# Praktikumsbericht: Entwicklung von Autoencodern für intelligente Agenten basierend auf neuartigen neuronalen Netzen

Dennis Loran

Institut für Programmstrukturen und Datenorganisation (IPD)
Betreuender Mitarbeiter: Daniel Zimmermann und Björn Jürgens

Ziel des Praktikas ist die Entwicklung mehrerer Autoencoder für das OpenAI-Procgen Spiel Heist. Die Autoencoder werden anschließend in eine Gym-Umgebung gepackt, die für CTRNN's verwendet werden können. Dieses Dokument ist als Handbuch zu verstehen. Zunächst einmal werden grob die Anforderungen für die entwickelten Autoencoder gestellt. Danach folgt eine Erläuterung zu der gewählten Architektur. Die einzelnen Komponenten werden dann im weiteren Kapitel detaillierter vorgestellt. Abschließend folgt eine Bewertung der entwickelten Autoencoder, sowie eine kurze Zusammenfassung über die Ergebnisse des Praktikas.

## 1. Anforderungen

Ziel dieses Praktika ist für das Procgen Spiel Heist mehrere Autoencoder zu entwickeln. Die entwickelten Autoencoder sollten dann im nächsten Schritt in eine Gym-Umgebung gewrapped werden, damit sie für ein CTRNN's verwendet werden können.

Ein Autoencoder ist ein Neurales Netz das verwendet wird um effiziente Codierungen zu lernen. Üblicherweise bestehen Autoencoder aus einem Encoder und einen Decoder. Der Encoder erhält das Eingabe Bild und erzeugt eine komprimierte Repräsentation (Encoding). Der Decoder erhält eine komprimierte Repräsentation und versucht damit das Eingabebild zu rekonstruieren (Siehe Abbildung 1) . Ziel ist es Decoder $Dec$ und Encoder $Enc$ so zu trainieren, dass $I = Dec(Enc(I))$ gilt.

### 1.1. Funktionale Anforderungen

Folgende Funktionale Anforderungen wurden an die Entwicklung der Autoencoder gestellt.

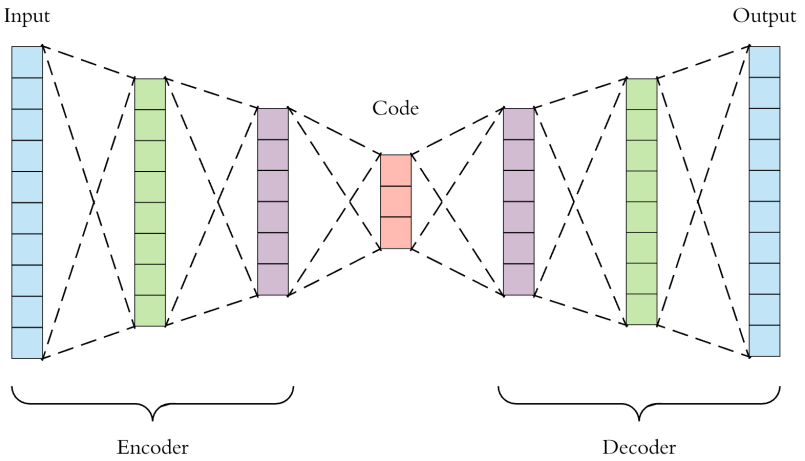



Abbildung 1: Typische Encoder-Decoder Architektur eines Autoencoders. Das Ergebnis der Codierung ist die benötigte Ausgabe eines Encoders

/FA10/ Für das Procgen Spiel Heist sollen mehrere Autoencoder entwickelt werden

/FA20/ Für das Trainieren der Autoencoder müssen Trainingsdaten erstellt werden

/FA30/ Die Autoencoder sollten trainiert werden können

/FA40/ Die Autoencoder müssen nach dem Training evaluiert werden können

/FA50/ Die Trainierten Autoencoder müssen in eine Gym Umgebung gewrapped werden

Da der Autoencoder in Kombination mit einem CTRNN's eingesetzt wird waren folgende Nicht Funktionale Anforderungen gestellt.

## 1.2. Nicht Funktionale Anforderungen an Autoencoder

/NFA10/ Die Autoencoder sollten mithilfe des Rahmenwerks *pytorch* entwickelt werden

/NFA20/ Die Codierung sollte *möglichst* klein sein. Ein Merkmalsvektor um die 100-Dimensionen war erstrebenswert, ein Merksmalsvektor der Größe 1200 allerdings auch noch akzeptabel.

/NFA30/ Der Autoencoder sollte in der Lage sein auch in Echtzeit eingesetzt werden zu können. Als Echtzeitfähig wurde eine Laufzeit von 5 Millisekunden pro Eingabe auf einer herkömmlichen CPU angenommen.

/NFA40/ Der Autoencoder sollte auch händisch anhand einiger Testdaten beurteilt werden. Dies liegt daran, dass der Durchschnittliche-Quadratische-Fehler als Messeinheit alleine nicht ausreicht. So kann Beispielsweise der Fehler klein sein, wesentliche Merkmale (Schlüssel,Schlösser,Kristalle) können aber fehlen.

