte es wäre eine Auflösung von $128 \times 128 \mapsto 512 \times 512$, weil mir dies logisch erschien. Ich suchte also weiter nach einem Datensatz mit grösseren Bildern. Es gibt viele grosse Gesichtsdatensätze wie **CelebA** [21], **Flickr-Faces-HQ** [16] oder **VGGFace2** [24]. Diese wären von der Auflösung her passend gewesen. In dieser Arbeit, war es jedoch das Ziel nicht nur Gesichter zu trainieren. Deshalb habe ich mich für den **Unsplash** Datensatz [32] entschieden.

2.5.1 Unsplash

Unsplash ist eine Webseite, welche Bilder von hoher Qualität von über 256 Tausend verschiedenen Photographen gratis zur Verfügung stellt. Neben einer API stellt Unsplash seit neuem einen freien Datensatz mit 25 Tausend Bildern und einen Lizenzpflichtigen Datensatz mit über 3 Millionen Bildern zur Verfügung. Letzterer wird in dieser Arbeit genutzt. Um die Bilder herunterzuladen und auf einer Festplatte zu speichern, habe ich ein Tool geschrieben, welche diese Arbeit automatisiert. Dieses arbeitet die URL-Links des Unsplash Datensatzes ab, ruft diese mit der richtigen Grösse auf und speichert sie. Dabei ist es möglich, die Bilder in unterschiedlichen Grössen zu speichern, um später während des Trainings keine Zeit für das verkleinern von Bildern zu verlieren. Dabei gibt es ein weiteres zeitliches Problem. Da jedes Bild als einzelne *.jpg* Datei abgeschpeichert wurde, muss jedes Bild einzeln dekomprimiert werden. Dadurch geht sehr viel Zeit während des Trainings verloren. Statt alle Bilder einzeln zu speichern, sollte man sie also besser zusammen speichern. Dies wird möglich, wenn man sie in einem *Hierarchical Data Format* speichert. Genau dies wurde in dieser Arbeit gemacht.

2.5.2 Hierarchical Data Format

Das **Hierarchical Data Format** bezeichnet Dateiformate wie **HDF5** und ist dafür ausgelegt, sehr grosse Mengen an Daten zu speichern. Es ist ausserdem darauf optimiert, schnell und effizient die Inhalte, das sind vor allem Tabellen, auszulsesen [4]. Dabei wird erstens Zeit gespart, da die Bilder weniger stark und nur einmal dekomprimiert werden müssen und zweitens wenig Arbeitsspeicher beansprucht, da der Zugriff effizienter ist. Der grosse Nachteil ist aber, dass für das Speichern von Bildern viel mehr Platz benötigt wird. Das HDF5 Datenformat speichert seine Inhalte wie schon gesagt in Tabellen, es komprimiert diese Tabellen also nur mit einer **Zip**-Kompression und nicht mit einer passenden Bildkompression wie z.B. in **Jpg** Dateien. Solche Zip-Kompressionen sind natürlich viel ineffizienter für Bilder. Deshalb benötigen 10^5 Bilder der Grösse 512×512 Pixel im HDF5 Format etwa 78 GB, wobei sie im jpg Format lediglich 4 GB beanspruchen.

2.5.3 Multiprocessing I

Da alle Bilder nur über URL Links im Unsplash Datensatz verfügbar sind, mussten alle Bilder heruntergeladen werden. Dieses Herunterladen braucht aber eine ziemlich lange Zeit. Deshalb habe ich versucht den ganzen Prozess mit *Multiprocessing* zu beschleunigen. Mit Multiprocessing kann ein Programm gleichzeitig verschiedene Dinge tun und ist dadurch schneller, als wenn die einzelnen Schritte nacheinander erledigt werden. Multiprocessing ist auch vergleichbar mit *Multithreading*. In Python ist Multithreading jedoch nicht wirklich möglich, da durch das *Global Interpreter Lock* verhindert wird, dass Threads parallel ausgeführt werden. Es ist möglich mehrere Threads zu erzeugen, sie werden aber immer noch hintereinander ausgeführt. Mit Multiprocessing kann man aber mehrere *Subprocesses* erzeugen die wirklich parallel ablaufen [2].

Mit dieser Veränderung konnten die Bilder, mit Hilfe von 10 gleichzeitig laufenden Prozessen, doppelt so schnell heruntergeladen werden.

2.6 Metriken

Eine der grössten Schwierigkeiten für Super-Resolution besteht darin, eine passende Verlustfunktion zu finden. Es ist schwierig mathematisch zu beschreiben, was für Eigenschaften ein Bild ausmacht, welches für das menschliche Auge echt wirkt. Bis heute hat man es nicht wirklich geschafft ein Mass für die Qualität eines Bildes zu finden. Doch wie vergleicht man dann verschiedene Methoden? Entweder man macht eine Umfrage und bittet andere Menschen die Bilder zu bewerten. Dies ist aber teuer und zeitaufwändig. Eine andere Möglichkeit wäre es trotz mangelnder Genauigkeit, mathematische Masse zu benutzen. Die zwei wichtigsten Metriken für die Beschreibung der Qualität eines Super-Resolution Bildes sind **PSNR** und **SSIM** [34].

PSNR steht für *Peak Signal-to-Noise Ratio*, es beschreibt wie stark das generierte Bild im Verhältnis zum echten Bild verrauscht ist. PSNR ist deshalb auch ein beliebtes Mass, um die Qualität einer Kompressionsmethode zu beschreiben. Definiert ist das Mass wie folgt [34]:

$$\mathrm{PSNR}(Y, G(\theta, X)) = 10 \log_{10} \left(\frac{whc \cdot L^2}{\sum_{i,j,k=1}^{w,h,c} (Y_{i,j,k} - G(\theta, X)_{i,j,k})^2} \right)$$

L beschreibt dabei den grössten Pixelwert, für 8-bit Bilder also 255. Wie man sieht, basiert PSNR hauptsächlich auf MSE. Deshalb hat das Mass auch die gleichen Probleme wie die Verlustfunktion. Das PSNR Mass bevorzugt also auch eher glatte Bilder ohne viele Details. Deshalb schneiden MSE-Methoden auch besser ab, als GAN-Methoden. Man kann sich fragen, warum man dann dieses Mass benutzt, aber in der Realität ist es einfach weit verbreitet. Wäre ein generiertes Bild perfekt, also genau gleich wie das echte Bild, dann ginge der PSNR-Wert im Grenzwert gegen ∞.

SSIM steht für *Structural Similarity*. SSIM vergleicht die Helligkeit, den Kontrast und die Struktur des Bildes. Die Definition dieses Masses ist deutlich komplizierter als beim PSNR Mass, deshalb wird die genaue Definition hier nicht beschprochen. Für ein perfektes generiertes Bild, wäre der SSIM-Wert gleich 1 [34].

3 Eigene Erweiterungen

3.1 Aufbau des Programmes

Ein weiteres Ziel dieser Arbeit war es, ein möglichst flexibles System für das Training von Super-Resolution Methoden zu errichten. Es sollte möglich sein, ohne grossen Aufwand weitere Verlustfunktionen, andere Architekturen und zusätzliche Interpolationsmethoden hinzuzufügen und nach Belieben ein oder zwei neuronale Netze zu verwenden. Mit Hilfe des Codes, welcher sich auf dem *Github-Repository*[2] befindet, können so Interessierte ihre eigenen Ideen verwirklichen. Zusätzlich wird dem Benutzer gleich gezeigt, ob ein erstelltes Modell funktioniert und Informationen wie Verlustwerte, Metriken, Trainingszeiten und Auslastung des Systemes werden am Schluss geplottet und gespeichert. Somit ist es sehr einfach, verschiedene Methoden und Trainingsdurchgänge zu vergleichen.

[2] Zu finden unter https://github.com/lukas-hueglin/comparison-of-SR-methods

3.1.1 Modularität

Um diese Flexibilität zu erreichen wurde ein *Modulares System* entwickelt, in welchem man einfach verschiedene Blöcke austauschen kann. Die Unterteilung der Super-Resolution Aufgabenstellung wurde von Wang et al. [34] in **Framework** und **Upsampling-/Vergrösserungs-Methode**, **Architektur** und **Verlustfunktion** unterteilt. Zusätzlich dazu wurden in dieser Arbeit weitere Aufgaben unterschieden:

Framework	Das Framework entscheidet, wann das Bild vergrössert wird. Entweder *Vorher*, *Nachher* oder *Schrittweise*. Es bindet automatisch die vorgegebene Upsampling Methode ein.
Upsampling Methode	Die Upsampling Methode bestimmt mit welcher Interpolierung das Bild vergrössert wird, z.B *Bicubic*, *Bilinear* und weitere. In dieser Arbeit wird sie aber ganz weggelassen, da die Vergrösserung nicht durch eine Interpolation sondern durch Pixel-shuffle zustande kommt.
Architektur	Die Architektur kann innerhalb einer Funktion bestimmt werden und wird so zurückgegeben.
Verlustfunktion	Die Verlustfunktion wird wie die Architektur innerhalb von einer Funktion definiert und kann dann benützt werden.
Pipeline	Die Pipeline bestimmt ob man gerade *trainieren*, *validieren* oder Bilder *generieren* will. Sie regelt ausserdem den ganzen Ablauf und den Datenfluss.
Laden der Daten	Damit das Laden der Bilder so effizient wie möglich abläuft, wird es von einer eigenen Klasse gesteuert und dann in die Pipeline eingespiesen.
Statistik	Das Erheben aller interessanten Daten wie *Verlustwerte*, *Metriken*, *Zeiten* und *Leistungsverbrauch* wird nebenher von einer Klasse geregelt.
Model	Mit dem Model werden Architektur und Verlustfunktion (und weiteres wie *Optimizer*) in ein neuronales Netz zusammengebunden.
Methode	Die Methode bestimmt die generelle Art, ob es nur ein neuronales Netz, oder ob es ein GAN mit mehreren neuronalen Netzen gibt.

Um eine bestimmte Methode zu erstellen kann man in einer Funktion Objekte von allen Klassen erstellen und diese miteinander verknüpfen. Abbildung 19 zeigt ein Beispiel für die SRGAN Methode, visualisiert in einem Baumdiagramm:

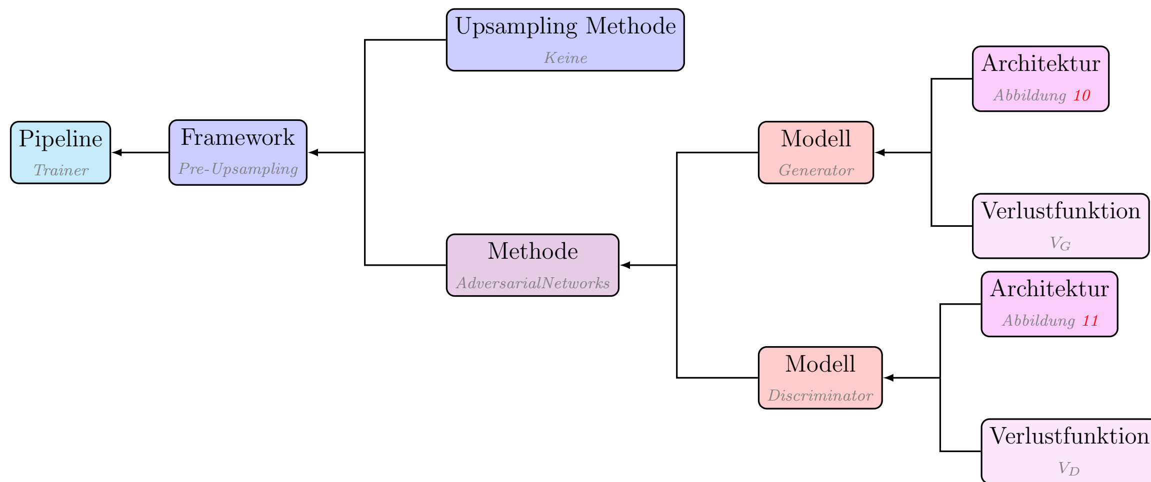


Abbildung 19: So sieht vereinfacht der Aufbau der SRGAN Methode aus.

3.1.2 Multiprocessing II

Um mehr Trainingsschritte durchführen zu können und im Allgemeinen ressourcenschonender zu sein, ist es natürlich das Ziel, möglichst schnell zu trainieren. Die Geschwindigkeit hängt einerseits von der Geschwindigkeit der Auswertung des neuronalen Netzes und des *Backpropagation* Algorithmus ab. Dies können wir aber nur schwer beeinflussen, da dieser Vorgang schon sehr optimiert auf der Grafikkarte abläuft. Andererseits hängt die Geschwindigkeit auch von der Ladezeit der Bilder ab. Diesen Vorgang müssen wir noch selber optimieren. Die einfachste Option wäre es, zuerst alle Bilder einer Epoche zu laden, abzuspeichern und dann während des Trainings darauf zuzugreifen. Dieser Weg ist auch in Abbildung 20 abgebildet. Dies ist natürlich langsam. Zuerst die Bilder laden, dann trainieren, nichts passiert gleichzeitig. Ausserdem müssen alle Bilder im Arbeitsspeicher gespeichert werden und es wird nicht dynamisch auf die .hdf5 Datei zugegriffen.

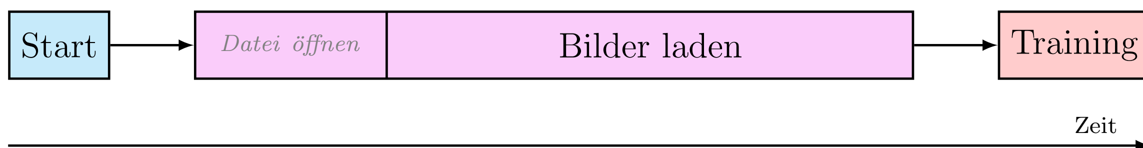


Abbildung 20: Das Laden der Bilder und das Training wird hintereinander ausgeführt.

Eine bessere Variante wäre es, wie in Abbildung 21 nur die Bilder eines einzelnen Batches zu laden, zu trainieren und gleichzeitig schon die nächsten zu laden. Das ganze Laden (ausser der erste Batch) passiert parallel zum Training. Die schon benutzten Bilder werden auch wieder vom Arbeitsspeicher freigegeben. Also perfekt oder? Nein nicht ganz. Das öffnen der .hdf5 Datei benötigt länger als das Trainieren (ca. 2-5 Sekunden), da sie noch dekomprimiert werden muss.

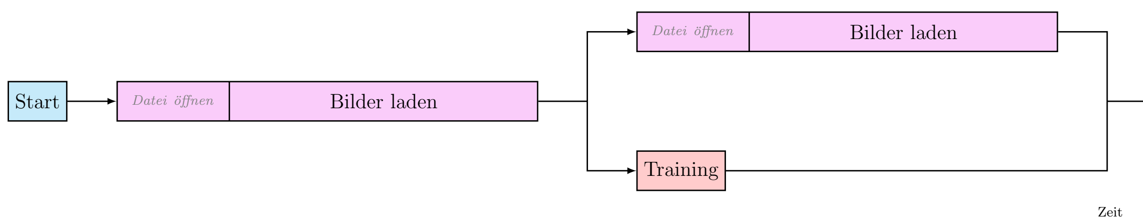


Abbildung 21: Das Laden der Bilder und das Training wird gleichzeitig ausgeführt. Einfachheitshalber wird die .hdf5 Datei aber wieder geschlossen.

Schliessen wir die Datei aber nicht mehr nach dem Laden eines Batches, sondern lassen wir

sie offen, so verlieren wir nur einmal etwas Zeit. Der Trainingsprozess schnappt sich dann einfach immer, wenn er neue Bilder braucht, die schon geladenen Bilder, welche auf dem gemeinsamen Speicher der beiden Prozesse gesichert sind.

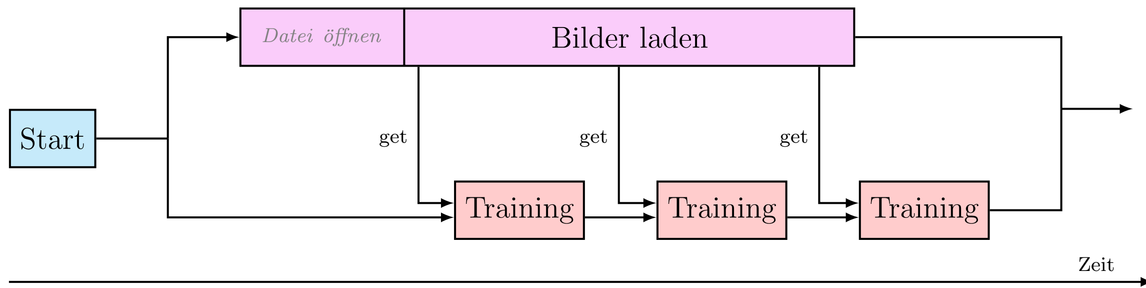


Abbildung 22: Das Laden der Bilder und das Training wird gleichzeitig ausgeführt. Die .hdf5 Datei wird aber erst am Ende der Epoche geschlossen.

Diese Erweiterung funktioniert so gut, dass nun die Auswertung des neuronalen Netzes und die Anpassung der Gewichte länger dauert als das Laden. In Abbildung 23 sieht man die benötigte Zeit fürs Laden der tiefaufgelösten Eingabebilder, der hochaufgelösten Bilder und das eigentliche Trainieren. Es ist klar ersichtlich, dass das Laden weniger lang dauert.

