LUMINOUS PRODUCTIONS

DirectX12を使用したシェーダー上への独自ニューラルネットワーク構築と描画品質向上のためのリアルタイムレンダリングへの応用

Construction of unique neural network model on shader using DirectX12, and applicate a real-time rendering for improved images quality.

Graphics Programmer
Kento Masuno

LUMINOUS PRODUCTIONS

本セッション

ニューラルネットワークのリアルタイムレンダリングへの応用

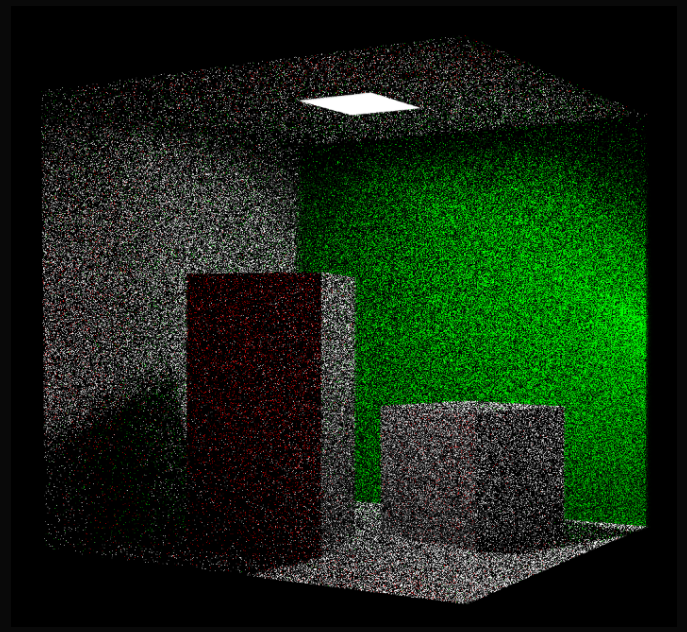
・従来のアルゴリズム処理では実現が難しい、ピクセルの補間のような表現が出来ないか？
・容量が大きくなるリソースデータを、ランタイムで超解像度化ができないか？
・ニューラルネットワークの重みデータの更新のみで、描画品質が向上できたりしないか？

→ そんな発想からニューラルネットワークの活用を検討

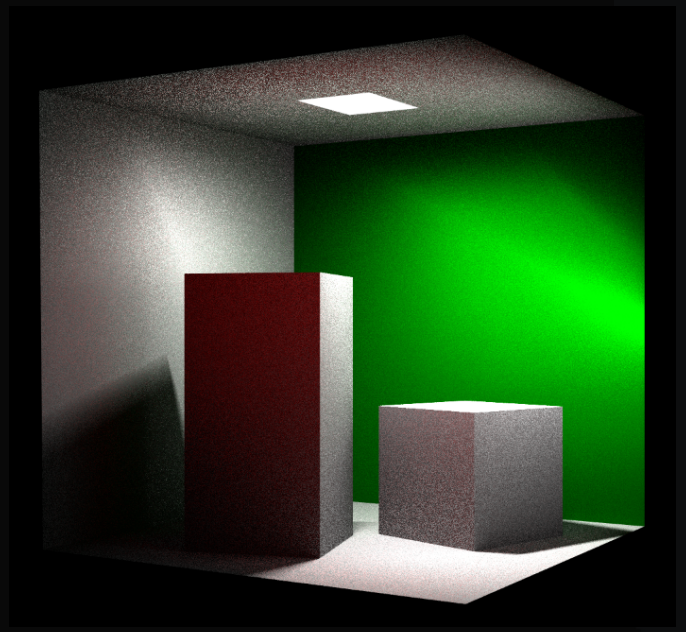
LUMINOUS PRODUCTIONS

Final Result

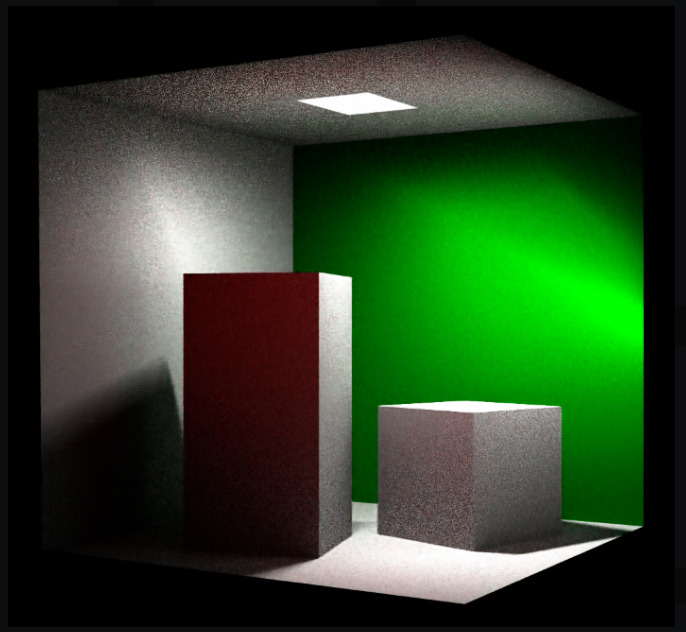
No generalization performance



Input Image
10spp



Ground Truth
100spp

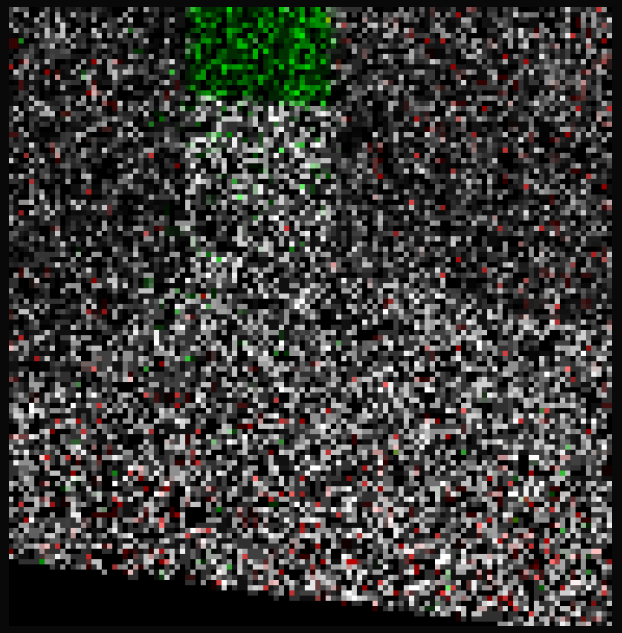


Denoising
Auto Encoder

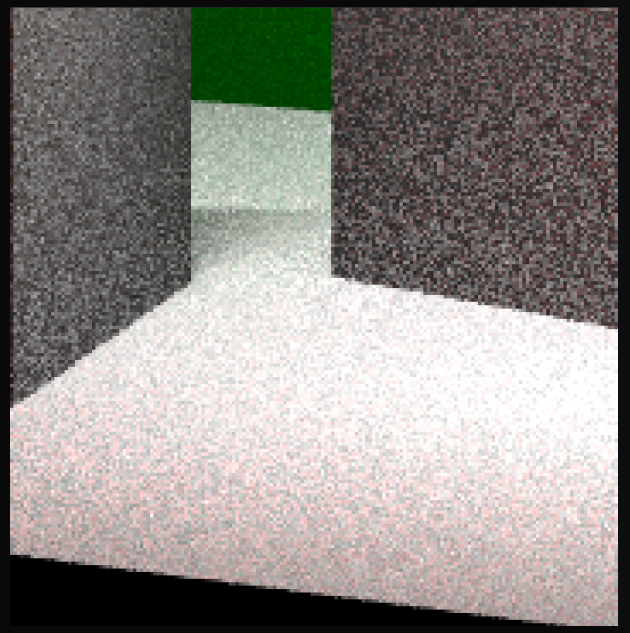
LUMINOUS PRODUCTIONS

Final Result

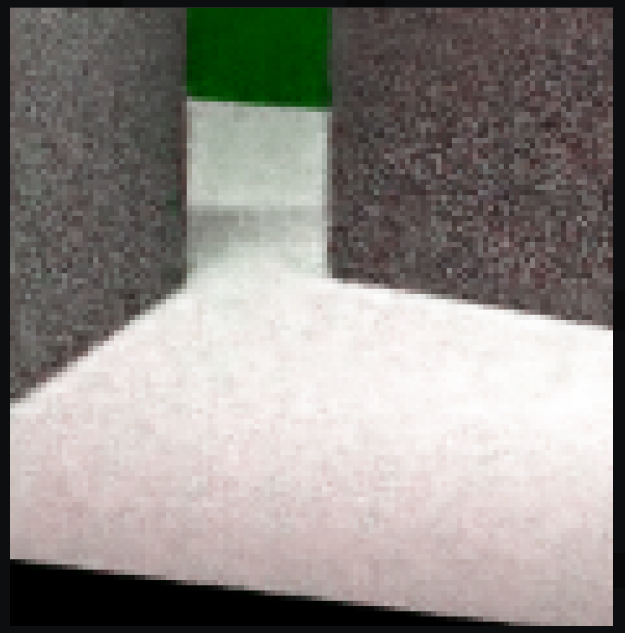
No generalization performance



Input Image
10spp



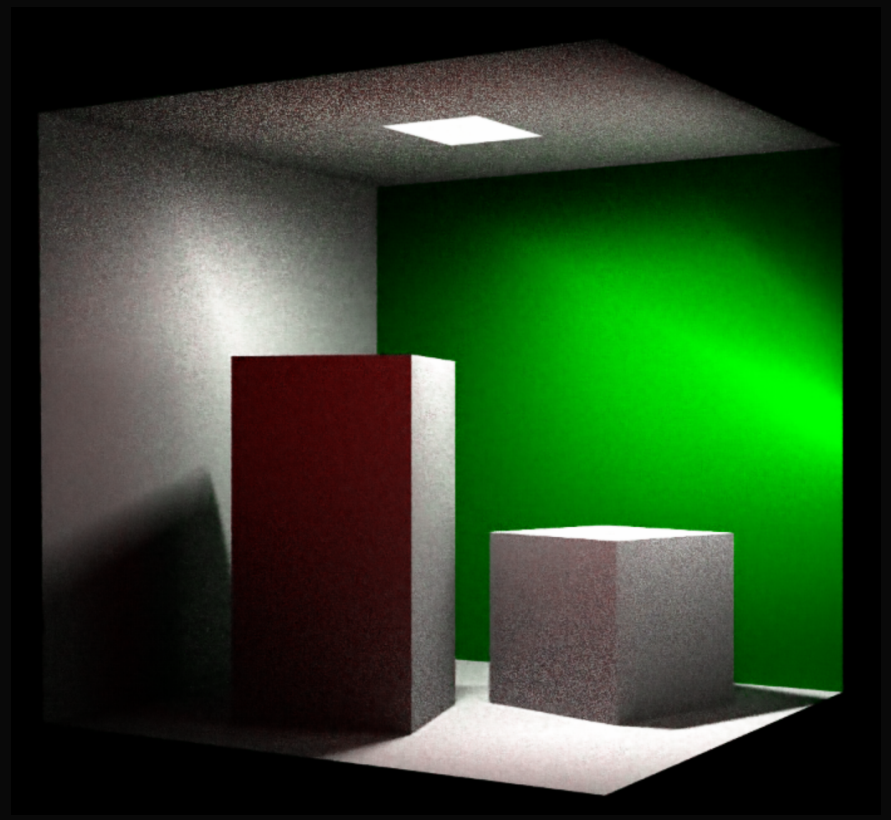
Ground Truth
100spp



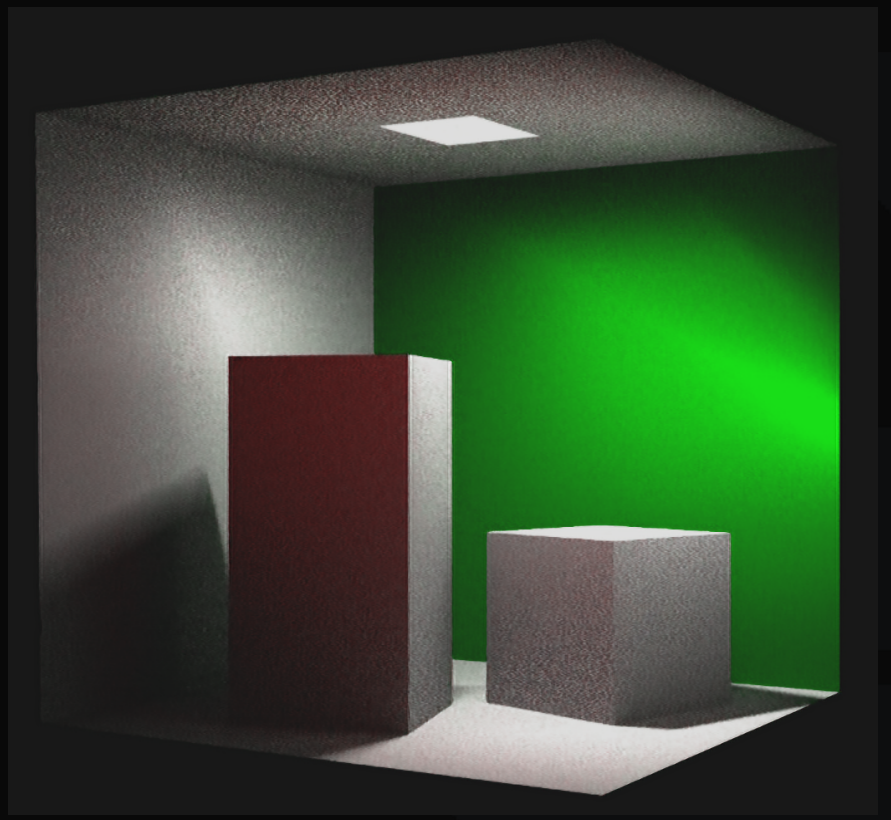
Denoising
Auto Encoder

LUMINOUS PRODUCTIONS

Final Result



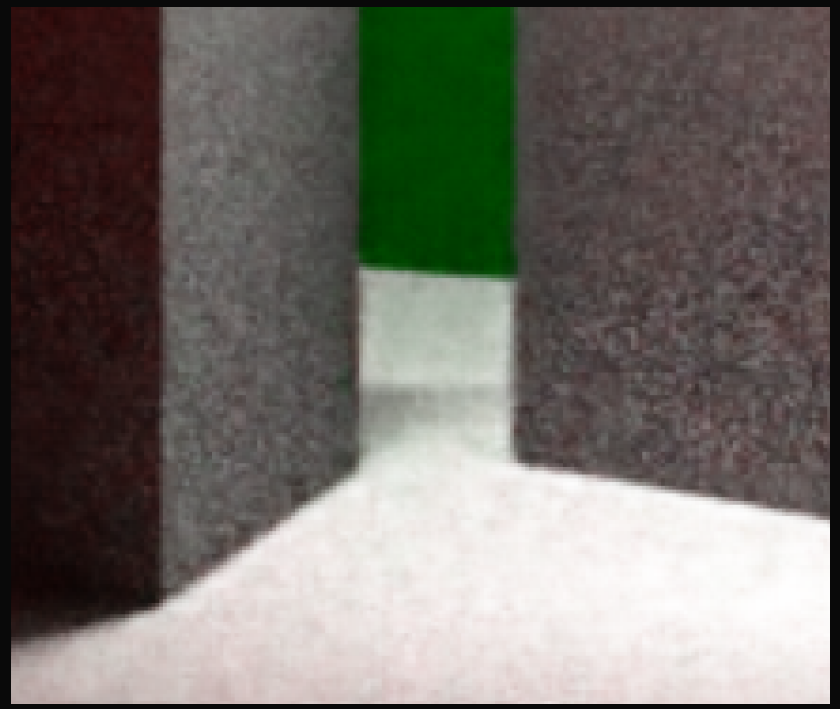
Bilinear



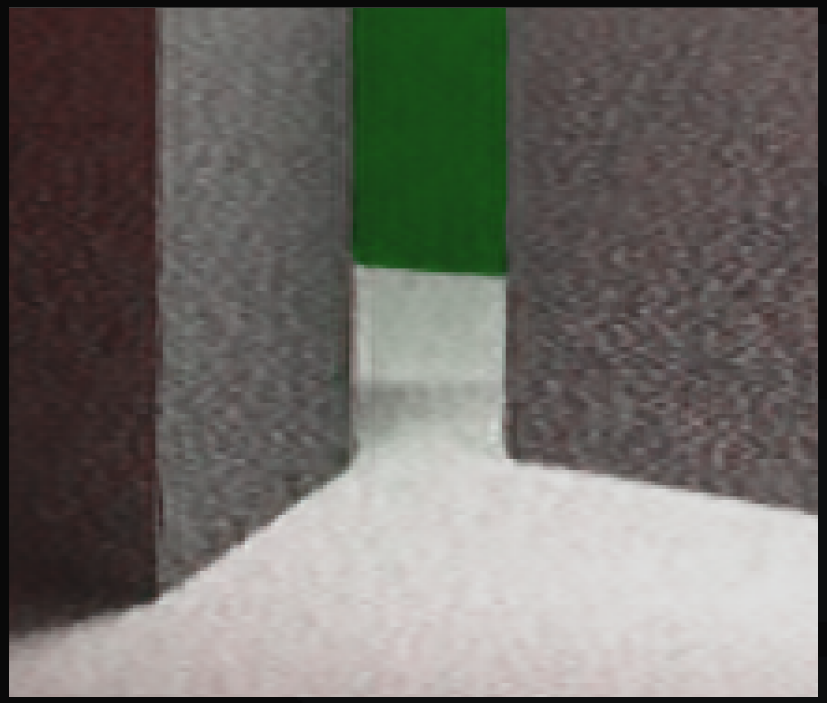
SRCNN
(※色域も変化)

LUMINOUS PRODUCTIONS

Final Result



Bilinear



SRCNN

LUMINOUS PRODUCTIONS

コンテンツ

- リアルタイムレンダリングの進化と機械学習
- 描画のためのニューラルネットワークとモデル構築
- シェーダー上へのニューラルネットワークの構築
- リアルタイムレンダリングへの機械学習の応用
- まとめ、質疑応答

※ 本セッションやプレゼン内でニューラルネットワークを NN と省略することがあります

LUMINOUS PRODUCTIONS

リアルタイムレンダリングの進化と機械学習

Real-time rendering and machine learning.