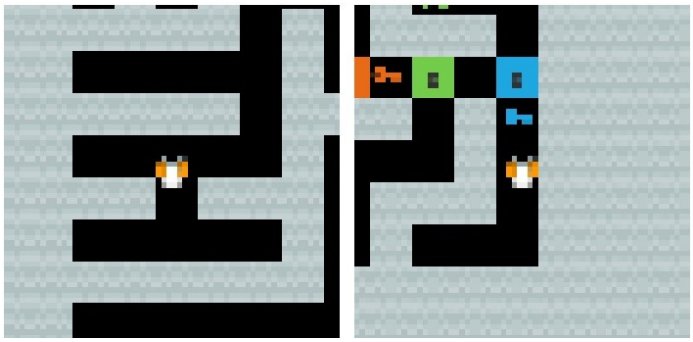


Abbildung 2: Zu sehen sind 2 Bilder, eins in dem der Agent keine Schlüssel sieht, rechts ein Agent der alle 3 Schlüssel sieht.

Es ist hier wichtig zu erwähnen, dass es sich um Konflikt zueinander stehenden Anforderungen handelt. Ein möglichst kleiner Merksmalsvektor benötigt ein relativ tiefes Neuronales Netz, was dazu führt, dass die Echtzeitfähigkeit nicht mehr garantiert werden kann. Wird hingegen auf Echtzeit optimiert, so können die Architekturen nicht mehr zu tief sein, was dazu führt, dass auch der Merkmalsvektor länger sein muss, um gute Ergebnisse zu erzielen. Es ist also sinnvoll nicht nur einen Autoencoder zu entwickeln, sondern mehrere mit unterschiedlichen optimierten Architekturen. Ein Bild lässt sich dabei als Array der Größe [64x64x3] darstellen. Somit besitzt ein Eingabebild eine Dimension von 64x64x3=12288 Dimensionen.

## 1.3. Procgen

Procgen ist ein von OpenAI entwickelterer Benchmark um die Performance von Reinforcement Learning Algorithmen zu messen. Procgen umfasst 16 einfach zu bedienende prozedural generierte Umgebungen. Intern verwenden Procgen Umgebungen eine Gym Umgebung, weshalb bereits entwickelte Algorithmen für Gym Umgebungen leicht wiederverwendet werden können. Eine detailliertere Beschreibung zu Gym Umgebungen kann in `https://gym.openai.com/docs/` gefunden werden.

### 1.3.1. Procgen Spiel Heist

In diesem Praktika sollte ein Autoencoder für das Procgen Spiel Heist entwickelt werden. Der Spieler (Im weiteren Agent genannt) hat die Aufgabe in einem generierten Labyrinth 1-3 Schlüssel zu finden, damit die jeweiligen Türen zu öffnen um einen gelben Kristal zu schnappen. Es gibt 3 Schlüsselfarben, Rot,Grün und Blau. Auch die Schlösser haben die Farbe Rot,Grün und Blau. Zum öffnen eines Schloss benötigt der Agent deshalb einen Schlüssel in zugehöriger Farbe. Der Agent hat bei der Suche einen beschränkten Sichtradius (Siehe Abbildung 2)

### 1.4. Ablauf Entwicklung

Bei der Entwicklung der Autoencoder wurde wie folgt vorgegangen

1. Erstellung des Training und Testdatensatzes
2. Implementierung gängiger Autoencoder
3. Trainieren der implementierten Autoencoder
4. Evaluierung der trainierten Autoencoder
5. Erstellen der Procgen-Umgebungen mit den Autoencodern

## 2. Architektur

Die Architektur wird in Abbildung 3 beschrieben. Die Grundsätzliche Pipeline läuft wie folgt ab. Zunächst einmal werden mithilfe der `Data Generation` Komponente Trainings und Testdaten generiert. In dieser sind die Trainings- und Testdaten für das Procgen Spiel Heist. Diese Daten werden dann auf die Festplatte gespeichert. Anschließend wird eine «`component`» `Autoencoder` erstellt. Diese Komponente definiert dabei das Gerüst des Autoencoders. Nach erstellung des Autoencoders muss dieser Trainiert werden. Dazu ist Komponente `Training Autoencoder` verantwortlich. Dazu benötigt diese einen «`component`» `Autoencoder` und auch die Trainings und Testdaten. Nach dem Training wird der trainierte Autoencoder auf die Festplatte gespeichert. Danach wird der Autoencoder nach ausgewählten Kritieren evaluiert. Dafür ist die `Evaluation Autoencoder` Komponente verantwortlich.

Der Trainierte Autoencoder wird dann verwendet um eine Gym-Umgebung zu erstellen. Das passiert innerhalb der `Autoencoder Procgen-Environment` Komponente. Diese Komponente ist diejenige die dann von einem CTRNN weiter verwendet werden kann.

Hier folgt eine Erklärung der jeweiligen Komponenten:

- **«Component» Autoencoder**: Eine Autoencoder Komponennte. Diese wird mithilfe des Rahmenwerks pytorch realisiert.
- **Data Generation:** Diese Komponente ist verantwortlich für das Generieren von Trainings und Testdaten. Die Daten werden anschließend auf der Festplatte abgespeichert
- **Training Autoencoder:** Diese Komponente erhält einen Autoencoder und trainiert diesen. Der Trainierte Autoencoder wird dann abschließend auf der Festplatte gespeichert.
- **Evaluation Autoencoder:** Diese Komponente lädt einen Autoencoder von der Fesplatte und Evaluiert diesen Anhand ausgewählter Kriterien
- **Autoencoder Procgen-Environment:** Diese Komponente lädt einen Autoencoder und baut mit diesem eine Gym-Environment.