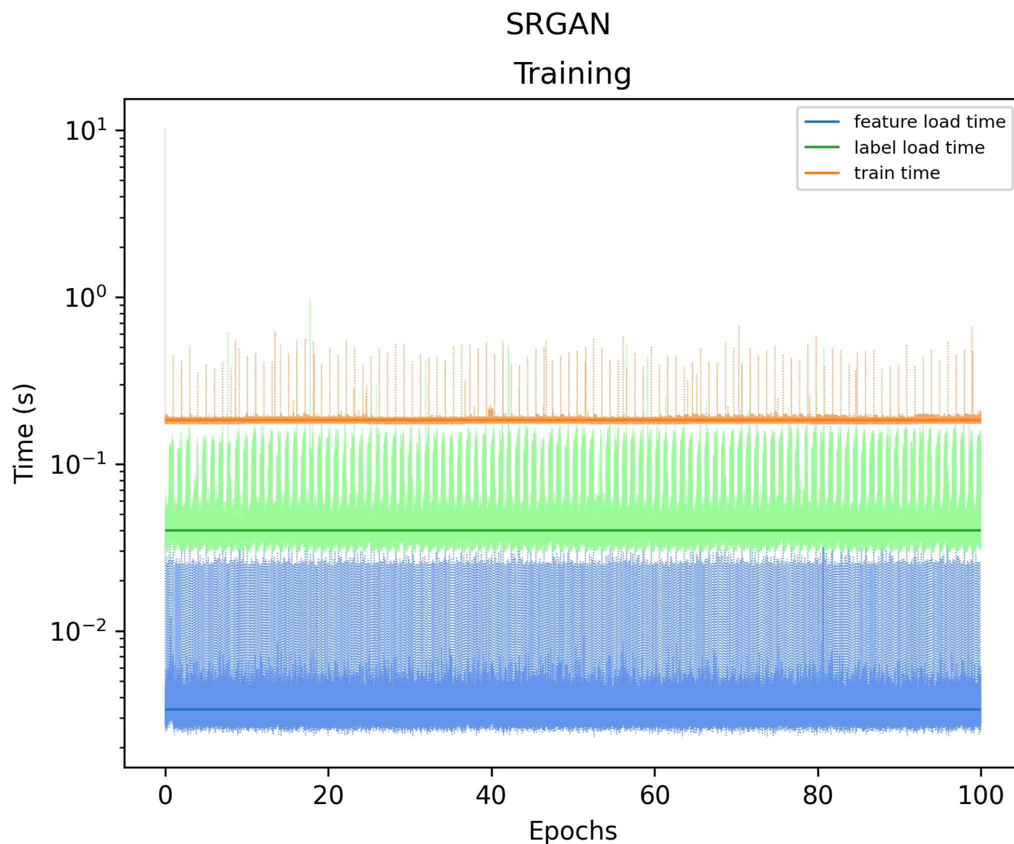


Abbildung 23: Das Laden der Bilder dauert viel kürzer als die Auswertung und die Anpassung der Gewichte des neuronalen Netzes.

3.2 Problematik beim Training von GANs

In Abschnitt 2.4.5 haben wir schon das *Vanishing Gradient* Problem angesprochen. Ist der Diskriminator perfekt, dann kann er *alle* Bilder korrekt zuordnen. $D(\theta_D, Y)$ ist dann im Extremfall

immer 1 und $D(\theta_D, G(\theta_G, X))$ ist immer 0, also maximal. Die Verlustfunktion des Generators

$$V_G\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

ist in diesem Fall 0, also maximal. Der Generator weiss nicht mehr, wie er seine Gewichte verändern soll, um sich zu verbessern. Daher ergibt sich laut Weng [36] ein grundlegendes Problem:

- Ist der Diskriminator zu schlecht, so erhält der Generator kein gutes Feedback, seine Verlustfunktion wiederspiegelt somit auch nicht die Realität (ob das Bild auch in den Augen eines Menschen gut ist oder nicht).
- Ist der Diskriminator zu gut, so verringert sich der Verlustwert des Generators praktisch auf 0, der Generator kann sich so nur noch langsam oder im schlimmsten Fall gar nicht mehr verbessern.

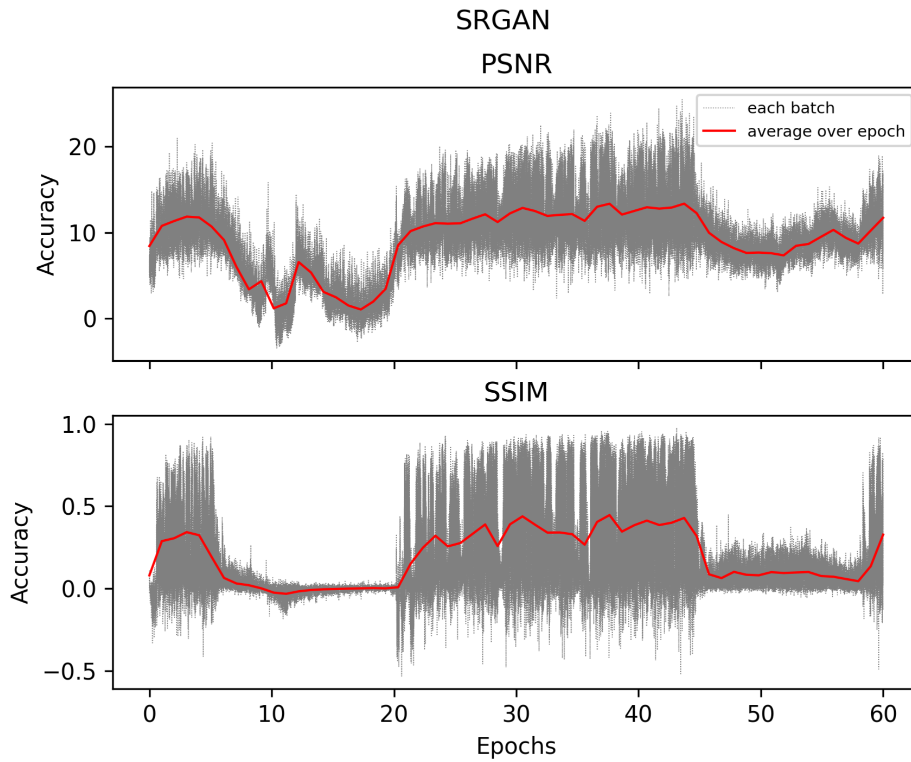
Laut Weng [36] besteht ein weiteres Problem auch darin, dass sich die beiden neuronalen Netze simultan verändern. Es kann nicht garantiert werden, dass die Modelle mit mehr Trainingsschritten gegen die richtige Lösung konvergieren. Dies kann man in den Trainingsdaten gut nachvollziehen. In Abbildung 24a sind die beiden bereits eingeführten Metriken dargestellt, die die Qualität der generierten Bilder beurteilen. Diese schwankt während des Trainingsprozesses. Interessant ist dabei auch, den Verlust des Generators und des Diskriminators zu betrachten (Abbildung 24b). Wir sehen, dass die Metrikwerte steigen, sobald sich der Verlustwert des Diskriminators kurz erhöht. Ansonsten bleibt der Verlustwert auf 0.

3.2.1 Bestrafungslösung

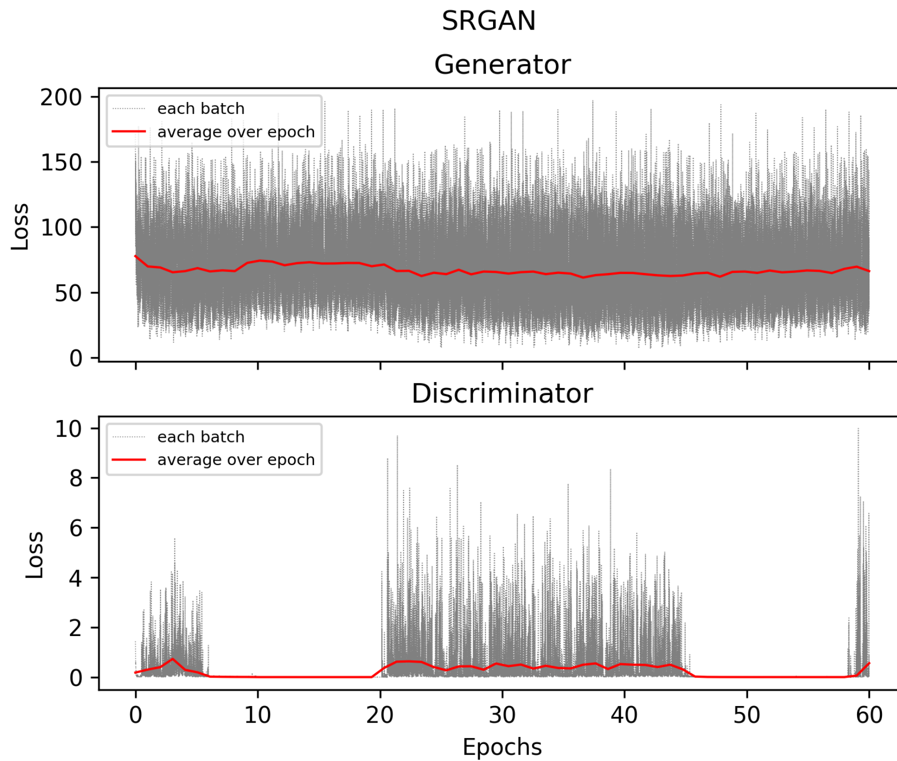
Zur Lösung dieses Problemes schlägt Weng [36] verschiedene Möglichkeiten vor, insbesondere das sogenannte *Wasserstein GAN*. Ich habe mir aber eine andere und eigene Lösung überlegt. Wir wollen ja die bestmöglichen Bilder erzeugen, das heisst, der Generator soll sein Optimum erreichen. Dabei wird $G(\theta_G, X) = Y$. Der Diskriminator erkennt keinen Unterschied mehr, $D(\theta_D, Y) = (D(\theta_D, G(\theta_G, X)) = \frac{1}{2}$. Die Verlustfunktion des Diskriminators

$$V_D\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = -\frac{1}{m}\sum_{k=1}^{m}\log\left(D\left(\theta_D, y_k\right)\right) - \frac{1}{m}\sum_{k=1}^{m}\log\left(1 - D\left(\theta_D, G\left(\theta_G, x_k\right)\right)\right)$$

wird im globalen Optimum zu $-2\log(\frac{1}{2}) = 2\log(2)$ [25, 36]. Würden wir den Verlustwert des Diskriminators bei $2\log(2)$ „festhalten“, dann könnte sich der Generator seelenruhig langsam seinem Optimum nähern. Diese Idee können wir verwirklichen, indem wir aufhören den Diskriminator zu Trainieren, wenn er zu gut ist. Auch wenn Goodfellow in seiner Nachfolgearbeit [9] davor warnt den Diskriminator zu schwächen, wird mit dieser Methode ein besseres Ergebnis erreicht. Wenn man den Diskriminator schwächt, ist die Wahrscheinlichkeit, dass der Diskriminator nicht sein ganzes Potential ausschöpfen kann, gross. Trotzdem wird der Diskriminator so nicht wirklich geschwächt, sondern nur relativ an die Leistung des Generators angepasst. Wird also der Generator immer besser, so wird dem Diskriminator erlaubt weiter zu trainieren und kann so sein Optimum erreichen.



(a) Die Qualität eines Bildes während eines eigenen Trainingdurchgangs.



(b) Die Verlustwerde von Generator und Diskriminator während eines eigenen Trainingdurchgangs.

Abbildung 24: Statistiken während eines eigenen Trainingdurchgangs eines SRGAN Modells.

3.3 Fourier-Verlustfunktion

Wie schon erwähnt, haben die Bilder, die mit einer MSE Verlustfunktion generierten wurden, laut Ledig et al. [19] keine *high frequency details*, sie sind geglättet. Alternativ zu den SRGANs könnte man einfach versuchen die MSE Verlustfunktion von SRResNet zu verändern. Der einzige Unterschied von SRGAN und SRResNet sind die „Verlustfunktionen“. SRResNet benutzt MSE und SRGAN den Diskriminator. Bis jetzt ist die SRGAN Methode eindeutig besser, aber es könnte ja möglich sein, eine mathematische Verlustfunktion zu erstellen, die mindestens gleich gut ist, wie ein Diskriminator.

Für die MSE Verlustfunktion lohnt es sich aber offenbar nicht, kleine Details zu erlernen, der Gradient (bzw. die Verringerung des Verlustes) ist zu gering. Wir müssen also die Belohnung für kleine Details vergrössern, um solche Details zu erzwingen. Man könnte zum Beispiel versuchen mit einer Art Low- und Highpass Filter verschiedene Bilder mit verschiedenen Mengen an Details zu erzeugen und diese dann mit unterschiedlicher Gewichtung per MSE zu untersuchen. Die Frage ist nur, wie man konkret Details von einem „Grundbild“ trennen kann. Eine Möglichkeit bietet die Fourier-Transformation. Diese Idee habe ich mir selbst überlegt und selber umgesetzt. Allerdings habe ich im Nachgang festgestellt, dass eine ähnliche Idee doch schon in einem Paper verwendet wurde [8].

Die Fourier-Analyse ist eine Möglichkeit, eine Funktion mit Hilfe einer unendlichen Reihe von Basisfunktionen auszudrücken. Im Gegensatz zur Taylorreihe benutzt man keine Polynome sondern Sinus- und Kosinusschwingungen als Basisfunktionen. Um dies zu verstehen, möchte ich wie in [35] Klänge zur Hilfe nehmen. In der Natur existieren keine reinen harmonische Schwingungen. Klänge bestehen immer aus einer Überlagerung von vielen harmonischen Schwingungen mit unterschiedlichen *Frequenzen* und *Amplituden*. Einzelne Töne bzw. harmonische Schwingungen sind in Abbildung 25 beispielhaft dargestellt.

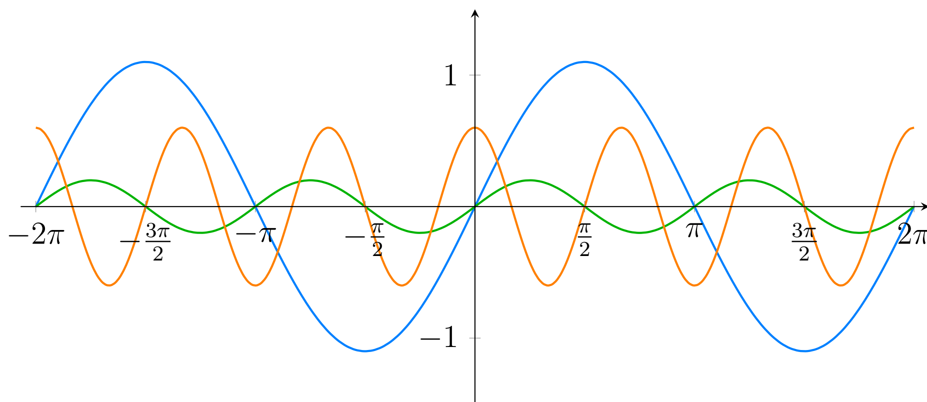


Abbildung 25: Verschiedene harmonische Schwingungen.

Aber natürlich hören wir die Schwingungen nicht einzeln, sondern eine Schwingung, welche die Überlagerung aller harmonischen Schwingungen darstellt. Eine solche Überlagerung zeigt Abbildung 26.

Das Ziel einer Fourier-Analyse ist es nun, diese einzelnen Schwingungen wieder zu extrahieren. Gleich können wir dies nun mit einer Pixelreihe aus einem Bild machen. Da ein Bild aber keine stetige Funktion ist, sondern nur aus Pixel mit einem bestimmten Farbwert bestehen, benützen wir eine diskrete Fourier-Transformation. Genauer gesagt, eine zweidimensionale schnelle Fourier-Transformation. Solche sind schon in Bibliotheken implementiert. Der Vorteil einer zweidimensionalen Fourier-Transformation ist, dass gleich ein ganzer Farbkanal analysiert wird und nicht nur eine Reihe des Bildes. Mit der Fourier-Transformation erhalten wir ein sogenanntes *Spektrum*. In einem solchen Spektrum wird die Amplitude in Abhängigkeit der Frequenz dargestellt. Wir wissen also, welche Frequenzen den grössten einen Einfluss haben. Nun

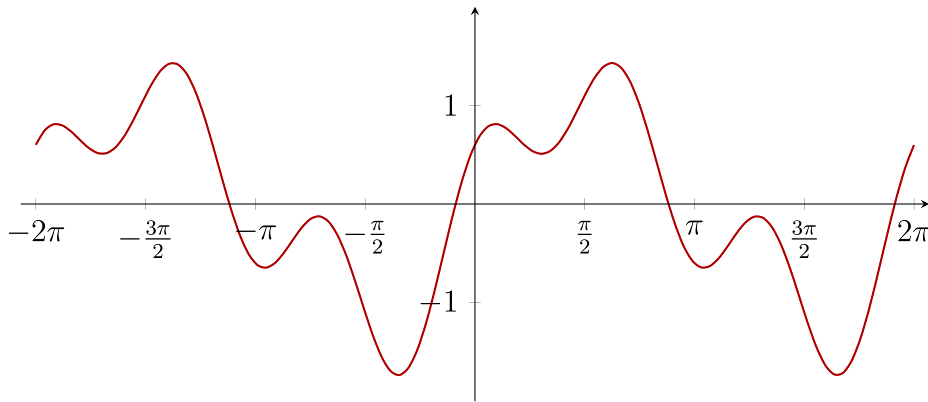


Abbildung 26: Die Summer der harmonischen Schwingungen aus Abbildung 25.

können wir bestimmte Frequenzen eliminieren, sodass diese gar nicht mehr vorkommen. Und wenn wir dann diese Spektren mit einer Inversen Fourier-Transformation wieder zurück in Bilder verwandeln, so erhalten wir Bilder mit unterschiedlichen Graden an Details. In Abbildung 27 ist ein Beispiel zu sehen.

