Berliner Hochschule für Technik Berlin
Fachbereich VI – Informatik und Medien

Bachelorarbeit

Erzeugung von Bildern mittels Neuronalen Netzen

Stefan Berger
Medieninformatik
Matrikel-Nr. 854184

Berlin, 6. April 2021

Betreut von: Prof. Dr. F. Gers

Gutachter: Prof. Dr. J. Schimkat

Zusammenfassung

Im Experiment und im Inhalt dieser Bachelorarbeit werden die Fragen
1. ...
2. ...
...
beantwortet.

Inhaltsverzeichnis

1. Einleitung

Obwohl die Idee für eine Maschine, die anhand eingegebener Daten selbständig Entscheidungen treffen kann und die ersten praktischen Ansätze für künstliche neuronale Netze schon einige Jahrzehnte alt sind, findet der Einsatz derartiger Algorithmen erst seit einigen Jahren statt. Viele Erfindungen, die vor 30 bis 50 Jahren in Filmen und Serien Science Fiction darstellten, sind inzwischen nicht nur Realität, sondern auch alltagstauglich. Zu den wichtigsten Beispielen zählen verbale Schnittstellen an Computersystemen und auch Armbanduhren, autonome Fahrzeuge etwa in Gestalt von Parkassistenten und verschiedene Verfahren zur biometrischen Identitätsprüfung.

Andere rasche technologische Fortschritte aus der jüngeren Vergangenheit haben nicht immer nur die Lebensqualität der beteiligten Personen erhöht, sondern stellten durch Missbrauch gelegentlich sogar Gefahren dar. So wie das Internet auch zur Verbreitung von Falschinformationen und die sichere Verschlüsselung von gespeicherten Daten auch für Erpressungen genutzt werden kann, ist die Generierung von täuschend echten Bildern unter Umständen geeignet, persönlichen, finanziellen oder anders gearteten Schaden zu verursachen.

Weiterhin existieren bei der Auswahl der Trainingsdaten für künstliche neuronale Netze rechtliche Grenzen. Bilddaten sind in mehr als ausreichenden Mengen vorhanden, berücksichtigen aber zum Beispiel nicht immer das Recht am eigenen Bild. Für diese Bachelorarbeit sind die Anforderungen an die Bildqualität außerdem sehr hoch, da möglichst auch Texturen und Lichtreflexionen erlernt werden sollen. Modellgrafiken mit geeigneten Material- und Beleuchtungseigenschaften gibt es zwar auch, aber nur in weitaus geringeren Mengen. Für solche Fremdarbeiten, die sehr arbeitsaufwendig sind, wäre auch die Klärung der Nutzungsrechte erforderlich geworden. Für Trainingsergebnisse, die das mentale Modell einer breiten Nutzergemeinschaft reflektieren, sind grundsätzlich auch Daten aus möglichst vielen verschiedenen Quellen erforderlich.

Es ist deshalb eine Brücke geschlagen worden zwischen Trainingsdaten, die zum einen aus zufällig ausgewählte Benutzereingaben bestehen, und solchen, die bestimmte Qualitätseigenschaften erfüllen und in beliebiger Menge erstellt werden können. Ein künstliches neuronales Netz soll aus Skizzen des “Quick, Draw!”-Datasets von Google hochwertig gestaltete Figuren generieren. TODO: Hierfür kommen als Ein- und Ausgabedaten sowohl Bilddateien als auch die jeweils zugrundeliegenden Dateiformate NDJSON und Wavefront OBJ infrage.

1). Ziel der Arbeit

Für diese Arbeit habe ich mir zum Ziel gesetzt, anhand wissenschaftlicher Literatur die Grundlagen und Methoden des maschinellen Lernens zu erarbeiten. Im Rahmen des Themas dieser Abschlussarbeit habe ich mit verschiedenen künstlichen neuronalen Netzen experimentiert, um Bilder zu generieren.

Im Zuge der Ausarbeitung des Konzeptes waren zunächst verschiedene Rahmenbedingungen zu bewerten. Trainingsdaten müssen in großen Mengen verfügbar sein. Urheberrechtliche oder gar Lizenzfragen sollten möglichst nicht auftreten. Weiterhin sollten die Eingabebilder eine gewisse Homogenität aufweisen, um ein Modell auf die Aspekte Textur und Shading trainieren zu können. Meine Recherche verschiedener Methoden zur Generierung von Bildern mittels künstlichen neuronalen Netzen ist im nächsten Abschnitt zu den bisherigen Arbeiten näher beschrieben.

Die Entscheidung fiel schließlich auf ein Machine-Learning-Modell, das unter anderem die Ergebnisse der Canny-Edge-Detection [1] eines Bildes, also nur die Umrisse des abgebildeten Objekts, in das Originalbild zurückübersetzen kann. Das Modell heißt Image-To-Image-Translation [2] oder kurz Pix2Pix. In meinen Experimenten habe ich das Modell erfolgreich darauf trainiert, handgezeichnete Skizzen in fotorealistische Abbildungen von Alltagsgegenständen zu übersetzen. Der Algorithmus ist in dem Abschnitt zu Image-To-Image-Translation des Kapitels 2 beschrieben.

Es gibt verschiedene denkbare praktische Anwendungsfälle für diese automatische Übersetzung. Vergleichbar mit Handschrifterkennung oder der Erkennung primitiver geometrischer Formen durch Geräte mit Stift- oder Fingereingabe kann die Erkennung von handgezeichneten Alltagsgegenständen etwa durch Smartboards die Illustration von Alltagssituationen erleichtern. In meinen Experimenten habe ich Skizzen von Autos in fotorealistische Abbildungen übersetzt. Damit ist die Darstellung einer Situation im Straßenverkehr vorstellbar, wie sie in einer Fahrschule vorteilhaft sein könnte. Ein weiterer konkreter Anwendungsfall ist die Inneneinrichtung einer Wohnung mit verschiedenen Möbelstücken.