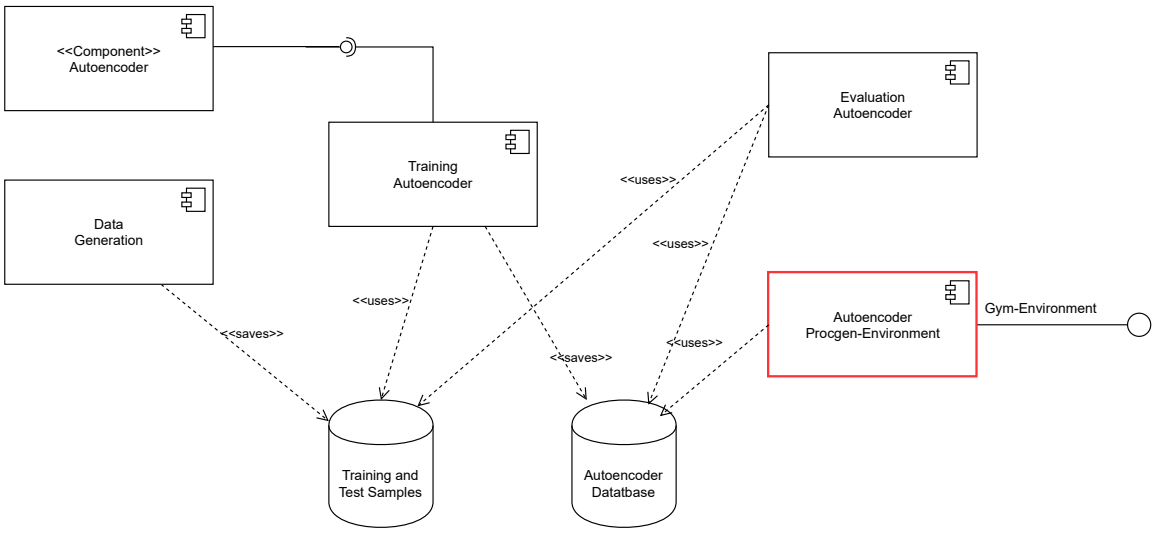



Abbildung 3: Informelles Komponenten Diagramm. Die rot gekennzeichnete Komponente ist die wesentliche Komponente die von einem CTRNN Anschließend verwenden werden kann

## 2.1. Data Generation

Zur Realisierung der Data Generation stehen 2 Skripte zu Verfügung.

`sample_generator_heist_basic.py` und `sample_generator_heist.py`

Für das Trainieren von Autoencoder benötigt man eine große Menge an Trainings und Testdaten. Diese lassen sich durch das Verwenden der Gym-Umgebung von Heist automatisch generieren. Dazu wird das Python Skript `sample_generator_heist_basic.py` verwendet. Die Grundlegende Idee ist es, ein Level zu laden, das Frame zu speichern und das nächste Level zu laden. Damit der Agent nicht ständig an der gleichen Stelle steht wird der Agent zusätzlich für 10 Frames zufällig bewegt.

Dieser Ansatz funktioniert bereits ausgesprochen gut. Es hat sich aber beim Training herausgestellt, das die Trainierten Autoencoder das Labyrinth perfekt darstellten, Schlüssel und Schlösser aber dafür häufig nicht beachtet werden. Das könnte daran liegen, dass die naive Generierung von Bildern zu einer ungleich Verteilten Menge an Bildern führen, bei denen die meisten Bilder keine Schlüssel und Schlösser enthalten. Um dieses Problem zu kompensieren wird das Skript `sample_generator_heist.py` verwendet.

Dabei wird zusätzlich für jedes generierte Bild überprüft, ob auf diesem ein Schlüssel oder Schloss dargestellt ist. Implementiert wurde diese Funktion mithilfe es Template Matching von OpenCV. Dazu wurden händisch einige Schlüssel und Schlösser ausgeschnitten und abgespeichert. Für ein Bild wird dann überprüft, ob dieses Template zu finden ist. Zu beachten ist dabei, dass das Template Matching einen hohen Rechenaufwand benötigt. Die Zeit für das Erstellen des Datensatz erhöht sich dadurch um den Faktor 20.