Abbildung 27: Fourier Bilder mit unterschiedlichen Frequenzbereichen. In der Mitte wurden tiefe ($0-8$-fache Frequenz der Grundschwingung) und rechts hohe Frequenzen ($2-256$-fache Frequenz der Grundschwingung) zurücktransformiert.

3.3.1 Fourier-Verlustfunktion für SRGAN

Wenn wir uns noch an die VGG19 Methode und die zugehörigen Feature Maps aus Abschnitt 2.4.6 zurückerinnern, dann erkennen wir Parallelen zu den Bildern der Fourier-Transformation aus Abbildung 27. Beide können Kanten und Objekte hervorheben. Zum Vergleich mit den VGG19 Featuremaps aus Abbildung 18 sind in Abbildung 28, das selbe Eingabebild sowie zugehörige Bilder mit einem eingeschränkten Frequenzbereich dargestellt.

Für die Fourier-Verlustfunktion wenden wir für das generierte und für das zugehörige hochaufgelöste Bild eine Fourier-Transformation an. Daraus werden vier Bilder mit unterschiedlichen Frequenzbereichen, wie in Abbildung 28, erzeugt. Analog zu VGG19 werden diese Bilder mit einer MSE-Verlustfunktion verglichen.

Leider ist es nicht möglich die VGG19 Verlustfunktion komplett mit der Fourier Verlustfunktion zu ersetzten. Man braucht noch etwas mehr Feedback aus einem normalen Pixelraum, z.B. per MSE oder per VGG19. Nach langem herumprobieren habe ich mich für folgende Mischung entschieden:

Abbildung 28: Fourier Bilder mir einem Frequenzbereich von $0-5$, $5-38$, $38-102$ und $102-256$-facher Frequenz der Grundschwingung (oben links nach unten rechts).

$$V_G\left(D\left(\theta_D, Y\right), G\left(\theta_G, X\right)\right) = 5 \cdot 10^2 V_{fourier} + V_{vgg} + 10^{-2} V_{adv}$$

Mit diesem Verhältnis unterstützt die Fourier Verlustfunktion die VGG19 Verlustfunktion und sollte auch vom GAN mehr Details fordern. Aber wie gut es wirklich funktioniert sehen wir später.

4 Resultate und Diskussion

4.1 Über den Trainingsvorgang

Durch die lizenzpflichtige Version des Unsplash Datensatzes, welche mir kostenlos zur Verfügung gestellt wurde, standen etwa 3 Millionen Bilder für das Training zur Verfügung. Zudem habe ich mir grosse Mühe gegeben, um einen grossen Datensatz herunterzuladen, zu speichern und wieder schnell zu laden. Schlussendlich wäre dies gar nicht nötig gewesen, da ich nur mit wenigen Bildern arbeiten konnte. Öffnet man eine *.hdf5* Datei, so werden die Daten in den Arbeitsspeicher geladen. Mit **32 GB RAM** lag die Grenze bei mir bei etwa 10^4 Bildpaaren (d.h. 10^4 tiefaufgelöste Bilder und 10^4 hochaufgelöste Bilder). Mehr war mit diesem Computer ohne Veränderungen des Ladevorganges einfach nicht möglich. Ein weiterer limitierender Faktor war die Grafikkarte, in meinem Fall die **NVIDIA GeForce RTX 2080**. Und auch wenn während des Trainings die Grafikkarte gar nicht so stark ausgelastet war, wie in Abbildung 29 zu sehen ist, stürzte das Programm bei der Initalisierung, bei einer zu grossen *Batch-Size* ab. Die Batch-Size gibt an, wie viele Bilder gleichzeitig verarbeitet werden und zu *einer* Veränderung der Gewichte beitragen. Je grösser die Batch-Size, desto präziser und mit einer besseren Qualität wird trainiert. Um bei der Initialisierung einen Absturz zu vermeiden, wurde für das Training aller Methoden mit einer Batch-Size von lediglich 2 gearbeitet. Das originale SRGAN benutzte eine Batch-Size von 16, was eine deutliche Verbesserung bedeuten würde.

Alle Methoden wurden mit 100 Epochen trainiert. Zudem habe ich von den 10^4 Trainingsbilder 20% zur Validierung verwendet. Das heisst, alle Methoden wurden mit $100 \cdot \frac{0.8 \cdot 10^4}{2} = 4 \cdot 10^5$ Trainingsschritten trainiert. Mehr wäre nicht möglich gewesen, da der Computer bereits so etwa 28 Tage durchgehend ausgelastet war. Im Paper von Ledig et al. [19] wurde SRResNet

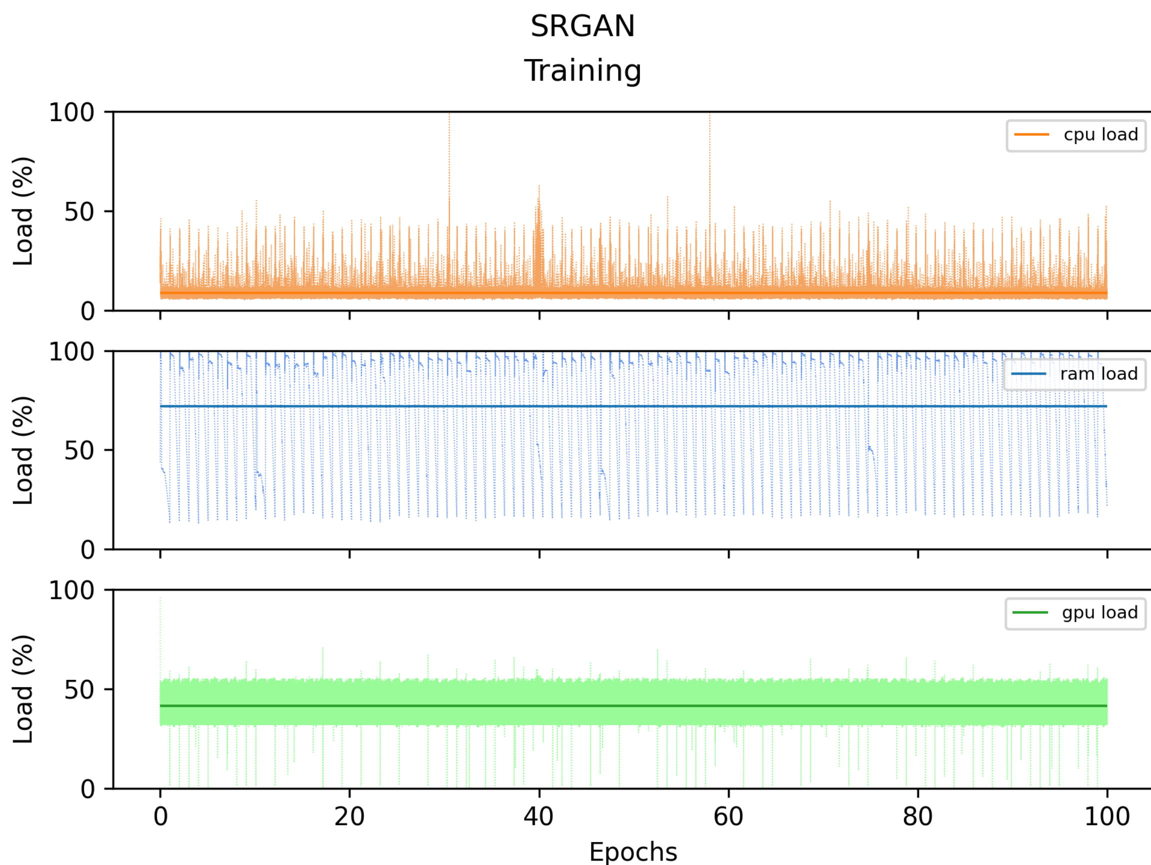


Abbildung 29: Die GPU (grün) ist durchschnittlich nur zu 40% ausgelastet, zu Beginn kann sie aber überlastet werden und abstürzen. Der Arbeitsspeicher (blau) stösst manchmal knapp an die komplettauslastung, was auch einen Absturz zufolge hätte.

mit 10^6 Trainingsschritten trainiert und dann für den Generator von SRGAN die Gewichte des schon vortrainierten SRResNet Modells zum Starten verwendet. Anschliessend wurde es mit $2 \cdot 10^5$ Trainingsschritten weitertrainiert. Die Modelle im Paper wurden also qualitativ besser und mehr trainiert. Es ist deshalb klar, dass meine Methoden nicht mithalten können.

4.2 Vergleich mit der originalen SRGAN Methode

Wie schon gesagt ist ein Vergleich zwischen meiner SRGAN Methode und den originalen Arbeiten nicht ganz fair, da die Qualität, der in meiner Arbeit erzeugten Bilder, tiefer sein wird. Zudem kommt dazu, dass die Auflösungen verschieden sind. Meine SRGAN Methode und die des originalen Papers [19] vergrössern zwar beide mit einer 4-fachen Vergrösserung in beide Richtungen (bzw. 16-fache Vergrösserung), meine Methode benutzt tiefaufgelöste Bilder von 128×128 px und die von Ledig et al. [19] tiefaufgeböste Bilder von 96×96 px. Es stellt sich also die Frage, welche Bilder man für einen Vergleich verwenden soll. Ledig et al. [19] verwendeten zum Testen Bilder aus den Datensätzen **Set5**, **Set14** und **BSD100**. Dies sind nur kleine Datensätzen mit 5, 14 und 100 Bildern. Das folgende Bild aus Set14 ist eines der wenigen Bilder mit einer Auflösung von 512×512. Ich weiss nicht, wie Ledig et al. [19] dieses Bild erzeugt haben, da ihr neuronales Netz nur Bilder von 384×384 Pixeln generiert. Möglicherweise haben sie das Bild in mehrern Stücken erzeugt. Dies ist natürlich ein grosser Vorteil und ihr Bild wird dadurch viel schärfer sein. Trotzdem, ist in Abbildung 30 ein Vergleich dargestellt.

Betrachtet man den Ausschnitt, so sieht man gut, dass meine eigene SRGAN Methode unschärfere Bilder erzeugt als SRGAN von Ledig et al. [19]. Dies war allerdings absehbar. Meine Methode ist aber etwas schärfer als Bicubic. Dafür ist mein Bild etwas stärker verpixelt.

Abbildung 30: Vergleich von verschiedenen SR-Methoden. Von links nach rechts die Bicubic-, eigene SRGAN-, originale SRGAN Methoden sowie das hochaufgelöste Bild. Zudem wurde ein Ausschnitt des Bildes vergrössert dargestellt.

Zusätzlich sind in Tabelle 1 die Gütemasse PSNR und SSIM über die Datensätze BSD100, Set14 und Set5 dargestellt. Die Werte der Gütemasse stimmen mit der Qualität der Bilder überein, man sieht, dass meine Methode wie erwartet weniger gut abschneidet als das originale SRGAN von Ledig et al. [19].

	Eigenes SRGAN		Originales SRGAN	
	PSNR	SSIM	PSNR	SSIM
BSD100	22.58	0.5550	25.16	0.6688
Set14	20.07	0.5295	26.02	0.7397
Set5	24.00	0.6534	29.40	0.8472

Tabelle 1: Methodenvergleich mit Hilfe zweier Gütemasse. Meine SRGAN Methode liefert weniger gute Resultate als die originale SRGAN Methode.

4.3 Die Trainingsprobleme von SRGAN

Wie schon in Abschnitt 3.2 erwähnt, ist es schwierig GANs zu trainieren. Diese Schwierigkeiten im GAN-Training sind sehr gut in den Ergebnissen dieser Arbeit erkennbar. Betrachtet man die Werte des PSNR Gütemasses, so erkennt man, dass das Training instabil war, da über die ganze Trainingszeit die Qualität der Bilder stark schwankte.

Die Qualität der Bilder war etwa bei der 50. Epoche am besten. Man hat also keine Garantie, dass man durch mehr Trainieren bessere Bilder erhält, das Modell konvergiert nicht in Richtung der besten Lösung. Auch hier passen diese Peaks in der Qualität mit den Schwächezeiten des Diskriminators übereinander. In Abbildung 32 ist der Verlustwert des Diskriminators dargestellt. Man sieht deutlich, dass der Diskriminator um die Epoche 50 herum schlecht arbeitet.

Ein weiteres Problem sind Artefakte, welche in den Bildern zu sehen sind. Zum einen sind es Gittermuster (Abbildung 33), welche eine direkte Folge des Pixel-shuffle-Upsamplings sind. Grundsätzlich gehen diese aber mit mehr Training weg.

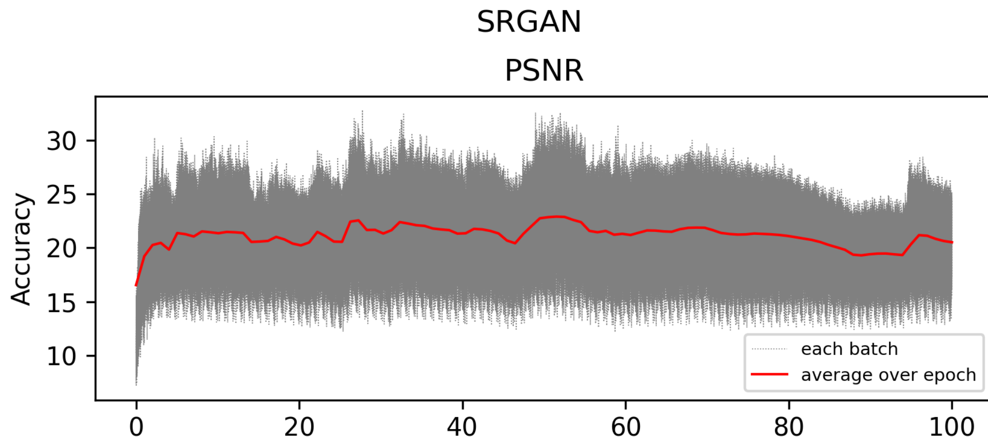


Abbildung 31: Die Qualität der generierten Bilder, ausgedrückt durch das Gütemass PSNR. Es ist sichtbar, dass das Training instabil ist.

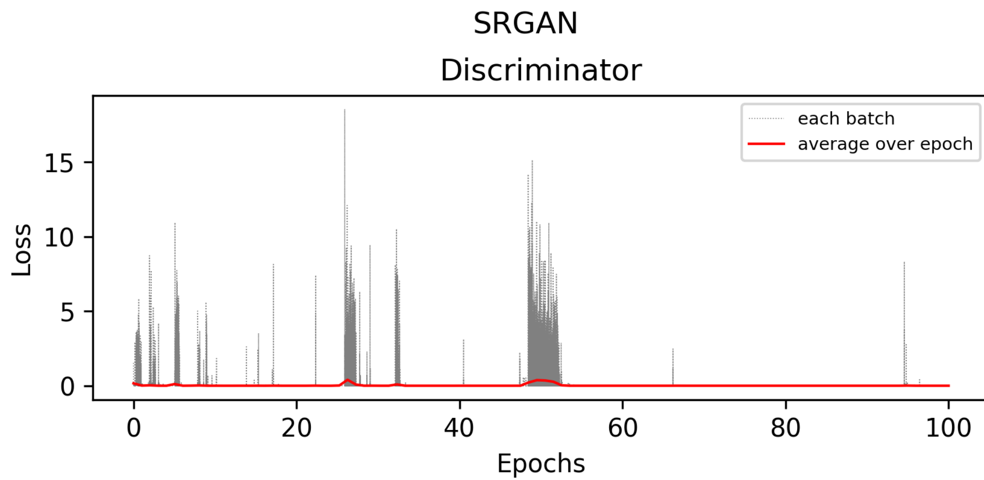


Abbildung 32: Die Verlustwerte des Diskriminators über den ganzen Trainingsvorgang. Die Epochen mit höherem Verlust stimmen mit den Peaks der Qualität überein.

Ganz erklären kann ich mir diese Gittermuster allerdings nicht. Ich nehme aber an, dass sie auch durch die Instabilität des Trainings zustande kommen. Denn eine weitere Quelle der Instabilität ist das Traineneren mit einer zu kleiner Batch-Size. Ein zweiter Artefakt entsteht nur bei einem blauen und einem weissen Hintergrund, wie in Abbildung 34. Diese farbigen Flecken sind typisch für einen zu schwachen Generator und kamen schon in manchen Trainingsversuchen von mir vor. In dem Durchgang, aus dem diese Bilder stammen, sind sie noch nicht so stark ausgeprägt und vergleichsweise klein. Aber mit etwas Glück, könnte dieser Artefakt in einem neuen Trainingsdurchlauf verschwinden.