Eine umfangreiche Sammlung an handgezeichneten Eingabebildern ist im Dataset des “Quick, Draw!”-Minigames von Google zu finden. Ziel des kostenlosen Browser-Spiels ist es, ein vorgebenes Motiv innerhalb 20 Sekunden mittels Maus, Touchpad oder einem vergleichbaren Eingabegerät so zu zeichnen, dass ein zeitgleich aktiver Algorithmus die Zeichnung klassifizieren kann. Das Dataset besteht aus den erstellten Zeichnungen, ist nach Motiven sortiert und ebenfalls kostenlos verfügbar. Somit ist es für meine Experimente bestmöglich geeignet. Die Verwendung des “Quick, Draw!”-Datasets ist in Kapitel 4 beschrieben.

Es ist in begrenztem Maße möglich, die Zahl der vorhandenen Eingabebilder durch Data Augmentation zu erhöhen [3]. Dabei werden die Bilder gespiegelt, gedreht, skaliert oder auf andere Weise verändert oder verfälscht. Es ist außerdem denkbar, die Eingabedaten selbst zu erstellen. Weil der Zusammenhang mit prozeduralen Shadern durch die Aufgabenstellung gegeben ist, habe ich mich dazu entschieden einen wesentlichen Teil der Eingabebilder zu entwerfen und automatisiert zu erstellen. Es sind Bilder verschiedener Alltagsgegegenstände entstanden, die sich in der Form unterscheiden, deren Texturen und Lichtverhältnisse sich aber gleichen. Dadurch war es mir möglich das Modell auf die Übersetzung der handgezeichneten Skizzen in passende, fotorealistische Bilder zu trainieren. Das Erstellen der 3D-Modelle sowie die Automatisierung ist ebenfalls in Kapitel 4 beschrieben.

Eingabedaten für das Training der Image-To-Image-Translation sind also zum einen handgezeichnete Skizzen und zum anderen fotorealistische Abbildungen alltäglicher Gegenstände. In der Literatur wird häufig das Bild, das übersetzt werden soll, als Input bezeichnet. Im Gegensatz dazu wird das Bild, in das übersetzt werden soll, Ground Truth oder Target genannt [3]. Ich werde in diesem Dokument dieselben Bezeichnungen verwenden.

Im Verlauf der Experimente habe ich den Trainingsfortschritt und -erfolg verschiedener Eingabebilder und Algorithmen betrachtet. Die interessantesten Größen beim maschinellen Lernen sind die Menge der Eingabedaten, die Anzahl der Trainingsdurchläufe und im Zusammenhang damit die Dauer des Trainings. Das tatsächliche Trainingsergebnis habe ich in Augenschein genommen und subjektiv für zufriedenstellend oder nicht zufriedenstellend befunden.

Abb. 1.1.: Das Modell generiert das Ergebnis schrittweise aus zufälligem Rauschen. Nach 15000 Trainingsschritten ist das Ergebnis zufriedenstellend.

2). Bisherige Arbeiten

Künstliche neuronale Netze finden erst seit wenigen Jahren breite Aufmerksamkeit, seit auch Heimcomputer in der Lage sind die hohe Anzahl der erforderlichen Rechenoperationen in annehmbarer Zeit auszuführen. Seitdem sind wenige, englischsprachige Einführungen in die Thematik entstanden. Ein häufig genanntes Buch ist “Deep Learning”, das online kostenfrei zugänglich ist [4]. Ebenfalls online kostenfrei ist das Buch “Dive into Deep Learning” [5]. Ein weiteres, praxisorientierteres Buch ist “Deep Learning with Python” [**chollet2017**], dessen zweite Auflage bald herausgegeben wird.

Als Architektur für ein künstliches neuronales Netz kommt für diese Arbeit zunächst derselbe Aufbau wie in “Image-To-Image-Translation” [2] zum Einsatz. Dieser setzt seinerseits auf Generative Adversarial Networks [6] und Conditional Generative Adversarial Networks [7] auf.

In “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” [8] werden ebenfalls realitätsnahe Ergebnisse erzielt. Anders als bei den bisher erwähnten Arbeiten wird Unsupervised Learning verwendet, um das künstliche neuronale Netz zu trainieren.

Blender ist ein 3D-Computergrafikprogramm mit Werkzeugen für die Modellierung und Animation von Objekten und Charakteren und zur Erstellung von Hintergrundszenen. Szenen können als Standbilder hergestellt werden. Animierte Sequenzen können für Videoproduktionen genutzt werden. Modelle und Szenen werden durch Farben und Texturen noch aufgewertet, wodurch brillante realistische Effekte produziert werden. Die Standbilder und Videos können als Kunstwerke oder als architektonisch oder wissenschaftliche Präsentationen Anwendung finden. [9]

2. Modell

1). Tensoren

Mehrdimensionale Arrays, in NumPy ndarray (N-Dimensional Array) genannt, werden auch Tensoren genannt. Gennerell verwenden alle derzeitigen Systeme für Maschinelles Lernen Tensoren als zugrundeliegende Datenstruktur. Tensoren sind elementar für dieses Fachgebiet. [3]

In seinem Kern ist ein Tensor ein Container für numerische Daten. Tensoren sind die Verallgemeinerung der Matrizen für eine beliebige Anzahl Dimensionen. Im Zusammenhang mit Tensoren werden die Dimensionen häufig Achsen gennannt. Die Anzahl der Achsen wird auch als Stufe bezeichnet. Ein Skalar ist ein Tensor nullter Stufe. Ein Vektor ist ein Tensor erster Stufe und eine Matrix ein Tensor zweiter Stufe. [**chollet201chollet2017dlpython7**]

Der Begriff der Dimension, der für Vektoren und Matrizen eindeutig ist, wird für Tensoren in der Literatur unterschiedlich verwendet. [3] In dieser Arbeit soll die Dimension einer Achse eines Tensors die Anzahl der numerischen Werte entlang dieser Achse bezeichnen.

Bilder erscheinen als Tensoren dritter Stufe, deren Achsen der Höhe, Breite und den Farbkanälen (Rot, Grün und Blau) entsprechen. [5]

Ein Tensor ist durch drei Schlüsselattribute definiert:

- Anzahl der Achsen (Stufe): Zum Beispiel hat ein Tensor dritter Stufe drei Achsen, und eine Matrix hat zwei Achsen. In Python-Bibliotheken wie NumPy oder Tensorflow wird sie auch als $ndim$ des Tensors bezeichnet.