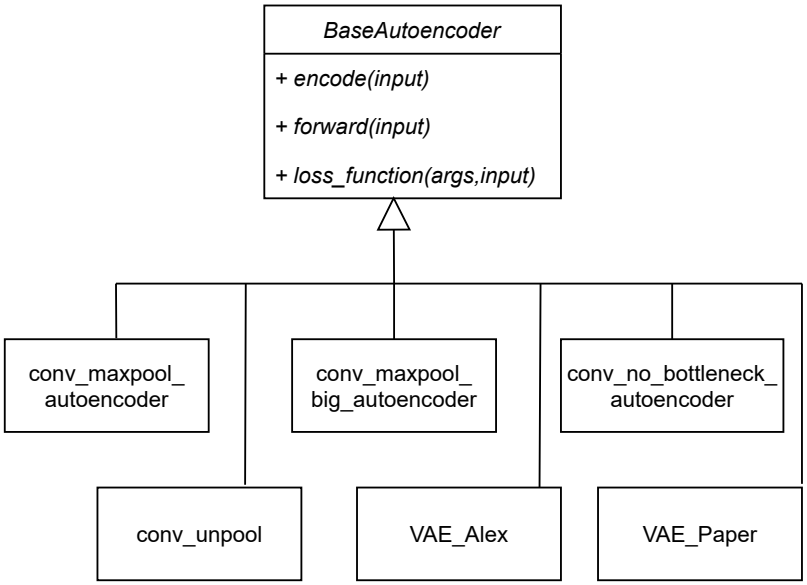



Abbildung 4: Klassendiagramm Autoencoder. Jede Erbende Klasse stellt dabei eine Autoencoder-Komponente dar

Nach dem Ausführen des Skriptes enstehen 2 Dateien, *training_samples_memory.npy* und *test_samples_memory.npy*. Das Testverhältnis kann in den beiden Skripten verändert werden, Standardmäßig beträgt es 9:1.

## 2.2. «Component» Autoencoder

Für die Implementierung der Autoencoder wurde das Rahmenwerk *pytorch* verwendet. Um die Autoencoder möglichst polymorph einsetzen zu können, wurde sich dafür entschieden eine Oberklasse `BaseAutoencoder` zu verwenden, von denen alle entwickelten Autoencoder erben (Siehe Abbildung 4)

### 2.2.1. Abstrakte Klasse `BaseAutoencoder.py`

Für das Training von Autoencodern mit pytorch ist es wichtig, dass jeder Autoencoder sowohl eine `forward` Funktion als auch eine `loss` Funktion besitzt. Zusätzlich ist für unseren Anwendungsfall natürlich wichtig, dass die Autoencoder eine Eingabe codieren können. Deshalb besitzt die Abstrakte Klasse folgende abstrakte Methoden:

- `encode(input)`: Methode die einen Batch an Bilder bekommt. Die Bilder werden durch durch den Encoder codiert. Das Ergebnis ist ein Batch mit den jeweiligen Codierungen
- `forward(input)`: Für das Training ist vom Rahmenwerk vorgeschrieben, dass eine forward Methode implementiert ist, die angibt, was bei einen Forward-Pass im Training passieren soll.

- `loss_function(input,output,args)`: Damit ein Netz überwacht trainiert werden kann ist es notwendig eine Bewertung für die jeweilig generierte Ausgabe zu geben. Dazu ist eine Fehlerfunktion notwendig. args enthält weitere benötigt Parameter.

### 2.2.2. Autoencoder

Die Architekturen der jeweiligen Autoencoder kann dem Anhang entnommen werden. Folgende Tabelle gibt für einen Autoencoder an, in welchen Format die Codierung ist und wie schnell der Encoder des Autoencoder ist.

| Autoencoder Name | Codierung | Encoder-Zeit GPU | Encoder-Zeit CPU |
|---|---|---|---|
| conv_maxpool_autoencoder | 32 | 0.4374ms | 1.211ms |
| conv_maxpool_big_autoencoder | 32 | 0.5217ms | 1.0303ms |
| conv_no_bottleneck_autoencoder | 128 | 0.5253ms | 1.8314ms |
| conv_unpool | [32x6x6] | 1.130ms | 2.673ms |
| VAE_Alex | 64 | 0.792ms | 1.688ms |
| VAE_Paper | 22 | :0.797ms | 1.646ms |

## 2.3. Training Autoencoder

Für das Training der Autoencoder steht das Skript `autoencoder_training.py` zur Verfügung. Zugehörige Hyperparameter können im `training_hyperparameter.yaml` modifiziert werden. Es folgt eine kurze Erklärung zu den jeweiligen Parametern:

- Daten-Parameter
  - `training_filepath`: Dateipfad zur Trainingsdatei
  - `test_filepath`: Dateipfad zur Testdatei
  - Der Name des gespeicherten Models und die Auswahl des Models werden als Kommondzeilen argumente angegeben.
  - `--name`: Name des gespeicherten Models
  - `--model_name`: Der Name des verwendeten Autoencoders. Siehe `Utils\model_factory` für
- Training-Parameter:
  - `batch_size`: Größe eines Batches
  - `epochs`: Anzahl an Epochen für das Training
  - `LR`: Lernrate im Training
  - `weight_decay`: Größe Verkleinerung des Einflusses der Gewichtsänderung jedem Gradient Descent Schritt.
  - `log_interval`: Wie häufig innerhalb einer Epoche eine Meldung zum Stand des Trainings gegeben wird.

  - `early_stopping`: Nach wie vielen konsekutiven Epochen ohne Testfehlerverbesserung das Training beendet werden soll.

- Logging-Parameter
  - `save_dir`: Ort an dem das Trainierte Model, sowie zugehörige Checkpoints gespeichert werden soll.
  - `seed`: interner Seed für Pytorch.