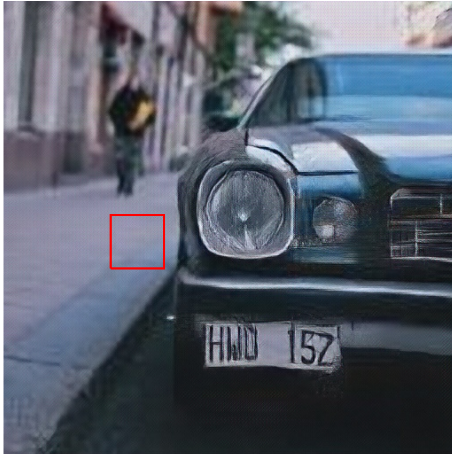


Abbildung 33: Beispiel von Artefakten. Man sieht links ein generiertes Bild von SRGAN, rechts ein Ausschnitt aus diesem Bild.

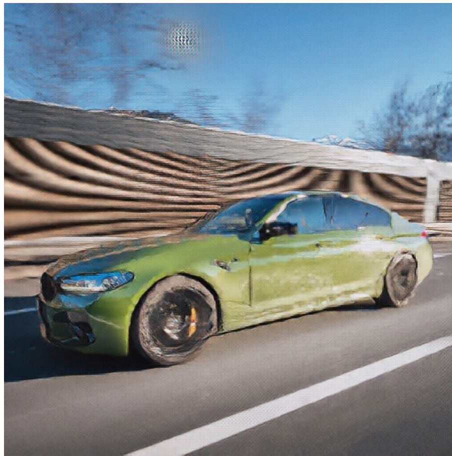


Abbildung 34: Beispiel von Artefakten. Auf blauen und weissen Hintergründen kann es zu merkwürdigen Flecken kommen.

4.4 SRGAN mit Fourier

Das Ziel mit SRGAN Fourier war es, mit Hilfe der Fourier Verlustfunktion, das neuronale Netz noch stärker auf kleine Deteils zu trimmen und eigentlich die VGG19 Verlustfunktion als Verbesserung abzulösen. Leider hat sich herausgestellt, dass die VGG19 Verlustfunktion nicht ersetzt werden kann, aber die beiden Verlustfunktionen harmonieren zusammen gut. Als Nebeneffekt der neuen Verlustfunktion hat sich das Training stark stabilisiert, der Generator und Diskriminator sind besser ausbalanciert. Eine ungewollte, aber hervorragende Verbesserung.

Der Verlustwert des Diskriminators ist nie 0, dass heisst es kann kein Vanishing Gradient entstehen. Ein Nachteil ist aber, dass das Training sehr langsam vorangeht, zum Schluss verbessert sich der Generator nur noch wenig.

Der grosse Nachteil dieser Methode ist die Trainingszeit. Während die SRGAN Methode für eine Epoche etwa 53 Minuten braucht (für 100 Epochen 3.7 Tage), benötigt SRGAN Fourier 122 Minuten (für 100 Epochen 8.5 Tage). Dies ist mehr als die doppelte Trainingszeit.

Wie in Abbildung 36 zu erkennen ist, hat sich die Bild-Ladezeit kaum verändert, aber die Auswertungszeit des neuronalen Netzes und die Anpassung der Gewichte dauert viel länger. Dieser Zeitanstieg liegt hauptsächlich in der Auswertung einer schnellen Fourier-Transformation. Es besteht aber Verbesserungspotential, da diese für jeden Farbkanal einmal ausgeführt wird.

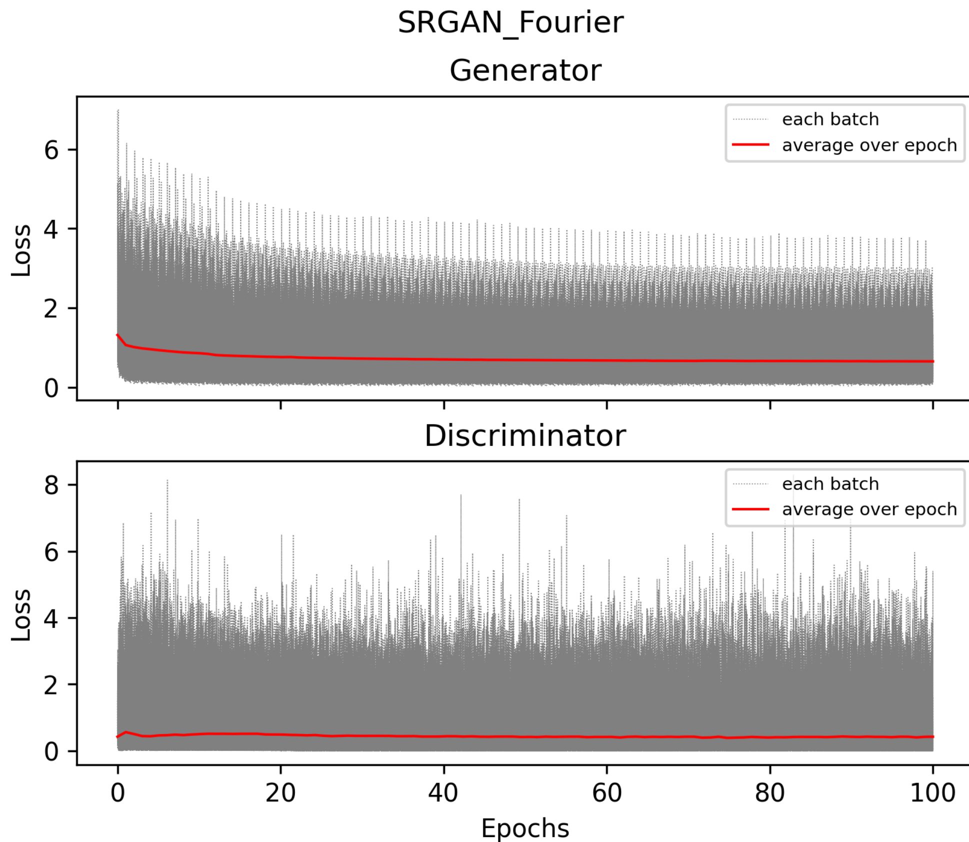


Abbildung 35: Die Verlustwerte verringern sich langsam und monoton. Der Verlustwert des Diskriminators ist nie auf 0.

Würde nur noch eine Fourier-Transformation gemacht werden, was theoretisch möglich sein müsste, dann könnte man sich einen Drittel der zusätzlichen Zeit sparen.

Betrachten wir zunächst die Gütemasse PSNR und SSIM in Tabelle 2.

	SRGAN		SRGAN Fourier	
	PSNR	SSIM	PSNR	SSIM
BSD100	22.58	0.5550	24.25	0.6240
Set14	20.07	0.5295	21.46	0.6032
Set5	24.00	0.6534	26.90	0.7480
Unsplash	23.23	0.5786	27.15	0.7300

Tabelle 2: Methodenvergleich mit den beiden Gütemasse PSNR und SSIM. Die SRGAN Fourier Methode schneidet deutlich besser als die SRGAN Methode ab.

Alle PSNR und SSIM Werte sind bei SRGAN Fourier höher. Auch wenn diese Gütemasse eigentlich mit Vorsicht beurteilt werden sollten, wird ihre Aussage, wie wir später sehen werden, von den erzeugten Bildern unterstützt. In Abbildung 37 wurde ein Bild einer Kamera mit *Bicubic*, *SRGAN* und *SRGAN Fourier* vergrössert. Zudem wurden vier Ausschnitte des Bildes ausgewählt, welche die Unterschiede verdeutlichen.

Das Bicubic Bild ganz Links ist natürlich eher unscharf, enthält aber keine Artefakte wie SRGAN. SRGAN Fourier enthält ebenfalls weniger Artefakte. Der Bereich um den Auslöser der Kamera und die Schrift sind undefiniert, man kann nicht wirklich erkennen was es ist bzw. was es heissen soll.

Betrachten wir den **ersten**, roten, Ausschnitt und vergleichen SRGAN mit SRGAN Fourier,

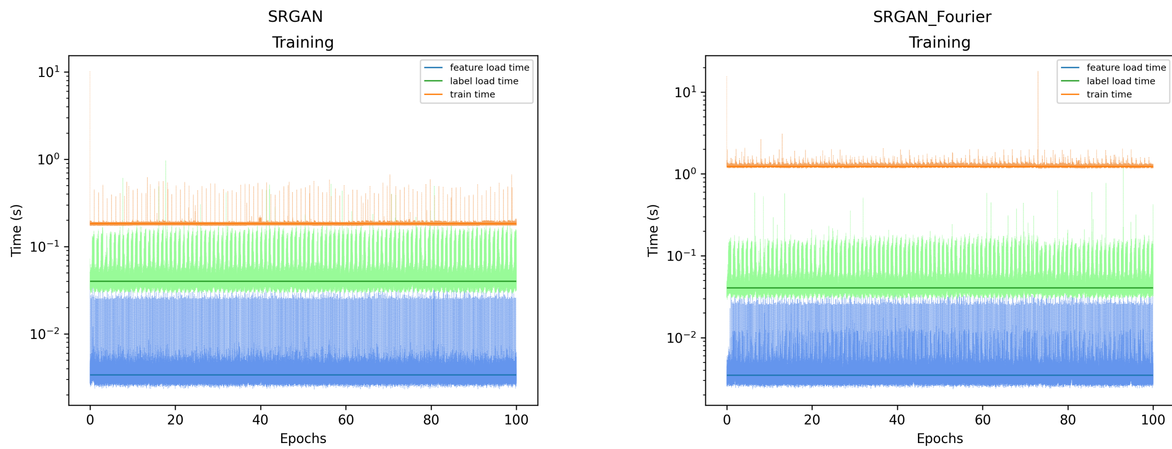


Abbildung 36: Die Zeit, die das Netzwerk braucht, um einen Batch auszuwerten und die Gewichte anzupassen, braucht bei SRGAN Fourier mehr als doppelt so lang wie beim gewöhnlichen SRGAN.

so stellen wir natürlich fest, dass Letzteres viel weniger Bildrauschen enthält. Die einzelnen Objekte wie das Lederband, die Metallschnalle und die Grashalme sind geglättet, aber ohne ineinander zu verfliessen. Die Kanten sind immer noch klar definiert. Dadurch lassen sich auch einzelne Grashalme erahnen. Zudem ist der Kontrast bei SRGAN Fourier höher, was für mich besser aussieht. Verglichen mit dem hochaufgelösten Bild stimmen die Farben (wie das Braun des Lederbandes) bei SRGAN Fourier besser überein als bei SRGAN.

Auch beim **zweiten**, orangen, Ausschnitt ist die Schnalle viel klarer definiert. Man kann erkennen, wie das Lederband über einen Metallhaken befestigt wird. Bei SRGAN ist es dagegen viel zu verpixelt.

Beim **vierten**, blauen, Ausschnitt ist zudem zu erkennen, dass SRGAN Fourier glatte Oberflächen zeichnen kann, aber trotzdem die vordere Kante scharf lässt. Dieser Ausschnitt hat praktisch die gleiche Qualität wie das hochaufgelöste Bild.

Zusammengefasst ist klar, dass SRGAN Fourier die Qualität der Bilder stark steigert. Sie haben keine Artefakte, sind glätter bzw. nicht verpixelt, ohne dabei die Kantenschärfe zu verlieren. Der einzige grosse Nachteil ist die Trainingszeit. Diese könnte aber noch verbessert werden. Die Zeit ein Bild zu generieren ist aber identisch, da die gleiche Architektur für den Generator benutzt wird.

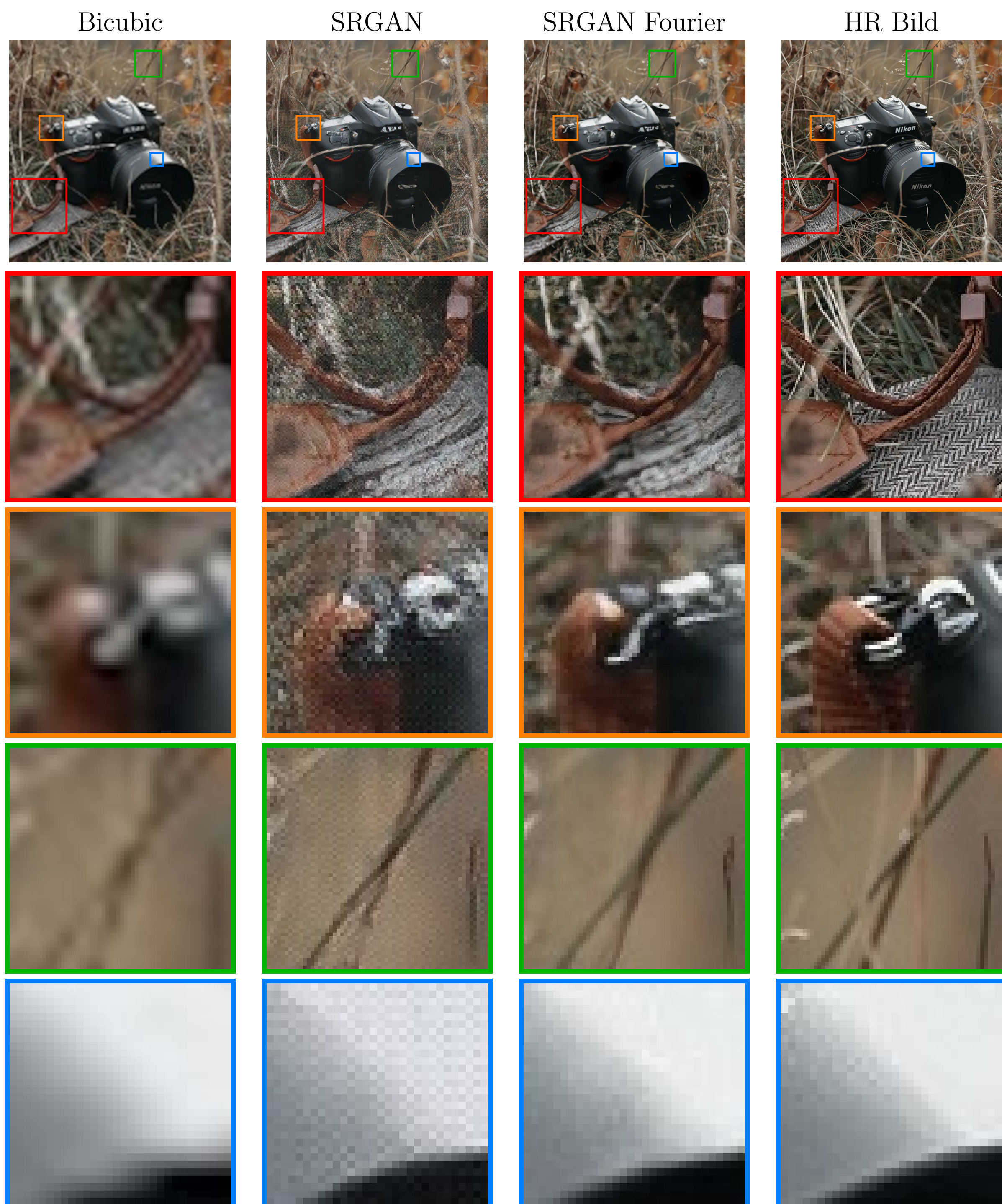


Abbildung 37: Vergleich von verschiedenen SR-Methoden. Von links nach rechts die Bicubic-, SRGAN-, SRGAN Fourier Methoden sowie das hochaufgelöste Bild. Zudem sind vier Ausschnitte des Bildes vergrössert dargestellt.

4.5 SRGAN mit limitiertem Diskriminator

Das Ziel der limitierten SRGAN Methode war es, die Architektur und die Verlustfunktion von SRGAN zu übernehmen und nur den Trainingsprozess abzuändern. Denn wie schon besprochen, ist die SRGAN Methode instabil. Der Diskriminator ist teilweise zu stark, wodurch es zu Vanishing Gradient kommt. Bei meiner eigenen limitierten SRGAN Methode passiert dies nicht. In Abbildung 38 ist erkennbar, dass der Generator sich monoton verbessert.