- Form: Dies ist ein Tupel aus Ganzzahlen, das die Dimension jeder Achse beschreibt. Beispiele sind $(3, 5)$ für die Form einer Matrix und $(3, 3, 5)$ für einen Tensor dritter Stufe. Ein Vektor hat eine Form mit einem einzelnen Element, etwa so: $(5,)$, während ein Skalar eine leere Form hat: $()$.

- Datentyp (in Python-Bibliotheken gewöhnlich $dtype$ genannt): Dies ist der Typ der im Tensor enthaltenen Daten. Zum Beispiel könnte der Datentyp eines Tensors $float16$, $float32$, $float64$, $uint8$ und so weiter sein.

[3]

Viele in dieser Arbeit verwendeten Tensoren beschreiben RBG-Bilddaten. Sie sind deshalb dritter Stufe, und die enthaltenen Daten sind Ganzzahlen zwischen 0 und 255. Die Form der durch das künstliche neuronale Netz generierten Bilder ist $(256, 256, 3)$. Die Dimensionen beschreiben hier also die Breite, Höhe und Farbtiefe der Bilddateien.

2). Logistische Regression

Ein vergleichsweise einfacher Algorithmus des Maschinellen Lernens ist Logistic Regression oder Softmax Regression. Es lassen sich damit Klassifizierungen durchführen. Bei binären Klassifizierungen werden die Eingaben in zwei Kategorien unterteilt. Häufige Beispiele sind fehlerfreie oder fehlerhafte Produktionsergebnisse, ärztliche Befunde einer bestimmten Krankheit oder Gesundheit und ob in einem Bild ein bestimmtes Objekt vorhanden ist oder nicht.

Bei der Multiclass Logistic Regression werden die Eingaben in mehr als zwei Klassen unterteilt. Es lassen sich damit beispielsweise in einem Bild verschiedene Arten von Fahrzeugen, Gegenständen oder Tieren unterteilen.

Diagramm: Ein-Schicht-NN (Die Ausgabeschicht wird nicht mitgezählt) Logistic Regeression kann mit einer einzelnen oder mehreren neuronalen Schichten imlementiert sein. Das künstliche neuronale Netz in Abb. x besitzt eine einzelne aus drei Einheiten bestehende Schicht. Eine Einheit besteht zunächst aus einer differenzierbaren Funktion. Oft ist das eine Multiplikation der Eingaben X mit einer Anzahl lernbaren Parametern W und eine anschließende Addition mit Bias-Werten b. Die Ausgabe einer Schicht wird oft mit $\hat{y}$ bezeichnet:

$$\hat{y} = WX + b \tag{2.1}$$

Dieser folgt eine Aktivierungsfunktion, die dem errechneten Wert einen anderen Wert zuordnet, der entweder nahe 0 oder nahe 1 liegt (gelegentlich nahe −1 oder nahe 1). Beispiele für Aktivierungsfunktionen sind Sigmoid:

$$sig(t) = \frac{1}{1 + e^{-t}} \tag{2.2}$$

und Rectified Linear Unit, kurz ReLU:

$$f(x) = max(0, x) \tag{2.3}$$

oder auch Leaky ReLU:

$$f(x) = max(0.01x, x) \tag{2.4}$$

In einigen künstlichen neuronalen Netzen werden zusätzlich die Eingaben für jede Schicht normalisiert, wodurch die Minimierung der Verlustfunktion (Gradient Descent) optimiert werden kann.

3). Deep Neural Networks

4). Convolutional Neural Networks

Für Bilddaten wendet ein CNN diese Operation typischerweise auf zwei dreidimensionale Matrizen an, nämlich einerseits auf die Eingabedaten und andererseits auf einen sogenannten Filter oder auch Kernel. Die Eingabedaten sind in der ersten Schicht des Netzes die RGB-Pixelinformationen und in allen weiteren konvolutionalen Schichten die Ausgabe der

vorherigen Schicht. Ein Filter ist eine Anzahl von trainierbaren Parametern, in diesem Fall auch eine . Beide Matrizen haben also die Form ($H\ x\ W\ x\ C$), wobei H die Höhe, W die Breite und C die RGB-Farbwerte repräsentiert.

Jede der drei Matrixdimensionen variiert üblicherweise zwischen den verschiedenen Schichten des Netzes. In fast allen CNNs ([4], [10], [11], [2]) nimmt die Kardinalität zunächst ab. Die Reduktion kann durch die Konvolution selbst entstehen oder durch Pooling-Schichten. Beim Pooling werden aus benachbarten Matrixkoeffizienten meist das Maximum, seltener der Durchschnitt oder andere Aggregierungen gebildet. Auf diese Weise wird das neuronale Netz darauf trainiert die relevanten Informationen zu extrahieren. [4]

Den konvolutionalen Schichten folgt in einigen Anwendungsfällen eine voll vernetzte Schicht (engl. Fully Connected Layer, FC), in der für jedes Neuron mit jedem Neuron der vorherigen Schicht eine Verbindung besteht. Besonders für die Bilderkennung ist diese Architektur gut geeignet. Die Ausgabe des neuronalen Netzes ist dann ein Vektor, beispielsweise von Wahrscheinlichkeitswerten für das Vorhandensein bestimmter Objekte und gegebenenfalls Bildkoordinaten der erkannten Objekte. Im Fall der Bildgenerierung ist die Ausgabe aber wieder eine Matrix von RGB-Pixelinformationen in der Form ($H\ x\ W\ x\ 3$).

5). U-Net-Architektur

Die bis hierhin beschriebenen neuronalen Netze besitzen eine gradlinige Struktur, in der die Ausgabe einer Schicht nur an die nächste Schicht übergeben wird. Bei zunehmender Anzahl der Schichten verbessert sich die Performance neuronaler Netze mit diesem Aufbau zunächst, aber verschlechtert sich bei zu vielen Schichten wieder. In einem Residual Neural Network (ResNet) verhindern zusätzliche Verbindungen zwischen nicht direkt aufeinanderfolgenden Schichten diesen Performanceverlust.