LUMINOUS PRODUCTIONS

コンピュータグラフィックス

最近のゲームグラフィックスで、キーワードとして挙げられる技術は何か？

- **リアルタイムレイトレーシング**
- **機械学習**

→ この2つの技術は今後のリアルタイムレンダリングを支える主要素になると仮定
DirectX Raytracing / DirectML

DirectX Raytracing

リアルタイムレイトレーシングの実現

- DirectX12 にレイトレーシング用のシェーダー機能が追加
- GPU 内の Raytracing Core を活用し、非常に高速なレイと三角形の交差判定を実現

従来のラスタライザでは難しかった表現がシンプルに実装可能に

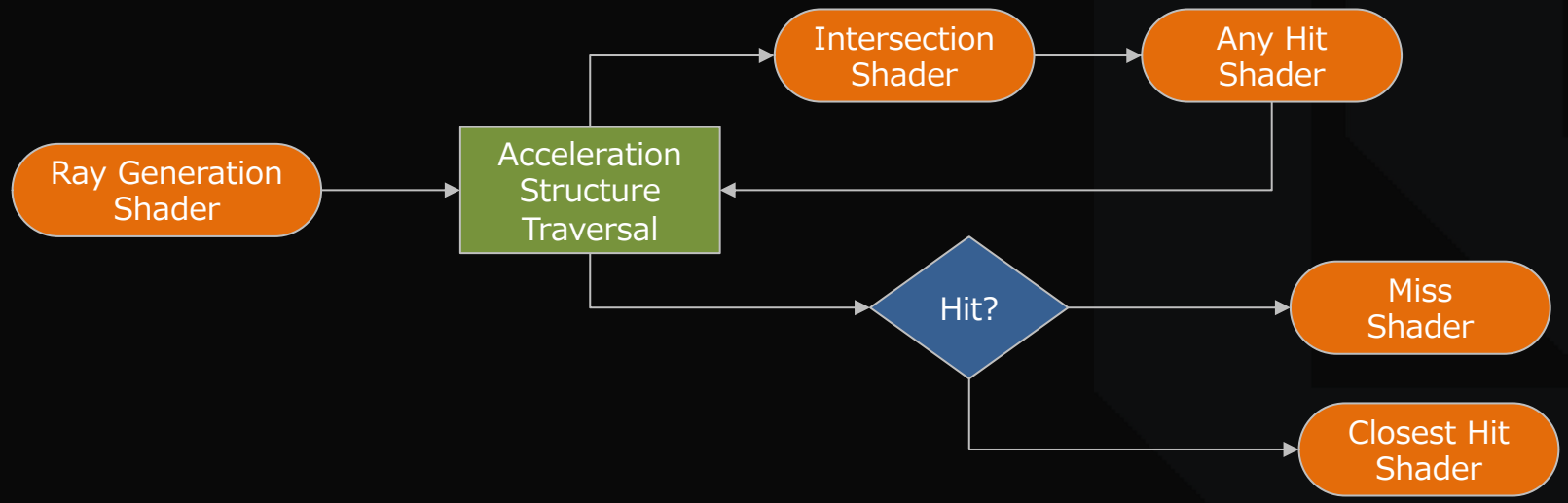
・グローバルイルミネーション (GI)
・正確な Reflection と Refraction 表現
・Shadow map の解像度に依存しない品質の高いソフトな影

DirectX Raytracing

ラスタライザとは仕組みが異なるレンダリングパイプラインを採用

・新しいシェーダーステージの追加

Ray Generation, Intersection, Any Hit, Miss, Closet Hit



https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing/

LUMINOUS PRODUCTIONS

Luminous Engine

弊社ゲームエンジン「Luminous Engine」 も DirectX Raytracing に対応済み
昨日行われた、下記 CEDEC セッションで詳細を発表

ルミナス・エンジンへのリアルタイムレイトレーシング実装事例の紹介
https://cedec.cesa.or.jp/2019/session/detail/s5ce39bf17ac37

LUMINOUS PRODUCTIONS

Direct Machine Learning

LUMINOUS PRODUCTIONS

Direct Machine Learning

DirectX Raytracing と同じタイミングで
DirectML (Machine Learning) という機械学習 API が DirectX Family 追加

DirectX からディープラーニングが構築可能になった
グラフィックス処理を始め、様々な機械学習分野のタスクへの適用が可能

- 超解像度
- スタイル変換
- デノイジング
- アンチエイリアス
- スーパーサンプリング

グラフィックス向けの想定されるタスクの一例

LUMINOUS PRODUCTIONS

Direct Machine Learning

DirectML は機械学習の **演算** を行える実行エンジン

→ NN 用の命令もあるが、比較的、低レベルな演算に特化した機能が提供

→ DirectX Family だからと言ってグラフィックス特化ではなく、様々な NN モデルに対応

ただし、ニューラルネットワークの学習は自前でやるか別の重みファイルを用意する必要がある

ハードウェアとして専用コアがあれば、非常に高速な推論処理が行われる

- 専用 HW が無い場合は、CPU (WARP) にフォールバックして動作が担保
- DirectML では HLSL などハードウェアに近いコーディングは行わない

Direct Machine Learning

DirectX アプリケーションからは D3D12 API と混在して処理が可能

→ Graphics / Compute と同じ Command Buffer に DirectML の命令を追加可能

D3D12 と DirectML は共通したバッファを参照できる

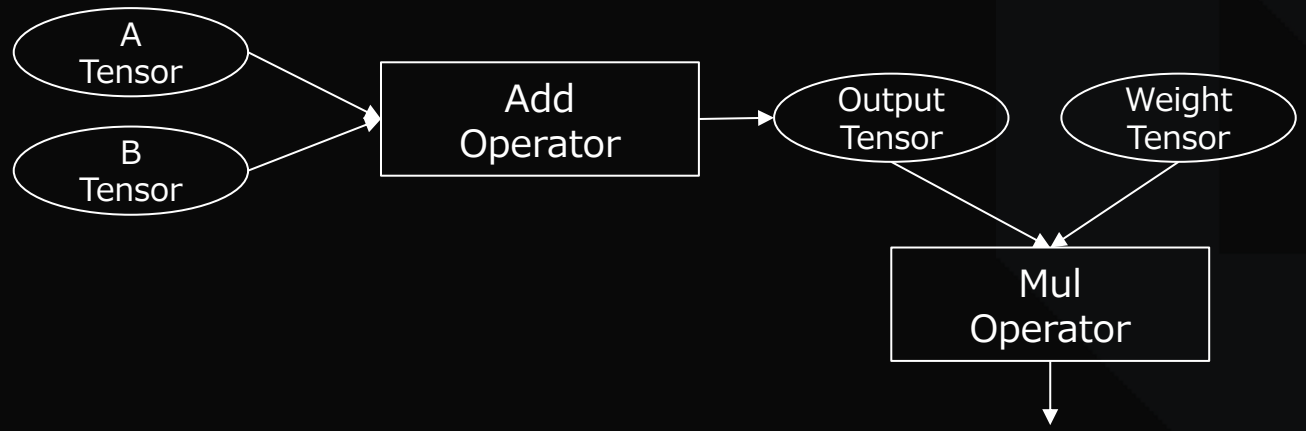
Graphics Task として描画したレンダリング結果を、DirectML に渡してバッファを操作し、
結果を再度 Graphics Task でバックバッファに書き込んで描画と言った一連の流れが可能になった

Command Buffer

Direct Machine Learning

DirectML を理解するためには、既存 NN フレームワークへの知識が必須

→ Tensor (データ) に対する Operation (操作) をデータフローグラフ的に記述していく
Tensorflow (≠Keras) に近いイメージだが、内部でグラフ情報は保持されない



DirectML 記述イメージ

LUMINOUS PRODUCTIONS

Direct Machine Learning

DirectML にも Tensor の概念が導入
既存 NN ライブラリのように、基本データ型は **Tensor** として定義

- D3D12 Buffer に Tensor が格納されていると考える。通常リソース *ID3D12Resource* と同じ
- Operator は Tensor を扱うための、下記のような Tensor レイアウト情報が必要

```
DML_BUFFER_TENSOR_DESC dmlBufferTensorDesc  = {};
dmlBufferTensorDesc.DataType                = DML_TENSOR_DATA_TYPE_FLOAT32;
dmlBufferTensorDesc.Flags                   = DML_TENSOR_FLAG_NONE;
dmlBufferTensorDesc.DimensionCount          = ARRAYSIZE(tensorSizes);
dmlBufferTensorDesc.Sizes                   = tensorSizes;
dmlBufferTensorDesc.Strides                 = nullptr;
dmlBufferTensorDesc.TotalTensorSizeInBytes  = DMLCalcBufferTensorSize(
    dmlBufferTensorDesc.DataType,
    dmlBufferTensorDesc.DimensionCount,
    dmlBufferTensorDesc.Sizes,
    dmlBufferTensorDesc.Strides);
```

DirectML Sample コードの一部。使用する Tensor 定義

LUMINOUS PRODUCTIONS

Direct Machine Learning

Tensor を操作するための Operator 定義
多くの Operator には処理する Tensor の Input と Output を指定

分かりやすい例だと、活性化関数 ReLU の Operator 定義は下記
この Operator は Input で入力された Tensor 要素の $f(x) = \max(0, x)$ を計算して出力

BLAS のような *DML_GEMM_OPERATOR_DESC (General Matrix Multiply)* 命令もあり
Input * Weight + bias の演算でニューロンの出力を計算

```
struct DML_ACTIVATION_RELU_OPERATOR_DESC {
    const DML_TENSOR_DESC *InputTensor;
    const DML_TENSOR_DESC *OutputTensor;
};
```

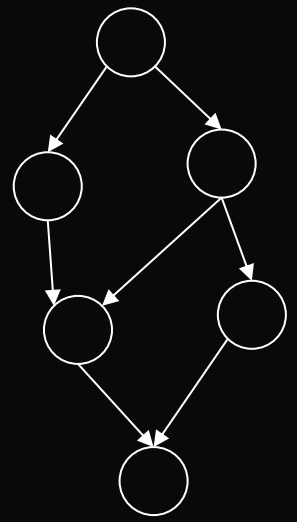
DML_ACTIVATION_RELU_OPERATOR_DESC structure
https://docs.microsoft.com/en-us/windows/win32/api/directml/ns-directml-dml_activation_relu_operator_desc

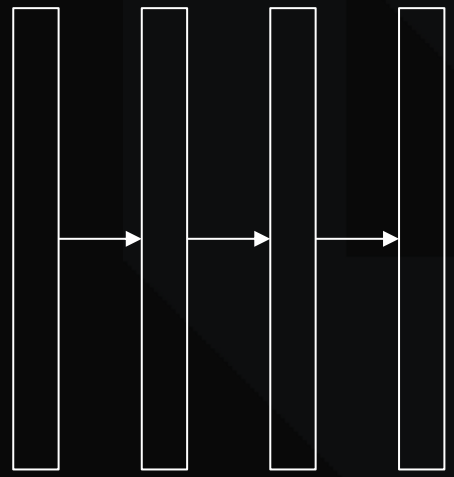
LUMINOUS PRODUCTIONS

Direct Machine Learning

このように Operator を複数組み合わせてデータフローグラフを構築し、
Tensor の処理結果を求めるのが DirectML のコードの流れ

Keras の様な高レベル API の場合は Layer 単位の追加でニューラルネットワークを構築出来るが、
Tensorflow や DirectML だとグラフの接続でデータを処理する、この考え方に近い



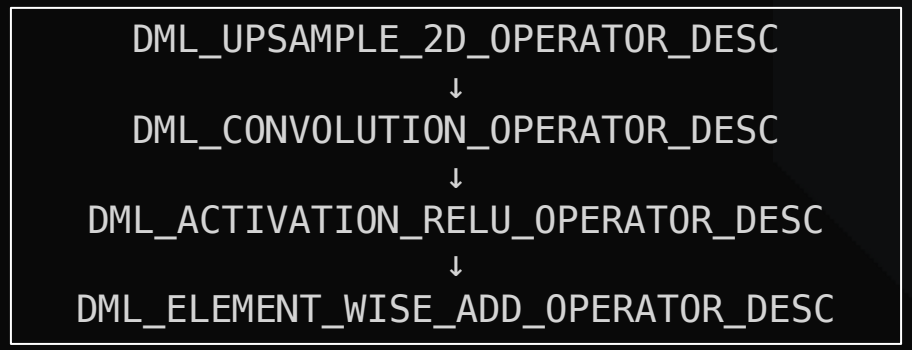


LUMINOUS PRODUCTIONS

Direct Machine Learning

DirectML には Super Resolution サンプルが提供されている
グラフィックスに機械学習応用を考えた時は、このソースコードから始めることをオススメ
　→ AI Core 搭載の最新 GPU であればリアルタイム 60 FPS に近い数字が出ている

描画結果のバッファに対して、ポスト処理的に複数の Operator でピクセルの処理を行い
結果をバックバッファに書き出す事で、超解像度の処理を D3D12 で実現



LUMINOUS PRODUCTIONS

Direct Machine Learning

ニューラルネットワークの重みデータは別の方法で用意する必要がある
既存 NN ライブラリで使われる ONNX のようなファイルも DirectML では直接扱えない

- Compute Shader 上で出力結果からの勾配を求め独自実装
- Tensorflow や PyTorch で学習した重み値のみを書き出して再利用なども検討
  既存のディープラーニングフレームワークであれば、専用の関数で重みが簡単に取れる

実行のための前処理、前提知識も多いため、実装はチャレンジング

LUMINOUS PRODUCTIONS

AI Core

ML API が策定された背景に、NN　処理を効率的に行う AI 専用コアの登場
一部の GPU やスマートフォンの SoC などにも搭載。名称や機能の詳細はそれぞれ異なる

専用コアは何を解決するか？

→ ニューラルネットワークの演算は、膨大な多次元テンソル演算の繰り返し
　専用コアはこの多要素の**行列アクセス**と**演算**を効率よく実行する

例えば、従来のコアでは数サイクル必要だった行列同士の乗算を
AI 専用コアでは1サイクルで実現するスループットを持つものも存在する

LUMINOUS PRODUCTIONS

AI Core

GPU はゲームなどの分野でラスタライズの効率化を目的として生まれたハードウェア

→ 現在ではゲームを超えて科学技術計算の分野で使われることが増加
そして一般ユーザーが持つ GPU にも AI Core を搭載するものが増えてきた

これが意味することは？

AI タスクを処理する専用 HW としての GPU 活用の幅が更に広がったと同時に、
グラフィックスと AI タスクの垣根が小さくなりつつある

DirectX という枠内であれば同じアプリでグラフィックスと AI の両タスクを実行可能

LUMINOUS PRODUCTIONS

Metal API

DirectX だけが特別な進化を遂げている訳ではない

→ グラフィックス API の機能拡張は Apple Metal が先行している
自社製品のため、仕様を切れる速度が速い

2017年に Metal Performance Shader (MPS) という仕組みを導入
Metal はいち早く Machine Learning も Raytracing にも対応した API

LUMINOUS PRODUCTIONS

Differentiable Renderer

リアルタイムレンダリングの分野からは少し離れるが
Differentiable Renderer (微分可能レンダラー) という仕組みが注目を集めている

- Neural 3D Mesh Renderer
- Tensorflow Graphics

シーンの生成に使われた Geometry, Light, Material, Transform などのパラメータを
レンダリング (ラスタライズ) 結果の微分により、チューニングしていく