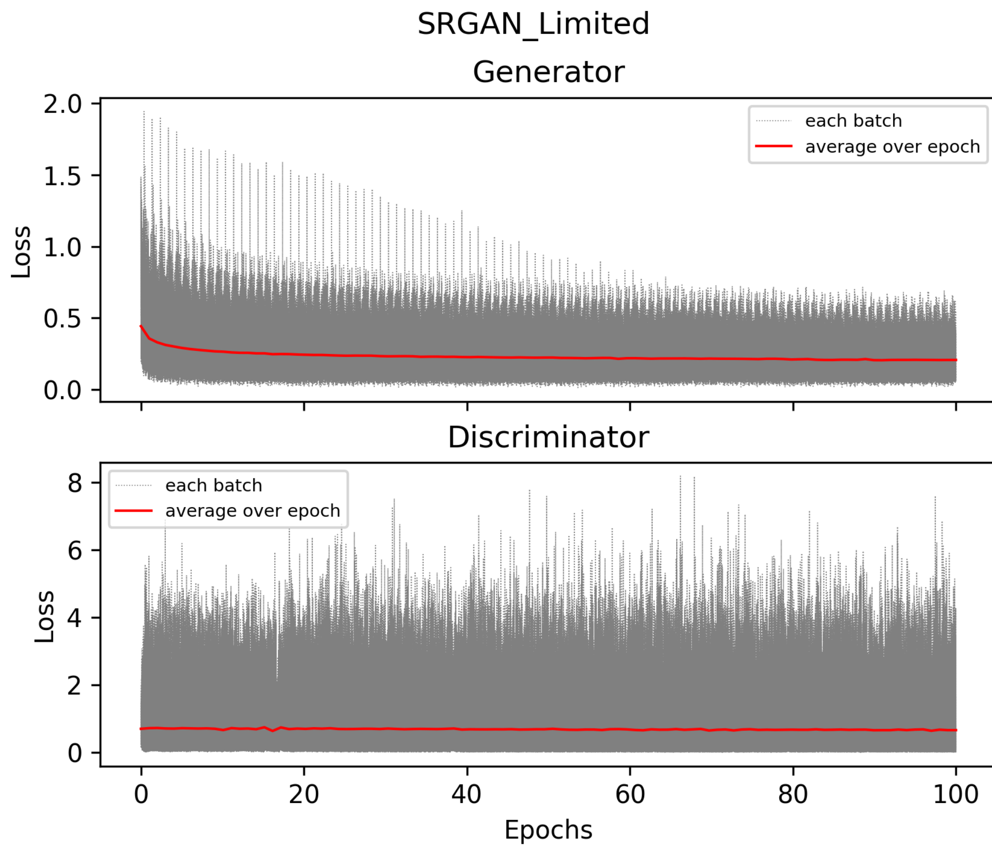


Abbildung 38: Verlauf der Verlustwerte des Generators und des Diskriminators während des Trainings. Die Verlustwerte des Generators verringern sich langsam und monoton. Der mittlere Verlustwert des Diskriminators liegt bei etwa ln(2).

Das heisst, dass die Qualität der generierten Bilder nicht schwankt, wodurch garantiert ist, dass mit mehr Trainingsschritten ein gleich gutes oder besseres Resultat erzielt wird. Der Verlustwert des Diskriminators liegt immer bei etwa $\ln(2) \approxeq 0.693$. Wie in Abschnitt 3.2.1 erwähnt, ist dies der Verlustwert des globalen Optimums.

Die Methode hat ihren Zweck also erfüllt, wie gross die Verbesserung der Bildqualität ist zeigt Abbildung 39.

Was schnell auffällt ist, dass die neue SRGAN Limited Methode viel weniger Artefakte hat. Das Gittermuster aus SRGAN ist verschwunden, nur an wenigen Stellen, z.B. im roten Ausschnitt, sind ein paar Verpixelungsflecken sichtbar. Auch in den anderen Ausschnitten sind bei SRGAN die Bilder stark verpixelt, bei SRGAN Limited dagegen nicht mehr. Im ausgewählten Bild kommt beim Fell SRGAN diese Verpixelung noch ein wenig zu Gute, aber bei anderen Bildern nicht mehr.

Vergleicht man SRGAN Fourier mit SRGAN Limited, so erkennt man nur wenige Unterschiede. Werden die kleinen Auschnitte Pixel für Pixel verglichen, dann gibt es natürlich Differenzen. Grundsätzlich wirken die Bilder aber sehr ähnlich. Der grösste Unterschied ist die Farbsättigung. Im roten und grünen Ausschnitt ist gut erkennbar, dass SRGAN Limited weniger, bzw. zu wenig gesättigte Farben hat. Das Braun des Felles wirkt ausgewaschen. In SRGAN Fourier und im originalen Bild wirken die Farben stärker.

Im orangen Ausschnitt hat das Bild von SRGAN Limited ein wenig mehr Kontrast und ist weniger geglättet als SRGAN Fourier. Im grünen Auschnitt erzeugt die limitierte SRGAN Methode eine Felltextur, welche zwar überhaupt nicht mit dem originalen Bild übereinstimmt, aber trotzdem detailreich ist. SRGAN Fourier ist oben links im Gegensatz dazu viel glatter.

Welche Methode nun besser ist, ist schwierig zu sagen. SRGAN Fourier erzeugt etwas glattere Bilder, was aber meiner Meinung nach auf die Wirkung des Bildes nicht viel Einfluss hat. Die Gütemass-Werte bestätigen leicht bessere Werte für SRGAN Fourier (Tabelle 3).

	SRGAN Fourier		SRGAN Limited	
	PSNR	SSIM	PSNR	SSIM
BSD100	24.25	0.6240	23.71	0.6134
Set14	21.46	0.6032	21.05	0.5969
Set5	26.90	0.7480	26.05	0.7316
Unsplash	27.15	0.7300	26.15	0.7190

Tabelle 3: Methodenvergleich mit Hilfe von zwei Gütemasse. Die SRGAN Fourier Methode schneidet etwas besser als die SRGAN Limited Methode ab. Dies muss aber nicht heissen, dass die Bilder in menschlicher Wahrnehmung wirklich besser sind.

Der PSNR Wert von SRGAN Fourier ist höher als der von SRGAN Limited und da dieser auf MSE basiert, präferiert er glatte Bilder. Die Farben haben schon einen grösseren Einfluss, diese könnten aber auch einfach in einem Bildbearbeitungsprogramm im nachhinein verbessert werden. SRGAN Fourier hat aber überhaupt keine Artefakte, SRGAN Limited dagegen schon.

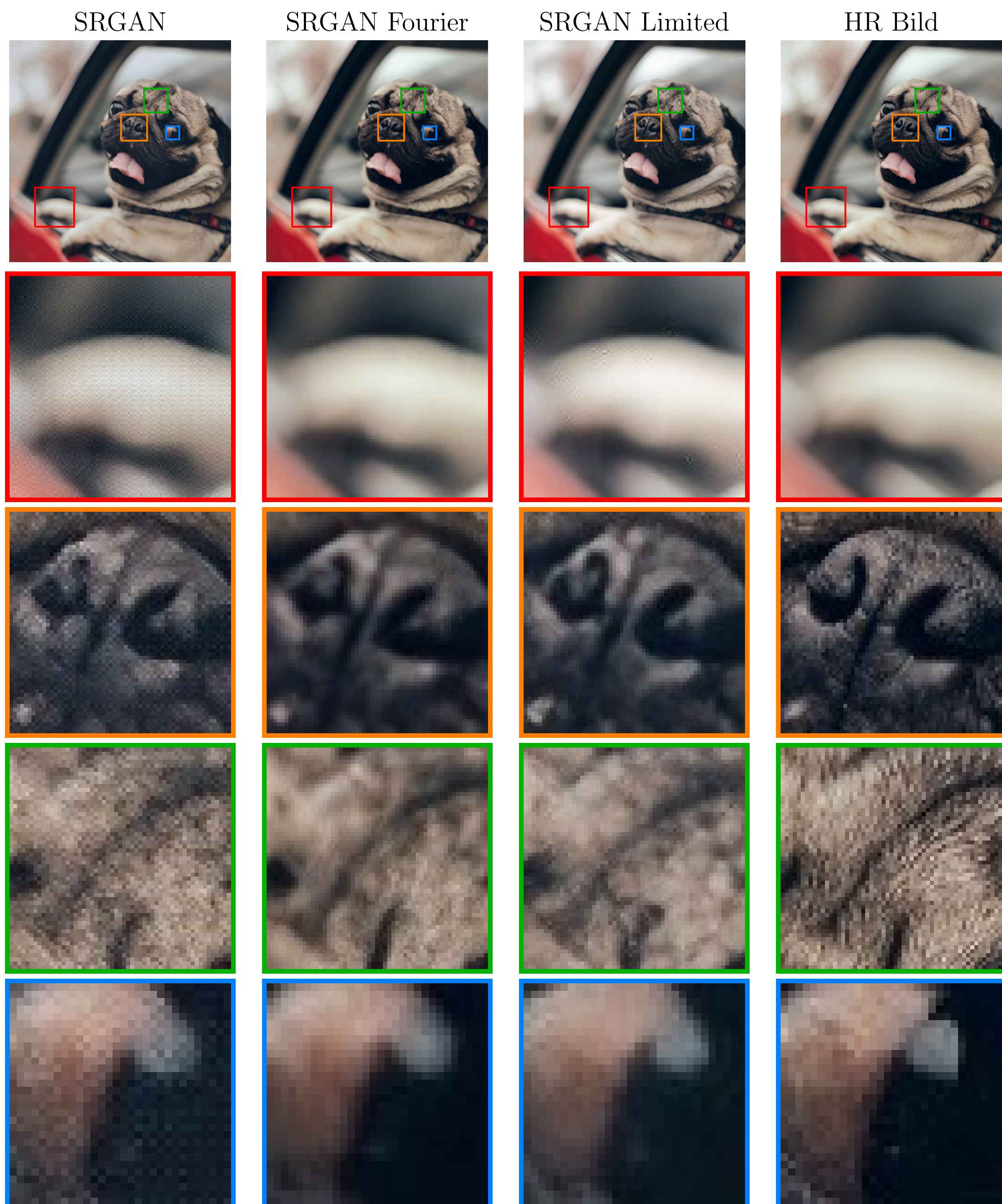


Abbildung 39: Vergleich von verschiedenen SR-Methoden. Von links nach rechts die SRGAN, SRGAN Fourier und SRGAN Limited Methoden sowie das hochaufgelöste Bild. Zudem sind vier Ausschnitte des Bildes vergrössert dargestellt.

4.6 SRResNet

Nun haben wir die Resultate aller komplizierten GAN-Methoden besprochen. Doch wie gut schneiden im Gegensatz dazu die einfacheren SRResNet Modelle ab? Diese haben keinen Diskriminator sondern nur eine analytische Verlustfunktion. Die *SRResNet* Methode benutzt die MSE-Verlustfunktion und die *SRResNet Fourier* benutzt die Fourier-Verlustfunktion. Von der normalen SRResNet Methode können wir nicht viel erwarten, da wir ja schon wissen, dass die generierten Bilder zu glatt sind und die Details fehlen. Versuchen wir nun trotzdem einmal, anhand der Abbildung 40, die Unterschiede aufzuzeigen.

Im **ersten**, rot gekennzeichneten, Bild, sieht man, dass SRGAN Fourier das schärfste Bild generiert. Das zweit schärfste Bild ist das von SRResNet gefolgt von SRResNet Fourier. Aber da SRGAN Fourier die meisten Details erzeugt, gibt es verpixelte Stellen die unrealistisch wirken. Weil bei SRResNet und SRResNet Fourier die Bilder stärker geglättet sind, können solche Artefakte gar nicht auftreten. Im roten Ausschnitt sieht das SRResNet Bild am besten bzw. nicht merkwürdig aus.

Diese erwähnten Artefakte liegen aber vor allem an Reflektionen an der Seitenpanele. Solche Reflektionen sind sehr schwierig zu erlernen. Im **vierten**, blau gekennzeichneten, Auschnitt erzeugt SRGAN Fourier sehr scharfe Kanten, viele Details und das ohne merkwürdige Artefakte zu generieren. Damit können SRResNet und SRResNet Fourier nicht mithalten.

Im **dritten**, grün gekennzeichneten, Ausschnitt kann man den Vorteil der Fourier-Verlustfunktion erkennen. SRGAN Fourier erzeugt schöne Rillen, auch wenn die des Originalbildes viel feiner sind. SRResNet hat im rechten Bereich Mühe und erzeugt einen unstrukturierten Bereich. Bei SRResNet Fourier kann man Rillen schon eher erkennen. Leider gibt es auch hier im rechten Bereich ein Gittermuster.

Schlussendlich wird klar sichtbar, dass SRGAN Fourier die schrärfsten Bilder mit den meisten Details erzeugt. Die SRResNet Methoden generieren glattere Bilder. Manchmal kann dies aber auch ein Vorteil sein, wenn man auf keinen Fall merkwürdige Artefakte will und die fehlenden Details in Kauf nimmt. Vergleicht man die beiden SRResNet Methoden, so erkennt man, dass die Fourier-Verlustfunktion eigentlich nichts bringt. Die Bilder sind einfach noch glatter und unschärfer. Zudem ist die Trainingszeit viel länger. SRResNet braucht für eine Epoche etwa 42 Minuten (für 100 Epochen 2.9 Tage), aber die SRResNet Fourier Methode braucht für eine Epoche etwa 113 Minuten (für 100 Epochen 7.9 Tage). Der tatsächliche Rechenaufwand für SRResNet Fourier lohnt sich nicht.

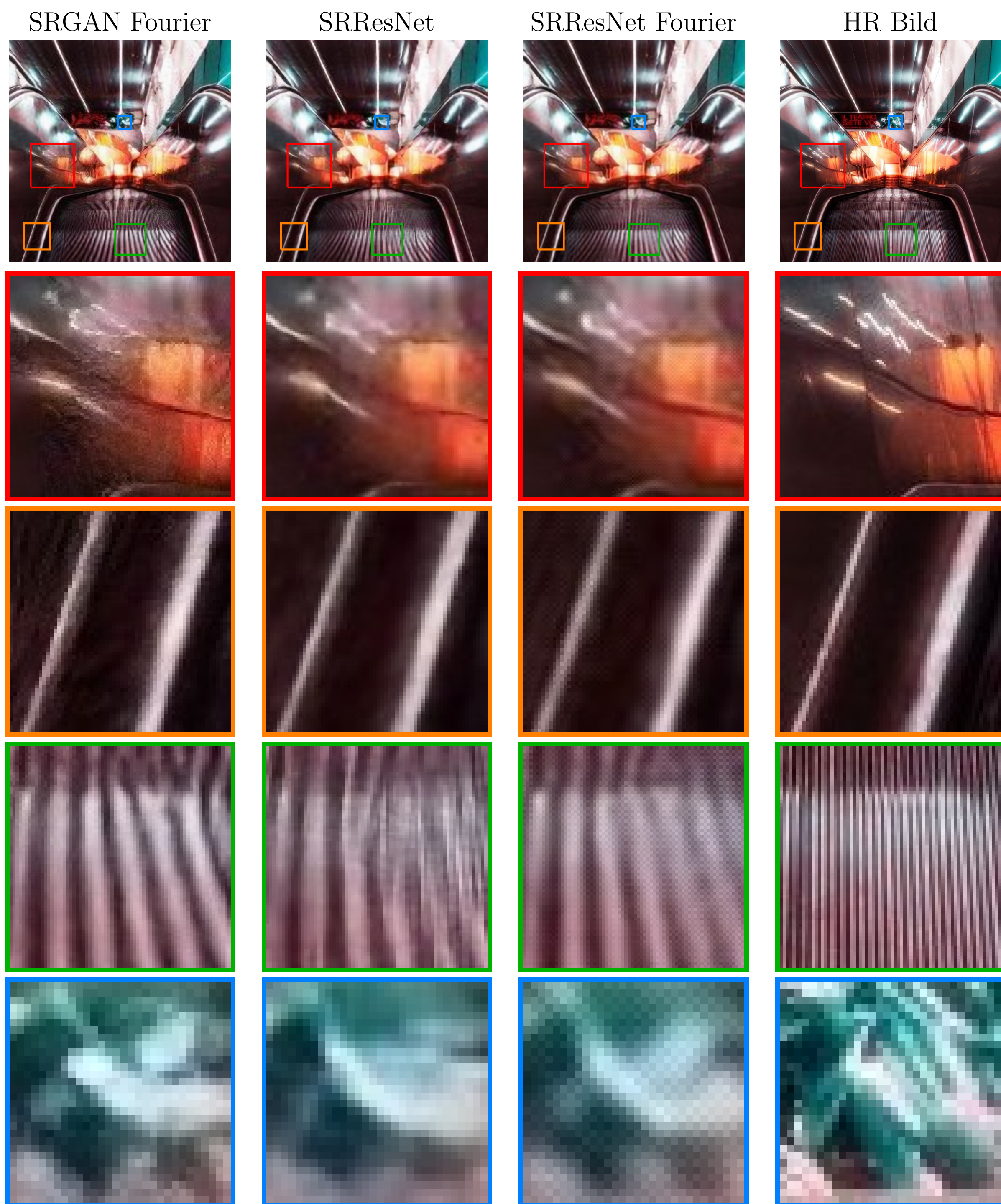


Abbildung 40: Vergleich von verschiedenen SR-Methoden. Von links nach rechts die SRGAN Fourier, SRResNet und SRResNet Fourier Methoden sowie das hochaufgelöste Bild. Zudem sind vier Ausschnitte des Bildes vergrössert dargestellt.

4.7 Schlussfolgerung

Diese Arbeit war für mich ein schwieriges Stück Arbeit und ich bin froh, dass ich diese nun abschliessen konnte. Eine Hauptschwierigkeit lag in der Theorie der verwendeten Methoden. Ich habe zuerst angefangen mit den Algorithmen zu arbeiten und diese auszuprobieren, ohne die Theorie dahinter zu verstehen. Als ich mich dann definitiv für das Thema entschieden hatte, musste ich mich mit der Theorie genauer befassen. Das war sehr schwierig, da mir die Grundlagen dazu fehlten. Ich habe einige Paper gelesen und ich denke, dass ich die Methoden im wesentlichen verstanden habe. Bei der praktischen Arbeit ergaben sich ebenfalls Schwierigkeiten, die ich überwinden musste. Beispielsweise musste ich verschiedene Möglichkeiten ausprobieren, um den Unsplash Datensatz effizient herunterzuladen und lokal abzulegen. Viel Spass hat mir das Nachdenken über die Softwarearchitektur, sowie das anschliessende Programmieren in Python gemacht. Ich konnte zwar bereits recht gut programmieren, aber ein so grosses und strukturiertes Programm habe ich bisher noch nicht geschrieben. Ich habe einiges dabei gelernt. Gefreut hat mich auch, dass sich die Idee einer Verlustfunktion basierend auf der Fourier-Transformation umsetzen liess und diese angepasste Methode gut funktionierte. Bei Herrn Dr. Jörg Bader möchte ich mich gerne für die Betreuung bedanken, seine Rückmeldungen und Vorschläge waren sehr hilfreich. Meiner Familie möchte ich ebenfalls für die Ratschläge und die Diskussionen danken.