Ein Beispielsaufruf für das Training ist

```
python autoencoder_training --name myModel --model_name Conv_Unpool
```

Generell verwendet das Training den Optimierer Adam. Die Trainingsdaten werden mithilfe das Skripts `NumpyDataLoader.py` in eine für pytorch verständliche Repräsentation gebracht. Während dem Training wird jeweils das momentan beste Model zusätzlich abgespeichert. Nach dem Training wird das Model evaluiert. Dazu wird das Skript `performance_eval.py` verwendet.

## 2.4. Evaluation Autoencoder


Abbildung 5: Auszug des Evaluation-Videos. Links zu sehen ist das Eingabebild. Rechts die Ausgabe des Autoencoders. Oben rechts ist zusätzlich der zugehörige Fehler abgebildet.

Für die Evaluation wird das Skript `performance_eval.py` verwendet. Folgende Aspekte werden Evaluiert:

- Laufzeit des Encoders auf einer CPU mit Batch-Size 1 und 64
- Laufzeit des Encoders auf einer GPU mit Batch-Size 1 und 64
- Es wird ein Video erstellt, wo für jede Testbild die Ausgabe des Encoders und der zugehörige Fehler dargestellt wird (Siehe Abbildung)

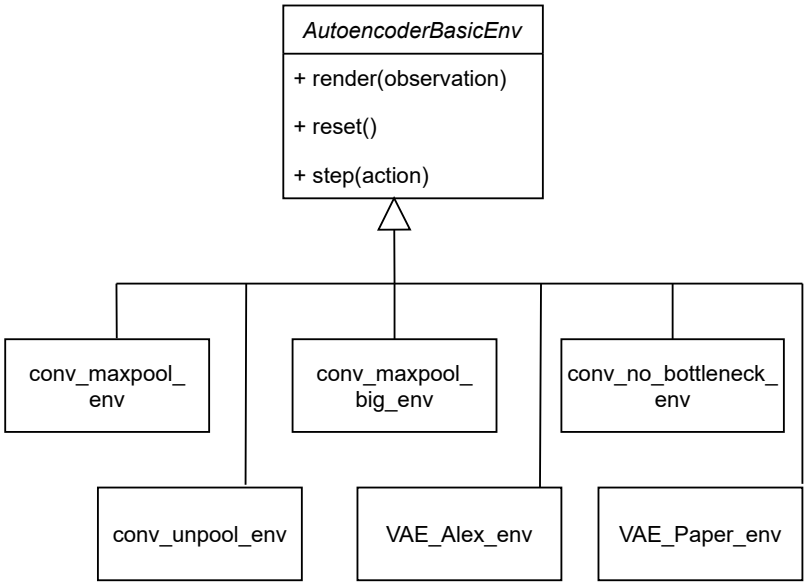



Abbildung 6: Informelles Komponenten Diagramm. Die rot gekennzeichnete Komponente ist die wesentliche Komponente die von einem CTRNN Anschließend verwenden werden kann

- Trainings und Test Loss werden in einem Diagramm dargestellt.

Es wird eine Video Datei erstellt, sowieso eine `summary.txt` wo die Laufzeiten und die Model-Architektur abgespeichert werden.

## 2.5. Procgen Umgebungen aus den Autoencoder

Nachdem die Autoencoder Trainiert worden sind, müssen diese noch in eine Gym-Umgebung gewrapped werden. Dies wird ähnlich wie in Abbildung 4 geregelt. Für jeden Autoencoder wird eine zugehörige Klasse erstellt. Diese erben alle von `AutoencoderBasicEnv.py`. Diese stellt bereits die Methoden `render`,`reset` und `step` zur Verfügung und müssen nicht überschrieben werden. Die Unterklassen müssen lediglich den Model Namen des Autoencoders (`path_stub`), einen Pfad für den gespeicherten Autoencoder (`path_model`) und die Größe der Codierung (`shape`) angeben. Abschließend müssen die Autoencoder noch in der `__init__.py` initalisiert werden.

# 3. Bewertung

In diesen Abschnitt soll eine kurze Bewertung der entwickelten Modelle, sowie Rückschlüsse für weitere Arbeiten gezogen werden.

Es hat sich in der Arbeit gezeigt, dass Varitonale Autoencoder tendenziell besser funktionieren als gewöhnliche Autoencoder. Der `VAE_Paper` funktioniert deutlich besser als als die gewöhnlichen Autoencoder (`conv_maxpool_autoencoder` `conv_maxpool_big_autoencoder`

conv_no_bottleneck_autoencode) bei gleicher Anzahl an Codierung Dimensionen. Der Hyperparameter lamda_kl_loss ist aber vergleichsweise schwer zu optimieren, ohne das es zum "Posterior Collapse"kommt. Dennoch reicht diese Art von Autoencoder für das Heist Spiel nicht aus. Das liegt aber nicht an den Autoencodern sondern am Spiel Heist selbst. Die Schlüssel sind die entscheidenden Elemente des Spiels. Diese sind aber nur einige Pixel groß, sodass ein Autoencoder diese eher ignoriert, da sie nicht groß zum gesammten Fehler beitragen. Ein Neuronales Netz optimiert deshalb eher darauf, dass das Labyrinth gut dargestellt wird.

Eine Möglichkeit diesen Fehler ein wenig zu verbessern wurde im VAE_Alex implementiert, in dem durch ein zusätzliches Neuronales Netz der Perceptual Loss eines Bildes berechnet wird. Dies führt zu deutlich besseren Ergebnissen, die Schlüssel werden deutlich häufiger erkannt. Dennoch ist das für eine Robuste Erkennung noch nicht ausreichend.