勾配が求められれば、ニューラルネットワークの中でパラメータの学習が可能

Neural 3D Mesh Renderer
https://arxiv.org/abs/1711.07566

LUMINOUS PRODUCTIONS

Tensorflow Graphics

Tensorflow の機能として Tensorflow Graphics がリリースされた意味は大きい
→ Computer Graphics, Computer Vision での NN 活用の幅が広がる

例えば、ラスタライザ結果の微分からレンダリングに使われたシーンパラメータの推定ができる
画像を与えたら、その画像を構成する 3D シーンの復元なども可能？

- ライティング環境
- ジオメトリー、アニメーション
- BRDF

LUMINOUS PRODUCTIONS

リアルタイムグラフィックス

ここ一年ほどの間でリアルタイムグラフィックスの分野は
”レイトレーシング“ と ”機械学習“ の2つの方向に進化が始まっている

API の対応によりグラフィックスとディープラーニングが一つのアプリに混在可能に

今後のリアルタイムグラフィックス →

- Raytracing
- Machine Learning

LUMINOUS PRODUCTIONS

リアルタイムグラフィックス

機械学習のグラフィックス分野への応用は始まったばかり

→ 機械学習を使わずにアルゴリズムを書いた方が早いものも多いため活用方法は模索
→ DirectML には API レベルでの学習機能が存在しない

DirectML のような仕組みも始まったばかり (2019年5月リリース)

・API 使用の難度の高さや、学習機能など重みデータ管理の問題
・対応 OS が Windows 10 1903 以上に限定
・専用ハードウェアコアの搭載の有無による実行速度の差など

LUMINOUS PRODUCTIONS

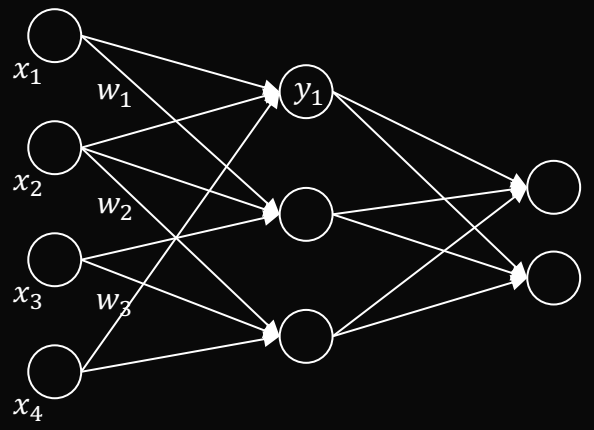
描画のためのニューラルネットワークとモデル構築

Neural network and image processing model.

ニューラルネットワーク

ニューラルネットワークは、ニューロン (ノード) の繋がりにより問題を解く数理モデル

- Tensorflow ではグラフの繋がりのイメージでネットワークを表現
- Keras では Layer (層) 間の繋がりで表される



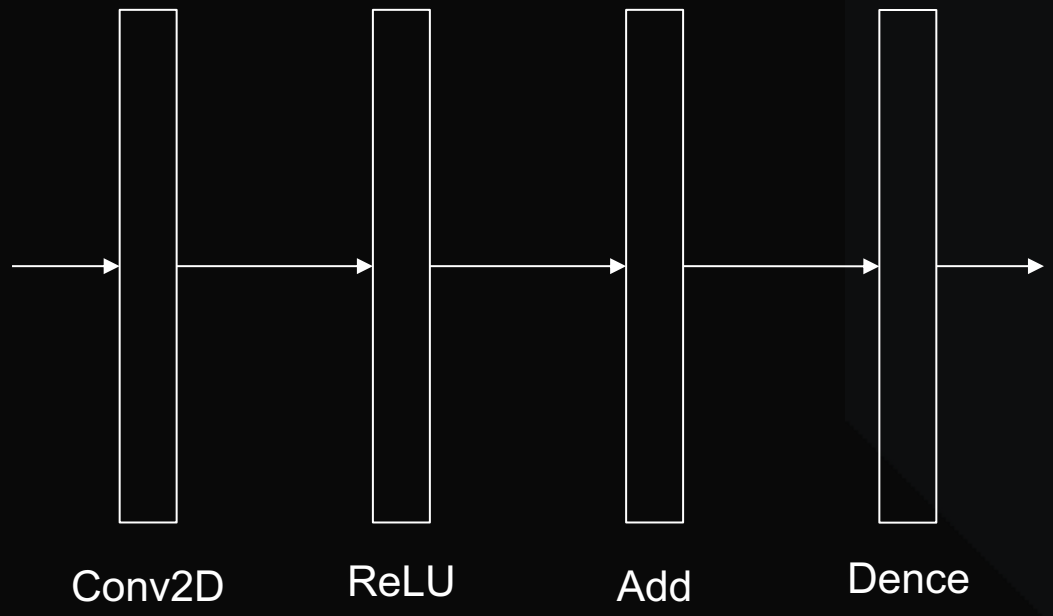
$$y_1 = \varphi\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

1つのニューロンへの入力 y を数式で表現
ニューラルネットワークの演算 (推論) 処理はこの繰り返し

LUMINOUS PRODUCTIONS

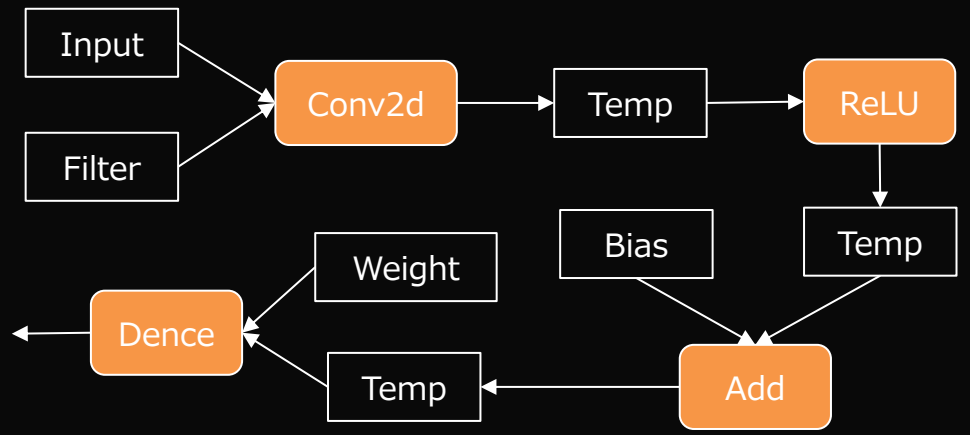
ニューラルネットワーク

ニューラルネットワークの構成図として良く見かける抽象的なレイヤー表現
これを少しだけ詳細が見える形にすると



ニューラルネットワーク

グラフの繋がりを表しており、入力に対して出力を返す関数になる
そして、入力した値が複数の関数を経由して出力層に到達する**合成関数**の構造



res = Dence(ReLU(Conv2d(a, f)) + b, w);

ニューラルネットワークを関数としてみると
実態のイメージがつきやすくなる

NVIDIA USING AI TO ACCELERATE YOUR GAME (PART 2)
https://devblogs.microsoft.com/directx/wp-content/uploads/sites/42/2018/03/WinML_acceleration_GDC.pdf

ニューラルネットワーク

「Neural Network」や「Deep Learning」は、未知の技術の印象が強かった

- 複雑で巨大なレイヤーのネットワークが、想像もつかない結果を生成するため
- GAN など画像生成を行うモデルは魔法としか思えない
- 何か特別な環境下で特殊なプログラミング言語や技法が必要？

実際、個々に構成する要素を紐解いていけば、関数の入出力の関係のみで構成されている事が分かる
多少の工夫は必要だが Compute Shader でも計算できる

Operator (関数) を通過する Tensor (データ) の流れを
意識すればニューラルネットワークは構築できる

LUMINOUS PRODUCTIONS

ニューラルネットワーク

仕組み自体は非常にシンプル
入力したデータが、複数の層 (関数) を経由して出力される合成関数の塊

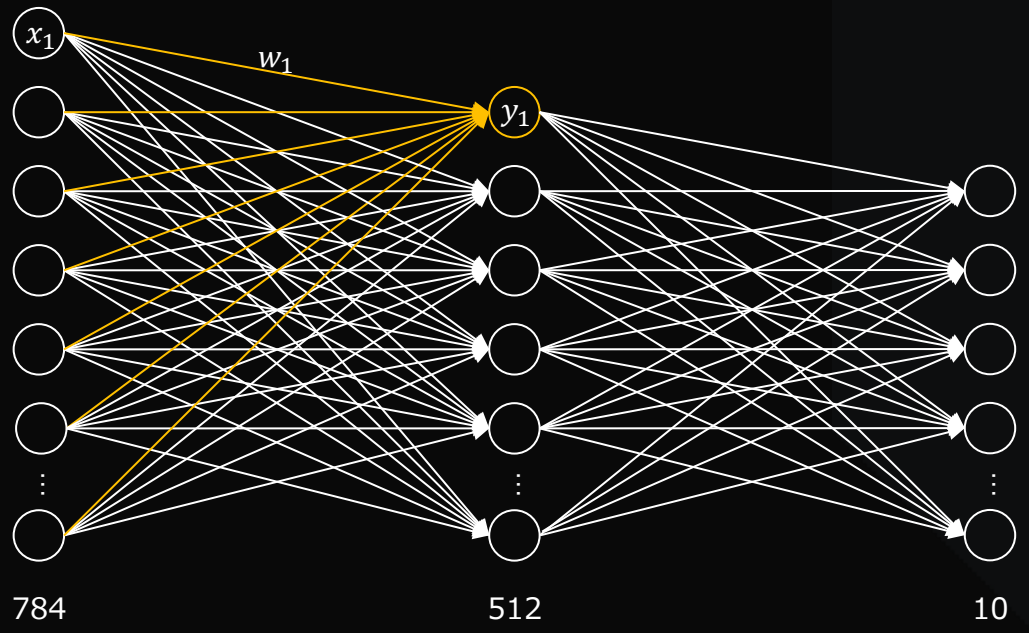
合成関数であるという考え方が、ニューラルネットワークでの学習を効率的に
　→ 連鎖律 (Chain Rule) という微分の性質を元にパラメータをチューニング

$$z = t^2 \qquad t = x + y$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$$

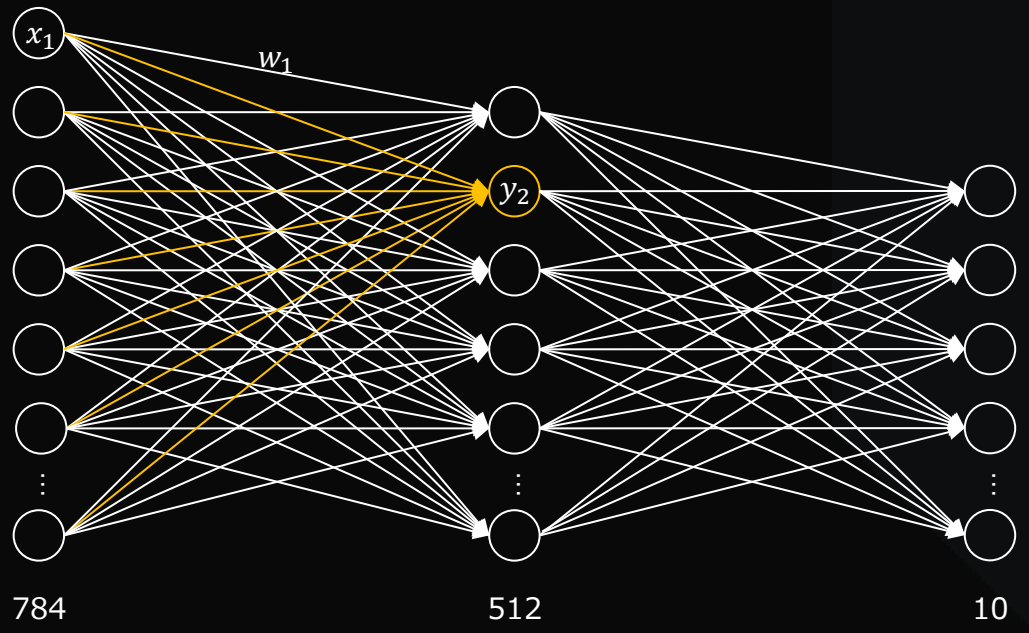
ニューラルネットワーク

ニューラルネットワークの基本的な構造として、全結合と呼ばれるノード間の接続がある
前の層のノードの全てが、次の層の一つのノードと接続している状態



ニューラルネットワーク

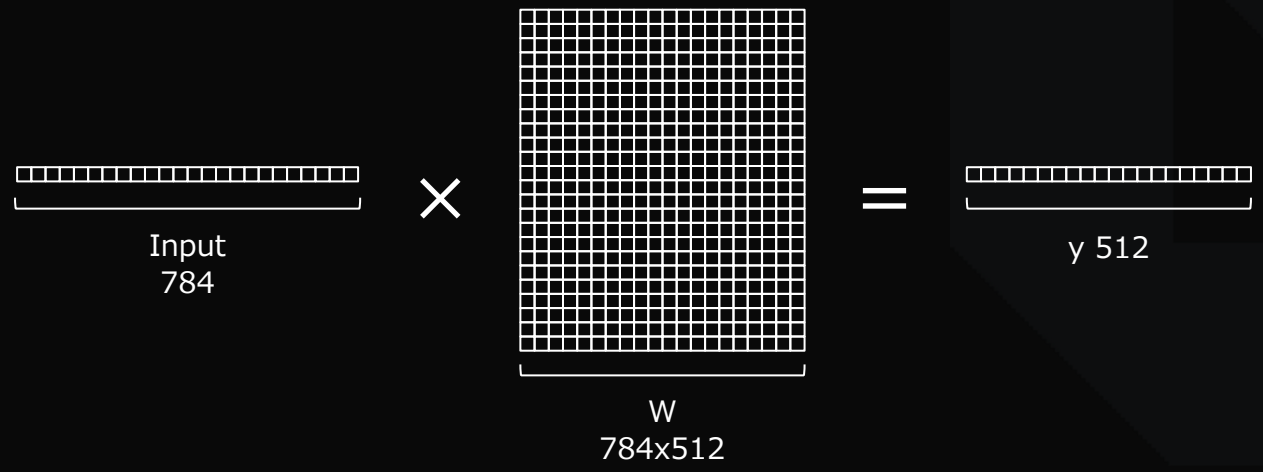
この複数の仮想的な接続と、接続間の入力と重みによって次のノードへの入力値を計算
全結合のネットワークであれば、全てのノードに対して下の計算を繰り返す



$$y_2 = \varphi\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

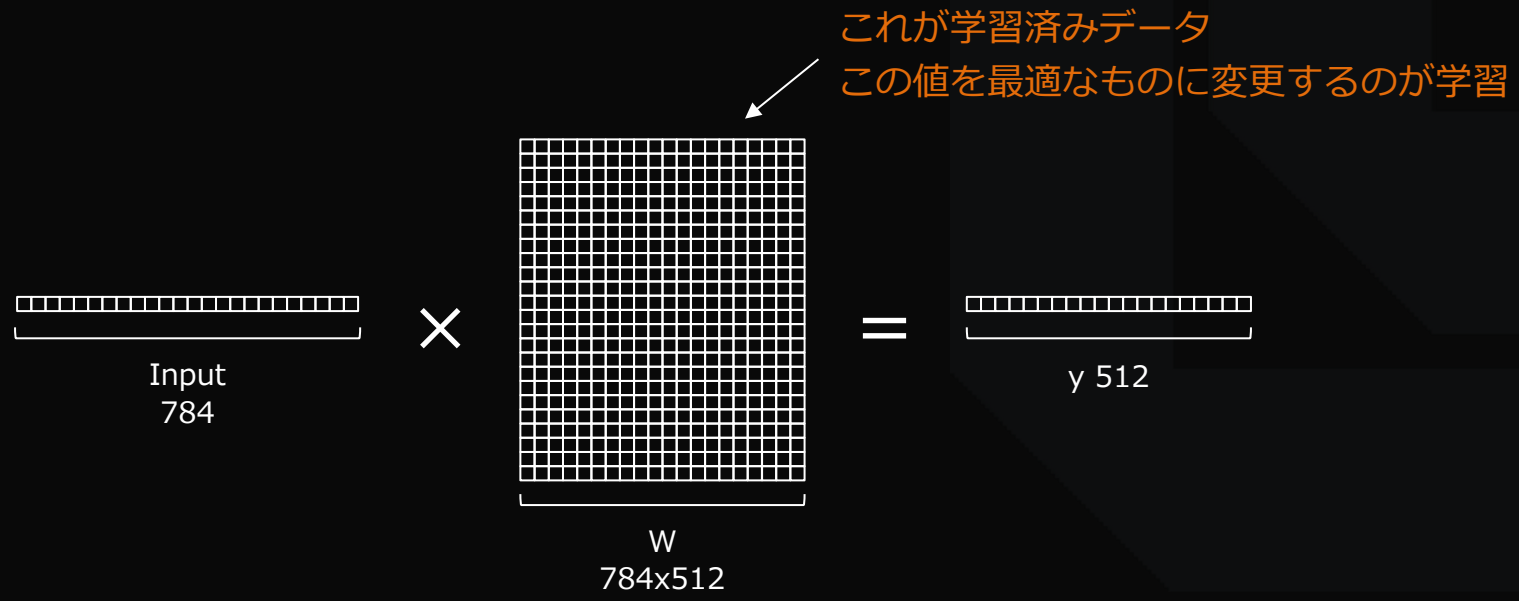
ニューラルネットワーク

NN の計算は単純な行列演算に落とし込める
例えば入力が784ベクトルあり、それぞれが重みを持ち512個のノードと接続している場合
(1, 784) x (784,512) = (1, 512) この計算で中間層のノードの値が求まる