Ein U-Net ist eine spezielle Form eines ResNets. Es hat eine annähernd symmetrische Struktur, in der sich die Kardinalitäten der Matrizen zuerst verringern und anschließend wieder erhöhen. U-Nets erzielen selbst mit wenigen Trainingsdaten gute Ergebnisse und benötigen dafür vergleichsweise wenig Rechenleistung. [12]

6). Generative Adversarial Networks

Ein Generative Adversarial Network (GAN)besteht zunächst aus einem Generator und einem Discriminator [6]. Der Generator lernt während des Trainings täuschend echt aussehende Bilddaten zu generieren. Der Discriminator wird dagegen darauf trainiert, echte von generierte Bildern zu unterscheiden. Anschließend können beide Modelle "gegeneinander antreten". Deswegen wird es Generative *Adversarial* Network genannt.

GANs unterscheiden sich von anderen Modellen durch ihren Aufbau und in Bezug auf das Trainingsziel. Künstliche neuronale Netze ermitteln häufig einen skalaren Wert wie beispielsweise einen Wahrscheinlichkeitswert und minimieren zu diesem Zweck eine Verlustfunktion. In einem GAN sind zwei CNNs im Einsatz. Das erste, der Generator, erstellt Tensoren n-ter Klasse. In diesem Beispiel sind das RGB-Bildinformationen. Das zweite CNN wird Discriminator genannt und bekommt als Eingabe die Ausgabe des ersten CNNs. Der Discriminator

wird darauf trainiert, generierte Bilder von Bildern aus dem Trainingsset zu unterscheiden. Er minimiert also eine Verlustfunktion. Der Generator wird darauf trainiert, diese Verlustfunktion zu maximieren. Dieser Vorgang stammt aus der Spieltheorie und heißt Minimax. [6]

Eine Erweiterung des GANs ist das Conditional Generative Adversarial Net (cGAN). In cGANs erhält der Generator zusätzliche Eingabedaten (y, "Ground Truth"), die Hinweise für die Generierung enthalten. Der Discriminator erhält diese zusätzlichen Daten ebenfalls, um die Erkennung während des Trainings zu optimieren. [7]

7). Image-To-Image-Translation

Der Image-To-Image-Translation-Algorithmus oder kurz Pix2Pix-Algorithmus verwendet ein GAN, um Bilder in Bilder zu übersetzen. Dafür kommen zwei CNNs zum Einsatz, nämlich eins für den Generator und eins für den Discriminator. [2]

Die Datei main.py befindet sich im Anhang und ist die Implementierung, die für diese Arbeit hauptsächlich verwendet wurde.

Die erste Funktion namens `load` öffnet eine Datei und liest diese als JPEG-Bilddatei. Sie wird verwendet, um die Trainingsdaten zu laden. Weil die Trainingsdaten aus zwei Bildern pro Datei bestehen, nämlich jeweils einem Eingabebild und dem erwarteten Ergebnis (Ground Truth), extrahiert die Funktion zusätzlich die beiden Bilder und gibt jede in einer eigenen Variablen zurück.

Die Funktion `resize` ist ebenfalls für die Verarbeitung zweier Bilder vorgesehen und skaliert beide auf die übergebene Breite und Höhe.

Die Funktion `random_crop` stellt einen zufälligen Ausschnitt der zwei Eingabebilder frei. Sie wird aufgerufen, nachdem die Eingabebilder hochskaliert wurden. In der Referenzimplementierung wie auch in dieser Arbeit beträgt die Bildgröße vorher $286x286$ Pixel und $256x256$ Pixel nach dem Freistellen. Die zurückgegebenen Bilder haben dann wieder die gleichen Abmessungen wie die Eingabedaten.

In der Funktion `normalize` werden die RGB-Werte der beiden Eingabebilder normalisiert. Dadurch sollen zu große Schritte auf dem Gradienten verhindert werden, die das Konvergieren verhindern könnten. [3] Die RGB-Werte sind zunächst als Ganzzahlen im Bereich von 0 bis 255 gegeben und werden in Fließkommazahlen im Bereich -1 bis 1 umgerechnet.

In `random_jitter` wird zunächst `resize` aufgerufen, um die Eingabebilder auf 286x286 Pixel zu skalieren. Anschließend wird `random_crop` auf die skalierten Bilder angewendet. Schließlich werden aufgrund einer Zufallszahl beide Eingabebilder mit einer Wahrscheinlichkeit von 50 Prozent gespiegelt.

Die beiden Funktionen `load_image_train` und `load_image_test` laden die Eingabe- und Referenzbilder (Ground Truth), skalieren diese auf 256x256 Pixel und normalisieren die Eingabedaten durch Aufrufe der Funktionen `load`, `resize` und `normalize`. Nur die Funktion zum Laden der Trainingsbilder ruft für die geladenen Bilder `random_jitter` auf.

Die Funktion `downsample` erzeugt eine Schicht eines CNNs mit optionaler Batch Normalization und der Leaky-ReLU-Aktivierungsfunktion. Diese Schichten werden im Encoder des Generators sowie im Discriminator verwendet. Batch Normalization kommt nur in der jeweils ersten Schicht des Generators und des Diskriminators nicht zum Einsatz.

In der Funktion `upsample` entstehen die übrigen CNN-Schichten des Generators. Der Entfaltung, bei der die Faltung der Funktion `downsample` umgekehrt wird (engl. Transposed Convolution [13]), folgt hier immer die Batch Normalization sowie optional Dropout mit einem Wahrscheinlichkeitswert von 0.5. Das ist in den ersten drei Schichten des Decoders im Generator der Fall und bedeutet, dass statistisch die Hälfte der Aktivierungen „fallengelassen" werden, also nicht in das Trainingsergebnis eingehen. Als Aktivierungsfunktion kommt ReLU zum Einsatz.

In `build_generator` werden die Schichten des Generators zusammengefügt. Im Encoder wird acht Mal `downsample` aufgerufen, bis die Dimensionen des Eingabetensors, die am Anfang der Bildgröße entsprechen (Breite und Höhe, also $256x256$ Pixel), durch die Konvolution auf 1 reduziert sind. Die Dimension, welche die Anzahl Farbkanäle der Eingabebilder repräsentiert, erhöht sich im Encoder auf den Wert 512. Die Werte dieser Achse werden als Features [5] [3] [13] bezeichnet.