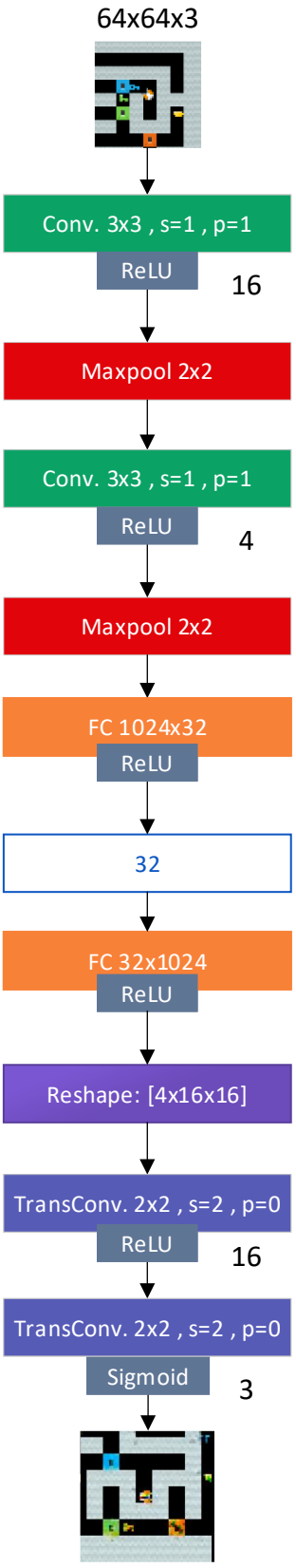
In der Arbeit hat sich heraugestellt, dass der Conv_Unpool Autoencoder mit Abstand die besten Ergebnisse erzielt. Dies liegt sicherlich auch an den deutlich größeren Merkmalsvektor, als auch tieferes Neuronales Netz.

In diesen Praktika lag der Fokus auf Neuronalen Netzen, die eine schnelle Ausführungszeit haben. Eine Weitergehende Arbeiten könnte deshalb die Entwicklung von Autoencoder sein, die auf einer Grafikkarte ausgeführt werden und somit deutlich tiefere Architekturen ermöglichen. Ein weiterer Aspekt könnte die Optimierung der Generierung von Trainingsdaten sein. Aktuell ist die Erstellung balancierter Trainingsdaten nicht parallelisiert und beansprucht deshalb einen hohen Zeitaufwand.

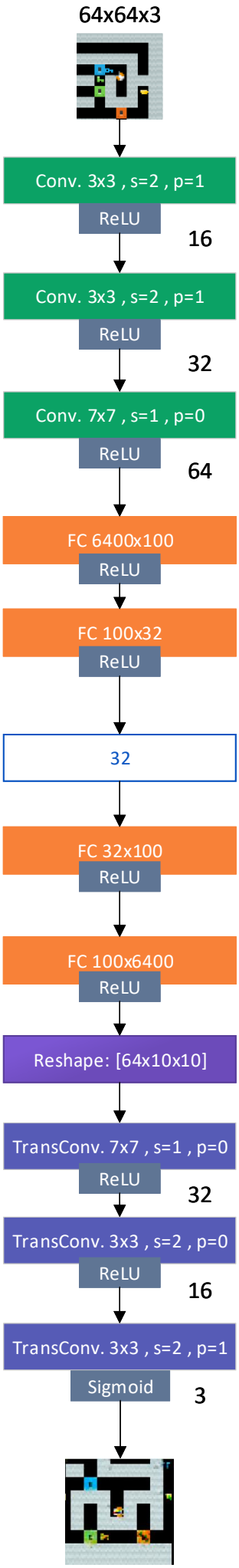
# A. Anhang

Im folgenden Finden sich die Architekturen der Autoencoder wieder. Die Netze sind dabei in Blöcke angegeben. Die Block-Interna sind innerhalb des Blockes beschrieben. Die Farben geben dabei verschiedene Block Arten an. Die zahl unterhalb eines Blocks gibt die Anzahl als Filter an. Der weiße Block, mit blauer Umrandung gibt die Ausgabegröße des Encoders an. Dieser Block trennt somit Encoder und Decoder Einheit eines Autoencoders.