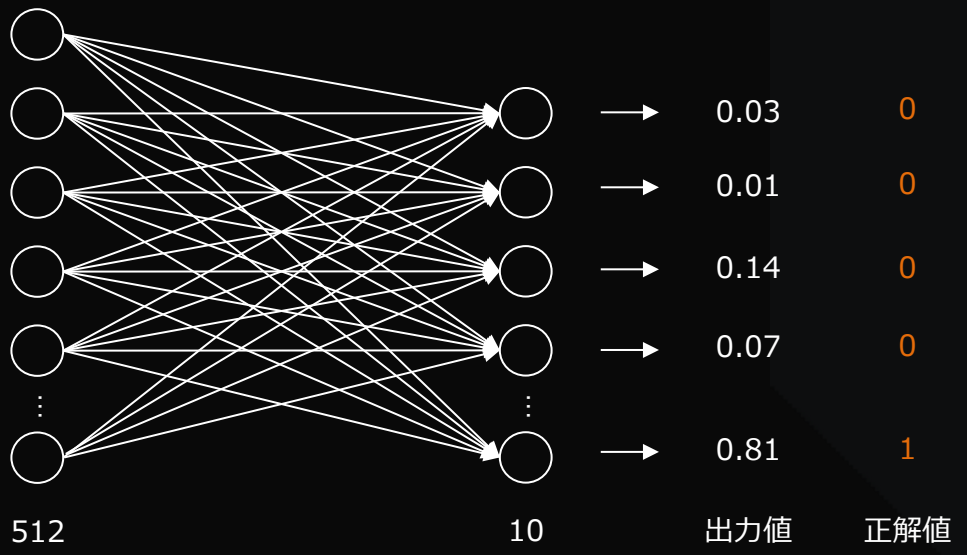
ニューラルネットワーク

“ニューラルネットワークの学習済み重みデータ” と言うと特別な印象を受けるが、
実態はネットワークを流れるノードと積(和)を取る実数データの塊
→ DirectML などでも単なる Tensor データとしてアプリ側で管理
ID3D12Resource で確保して *CreateCommittedResource()* で転送するバッファに過ぎない



ニューラルネットワーク

このアフィン変換を出力層まで繰り返し行うことで最終的な NN からの出力値が求まる
出力値と正解データ（教師データ）との差を計算して NN のパラメータを調整



LUMINOUS PRODUCTIONS

損失関数の定義

NN でのクラス分類のタスクでよく用いられるのは交差エントロピー誤差
→ ネットワークのパラメータを調整する指標

例えば、今回の講演内容のように画像処理的にピクセルを扱う場合、
教師データとなる画像との差が分かればいいため、平均二乗誤差を用いる

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

LUMINOUS PRODUCTIONS

Peak signal-to-noise ratio

損失関数と似ているが、画像の劣化具合を評価するための関数として PSNR を用いる
画像処理の分野で使われる事が多いため、損失関数とは別に分かりやすい参考値

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

$$PSNR = 10 * \log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

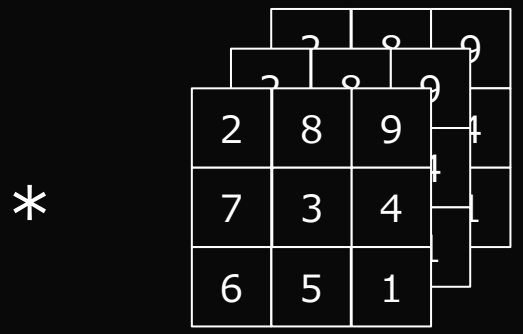
LUMINOUS PRODUCTIONS

Convolutional Neural Network

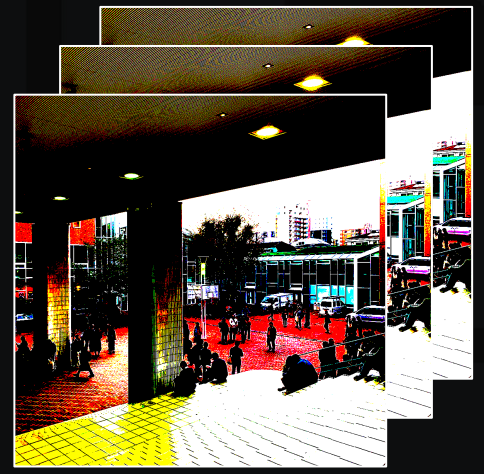
Convolutional Neural Network

ニューラルネットワークで画像処理・画像生成系のタスクを実行する場合は CNN が使われる
入出力として扱われる、特徴マップ、フィルター、畳み込みのパラメータで構成

Input



Filter

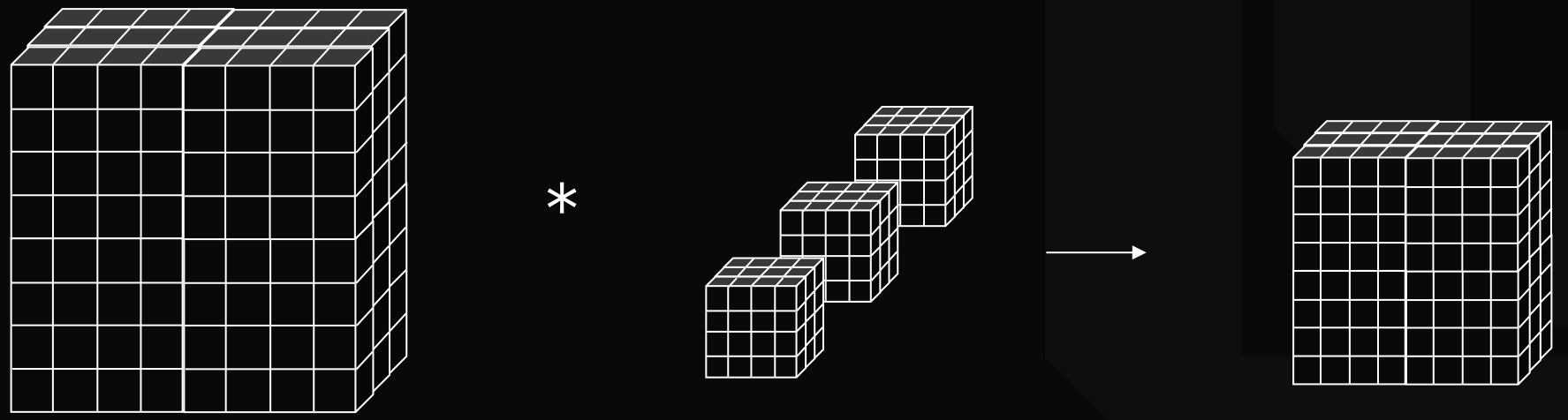


Feature Map

LUMINOUS PRODUCTIONS

Convolutional Neural Network

CNN の画像データの扱いは少し難しい。プログラムで処理する場合は、
下記のような3次元データの入力と3次元のフィルター(個数分) との計算で特徴マップが生成

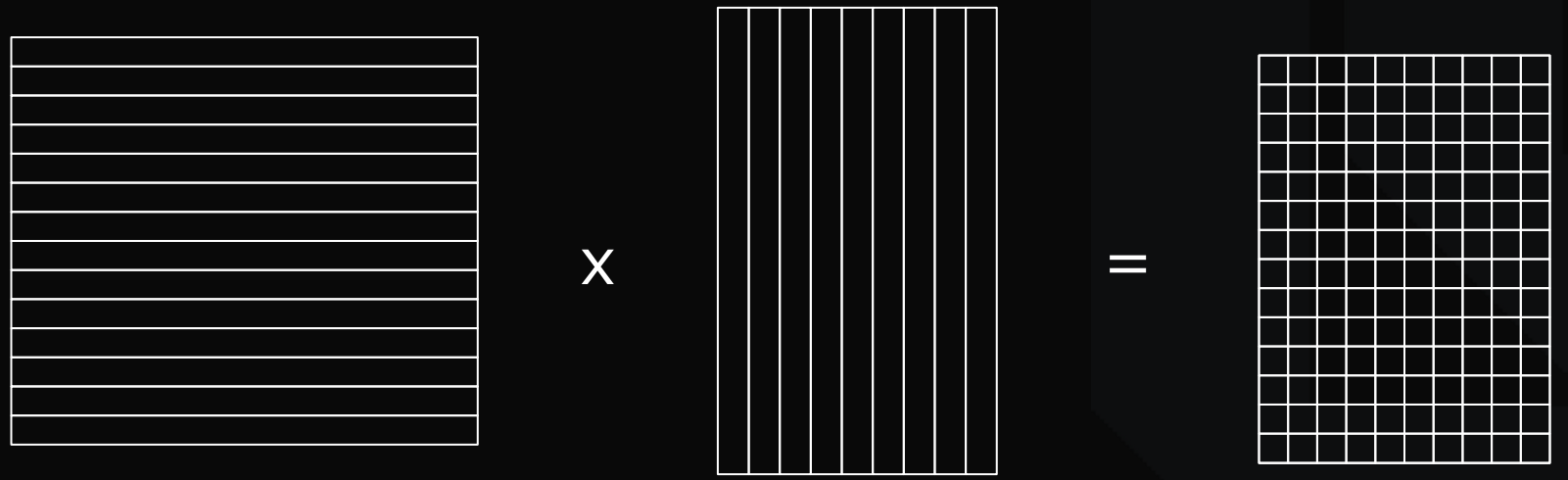


Input　　Filter　　Feature Map

LUMINOUS PRODUCTIONS

Convolutional Neural Network

CNN の計算は多次元の Input の値と Filter を 二次元の行列積として求める
そして再度 Feature Map に戻すという操作が入る



LUMINOUS PRODUCTIONS

シェーダー上へのニューラルネットワークの構築

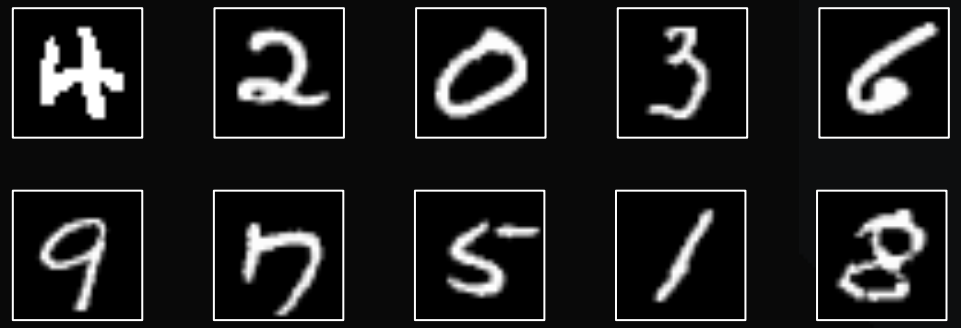
Build a neural network on compute shader.

LUMINOUS PRODUCTIONS

MNIST

画像データの 0-9 多値分類を Compute Shader 上に実装を行う
シェーダー上で分類問題を解くことに意味は無いが、最初のステップとして実装

→ 60000個の Training Data と 10000個の Validation Data データ

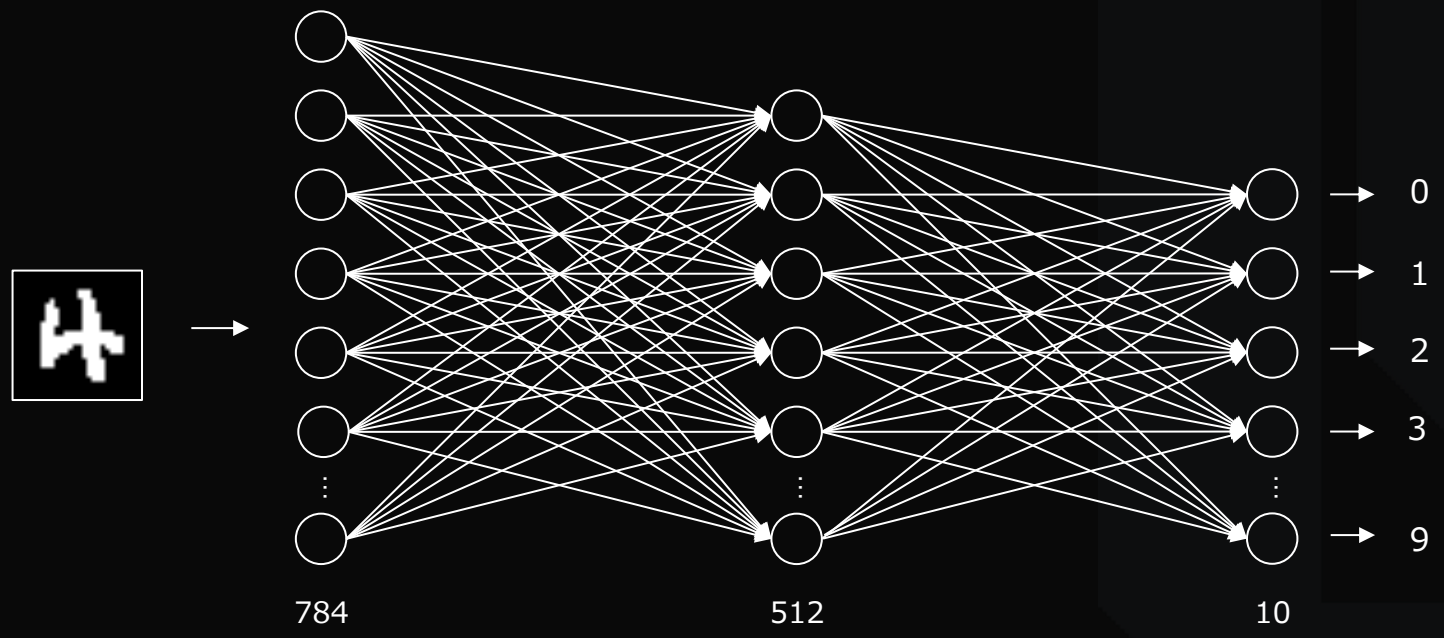


↑分かりやすいように拡大していますが
1枚 28x28px (784次元) の小さなデータです

LUMINOUS PRODUCTIONS

MNIST

ニューラルネットワークでの画像分類

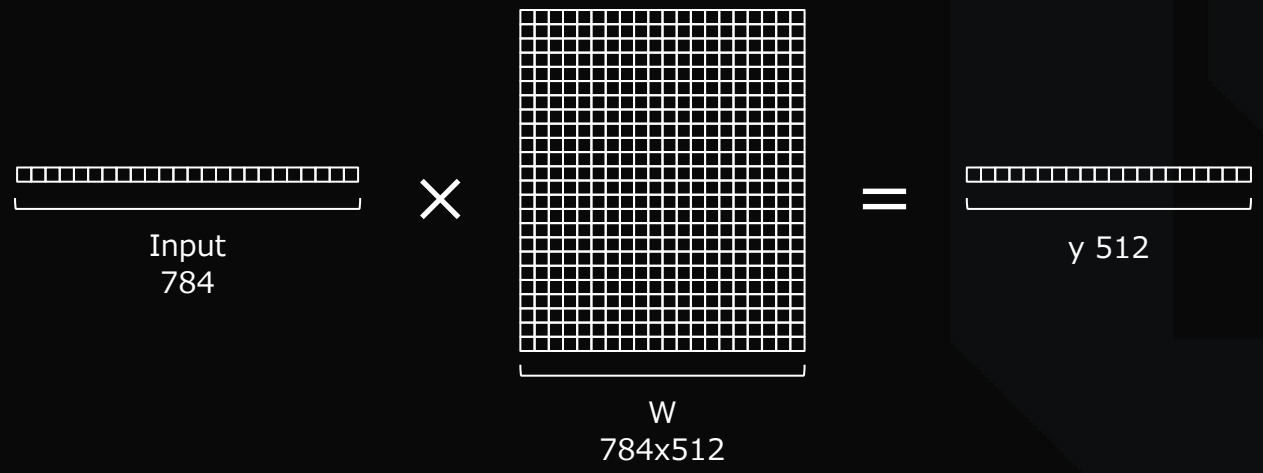


画像の 1px がニューラルネットワークのニューロンの一つの入力とする。
784 次元のベクトルが各ニューロンに入力として対応し、出力として 10 ノード存在

LUMINOUS PRODUCTIONS

MNIST

このノードの接続は実際に行われる計算は、先ほど見たこの行列の掛け算



Tensorflow (Keras) での実装

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation=tf.nn.relu),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam’,
              loss='sparse_categorical_crossentropy‘,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Get Started with TensorFlow
https://www.tensorflow.org/tutorials