Der Decoder ruft seinerseits sieben Mal `upsample` auf und führt eine weiter Entfaltung durch, wodurch die Dimensionen des bearbeiteten Tensors wieder jeweils dieselbe Form wie in den Eingabebildern annehmen. Schließlich werden die Skip-Connections zwischen den Schichten 1 bis 8 des Encoders und des Decoders hergestellt.

Im Ausführungsstrang wird zunächst eine Option des GPU-Speichermanagements gewählt, um die Größe des reservierten Speichers dynamisch anwachsen zu lassen. [13] Zu Beginn der Experimente hat sich diese Einstellung als vorteilhaft herausgestellt.

3. Implementierung

1). Entwicklungsumgebung

1.1). Ubuntu Linux

1.2). Python

1.3). Tensorflow

1.4). CUDA

Cuda ist eine NVIDIA-proprietäre Hardware- und Software-Architektur.

Es ist das Schema, nach dem NVIDIA-Grafikkarten gebaut wurden, die sowohl traditionelle Grafik-Rendering-Aufgaben als auch allgemeine Aufgaben durchführen können. Zum Programmieren der CUDA GPUs wird die Sprache CUDA C verwendet. CUDA C ist im Wesentlichen die Programmiersprache C mit einer Handvoll Erweiterungen, welche die Programmierung hoch parallelisierter Maschinen wie NVIDIA GPUs ermöglichen. [14]

Anders als frühere GPU-Generationen, die Rechenressourcen in Vertex- und Pixelshader aufteilten, enthält die CUDA-Architektur eine einheitliche Shader-Pipeline, welche die Zuordnung allgemeiner Berechnungen zu jeder arithmetisch-logischen Einheit (ALU) auf dem Chip durch ein Programm erlaubt. Diese ALUs wurden mit einem Befehlssatz entworfen, der für allgemeine Berechnungen statt für spezielle Grafikberechnungen zugeschnitten ist. Weiterhin wurde den Execution Units auf der GPU freier Lese- und Schreibzugriff auf den Speicher sowie Zugriff auf einen softwaregesteuerten Cache, genannt Shared Memory, gegeben. [14]

Zusätzlich zu der Sprache für das Schreiben von Code für die GPU stellt NVIDIA einen spezialisierten Hardwaretreiber zur Verfügung, der die hohe Rechenleistung der CUDA-Architektur ausschöpft. Kenntnis der OpenGL- oder DirectX-Programmierschnittstellen ist nicht länger erforderlich. [14]

1.5). Image-To-Image-Translation in Python

Es besteht eine Auswahl an Beispielimplementierungen für Image-To-Image-Translation. Das Original-Paper verweist auf ein GitHub-Repository, das eine Lua-Implementierung zur Verfügung stellt. Wegen der besseren Eignung für Experimente auf einem lokalen Rechner durch CUDA-Unterstützung mit Tensorflow wird in dieser Arbeit Python verwendet.

Die verwendete Beispielimplementierung ist zum Zeitpunkt der Erstellung dieses Dokumentes unter https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/pix2pix.ipynb zu finden. Eine an die aktuelle Version von Tensorflow und die Anforderungen dieser Arbeit angepasste Version befindet sich im Anhang.

Die wichtigste Änderung ist das selbst erstellte Dataset, das statt der im Beispiel verwendeten Gebäudefassaden geladen wird. Es wurden Ausgaben entfernt, wegen derer die Ausführung des Skriptes unterbrochen wurde. Dabei wurden Bildinformationen und Beispielbilder nach dem Laden des Datasets, dem Aufteilen in Eingabebild und Ground Truth, der Erzeugung von Bildrauschen, Down- und Upsampling sowie den ersten Trainingsdurchläufen ausgegeben, die während des Trainings der Skizzen- und Renderbilder nicht betrachtet werden mussten. Außerdem wurden bei jedem Training Diagramme des Generators und des Discriminators erzeugt.

4. Experimente und Resultate

1). Vorbereitung der Eingabedaten

Die Trainingsdaten liegen für das "Quick, Draw!"-Dataset im NDJSON-Format und als Blender-Dateien beziehungsweise Wavefront OBJ-Dateien vor. Die effizienteste Möglichkeit, die Skizzen und 3D-Modelle für die Verarbeitung in einem CNN vorzubereiten, ist das Rendern und Speichern als Bilddateien. Als Dateiformat kommen JPEG oder PNG infrage. Beide Formate können leicht als Trainingsset mit Tensorflow geladen werden.

NDJSON steht für Newline Delimited JavaScript Object Notation. In einer solchen Datei sind also zeilenweise JSON-Objekte gespeichert. Für jede Zeichnung sind Angaben zu Motiv, Ort und Zeit enthalten. Außerdem ist angegeben, ob die künstliche Intelligenz des Minispiels das Motiv in der Zeichnung korrekt klassifiziert, also erkannt hat. Jede Zeichnung hat weiterhin eine eindeutige ID.

Der relevanteste Teil ist die "Drawings"-Eigenschaft der JSON-Objekte, ein mehrdimensionales Array mit Bildkoordinaten. Es enthält mindestens X-Koordinaten und Y-Koordinaten in jeweils einem Array im "Drawings"-Array. Indem Linien zwischen den Bildkoordinaten in der Reihenfolge der Arrayelemente in ein Bild gezeichnet und in einer Bilddatei gespeichert werden, können die Zeichnungen in beliebigen Bilddateiformaten gespeichert werden. Für diese Arbeit wurde die Konvertierung ebenfalls in Python realisiert.

Bei der Verarbeitung in einem Convolutional Neural Network spielt die Bildgröße in Bezug auf die Verarbeitungszeit eine wichtige Rolle. Bildformate der Größe 256x256 oder kleiner sind üblich und gut geeignet. Für ein GAN ist es zwar nicht erforderlich, aber sinnvoll, für Ein- und Ausgabedaten dieselben Dimensionen festzulegen. Bei der Bildgenerierung aus Skizzen unterscheiden sich Ein- und Ausgabdedaten natürlich bei die Farbtiefe. Während die Skizzen Graustufenbilder sind, besitzen die generierten Bilder drei Farbkanäle (RGB).