Literatur

[1] Kreuzentropie. https://de.wikipedia.org/wiki/Kreuzentropie. Accessed: 2022-09-17. 14

[2] Multiprocessing - process-based parallelism. https://docs.python.org/3/library/multiprocessing.html. Accessed: 2022-09-15. 22

[3] Understanding generative adversarial networks (gans). https://towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29. Accessed: 2022-08-03. 10, 11

[4] Hierarchical data format. https://de.wikipedia.org/wiki/Hierarchical_Data_Format, Sep 2021. Accessed: 2022-08-16. 22

[5] Bicubic interpolation. https://en.wikipedia.org/wiki/Bicubic_interpolation, Sep 2022. Accessed: 2022-09-22. 7

[6] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. 17

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. 21

[8] Dario Fuoli, Luc Gool, and Radu Timofte. Fourier space losses for efficient perceptual image super-resolution. pages 2340–2349, 10 2021. 29

[9] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016. 9, 14, 27

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. 10, 14, 20

[11] Hayit Greenspan. Super-resolution in medical imaging. *The computer journal*, 52(1):43–63, 2009. 5

[12] Hayit Greenspan, G Oz, N Kiryati, and SLBG Peled. Mri inter-slice reconstruction using super-resolution. *Magnetic resonance imaging*, 20(5):437–446, 2002. 5

[13] Dianyuan Han. Comparison of commonly used image interpolation methods. In *Conference of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 1556–1559. Atlantis Press, 2013. 7

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016. 18, 48

[15] Jithin Saji Isaac and Ramesh Kulkarni. Super resolution techniques for medical image processing. In *2015 International Conference on Technologies for Sustainable Development (ICTSD)*, pages 1–6. IEEE, 2015. 5

[16] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4401–4410, 2019. 22

[17] Hansruedi Künsch, Nora Mylonas, Hansjürg Stocker, Eva Frenzel, and Fabian Glötzner. *Stochastik*. DMK Deutschschweizerische Mathematikkommission des VSMP, 1 edition, 2018. 10

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015. 8

[19] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017. 9, 16, 17, 20, 21, 22, 29, 31, 32, 33, 48

[20] Zheng Li, Yongcheng Wang, Ning Zhang, Yuxi Zhang, Zhikang Zhao, Dongdong Xu, Guangli Ben, and Yunxiao Gao. Deep learning-based object detection techniques for remote sensing images: A survey. *Remote Sensing*, 14(10):2385, 2022. 5

[21] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015. 22

[22] Patric Müller. Woche 7: Maximum-likelihood-schätzung. https://ethz.ch/content/dam/ethz/special-interest/math/statistics/sfs/Education/AdvancedStudiesinAppliedStatistics/course-material-1921/W'keitundStatistik/slides07.pdf, Jun 2019. ETHZ. 12

[23] Kamal Nasrollahi and Thomas B Moeslund. Super-resolution: a comprehensive survey. *Machine vision and applications*, 25(6):1423–1468, 2014. 5

[24] Omkar M. Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. In *British Machine Vision Conference*, 2015. 22

[25] Tariq Rashid. *GANs mit PyTorch selbst programmieren*, volume 1. O'Reilly, 2020. 27

[26] Daniel Rowe. Bilinear, bicubic, and in between spline interpolation. https://www.mssc.mu.edu/~daniel/pubs/RoweTalkMSCS_BiCubic.pdf, Feb 2018. Accessed: 2022-09-22. 8

[27] Lothar Schermelleh, Alexia Ferrand, Thomas Huser, Christian Eggeling, Markus Sauer, Oliver Biehlmaier, and Gregor PC Drummen. Super-resolution microscopy demystified. *Nature cell biology*, 21(1):72–84, 2019. 5

[28] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *CoRR*, abs/1609.05158, 2016. 20

[29] Yaron M Sigal, Ruobo Zhou, and Xiaowei Zhuang. Visualizing and discovering cellular structures with super-resolution microscopy. *Science*, 361(6405):880–887, 2018. 5

[30] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 21

[31] Anushka Singh. Gan: Generative adversarial network. https://medium.com/analytics-vidhya/gan-generative-adversarial-network-fbef2a96e183, Jul 2021. Accessed: 2022-10-28. 15

[32] Unsplash. Access the world's largest open library dataset for free. https://unsplash.com/data. Accessed: 2022-09-11. 22

[33] Eric Van Reeth, Ivan WK Tham, Cher Heng Tan, and Chueh Loo Poh. Super-resolution in magnetic resonance imaging: a review. *Concepts in Magnetic Resonance Part A*, 40(6):306–325, 2012. 5

[34] Zhihao Wang, Jian Chen, and Steven CH Hoi. Deep learning for image super-resolution: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 43(10):3365–3387, 2020. 8, 23, 24

[35] Edmund Weitz. *Fourier-Analysis*, page 649–667. Springer, 2018. 29

[36] Lilian Weng. From GAN to WGAN. *CoRR*, abs/1904.08994, 2019. 27

[37] Liangpei Zhang, Hongyan Zhang, Huanfeng Shen, and Pingxiang Li. A super-resolution reconstruction algorithm for surveillance images. *Signal Processing*, 90(3):848–859, 2010. 5

Abbildungsverzeichnis

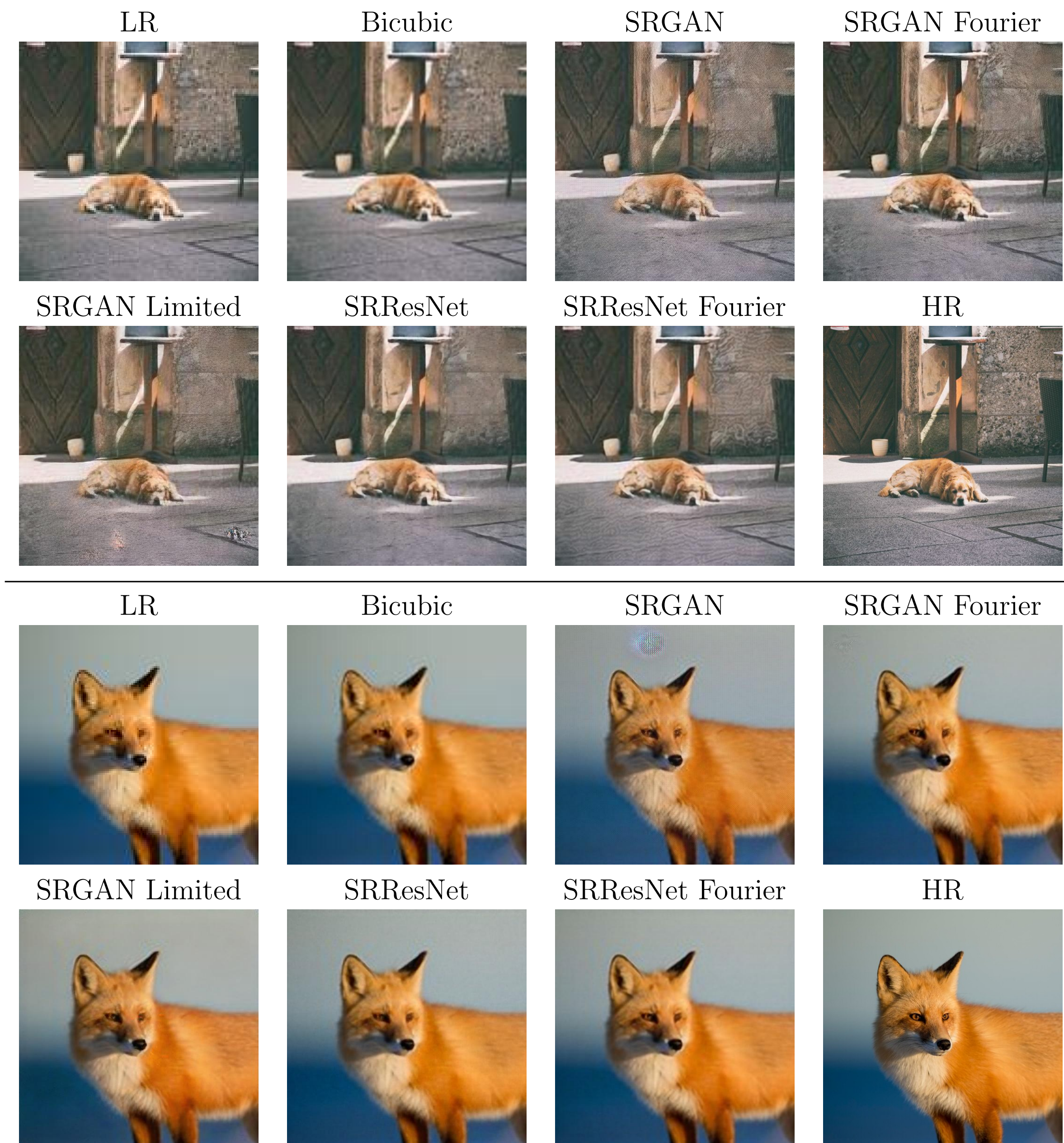
1 Das Eingabebild (128 × 128) wird per Super-Resolution zum generierten Bild vergrössert (512 × 512). . . . 6
2 Zwischen die schon vorhandenen Pixel müssen weitere Pixel eingefügt werden. . . . 7
3 Zwischen dieser Reihe von Bildpunkten soll interpoliert werden. . . . 7
4 Eine lineare Interpolation zwischen x_i und x_{i+1}. . . . 8
5 Eine kubische Interpolation zwischen x_i und x_{i+1}. . . . 8
6 Das mittlere Bild (SRResNet) ist mit der Verlustfunktion MSE erzeugt und wirkt geglättet. . . . 9
7 Illustration einer Wahrscheinlichkeitsverteilung am Beispiel der Normalverteilung. . . 11
8 Die Zufallsvariable $Y = F^{-1}(X)$ soll der Verteilung F folgen. . . . 11
9 Die Analogie des Geldfälschens verglichen mit dem GAN-Algorithmus angewandt auf Super-Resolution. . . . 15
10 Die Architektur des Generators der SRGAN Methode aus [19]. . . . 16
11 Die Architektur des Diskriminators der SRGAN Methode aus [19]. . . . 17
12 Ein Filter (grün) wandert mit vorgegebener Grösse (3 × 3 px) und Schrittweite (1 px) über das ganze Bild. . . . 17
13 Ein Convolutional Layer erkennt Linien bzw. Kanten und Formen, indem stückweise das Bild mit einem Filter, welcher Strukturen im Bild via Skalarprodukt entdecken kann, „abgetastet“ wird. . . . 18
14 Vergleich von zwei neuronalen Netzen mit unterschiedlicher Anzahl Schichten (Layer). Auf den x-Achsen sind die Anzahl Trainingsschritte abgebildet. Links bezeichnet die y-Achse den Trainingsfehler und rechts den Testfehler. In beiden Abbildungen ist zu erkennen, dass das 20-schichtige Netz besser abschneidet als das Grössere. Abbildung aus [14]. . . . 18
15 Schematische Darstellung eines Residual Blocks der aus zwei Convolutional Layer bestehtDurch die skip connection wird der Eingabewert wieder hinzuaddiert. Dadurch müssen nur die Differenzen gelernt werden. . . . 19
16 Die Feature Maps werden aneinandergereiht, wodurch das Bild grösser wird. . . . 19
17 Bilder mit unterschiedlichen Gewichtungen der Verlustfunktionen V_{con} und V_{adv}. Links ist das Gewicht von V_{con} viel zu klein, rechts viel zu gross. . . . 21
18 Links das Eingabebild, rechts 10 von 256 Feature Maps. . . . 21
19 So sieht vereinfacht der Aufbau der SRGAN Methode aus. . . . 25
20 Das Laden der Bilder und das Training wird hintereinander ausgeführt. . . . 25
21 Das Laden der Bilder und das Training wird gleichzeitig ausgeführt. Einfachheitshalber wird die .hdf5 Datei aber wieder geschlossen. . . . 25
22 Das Laden der Bilder und das Training wird gleichzeitig ausgeführt. Die .hdf5 Datei wird aber erst am Ende der Epoche geschlossen. . . . 26
23 Das Laden der Bilder dauert viel kürzer als die Auswertung und die Anpassung der Gewichte des neuronalen Netzes. . . . 26
24 Statistiken während eines eigenen Trainingdurchgangs eines SRGAN Modells. . . . 28
25 Verschiedene harmonische Schwingungen. . . . 29
26 Die Summer der harmonischen Schwingungen aus Abbildung 25. . . . 30
27 Fourier Bilder mit unterschiedlichen Frequenzbereichen. In der Mitte wurden tiefe ($0-8$-fache Frequenz der Grundschwingung) und rechts hohe Frequenzen ($2-256$-fache Frequenz der Grundschwingung) zurücktransformiert. . . . 30
28 Fourier Bilder mir einem Frequenzbereich von $0-5$, $5-38$, $38-102$ und $102-256$-facher Frequenz der Grundschwingung (oben links nach unten rechts). . . . 31

29 Die GPU (grün) ist durchschnittlich nur zu 40% ausgelastet, zu Beginn kann sie aber überlastet werden und abstürzen. Der Arbeitsspeicher (blau) stösst manchmal knapp an die komplettauslastung, was auch einen Absturz zufolge hätte. 32
30 Vergleich von verschiedenen SR-Methoden. Von links nach rechts die Bicubic-, eigene SRGAN-, originale SRGAN Methoden sowie das hochaufgelöste Bild. Zudem wurde ein Ausschnitt des Bildes vergrössert dargestellt. 33
31 Die Qualität der generierten Bilder, ausgedrückt durch das Gütemass PSNR. Es ist sichtbar, dass das Training instabil ist. 34
32 Die Verlustwerte des Diskriminators über den ganzen Trainingsvorgang. Die Epochen mit höherem Verlust stimmen mit den Peaks der Qualität überein. 34
33 Beispiel von Artefakten. Man sieht links ein generiertes Bild von SRGAN, rechts ein Ausschnitt aus diesem Bild. 35
34 Beispiel von Artefakten. Auf blauen und weissen Hintergründen kann es zu merkwürdigen Flecken kommen. 35
35 Die Verlustwerte verringern sich langsam und monoton. Der Verlustwert des Diskriminators ist nie auf 0. 36
36 Die Zeit, die das Netzwerk braucht, um einen Batch auszuwerten und die Gewichte anzupassen, braucht bei SRGAN Fourier mehr als doppelt so lang wie beim gewöhnlichen SRGAN. 37
37 Vergleich von verschiedenen SR-Methoden. Von links nach rechts die Bicubic-, SRGAN-, SRGAN Fourier Methoden sowie das hochaufgelöste Bild. Zudem sind vier Ausschnitte des Bildes vergrössert dargestellt. 38
38 Verlauf der Verlustwerte des Generators und des Diskriminators während des Trainings. Die Verlustwerte des Generators verringern sich langsam und monoton. Der mittlere Verlustwert des Diskriminators liegt bei etwa ln(2). 39
39 Vergleich von verschiedenen SR-Methoden. Von links nach rechts die SRGAN, SRGAN Fourier und SRGAN Limited Methoden sowie das hochaufgelöste Bild. Zudem sind vier Ausschnitte des Bildes vergrössert dargestellt. 41
40 Vergleich von verschiedenen SR-Methoden. Von links nach rechts die SRGAN Fourier, SRResNet und SRResNet Fourier Methoden sowie das hochaufgelöste Bild. Zudem sind vier Ausschnitte des Bildes vergrössert dargestellt. 43

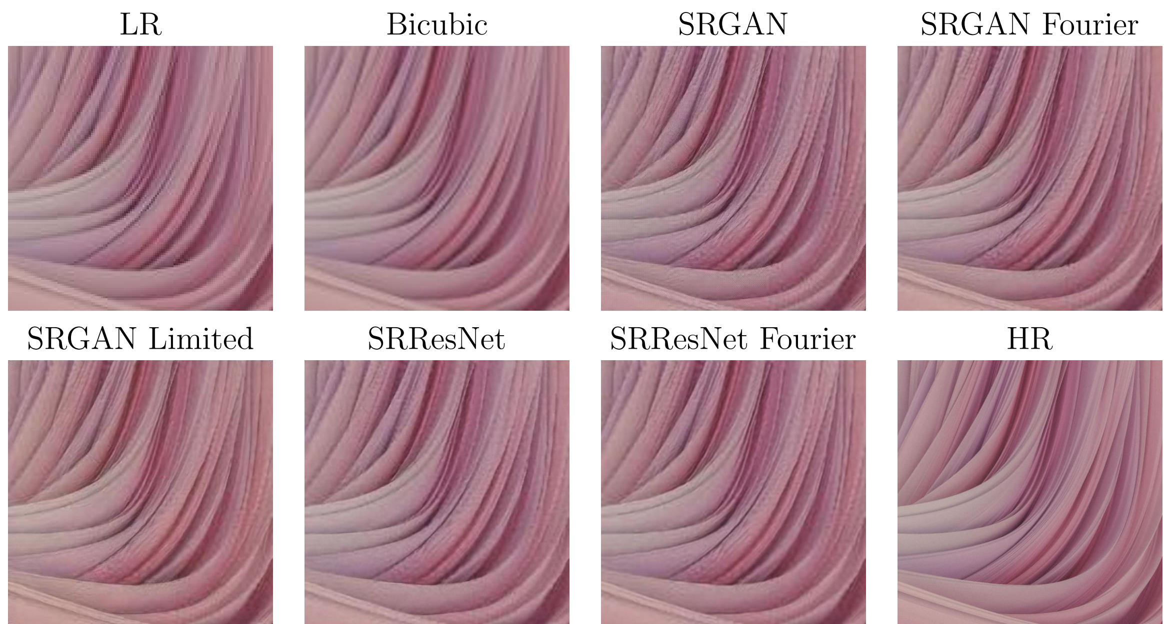
Anhang

Auf den nachfolgenden Seiten werden einige Bilder gezeigt, welche mit den in dieser Arbeit untersuchten Methoden generiert wurden. Zum Vergleich sind zudem die Ausgangsbilder in tiefer Auflösung (LR, 128×128), sowie als Referenz die selben Bilder in der hohen Auflösung (HR, 512×512) gezeigt.