Tensorflow (Keras) での実装

```
import tensorflow as tf
```
ライブラリインポート

```
mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```
データセットの準備

```
model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation=tf.nn.relu),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```
ネットワークのレイヤー定義

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```
学習方法の設定

```
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```
学習の実行と評価

Get Started with TensorFlow
https://www.tensorflow.org/tutorials

LUMINOUS PRODUCTIONS

Keras コード

ネットワークのレイヤー定義

- Input は 784 (28x28) 次元のベクトル
- 512 個の中間層を持つレイヤーとの全結合 (1個の入力が 512 個と結合)
- 20% のノードを推論精度の向上のためにドロップアウトさせる
- Softmax という関数を通して10個のニューロンから確率として出力

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

LUMINOUS PRODUCTIONS

Keras コード

ネットワークの学習と検証

- 最適化関数は Adam
- 損失関数は Cross Entropy 誤差
- 60000個の Training データを 5 epochs 繰り返す
- 学習が終わった後に evaluate() で現在の認識精度を確認

```
model.compile(
    optimizer='adam‘,
    loss='sparse_categorical_crossentropy‘,
    metrics=['accuracy'])

    model.fit(x_train, y_train, epochs=5)
    model.evaluate(x_test, y_test)
```

LUMINOUS PRODUCTIONS

Tensorflow (Keras) での実装

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation=tf.nn.relu),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam’,
              loss='sparse_categorical_crossentropy‘,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Get Started with TensorFlow
https://www.tensorflow.org/tutorials

LUMINOUS PRODUCTIONS

MNISTを解くネットワーク

この Keras コードを実行すると、数字画像の分類が 97.8% の精度 (正解率) が出る

→ 同等の精度を実現するアルゴリズムを機械学習無しに実装しようとすると困難

```
Epoch 1/5
60000/60000 [==============================] - 5s 80us/sample - loss: 0.2186 - acc: 0.9355
Epoch 2/5
60000/60000 [==============================] - 4s 73us/sample - loss: 0.0970 - acc: 0.9703
Epoch 3/5
60000/60000 [==============================] - 5s 75us/sample - loss: 0.0698 - acc: 0.9779
Epoch 4/5
60000/60000 [==============================] - 4s 74us/sample - loss: 0.0545 - acc: 0.9821
Epoch 5/5
60000/60000 [==============================] - 5s 87us/sample - loss: 0.0424 - acc: 0.9858
10000/10000 [==============================] - 0s 44us/sample - loss: 0.0697 - acc: 0.9784

[0.06973317598235444, 0.9784]
```

acc: 0.978 (97.8%の正解率)

LUMINOUS PRODUCTIONS

NN on Compute Shader

LUMINOUS PRODUCTIONS

Compute Shader

Compute Shader にタスクを持ってくる際に必要なのは、何をどの粒度で計算するか
一度に実行できるのは、一つのメイン関数 (エントリーポイント) のみ

- DirectML のように個々の演算をシェーダーメイン関数として定義して
  D3D12 上で組み合わせて計算する流れ
- 一度の Dispatch で一つのシェーダーメイン関数内で全て完結してしまうか

LUMINOUS PRODUCTIONS

Compute Shader

基本的には DirectML のように個々の演算に処理を分割するのが良い
演算ごとに Input/Output リソースを設定して Dispatch

- 大規模なネットワークの場合、巨大なシェーダーになるとデバッグが困難
- 特に CNN 実装は処理と扱うデータが巨大

個々の演算子は、適切な粒度で分割することをお勧め
・ReLU など活性化関数単位は細かすぎて管理が大変
・例えば CNN + Activation の粒度。Keras の Layer 実装が非常に参考になる

LUMINOUS PRODUCTIONS

Our Network

ただ、今回実装する MNIST は入力、中間層、出力層があるだけのシンプルなネットワーク
順伝播も逆伝播の機能も一つの CS ファイル内に落とし込む

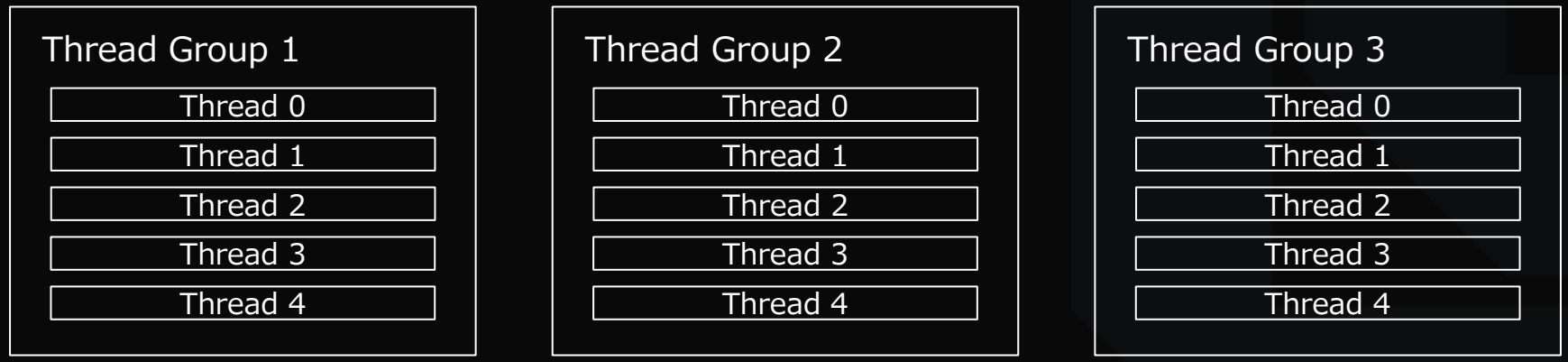
- **CSPredict.cs**
  バッチ対応の推論処理を実行。結果と学習に必要なデータを UAV に書き出す

- **CSTrain.cs**
  CSPredict での結果を元に学習処理を行う。重み、バイアスの更新

これらの処理を Compute Shader に落とし込む

Compute Shader

DirectX の Compute Shader は Thread Group と Thread から構成
CommandList::Dispatch(x, y, z) 命令で Thread Group 数を指定し、
Thread Group 内で起動する Thread 数は Compute Shader のメイン関数に指定



一度の Dispatch 呼び出しで、起動する Compute Shader のプログラムは全て同じ

Compute Shader

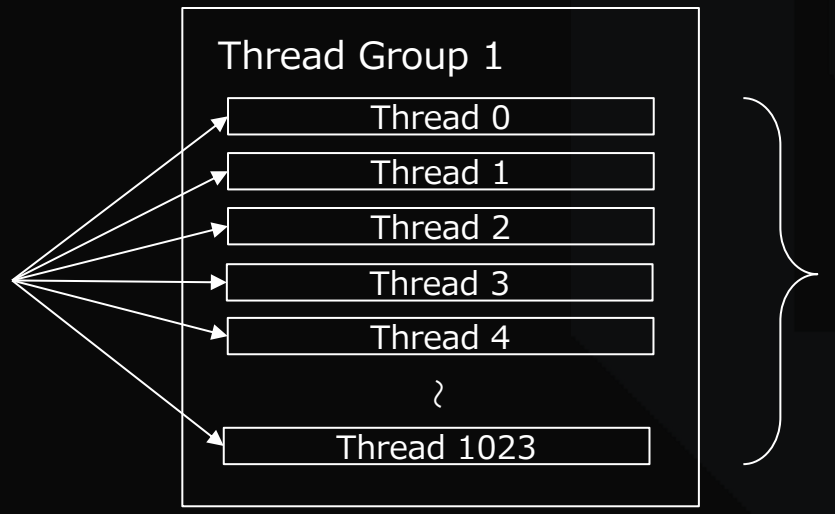
1 Thread Group 内で起動できる Thread 数の上限は 1024
1024 Threads は group shared という高速な共有メモリーにアクセスできる
今回、この 1024 Threads 内を使用して、データ1件分を処理する並列実行を考える

Input Data

```
[numthreads(32, 32, 1)]
void CSTrain() {

Texture2D<float> data = InputData[Gid.x];
OutputData[0] = data.Load(int3(0,0,0));

                ⋮
```



1 Thread Group 内では
最大 1024 Threads まで

LUMINOUS PRODUCTIONS

CSPredict

Predict

Compute Shader で以下の UAV を使うためのリソースを確保

```
Texture2D<float>    trainData[IMAGE_NUM]    : register(t0, space1);
ByteAddressBuffer   trainLabel              : register(t3);

RWStructuredBuffer  DebugOutputBuffer       : register(u0);
RWBuffer<float>     LayerOutput             : register(u1);

RWBuffer<float>     LayerInput              : register(u2);
RWBuffer<float>     LayerWeight             : register(u3);
RWBuffer<float>     LayerBias               : register(u4);
RWBuffer<float>     WorkingBuffer           : register(u5);



// 計算結果の保存用
groupshared float layer1Neuron[512];
groupshared float layer2Neuron[10];
```

Pseudo code

LUMINOUS PRODUCTIONS

Predict

入力としての784ニューロンを処理する
重みデータとの乗算行いバイアスを足して活性化関数を通す

```
if (GI < 512)
{
    float neuron1 = 0.0f;
    for (int i = 0; i < 784; i++) {
        float data = trainData[dataIndex].Load(int3((i % 28), (i / 28), 0));
        LayerInput[(Gid.x * 784) + i] = data;
        neuron1 += dot(data, LayerWeight[(i * 512) + GI]);
    }

    neuron1 += LayerBias[GI];

    float result = 0.0f;
    ActivationReLU(neuron1, result);

    layer1Neuron[GI] = result;
}

GroupMemoryBarrierWithGroupSync();
```

Pseudo code

Predict

中間層から出力層への出力を求める行列計算

```
if (GI < 10)
{
    float neuron2 = 0.0f;
    for (int i = 0; i < 512; i++) {
        float data = layer1Neuron[i];
        LayerInput[(Gid.x * 512) + i] = data;
        neuron2 += dot(data, LayerWeight[(i * 10) + GI]);
    }

    neuron2 += LayerBias[GI];
    layer2Neuron[GI] = neuron2;
}

GroupMemoryBarrierWithGroupSync();
```

Pseudo code

Predict

最後に Softmax 関数を通して確率に変換して、出力用のバッファに代入

```
if (GI == 0)
{
    float result[10];
    ActivationSoftmax(layer2Neuron, result);

    float oneHotLabel[10];

    [unroll]
    for (int i = 0; i < 10; ++i) {
        LayerOutput[(Gid.x * 10) + i] = result[i];
        oneHotLabel[i] = 0.0f;
    }

    int maxIdx = GetMaxIndex(result);

    DebugOutputBuffer[Gid.x].x = oneHotLabel[maxIdx];
    DebugOutputBuffer[Gid.x].y = LossCrossEntropy(oneHotLabel, result);
}
```

Pseudo code

LUMINOUS PRODUCTIONS

CSTraining

Training

順伝播とは逆に進む。Predict での出力結果と正解値との誤差をネットワークで伝播

```
if (GI < 64)
{
    int dataIndex = idxArray[GI / 4][GI % 4];
    uint label = trainLabel.Load(dataIndex * 4);

    oneHotLabel[label] = 1.0f;

    for (int j = 0; j < 10; ++j) {
        output[GI][j] = (LayerOutput[(GI * 10) + j] - oneHotLabel[j]) / 64.0f;
    }

    oneHotLabel[label] = 0.0f;
}
```

Pseudo code

Training

左がアフィン変換の逆伝播コードで、右が ReLU の逆伝播コード
出力層からの勾配に、重みデータの転置行列を掛けることで入力層からの勾配を計算

```
if (GI < 64)
{
    for (int j = 0; j < 512; ++j) {
        float dc = 0.0f;
        for (int k = 0; k < 10; ++k) {
            dc += output[GI][k] * LayerWeight[(j * 10) + k];
        }
        WorkingBuffer[(GI * 512) + j] = dc;
    }
}
```

```
// ReLU
if (GI < 64)
{
    for (int j = 0; j < 512; ++j) {
        if (LayerInput[(GI * 512) + j] <= 0) {
            WorkingBuffer[(GI * 512) + j] = 0.0f;
        }
    }
}
```

Pseudo code

$$\frac{\partial \mathrm{L}}{\partial \mathrm{X}} = \frac{\partial L}{\partial Y} W^t$$

Training

左が重みのパラメータの更新で、右がバイアスの更新
重みのパラメータの更新は、入力の転置と出力層からの勾配の行列の積を計算
バイアスの場合はそのまま出力層からの勾配でパラメータを更新

```
if (GI < 512)
{
    for (int j = 0; j < 10; ++j) {
        float dc = 0.0f;
        for (int k = 0; k < 64; ++k) {
            dc += LayerInput[k * 512 + GI] * output[k][j];
        }
        LayerWeight[(GI * 10) + j] -= lerningLate * dc;
    }
}
```

```
if (GI < 10)
{
    float db = 0.0f;
    for (int j = 0; j < 64; ++j) {
        db += output[j][GI];
    }
    LayerBias[GI] -= lerningLate * db;
}
```

Pseudo code

$$\frac{\partial \mathrm{L}}{\partial W} = X^t \frac{\partial L}{\partial Y}$$

LUMINOUS PRODUCTIONS

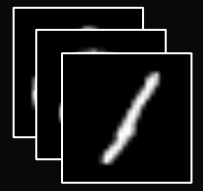
Result

LUMINOUS PRODUCTIONS

評価とトレーニング

ここまでで作成されたニューラルネットワークの評価とトレーニングを行う
MNIST データセットの 60000個の画像で学習し、最後に 10000個のデータを用いて精度を確認

学習データ



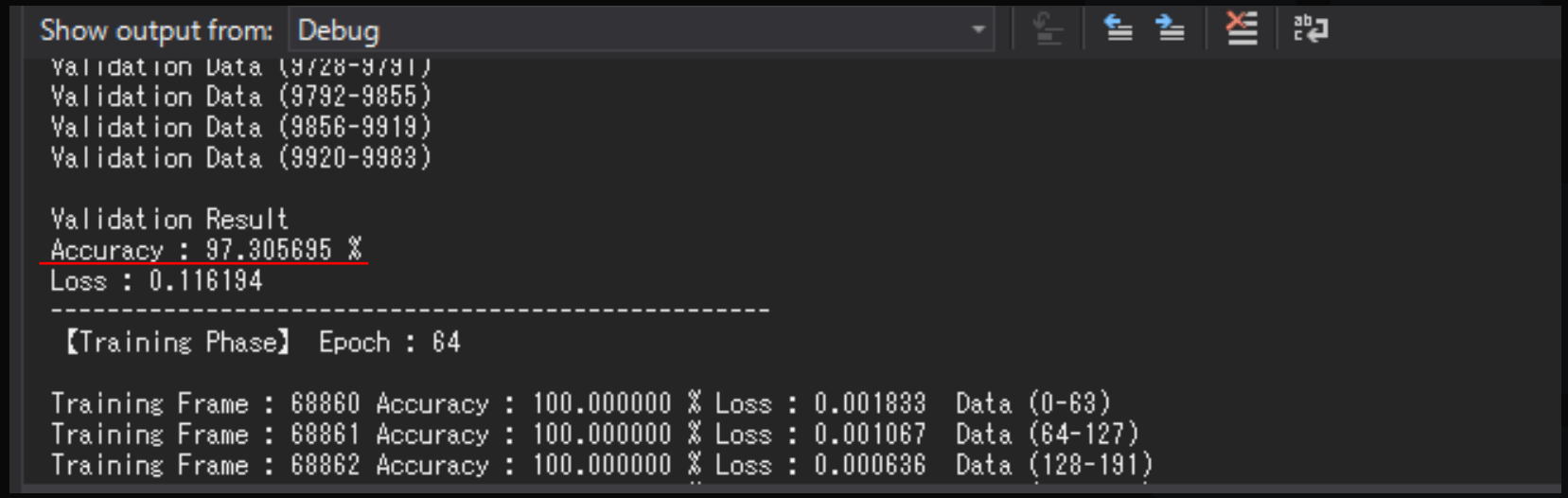
検証データ

学習用データと検証用データを繰り返し入力を行い、学習を進め
損失関数が 0 に近づいているか、Loss と Val Loss の低下を確認する

今回、Compute Shader の中で 64 Epoch ほど回した結果

Result

正解率 97.3% を記録。
Keras での実行結果 97.8% に近い数字。CS 実装でも既存 NN フレームワークと同じ処理が可能



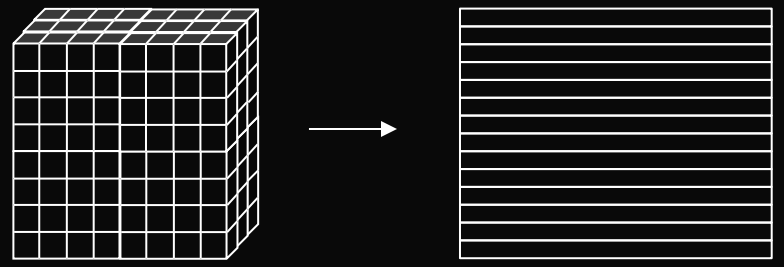
0.5% の違いは、Keras は Adam だが、今回でのテストでは SGD を使用したため

LUMINOUS PRODUCTIONS

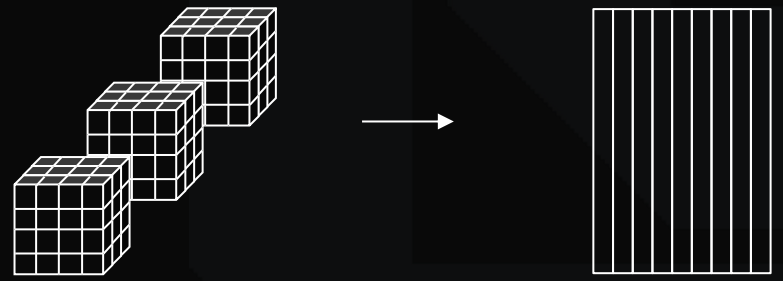
Convolutional Neural Network

Convolutional Neural Network

Convolutional Layer は入力データと Filter を行列として展開して行列積を求める
単純な行列積の後はバイアスを通して活性化関数を通すだけで出力が求まる



入力データの行列への展開



Filter データの行列への展開

Convolutional Neural Network

Compute Shader 上での CNN 実装例