Sowohl während der Entwicklung als auch zur Laufzeit kann es vorteilhaft sein, Farbwerte statt auf der oft verwendeten Skala von 0 bis 255 als Fließkommazahlen im Bereich $0,0$ bis $1,0$ darzustellen. Auch für die Ausgaben der einzelnen Schichten eines künstlichen neuronalen Netzes kann diese Normalisierung durchgeführt werden (TODO: BatchNorm).

Ein ebenso wichtiger wie aufwendiger Vorgang ist die Klassifizierung der Trainingsdaten, also die Zuweisung von Eingaben zu den erlernbaren Ergebnissen. Aufgrund der selbst erstellten Ausgabebilder existieren für diesen Zweck keine vorgefertigten Datasets. Die Sortierung und Zuweisung erfolgt deshalb manuell.

2). Anwendung herkömmlicher Shader

Shader definieren die Interaktion des Lichts mit der Oberfläche des Objekts. Dabei können Shader aus einem oder mehreren BSDFs (Bidirectional Scattering Distribution Function)

bestehen, die wiederum von Mix-und Add-Shadern in der Zusammensetzung gemischt werden. [15]

In der Computergrafik wird die Darstellung der Oberfläche eines Objekts durch die drei Faktoren Material, Textur und Ausleuchtung bestimmt. Material ist die Grundfarbe der Oberfläche. Textur sind die physischen Charakteristiken der Oberfläche, und Ausleuchtung ist die Hintergrundbeleuchtung oder Licht, welches von Lichtern (Lampen) emittiert wird. [9]

In der Computergrafik ist ein Material die Farbe eines Objektes. Es legt fest, wie das sichtbare Spektrum des Lichts von der Oberfläche des Objekts reflektiert wird. Ein Material legt außerdem fest, ob die Oberfläche matt oder metallisch erscheint. [9]

Material (Farbe) kann entsprechend der drei Farbschemas RGB, HSV oder Hex dargestellt werden. Wie die Farbe in jedem Schema erscheint ist auch von einem Alpha-Wert, welcher für die Menge der Transparenz steht, abhängig. [9]

Texturen definieren das physische Erscheinungsbild einer Oberfläche, also etwa wie glatt oder uneben diese erscheint, oder ihre Struktur, welche die visuelle Wahrnehumg der physischen Beschaffenheit des Objekts definiert. Diese Definition bestimmt, woraus die Oberfläche besteht, wie Holz, Ziegelsteine, Wasser und so weiter. [9]

Texturen werden durch Algorithmen generiert, wie sie in Blender integriert sind (prozedurale Texturen) oder aus Bilddateien (Bildtexturen). [9]

Die Qualität eines Bildes hängts direkt von der Effektivität des Shading-Algorithmus ab, der wiederum von der Modellierungsmethode des Objektes abhängt. Zwei wesentliche Methoden der Objektbeschreibung werden häufig verwendet, nämlich Oberflächendefinition mittels mathematischer Gleichungen und Oberflächenapproximation durch Mosaike aus polygonalen Flächen. [16]

Polygonobjekte sind anfangs immer ungeglättet, das heißt, dass beim Rendern oder in der schattierten Ansicht zunächst immer die einzelnen Flächen zu sehen sind, aus denen sich das Objekt zusammensetzt. Eine mögliche, wenn auch in Sachen Renderzeit und Speicherverbrauch ungünstige Methode wäre, einfach das Objekt so weit in kleinere Flächen zu unterteilen, dass beim Rendern eine Fläche pro Pixel gerendert wird. In der praktischen Arbeit verwendet man deshalb einen Trick, bei dem die Übergänge zwischen den einzelnen Flächen “glattgerechnet” werden. Übliche Verfahren hier sind das Gouraud oder Phong Shading. [15]

3). Hyperparameter

Die Pix2Pix-Referenzimplementierung ist bereits für die Übersetzung von Skizzen in Fotos eingestellt. Für das Training waren anfangs mehrere Tausend Epochs, also Trainingsdurchläufe, erfoderlich, um zufriedenstellende Ergebnisse zu sehen. TODO: Diese Zahl konnte durch ... verringert werden.

Die Eingabebilder sind 256x256 Pixel groß und besitzen einen Farbkanal für Graustufen. Sie werden am Anfang des Trainingsprozesses durch sogenanntes Jittering augmentiert. Dabei werden die Bilder zuerst auf 286x268 Pixel vergrößert und anschließend auf einen zufälligen 256x256 Pixel großer Ausschnitt wieder verkleinert. Diese Pixelgrößen können im Experiment geändert werden.

Der Adam Optimierer [17] erhält für die Learning-Rate den Wert $0,0002$. Das Momentum ist auf $0,9$ voreingestellt. Diese beiden Werte beinflussen die Lerngeschwindigkeit und sind in begrenztem Maße anpassbar.

4). Performancebeobachtungen

5. Diskussion

Literatur

1. CANNY, J.: A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 1986, Jg. 8, Nr. 6, S. 679–698. ISSN 0162-8828. Abgerufen unter DOI: 10.1109/TPAMI.1986.4767851.
2. ISOLA, P. u. a.: *Image-to-Image Translation with Conditional Adversarial Networks*. 2018. Abgerufen unter arXiv: 1611.07004 [cs.CV].
3. CHOLLET, F.: *Deep Learning with Python*. 1st. USA: Manning Publications Co., 2017. ISBN 1617294438.
4. GOODFELLOW, I. u. a.: *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
5. ZHANG, A. u. a.: *Dive into Deep Learning*. 2020. https://d2l.ai.
6. GOODFELLOW, I. J. u. a.: *Generative Adversarial Networks*. 2014. Abgerufen unter arXiv: 1406.2661 [stat.ML].
7. MIRZA, M.; OSINDERO, S.: *Conditional Generative Adversarial Nets*. 2014. Abgerufen unter arXiv: 1411.1784 [cs.LG].
8. RADFORD, A. u. a.: *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016. Abgerufen unter arXiv: 1511.06434 [cs.LG].
9. BLAIN, J. A K Peters/CRC Press, 2020. Auch verfügbar unter: https://doi.org/10.1201/9781003093183.
10. LECUN, Y. u. a.: Object Recognition with Gradient-Based Learning. In: *Contour and Grouping in Computer Vision*. Springer, 1999.
11. RONNEBERGER, O. u. a.: U-Net: Convolutional Networks for Biomedical Image Segmentation. In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 2015, Bd. 9351, S. 234–241. LNCS. Auch verfügbar unter: http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a. (available on arXiv:1505.04597 [cs.CV]).
12. HE, K. u. a.: *Deep Residual Learning for Image Recognition*. 2015. Abgerufen unter arXiv: 1512.03385 [cs.CV].
13. ZACCONE, G.; KARIM, M. R.: *Deep Learning with TensorFlow - Second Edition: Explore Neural Networks and Build Intelligent Systems with Python*. 2nd. Packt Publishing, 2018. ISBN 1788831101.
14. SANDERS, J.; KANDROT, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN 0131387685.
15. WARTMANN, C.: *Das Blender-Buch - 3D-Grafik und Animation mit Blender*. 5th. dpunkt.verlag, 2014.