Leider sind die Unterschiede in der gedruckten Version dieser Arbeit praktisch nicht erkennbar. Es wird daher empfohlen die Bilder in der digitalen Version (PDF) zu betrachten.

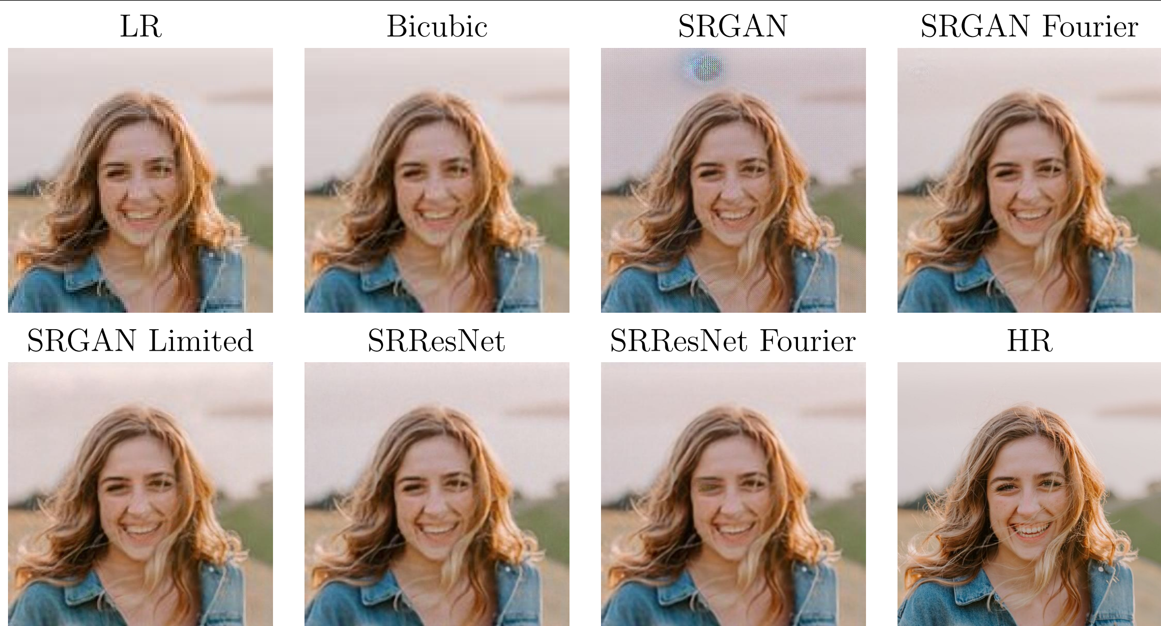


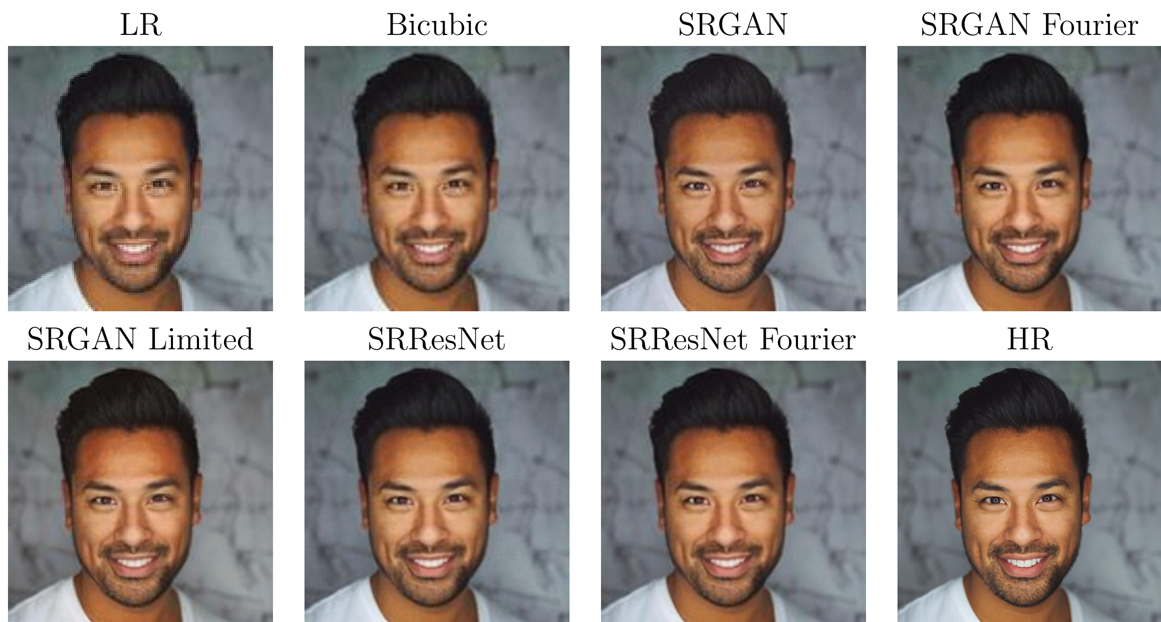
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

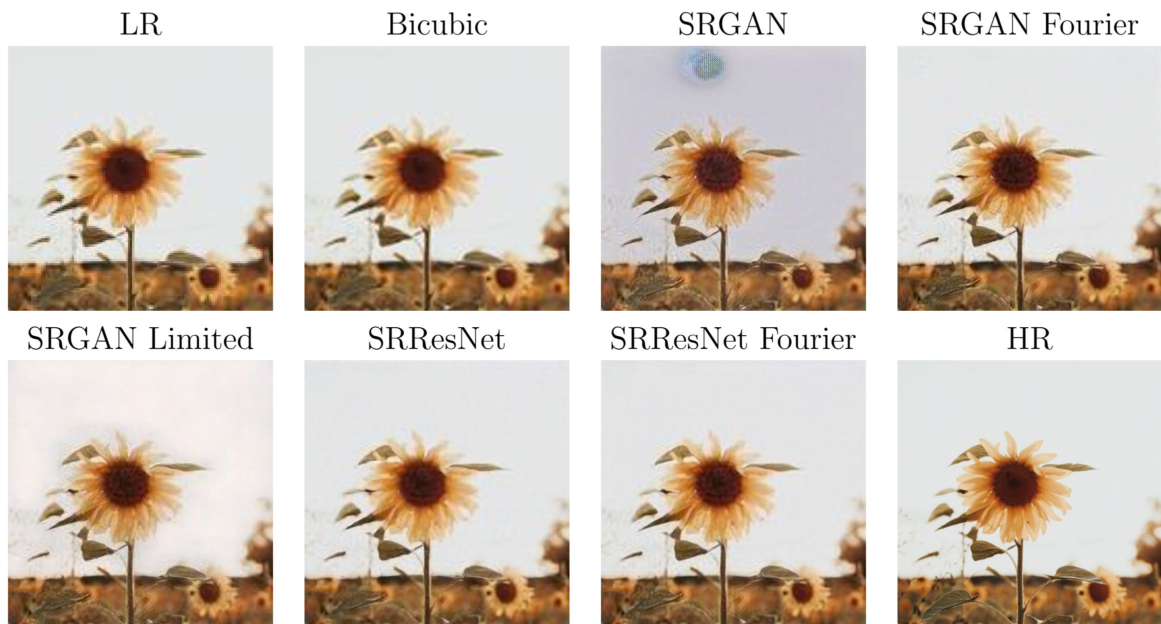
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

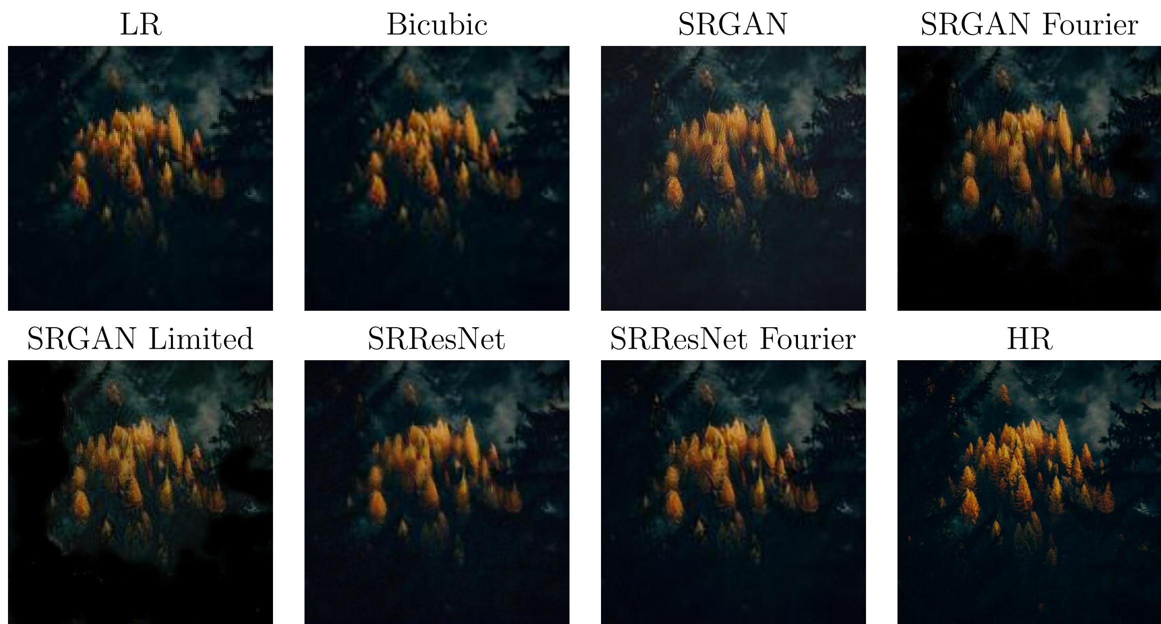

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

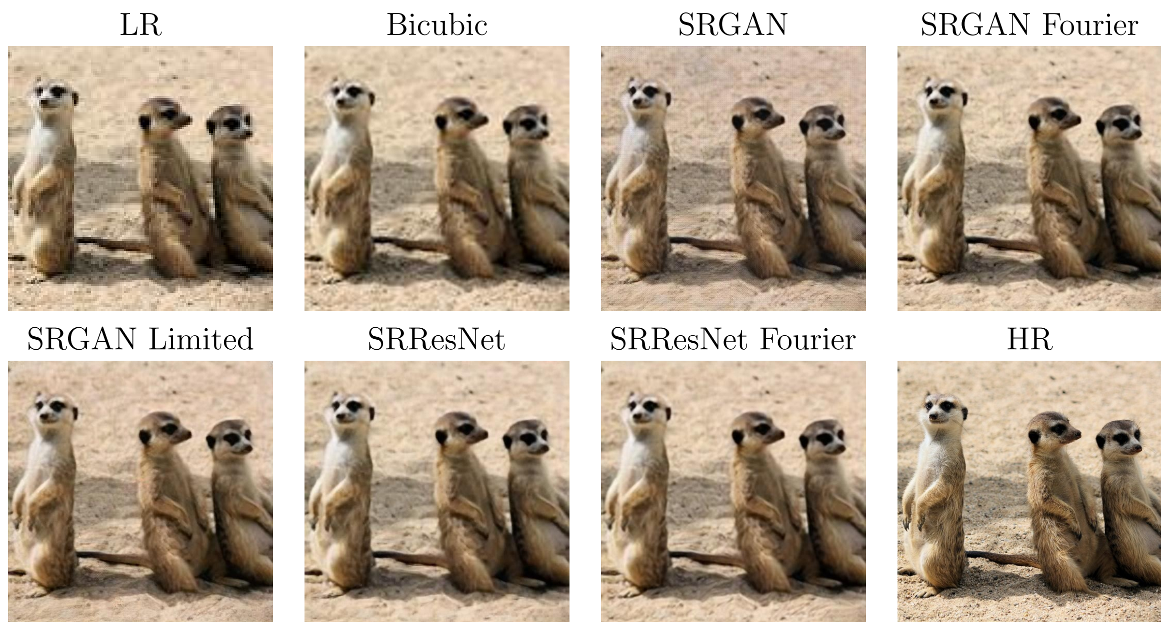
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

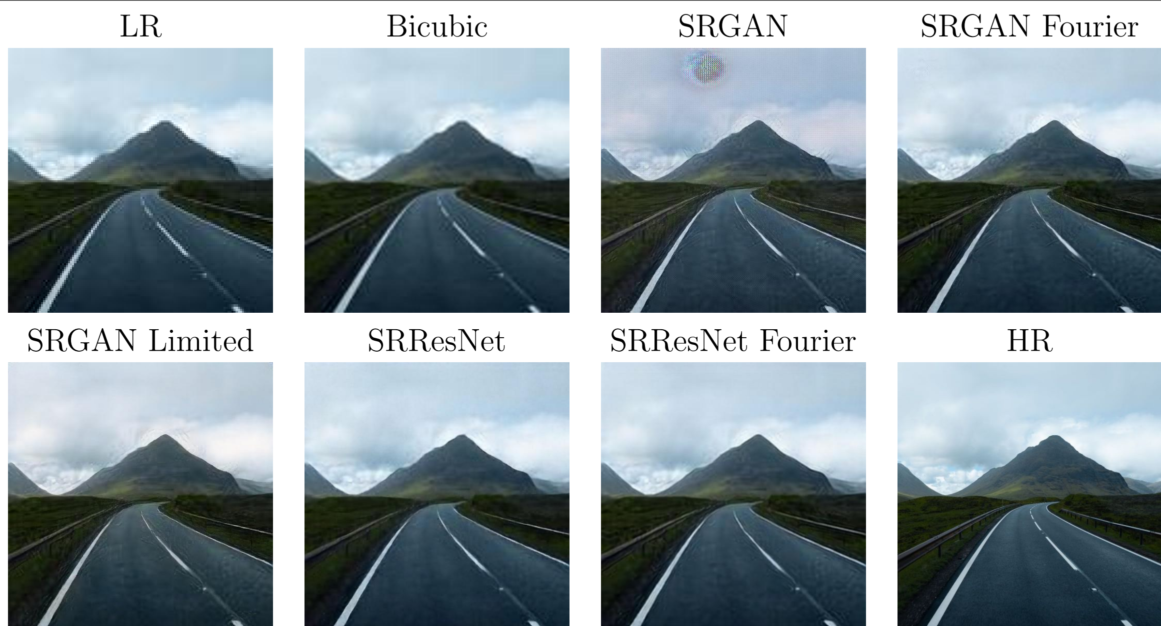
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