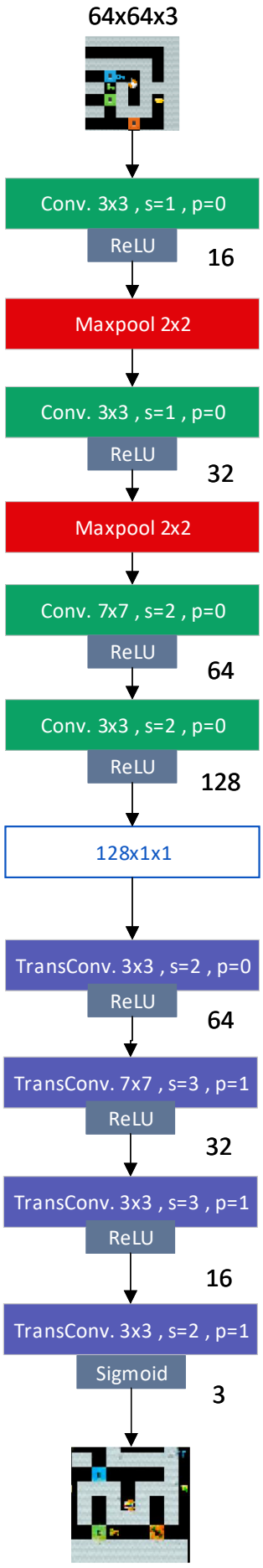
## Conv_maxpool

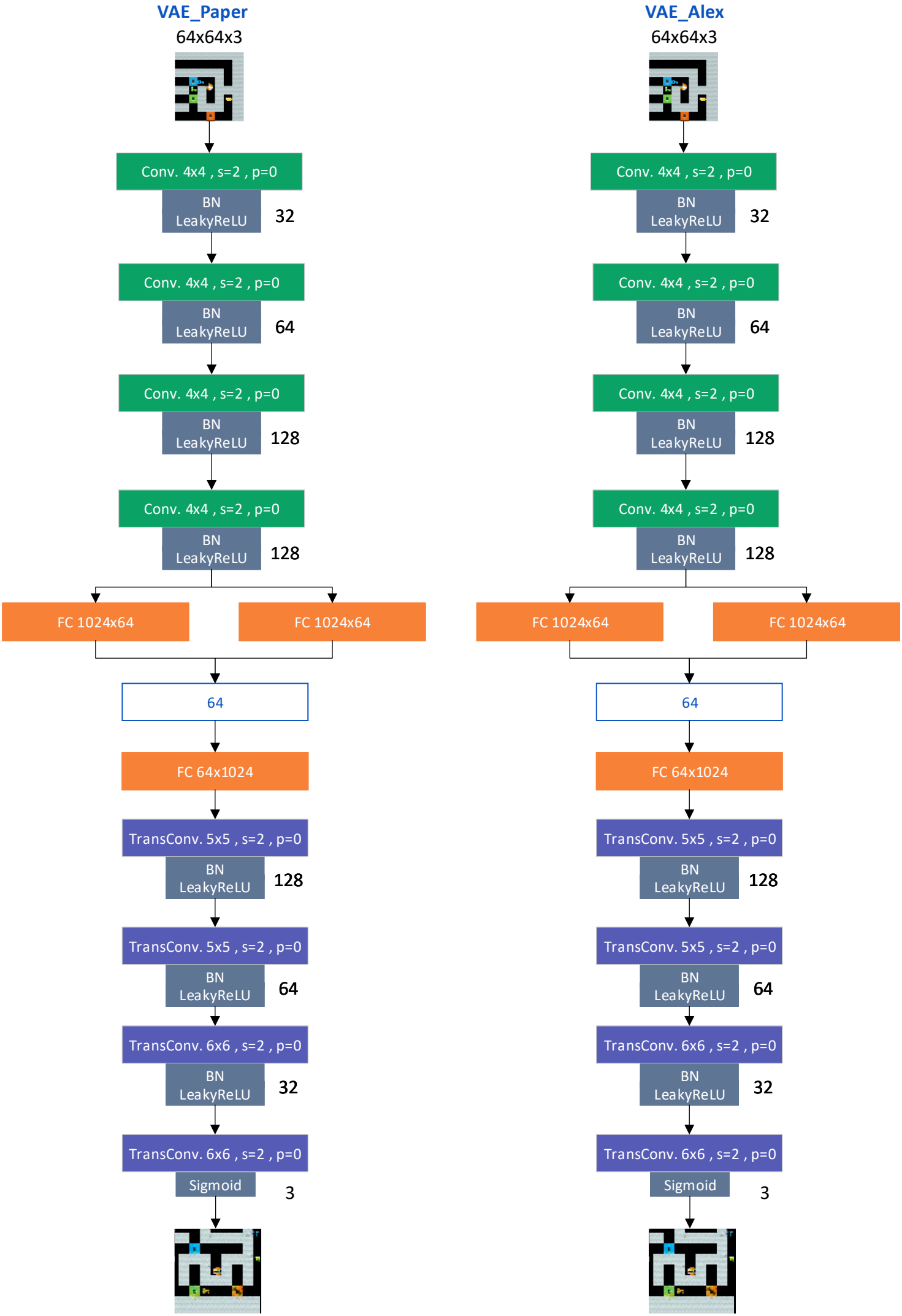



## Conv_maxpool_big

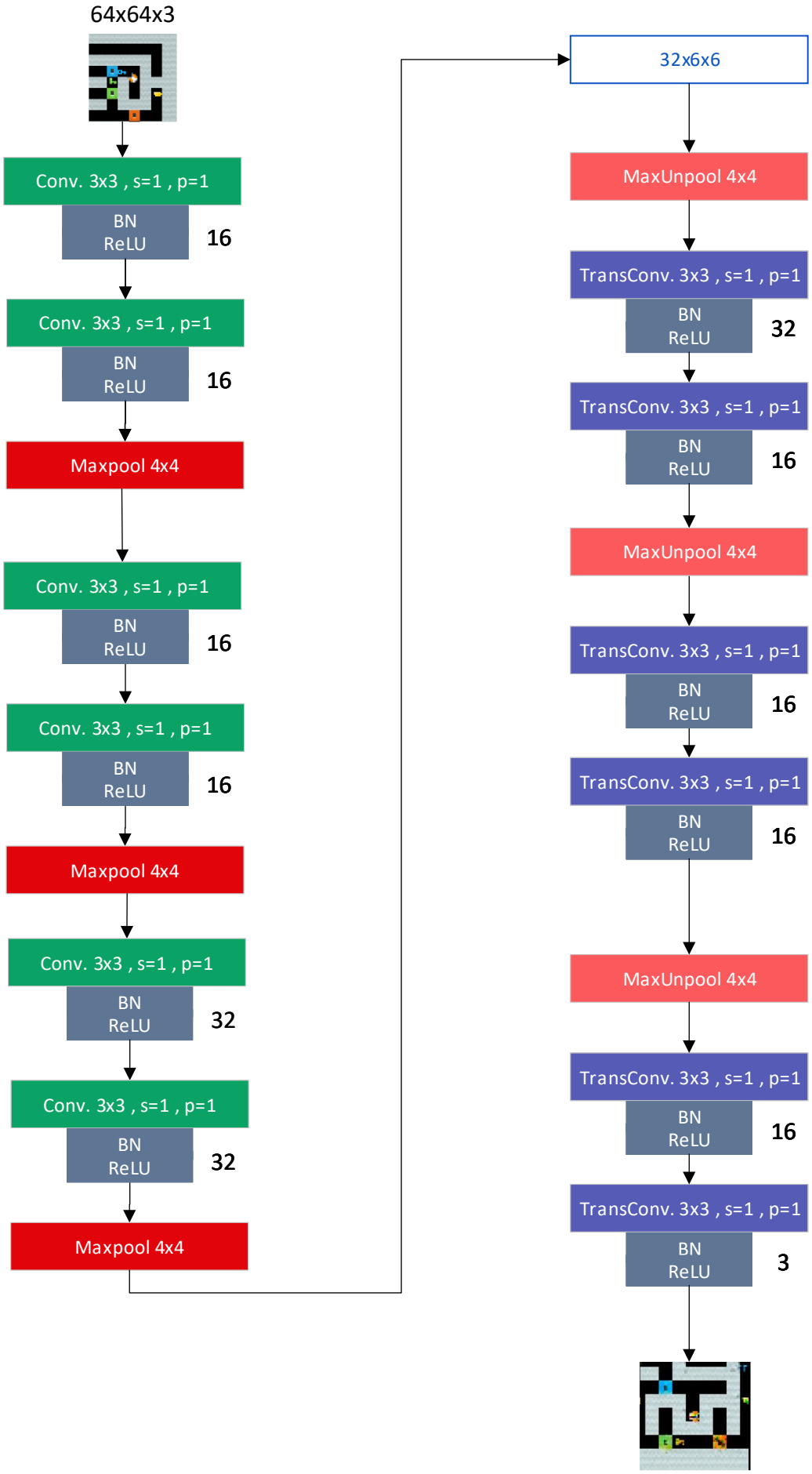



## Conv_no_bottleneck




VAE_Paper
64x64x3
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
32
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
64
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
128
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
128
FC 1024x64
FC 1024x64
64
FC 64x1024
TransConv. 5x5 , s=2 , p=0
BN
LeakyReLU
128
TransConv. 5x5 , s=2 , p=0
BN
LeakyReLU
64
TransConv. 6x6 , s=2 , p=0
BN
LeakyReLU
32
TransConv. 6x6 , s=2 , p=0
Sigmoid
3
VAE_Alex
64x64x3
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
32
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
64
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