```
{
        int writeSize = IM2COL_OUTPUT_SIZE(fs.x, fs.y, ch, outFilter.x, outFilter.y );
        int writeStartPos = AddWorkBuffer(BUFFER_CNN_0_MNIST_TO_COL, writeSize);

        IM2COL_MNIST_INPUT(trainData[0], 0, writeStartPos, insize.x, insize.y, ch, fs.x, fs.y, pad.x, pad.y, st.x, st.y);

        int weightStart, biasStart, keepInputStart;
        GetWeightBiasKeepInputOffset(MNIST_CNN_LAYER_0, weightStart, biasStart, keepInputStart);

        int filterStartPos = AddWorkBuffer(BUFFER_CNN_0_FILTER, fn * fs.x * fs.y * ch); // (32, 9) mat
        GET_FILTER_MATRX(filterStartPos, weightStart, fn, fs.x, fs.y, ch);

        AddWorkBuffer(BUFFER_CNN_0_RESULT_MUL_WEIGHT, 676 * 32);

        MUL_AT_BT(  WorkingBuffer, GetStartPos(BUFFER_CNN_0_RESULT_MUL_WEIGHT), // Result
                    WorkingBuffer, GetStartPos(BUFFER_CNN_0_MNIST_TO_COL),  // im2col (9, 676) mat
                    WorkingBuffer, GetStartPos(BUFFER_CNN_0_FILTER), // Filter (32, 9) mat
                    9, 676, 32, 9);

        ADD_BIAS(WorkingBuffer, GetStartPos(BUFFER_CNN_0_RESULT_MUL_WEIGHT), 676, 32, biasStart);
        ACTIVATION_RELU(WorkingBuffer, GetStartPos(BUFFER_CNN_0_RESULT_MUL_WEIGHT), 676, 32);
        COPY_BUFFER(LayerKeepInputs, keepInputStart, WorkingBuffer, GetStartPos(BUFFER_CNN_0_), GetBufferSize(BUFFER_CNN_0_));
}
```

Pseudo code

LUMINOUS PRODUCTIONS

HLSL

シェーダーは任意のサイズの配列を扱う関数は作れない。固定サイズのみ
当たり前だがポインタも使えない

Stack / Queue や Tree Structure など、自分でオフセット計算でやれないことは無いが
無闇にレジスタを消費するような構造は実行速度を低下させる

→ 単純な関数化を超えて、再利用性を求めるコードはシェーダー上では合わない
GPU プログラミングと CPU プログラミングの思想の違い

HLSL

任意の配列の関数は、マクロの実装で代用

```
/*
    NOTE:
        C = A x B を計算します。計算結果は C (A row, B cul) になります。
        C は (A row, B cul) サイズの領域を確保しておく必要があります
        C(3,3) = A(3,5) * B(5,3)
*/
#define MUL_A_B(C, co, A, ao, B, bo, a_row, a_cul, b_row, b_cul)\
for (int i = 0; i < a_row; ++i) {\
    for (int j = 0; j < b_cul; ++j) {\
        float dc = 0.0;\
        for (int k = 0; k < a_cul; ++k) {\
            dc += A[ao+(i * a_cul + k)] * B[bo+(k * b_cul + j)];\
        }\
        C[co+(i * b_cul + j)] = dc;\
    }\
}\
```

Pseudo code

HLSL

Compute Shader 上で実装するための工夫

```
#define ADD_BIAS(A, ao, row, col, offset)\
    for (int i = 0; i < row; ++i) {\
        for (int j = 0; j < col; ++j) {\
            A[ao+(i * col + j)] += LayerBias[offset + j];\
        }\
    }\

#define ACTIVATION_RELU(A, ao, row, col)\
    for (int i = 0; i < row; ++i) {\
        for (int j = 0; j < col; ++j) {\
            A[ao+(i * col + j)] = max(0.0f, A[ao+(i * col + j)]);\
        }\
    }\

#define COPY_BUFFER(DEST, destOffset, SRC, srcOffset, copySize)\
    for (int i = 0; i < copySize; ++i) {\
        DEST[destOffset+(i)] = SRC[srcOffset+(i)];\
    }\
```

Pseudo code

LUMINOUS PRODUCTIONS

Weight data

LUMINOUS PRODUCTIONS

Weight Data

ニューラルネットワークの重みデータは単純な Raw Data として管理
　→ float32 で 10個の重みデータがあればそのまま 40byte のファイル

Weight, Bias データは Compute Shader 上では UAV (RWBuffer) として読み書き
(推論時のみであれば SRV として扱う)

学習時に更新された重みデータは GPU からリードバックしてそのままファイル出力
読み込み時も、ロードした重みデータをそのまま Upload Heap -> Default Heap

Weight Data

Weight, Bias バッファと一緒に D3D12 の一般的な Read Back 用のバッファを定義
アプリ終了時に CopyResource して内容を取得

```
m_device->CreateCommittedResource( &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_READBACK),
                                   D3D12_HEAP_FLAG_NONE,
                                   &CD3DX12_RESOURCE_DESC::Buffer(bufferSize),
                                   D3D12_RESOURCE_STATE_COPY_DEST,
                                   nullptr,
                                   IID_PPV_ARGS(&readBackBuffer));
```

```
m_commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition( srcbuffer.Get(),
                                  D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
                                  D3D12_RESOURCE_STATE_COPY_SOURCE));

m_commandList->CopyResource(destbuffer.Get(), srcbuffer.Get());
```

LUMINOUS PRODUCTIONS

Weight Data

Map, Unmap して CPU から読み出し、標準関数でそのままディスク出力

```
CD3DX12_RANGE readRange(0, 0);
float* mappedData = nullptr;
ThrowIfFailed(readBackbuffer->Map(0, &readRange, reinterpret_cast<void**>(&mappedData)));
{
    std::ofstream fout(outputBinName, std::ios::binary | std::ios::trunc);
    fout.write((const char*)& mappedData[0], sizeof(float) * elementNum);
    fout.close();
}
#if ENABLE_OUTPUT_WEIGHT_TEXT_MODE
{
    std::ofstream fout(outputTextName);
    for (int i = 0; i < elementNum; ++i) {
        fout << mappedData[i] << std::endl;
    }
    fout.close();
}
#endif
readBackbuffer->Unmap(0, nullptr);
```

LUMINOUS PRODUCTIONS

Weight Data

複数層の Weight, Bias データもまとめて一つのファイルで管理
ネットワークの構造が変わらなければ、Raw data のレイアウト位置も固定
　→ レイヤー情報から、自身の重みデータ (バイト) 数を計算して定数バッファに渡して参照

```
// 1層目 CNN Layer のパラメータ
{
    // 1層目は 0 から参照して良い
    m_constantBufferData.weightsStartOffset[MNIST_CNN_LAYER_0] = wnum * sizeof(float); // 0
    m_constantBufferData.biasStartOffset[MNIST_CNN_LAYER_0] = bnum * sizeof(float); // 0

    int filterNum = 32;
    int filterSize = 3;
    int channelNum = 1; // RGB なら 3

    // Convolution Layer のパラメータ数 (個数)
    int twnum = (channelNum * filterNum * filterSize * filterSize);
    int tbnum = filterNum;

    PrintDebugParamNum(L"CNN Layer0", twnum, tbnum);
}
```

LUMINOUS PRODUCTIONS

Working Buffer

LUMINOUS PRODUCTIONS

Working Buffer

シェーダー上で確保可能なレジスタ数は大きく制限がある
　→ 1 Thread 内で多くの一時変数を使用すると GPU 内の Thread 数が低下し速度低下を招く

動作環境にもよるがシェーダー内で確保可能な一時変数の上限は概ね 65536
そのため　`float temp[256][256];`　確保するとシェーダーで扱える変数の上限に達する

　→ 一時作業用に UAV でアクセスするバッファを定義

Working Buffer

計算用の一時的な Work Buffer として巨大なデバイスメモリー (UAV) を確保
- レンダーターゲットに対して CNN をやる場合など一時的に広い作業領域が必要
- 単体の D3D12Resource で Compute Shader から大容量バッファへのアクセスが可能

```
m_workingBufferSize = 1920 * 1080 * sizeof(float);
m_workingBufferSize *= 240; // Test 1,990,656,000 Byte ≒ 2.0GB
```

```
// Working Buffer UAV
{
    D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};
    uavDesc.Format = DXGI_FORMAT_R32_FLOAT;
    uavDesc.ViewDimension = D3D12_UAV_DIMENSION_BUFFER;
    uavDesc.Buffer.FirstElement = 0;
    uavDesc.Buffer.NumElements = m_workingBufferSize / sizeof(float);
    uavDesc.Buffer.StructureByteStride = 0;
    uavDesc.Buffer.CounterOffsetInBytes = 0;
    uavDesc.Buffer.Flags = D3D12_BUFFER_UAV_FLAG_NONE;
    md3dDevice->CreateUnorderedAccessView(m_workingBuffer.Get(), nullptr, &uavDesc, WorkingBufferUAV);
}
```

Optimize

Compute Shader 最適化

- Thread Group 起動単位と group shared へのアクセス
  遊んでいるThread を最小限に

- 4x4 以上の多次元配列の計算時
  4 要素ごとに float4 での dot を取るなど

- UAV への書き戻しを最小限に
  Dispatch を分割することにより、必然的に計算過程で UAV を複数経由するため

- パラメータを Texture 化して、Gather 命令でアクセス

LUMINOUS PRODUCTIONS

リアルタイムレンダリングへの機械学習の応用

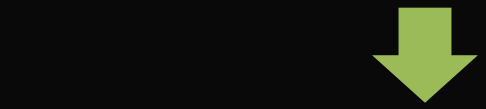
Applicate a machine learning to real-time rendering.

LUMINOUS PRODUCTIONS

リアルタイムレンダリング

次世代のリアルタイムレンダリングの処理は
レンダリングパイプライン上に構築された NN により描画品質の向上が行われる

その仮説を立てた時に、パイプライン上に NN を構築するとしたらどのような実装となるか？

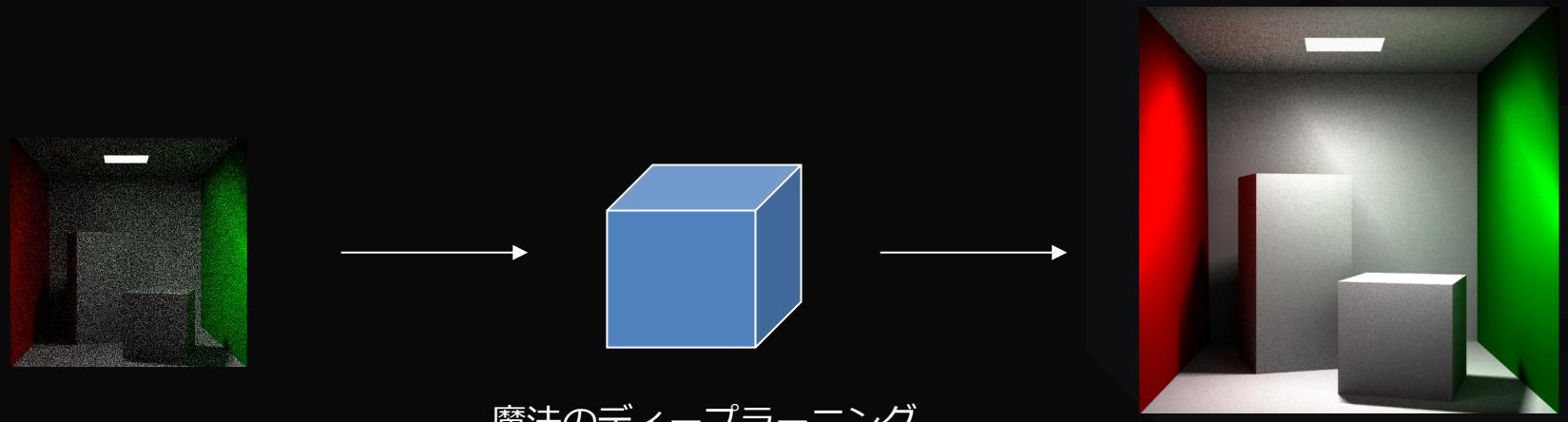


一つの巨大なディープラーニングで構築されたネットワークでピクセルの推定を行うか
あるいは、**個々の機能を持つネットワークの組み合わせ**によりピクセルの推定を行うか

LUMINOUS PRODUCTIONS

リアルタイムレンダリング

理想だけを言えば一つのニューラルネットワークが存在しており、
そこに画像を入力すると品質が向上した画像が出力されるという流れ



LUMINOUS PRODUCTIONS

リアルタイムレンダリング

教師データの写像先ピクセルだけを与えて学習させる
しかし汎化性能は無く、一つのネットワークでの実現は困難
　→ Denoising に使用するネットワークと Super-Resolution などでは構造が異なる

そのため、個々の機能を持ったネットワークの集合によりパイプラインの構築を考える
従来の Pixel, Compute で実装していたポストフィルターの延長

必要なポストフィルターの処理を追加していくように、
必要なニューラルネットワークのフィルターをパイプラインに追加して性能品質を図る

リアルタイムレンダリング

Pseudo code だが、理想的にはこのように追加
複数のニューラルネットワーク上をレンダリング結果 (SRV) が流れていくイメージ

```
RenderingSystem::BuildNeuralNetwork()
{
    AddNeuralNetwork (new DenoisingNeuralNetwork());
    AddNeuralNetwork (new AntiAliasingNeuralNetwork());
    AddNeuralNetwork (new SuperSamplingNeuralNetwork());
    AddNeuralNetwork (new SuperResolutioNeuralNetworkN());
    AddNeuralNetwork (new StyleChangeNeuralNetworkN());
}