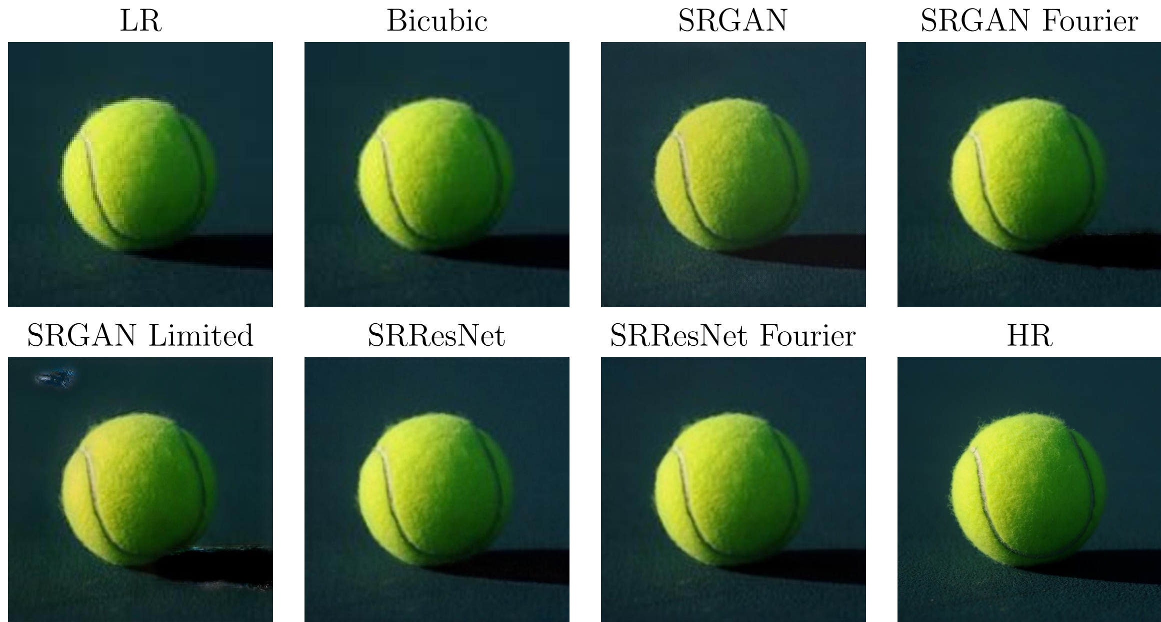

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

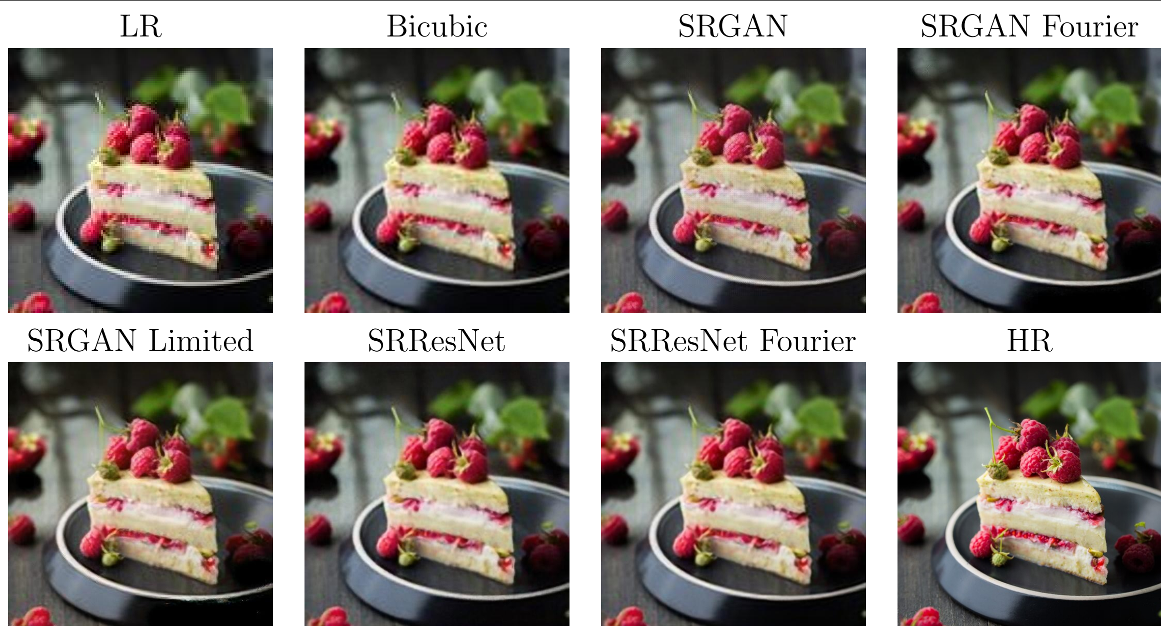

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

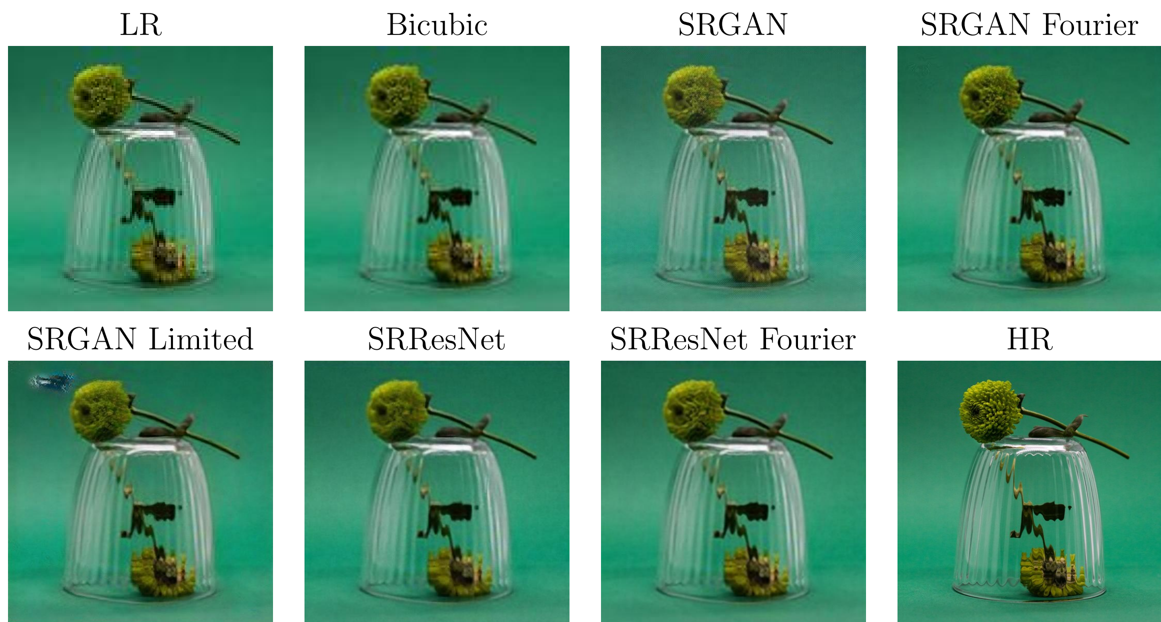
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

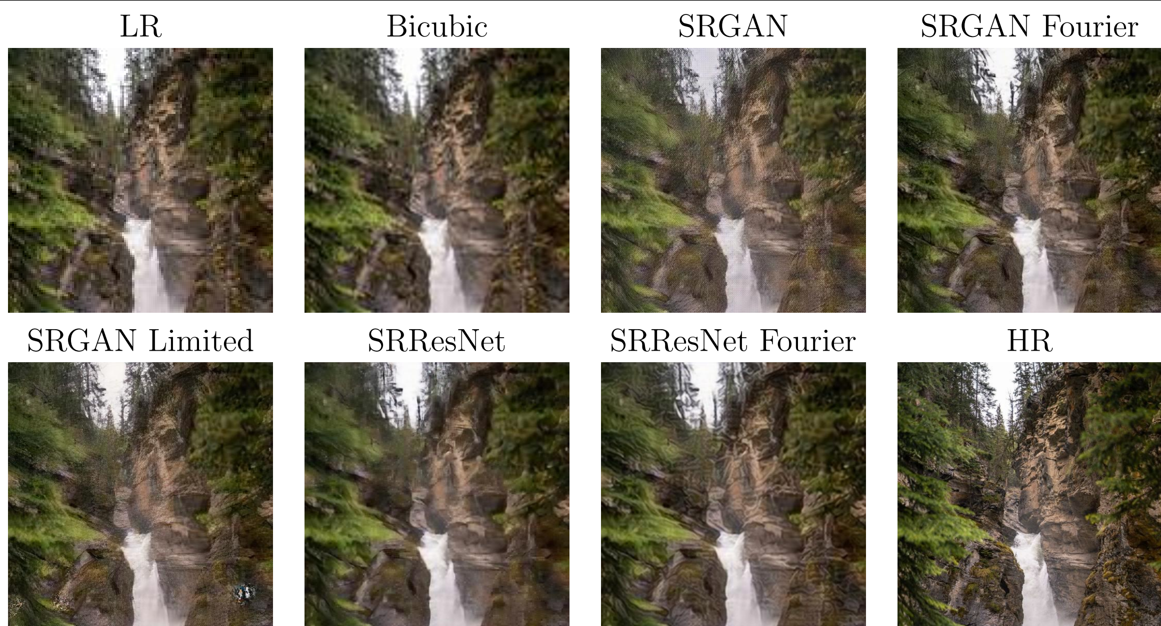
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

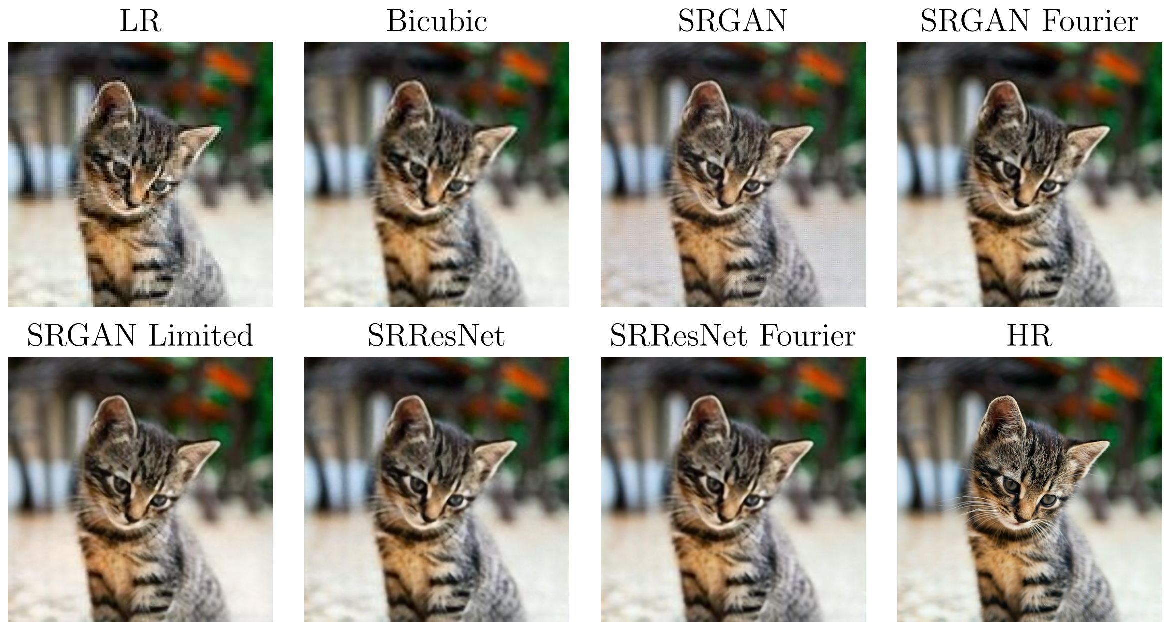

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

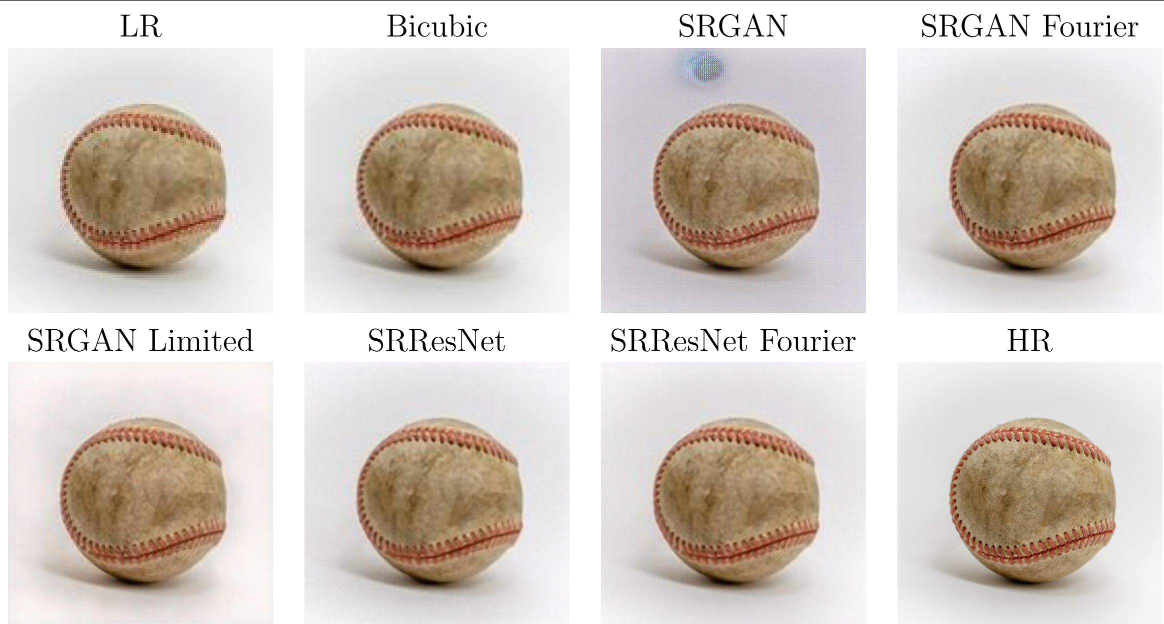

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

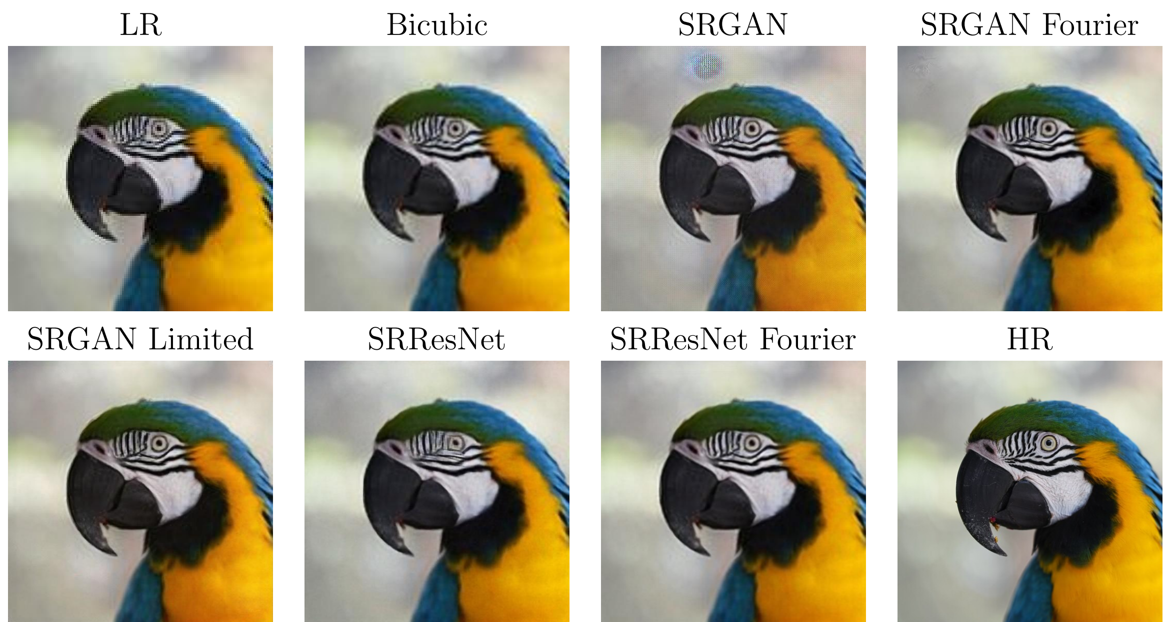
LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

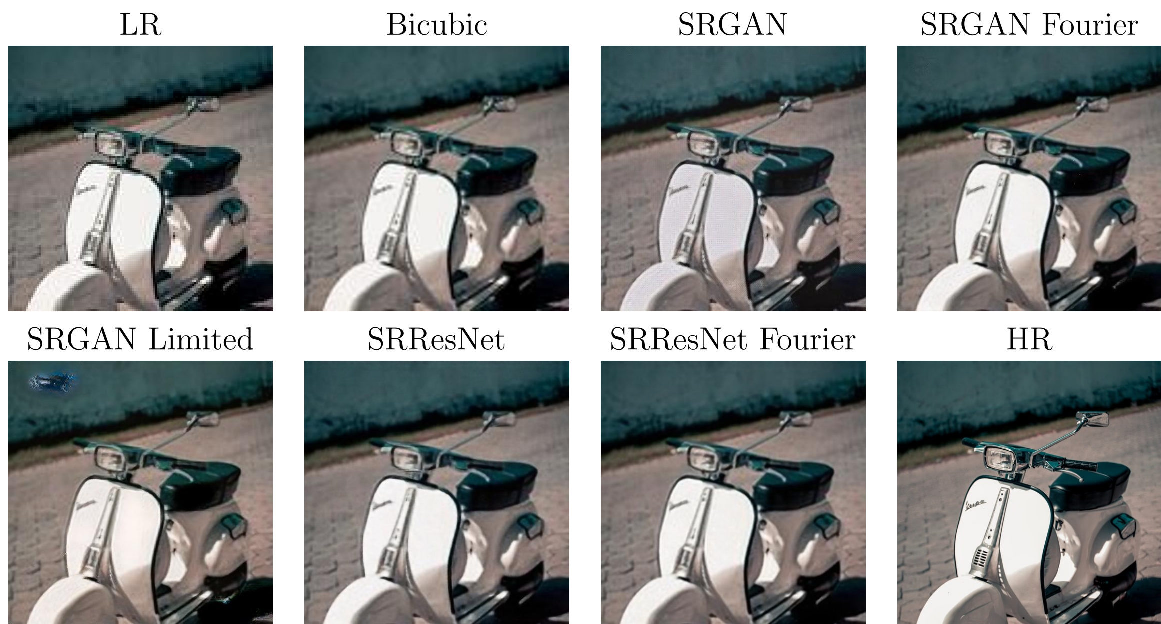

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR

LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR


LR
Bicubic
SRGAN
SRGAN Fourier
SRGAN Limited
SRResNet
SRResNet Fourier
HR