16. PHONG, B. T.: Illumination for Computer Generated Pictures. *Commun. ACM*. 1975, Jg. 18, Nr. 6, S. 311–317. ISSN 0001-0782. Abgerufen unter DOI: 10.1145/360825.360839.

17. KINGMA, D. P.; BA, J.: *Adam: A Method for Stochastic Optimization*. 2017. Abgerufen unter arXiv: 1412.6980 [cs.LG].

Bildnachweis

Abkürzungs- und Symbolverzeichnis

Abkürzungen

AC	Air Compressor, Luftverdichter
APH	Air Preheater, Luftvorwärmer
CC	Combustion Chamber, Brennkammer
EXP	Expander
HRSG	Heat Recovery Steam Generator, Abhitzekessel

Lateinische Symbole

c	Spezifische Kosten je Exergieeinheit, €/J_{ex}
$\dot{C}$	Kostenstrom, €/h
CC	Kapitalgebundene Kosten, €
cf	Capacity Factor, Jährliche Auslastung, –
e	Spezifische Exergie, J/kg
$\bar{e}$	Spezifisch molare Exergie, J/mol
$\dot{E}$	Exergiestrom, W
f	Exergoökonomischer Faktor, –
fc	Spezifische Brennstoffkosten, €/J
FC	Brennstoffkosten, €
h	Spezifische Enthalpie, J/kg
$\dot{H}$	Enthalpiestrom, W
HHV	Brennwert, J/kg

Griechische Symbole

Δ	Differenz
ε	Exergetischer Wirkungsgrad, –
η_s	Isentroper Wirkungsgrad, –
κ	Isentropenexponenten, –
λ	Luftzahl, –

Hoch- und tiefgestelle Indizes

0	Referenzpunkt, Thermodynamische Umgebung
a	Avarage, Mittlere
D	Destruction, Vernichtung
F	Fuel, Brennstoff, Aufwand
net	Netto