RenderingSystem::DispatchNeuralNetwork()
{
    DispatchDenoising (RenderTarget, outDenoiser);
    DispatchAntiAliasing (outDenoiser, outputAA));
    DispatchSuperSampling (outputAA, outSS));
    DispatchSuperResolutio (outSS, outSR));
    DispatchStyleChange (outSR, outSC));
    DrawRect (outSC, RenderTarget);
}
```

必要な処理を追加

Pseudo code

LUMINOUS PRODUCTIONS

リアルタイムレンダリング

目的によりネットワークを細分化した方が作りやすく学習も進みやすい
巨大なニューラルネットワークは構造が変化すれば、学習は一からやり直しが必要なため

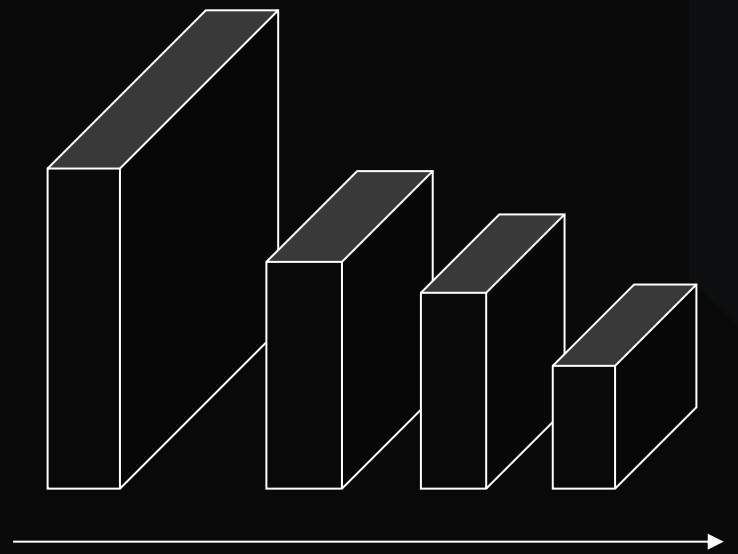
新しい手法が出てきた時も、そのネットワーク部分だけを差し替えられる
→ 新しい超解像度技術が出てきたら、その部分だけネットワーク・重みファイルの更新

LUMINOUS PRODUCTIONS

Encoder-Decoder

画像処理の分野で NN を用いる場合は、畳み込み処理が一般的に使用される
NN は Convolution Layer を多層に組み合わせて構成

・画像の特徴を抽出するためのフィルター
・レンダーターゲットはサイズが大きいため畳み込んで縮小

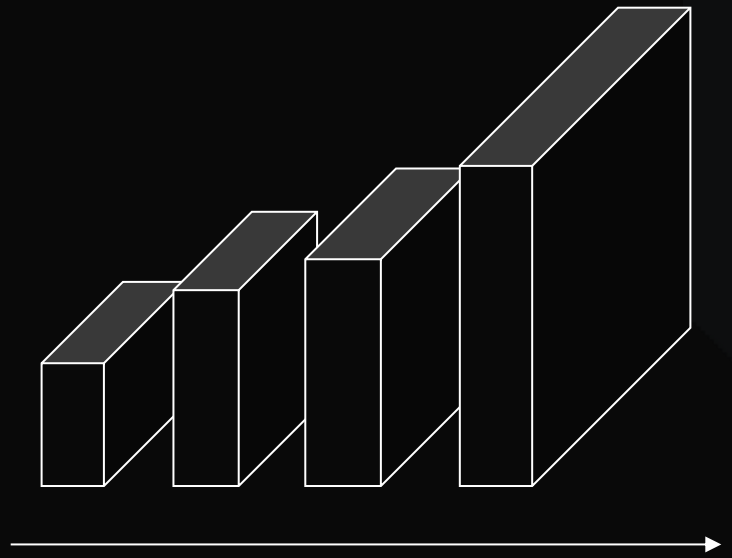


LUMINOUS PRODUCTIONS

Encoder-Decoder

通常の Convolution Layer と対になる形で
Transposed Convolution (Deconvolution) Layer と呼ばれる物が後段に配置
縮小した特徴マップの拡大が行われる

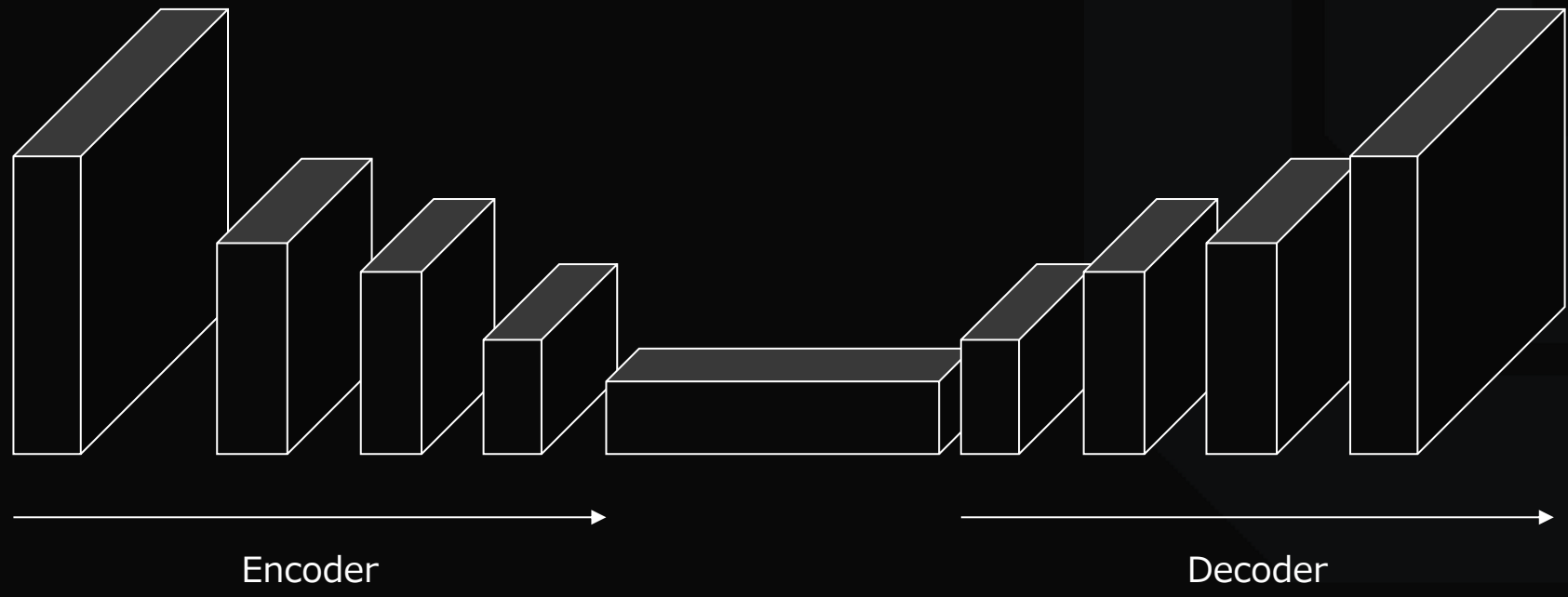
・モデルにもよるが、単純な Upsampling コピーで拡大する場合も



LUMINOUS PRODUCTIONS

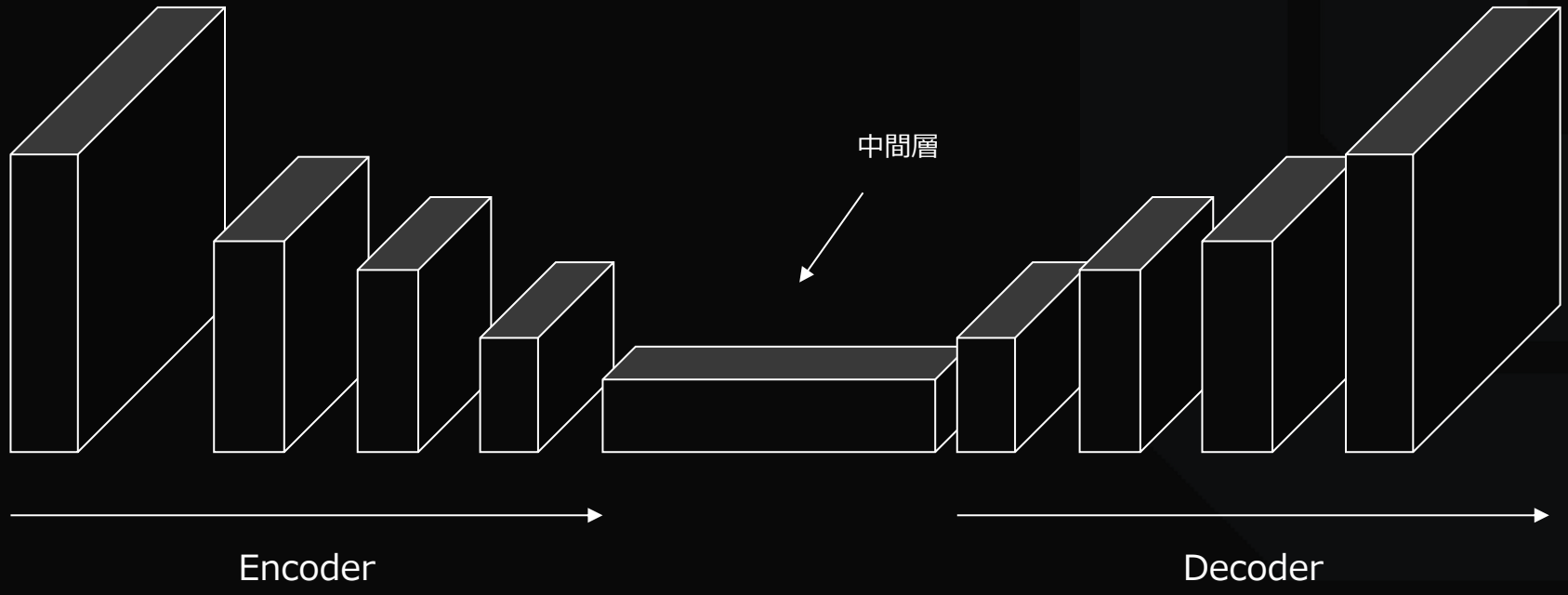
Encoder-Decoder

畳み込みを使って、画像処理のタスクとしてニューラルネットワークを用いる場合、
このような Encoder Decoder のネットワークとして構成されるものが多い



Encoder-Decoder

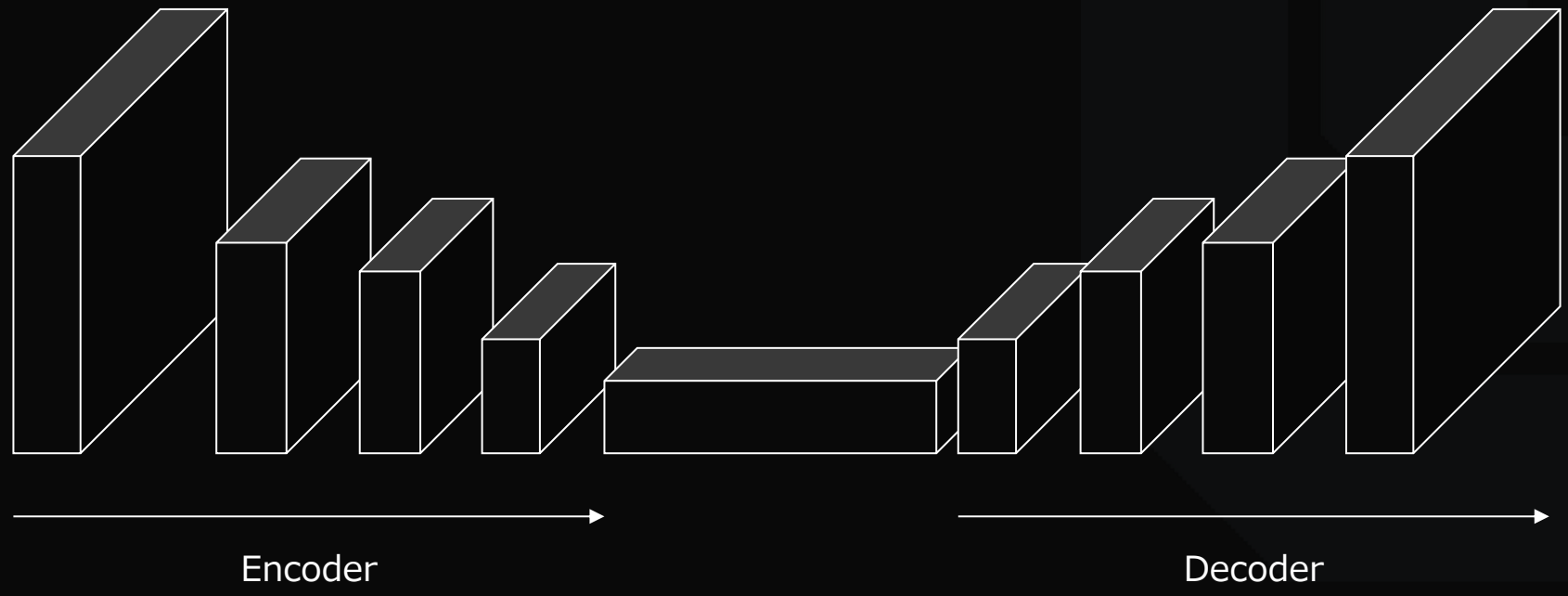
中間層部分に入力データの特徴を表す情報が溜まるように学習が行われ、
その特徴データを別の表現としてデコードする



LUMINOUS PRODUCTIONS

Encoder-Decoder

この仕組みで画像の自動着色やスタイル変換、画像生成などが行える事が分かってきたため
画像処理に置いてはこのような Encoder Decoder モデルが使用されることが多い



LUMINOUS PRODUCTIONS

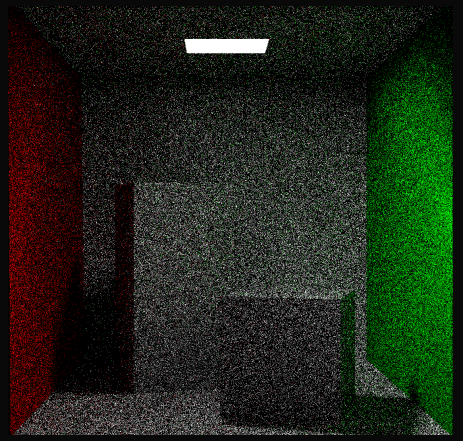
Denoising Auto Encoder

LUMINOUS PRODUCTIONS

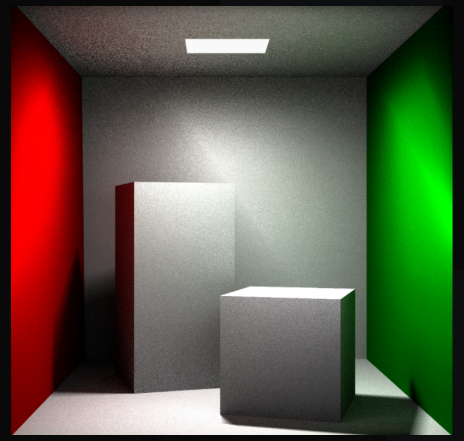
Denoiser

ノイズ除去のためのニューラルネットワークの必要性を考える