128
Conv. 4x4 , s=2 , p=0
BN
LeakyReLU
128
FC 1024x64
FC 1024x64
64
FC 64x1024
TransConv. 5x5 , s=2 , p=0
BN
LeakyReLU
128
TransConv. 5x5 , s=2 , p=0
BN
LeakyReLU
64
TransConv. 6x6 , s=2 , p=0
BN
LeakyReLU
32
TransConv. 6x6 , s=2 , p=0
Sigmoid
3

## Conv_Unpool


64x64x3
Conv. 3x3 , s=1 , p=1
BN
ReLU
16
Conv. 3x3 , s=1 , p=1
BN
ReLU
16
Maxpool 4x4
Conv. 3x3 , s=1 , p=1
BN
ReLU
16
Conv. 3x3 , s=1 , p=1
BN
ReLU
16
Maxpool 4x4
Conv. 3x3 , s=1 , p=1
BN
ReLU
32
Conv. 3x3 , s=1 , p=1
BN
ReLU
32
Maxpool 4x4
32x6x6
MaxUnpool 4x4
TransConv. 3x3 , s=1 , p=1
BN
ReLU
32
TransConv. 3x3 , s=1 , p=1
BN
ReLU
16
MaxUnpool 4x4
TransConv. 3x3 , s=1 , p=1
BN
ReLU
16
TransConv. 3x3 , s=1 , p=1
BN
ReLU
16
MaxUnpool 4x4
TransConv. 3x3 , s=1 , p=1
BN
ReLU
16
TransConv. 3x3 , s=1 , p=1
BN
ReLU
3