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

A. Quelltexte

1). Image-To-Image-Translation main.py

```
  import tensorflow as tf

import os
import time
import datetime

from matplotlib import pyplot as plt
from IPython import display

from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession


def load(image_file):
    # Read and decode an image file to a uint8 tensor
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)

    # Split each image tensor into two tensors:
    w = tf.shape(image)[1]
    w = w // 2
    input_image = image[:, :w, :]
    real_image = image[:, w:, :]

    # Convert both images to float32 tensors
    input_image = tf.cast(input_image, tf.float32)
    real_image = tf.cast(real_image, tf.float32)

    return input_image, real_image


def resize(input_image, real_image, height, width):
    input_image = tf.image.resize(input_image, [height, width],
                                  method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    real_image = tf.image.resize(real_image, [height, width],
                                 method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

    return input_image, real_image


def random_crop(input_image, real_image):
    stacked_image = tf.stack([input_image, real_image], axis=0)
    cropped_image = tf.image.random_crop(
        stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])

    return cropped_image[0], cropped_image[1]


# Normalizing the images to [-1, 1]
def normalize(input_image, real_image):
    input_image = (input_image / 127.5) - 1
    real_image = (real_image / 127.5) - 1

    return input_image, real_image
```

```
@tf.function()
def random_jitter(input_image, real_image):
    # Resizing to 286x286
    input_image, real_image = resize(input_image, real_image, 286, 286)

    # Random cropping back to 256x256
    input_image, real_image = random_crop(input_image, real_image)

    if tf.random.uniform(()) > 0.5:
        # Random mirroring
        input_image = tf.image.flip_left_right(input_image)
        real_image = tf.image.flip_left_right(real_image)

    return input_image, real_image


def load_image_train(image_file):
    input_image, real_image = load(image_file)
    input_image, real_image = random_jitter(input_image, real_image)
    input_image, real_image = normalize(input_image, real_image)

    return input_image, real_image


def load_image_test(image_file):
    input_image, real_image = load(image_file)
    input_image, real_image = resize(input_image, real_image,
                                     IMG_HEIGHT, IMG_WIDTH)
    input_image, real_image = normalize(input_image, real_image)

    return input_image, real_image


def downsample(filters, size, apply_batchnorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)

    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2D(filters, size, strides=2, padding='same',
                               kernel_initializer=initializer, use_bias=False))

    if apply_batchnorm:
        result.add(tf.keras.layers.BatchNormalization())

    result.add(tf.keras.layers.LeakyReLU())

    return result


def upsample(filters, size, apply_dropout=False):
    initializer = tf.random_normal_initializer(0., 0.02)

    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
                                        padding='same',
                                        kernel_initializer=initializer,
                                        use_bias=False))

    result.add(tf.keras.layers.BatchNormalization())

    if apply_dropout:
        result.add(tf.keras.layers.Dropout(0.5))

    result.add(tf.keras.layers.ReLU())

    return result
```

```
def build_generator():
    inputs = tf.keras.layers.Input(shape=[256, 256, 3])

    down_stack = [
        downsample(64, 4, apply_batchnorm=False),  # (batch_size, 128, 128, 64)
        downsample(128, 4),  # (batch_size, 64, 64, 128)
        downsample(256, 4),  # (batch_size, 32, 32, 256)
        downsample(512, 4),  # (batch_size, 16, 16, 512)
        downsample(512, 4),  # (batch_size, 8, 8, 512)
        downsample(512, 4),  # (batch_size, 4, 4, 512)
        downsample(512, 4),  # (batch_size, 2, 2, 512)
        downsample(512, 4),  # (batch_size, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True),  # (batch_size, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True),  # (batch_size, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True),  # (batch_size, 8, 8, 1024)
        upsample(512, 4),  # (batch_size, 16, 16, 1024)
        upsample(256, 4),  # (batch_size, 32, 32, 512)
        upsample(128, 4),  # (batch_size, 64, 64, 256)
        upsample(64, 4),  # (batch_size, 128, 128, 128)
    ]

    initializer = tf.random_normal_initializer(0., 0.02)
    last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                           strides=2,
                                           padding='same',
                                           kernel_initializer=initializer,
                                           activation='tanh')  # (batch_size, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = tf.keras.layers.Concatenate()([x, skip])

    x = last(x)

    return tf.keras.Model(inputs=inputs, outputs=x)


def generator_loss(disc_generated_output, gen_output, target):
    gan_loss = loss_object(tf.ones_like(disc_generated_output), disc_generated_output)

    # Mean absolute error
    l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

    total_gen_loss = gan_loss + (LAMBDA * l1_loss)

    return total_gen_loss, gan_loss, l1_loss


def build_discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
```

```
    tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

    x = tf.keras.layers.concatenate([inp, tar])  # (batch_size, 256, 256, channels*2)

    down1 = downsample(64, 4, False)(x)  # (batch_size, 128, 128, 64)
    down2 = downsample(128, 4)(down1)  # (batch_size, 64, 64, 128)
    down3 = downsample(256, 4)(down2)  # (batch_size, 32, 32, 256)

    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3)  # (batch_size, 34, 34, 256)
    conv = tf.keras.layers.Conv2D(512, 4, strides=1,
                                  kernel_initializer=initializer,
                                  use_bias=False)(zero_pad1)  # (batch_size, 31, 31, 512)

    batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

    leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

    zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu)  # (batch_size, 33, 33, 512)

    last = tf.keras.layers.Conv2D(1, 4, strides=1,
                                  kernel_initializer=initializer)(zero_pad2)  # (batch_size, 30, 30, 1)

    return tf.keras.Model(inputs=[inp, tar], outputs=last)


def discriminator_loss(disc_real_output, disc_generated_output):
    real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

    generated_loss = loss_object(tf.zeros_like(disc_generated_output), disc_generated_output)

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss


def generate_images(model, test_input, tar, image_index):
    prediction = model(test_input, training=True)
    plt.figure(figsize=(15, 15))

    display_list = [test_input[0], tar[0], prediction[0]]
    title = ['Input Image', 'Ground Truth', 'Predicted Image']

    for i in range(3):
        plt.subplot(1, 3, i + 1)
        plt.title(title[i])
        # Getting the pixel values in the [0, 1] range to plot.
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.savefig('results/' + str(image_index) + '.jpg')


@tf.function
def train_step(input_image, target, step):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        gen_output = generator(input_image, training=True)

        disc_real_output = discriminator([input_image, target], training=True)
        disc_generated_output = discriminator([input_image, gen_output], training=True)

        gen_total_loss, gen_gan_loss, gen_l1_loss = generator_loss(disc_generated_output, gen_output, target)
        disc_loss = discriminator_loss(disc_real_output, disc_generated_output)

    generator_gradients = gen_tape.gradient(gen_total_loss,
                                            generator.trainable_variables)
    discriminator_gradients = disc_tape.gradient(disc_loss,
                                                 discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(generator_gradients,
```

```
                                            generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                                discriminator.trainable_variables))

    with summary_writer.as_default():
        tf.summary.scalar('gen_total_loss', gen_total_loss, step=step // 1000)
        tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=step // 1000)
        tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=step // 1000)
        tf.summary.scalar('disc_loss', disc_loss, step=step // 1000)


def fit(train_ds, test_ds, steps):
    example_input, example_target = next(iter(test_ds.take(1)))
    start = time.time()

    for step, (input_image, target) in train_ds.repeat().take(steps).enumerate():
        if step % 1000 == 0:
            display.clear_output(wait=True)

            if step != 0:
                print(f'Time taken for 1000 steps: {time.time() - start:.2f} sec\n')

            start = time.time()

            generate_images(generator, example_input, example_target, step.numpy() // 1000)
            print(f"Step: {step // 1000}k")

        train_step(input_image, target, step)

        # Training step
        if (step + 1) % 10 == 0:
            print('.', end='', flush=True)

        # Save (checkpoint) the model every 5k steps
        if (step + 1) % 5000 == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)


if __name__ == '__main__':
    config = ConfigProto()
    config.gpu_options.allow_growth = True
    session = InteractiveSession(config=config)

    # Adjust this value to the number of training images
    BUFFER_SIZE = 400
    # The batch size of 1 produced better results for the U-Net in the original pix2pix experiment
    BATCH_SIZE = 1
    # Each image is 256x256 in size
    IMG_WIDTH = 256
    IMG_HEIGHT = 256

    PATH = '../PIX2PIX/images/combined/candles/'
    train_dataset = tf.data.Dataset.list_files(PATH + 'train/*.png')
    train_dataset = train_dataset.map(load_image_train,
                                      num_parallel_calls=tf.data.AUTOTUNE)
    train_dataset = train_dataset.shuffle(BUFFER_SIZE)
    train_dataset = train_dataset.batch(BATCH_SIZE)

    try:
        test_dataset = tf.data.Dataset.list_files(str(PATH + 'test/*.png'))
    except tf.errors.InvalidArgumentError:
        test_dataset = tf.data.Dataset.list_files(str(PATH + 'val/*.png'))
    test_dataset = test_dataset.map(load_image_test)
    test_dataset = test_dataset.batch(BATCH_SIZE)

    OUTPUT_CHANNELS = 3

    generator = build_generator()
```

```
LAMBDA = 100

loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)

discriminator = build_discriminator()

generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

log_dir = "logs/"

summary_writer = tf.summary.create_file_writer(
    log_dir + "fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))

fit(train_dataset, test_dataset, steps=40000)

# Restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

# Run the trained model on a few examples from the test set
index = 1000
for inp, tar in test_dataset.take(5):
    generate_images(generator, inp, tar, index)
    index = index + 1
```