例えば Path Tracing で描画を行なった際はサンプリング数が品質に直結
しかし、実行速度とのトレードオフになるため簡単には増やせない場面も
→ 通常は少ないサンプリング数でデノイズを行う



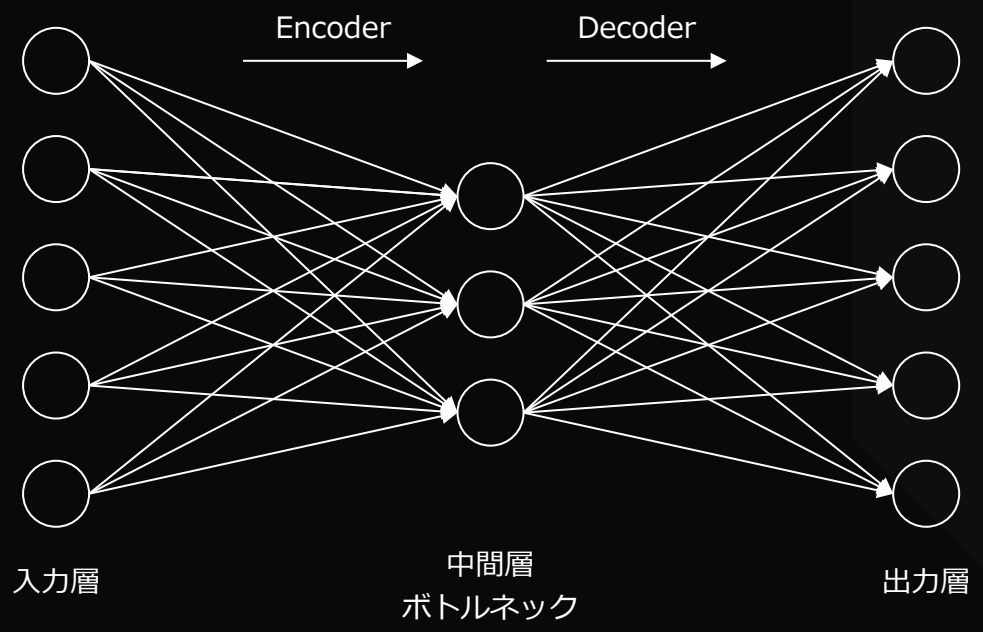
10spp



100spp

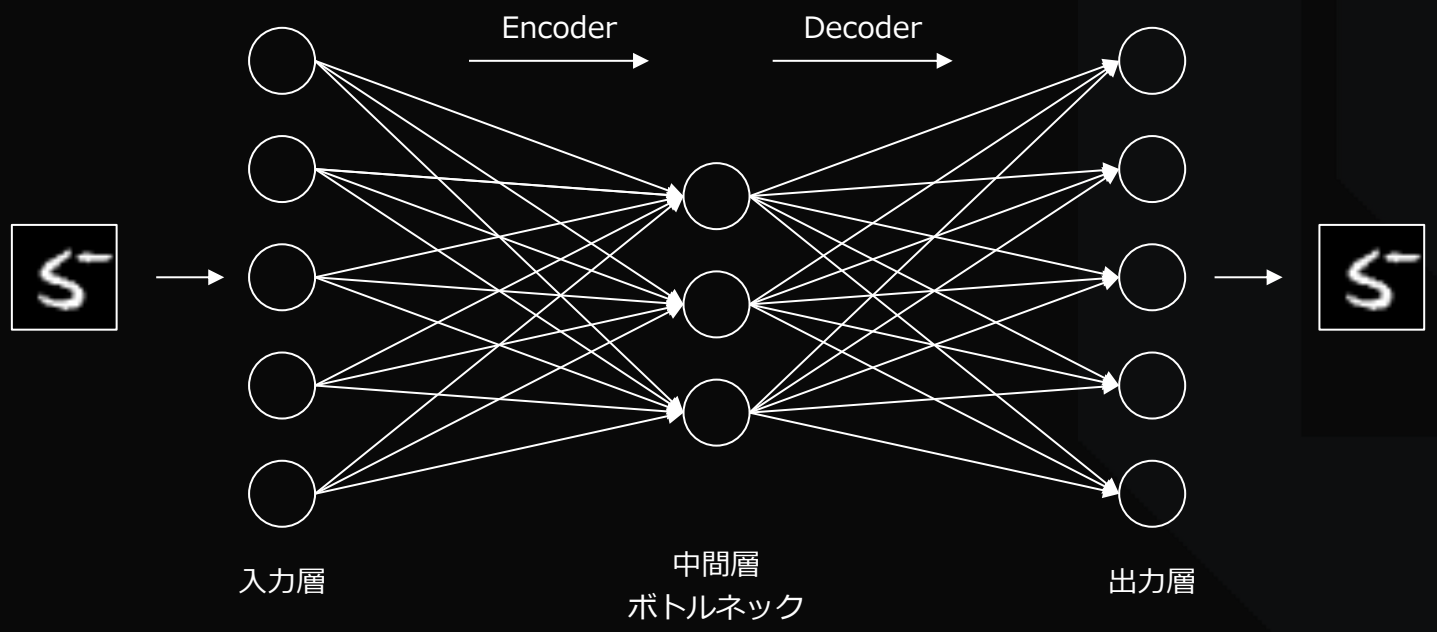
Auto Encoder

今回、ノイズ除去のためのネットワークとして Auto Encoder を用いる
Auto Encoder は入力層よりも次元の少ない中間層(ボトルネック)を持つネットワーク



Auto Encoder

特徴としては、入力と出力が同じになるようにネットワークを学習させる
中間層が少ないデータ(特徴)によって、元の入力を復号するようにパラメータを調整



Auto Encoder

この中間層のボトルネックが教師データを再現するための情報を保持

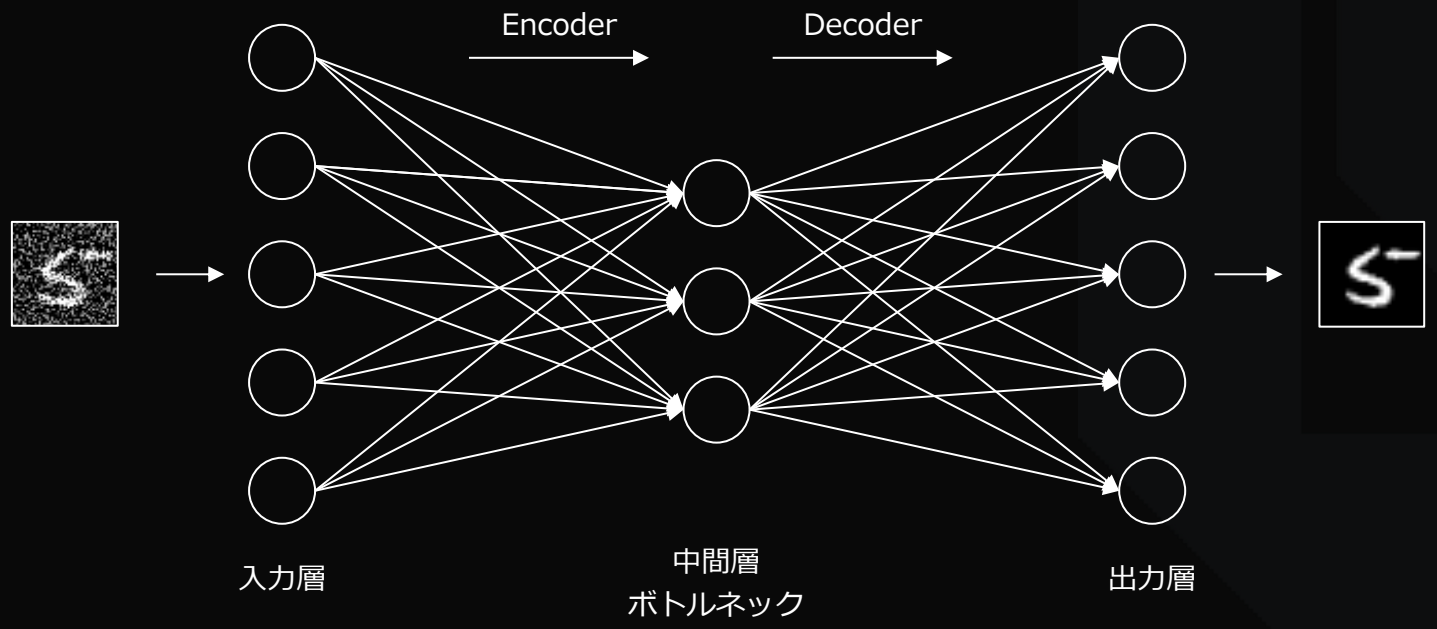
→ 潜在変数を学習により獲得

- Encoder が少ない特徴量からでも値を復元できるように符号化を行い
- Decoder が少ない特徴量からでも値を復元できるように復号化を行う

Latent variable

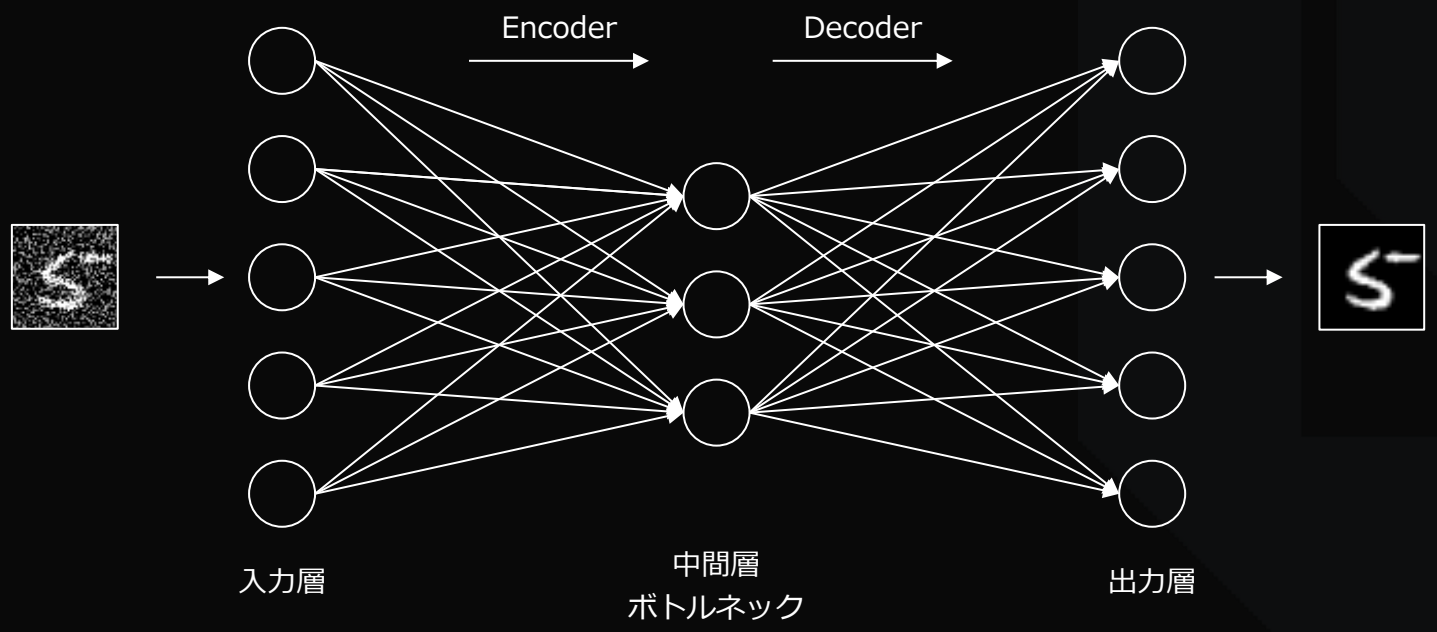
Denoising Auto Encoder

この Auto Encoder の仕組みに対して、ノイズが含まれる画像を入力し
ノイズが含まれない画像を教師データとして与える



Denoising Auto Encoder

Encoder がノイジーな画像の中から、
教師データへの復号に必要な情報のみを抽出するように学習。Decoder は元の画像に復号



Denoising Auto Encoder

このようなネットワークの事を Denoising Auto Encoder と呼ぶ

→ 今回は画像を対象として扱うが、画像以外の情報でも Denoising Auto Encoder は使われる

この仕組みによって、ネットワークに画像のノイズ除去の効果が生れる

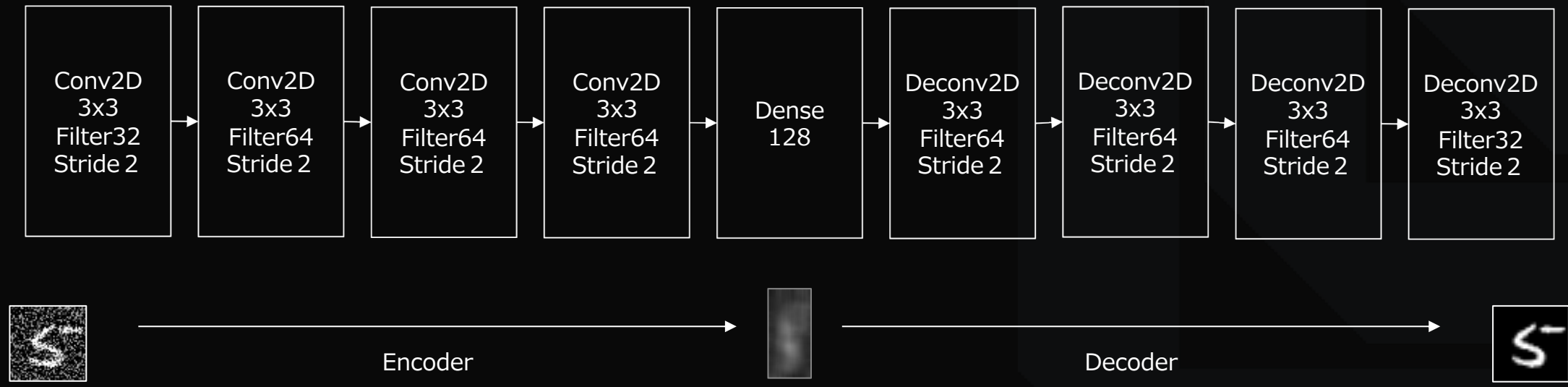
Latent variable

LUMINOUS PRODUCTIONS

Denoising Auto Encoder

ネットワークの構成

Convolution と Deconvolution を繰り返す構成

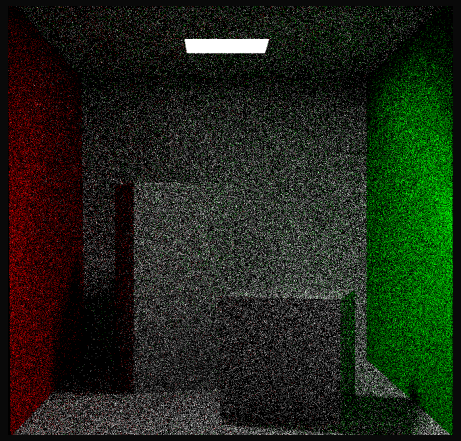


LUMINOUS PRODUCTIONS

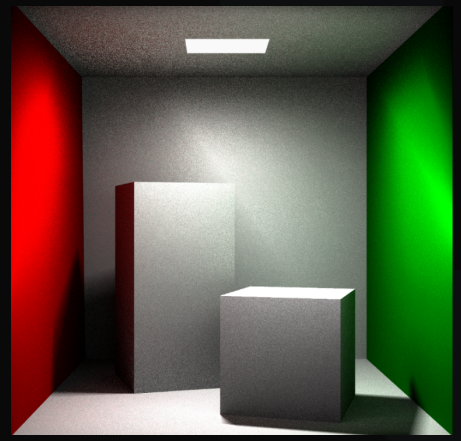
Denoising Auto Encoder

DXR で Path Tracing を構築し 100spp (samples per pixel) の結果を教師データとして Denoising Auto Encoder のネットワークを学習

入力データは 10spp の非常にノイジーなシーンを適用



10spp



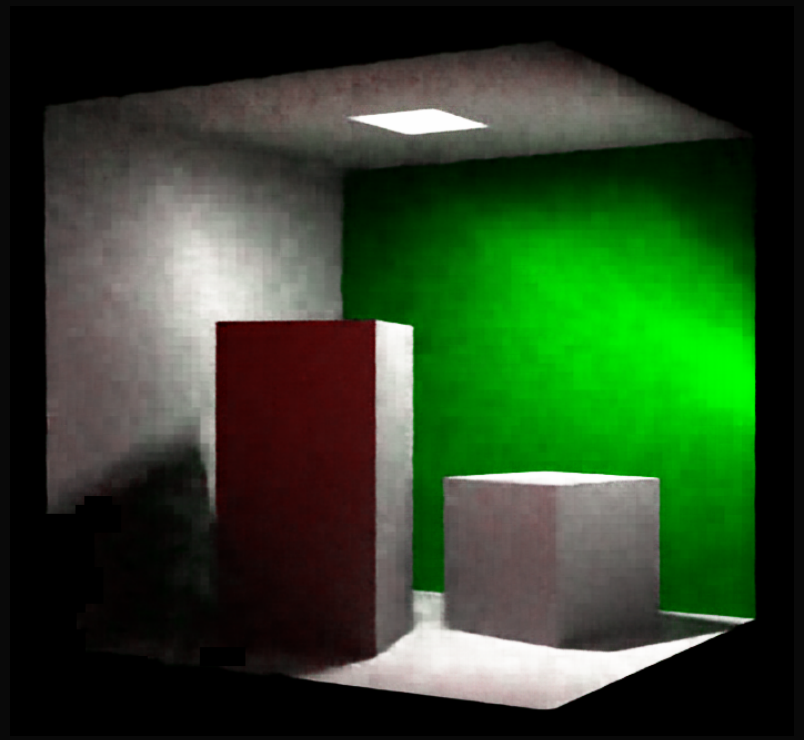
100spp

LUMINOUS PRODUCTIONS

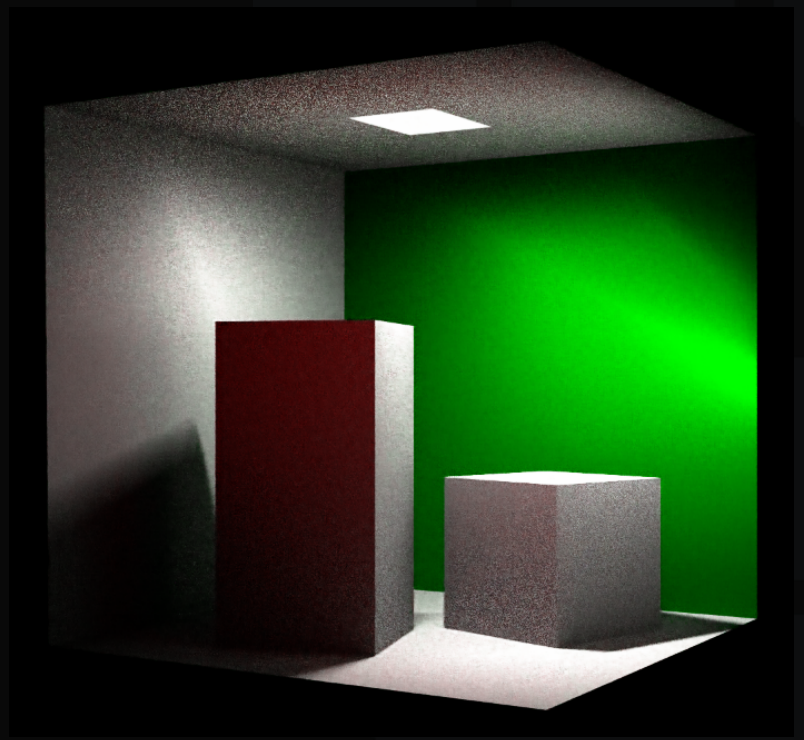
Denoising Auto Encoder

Epoch 数による比較

No generalization performance



50 Epoch
30.8 dB

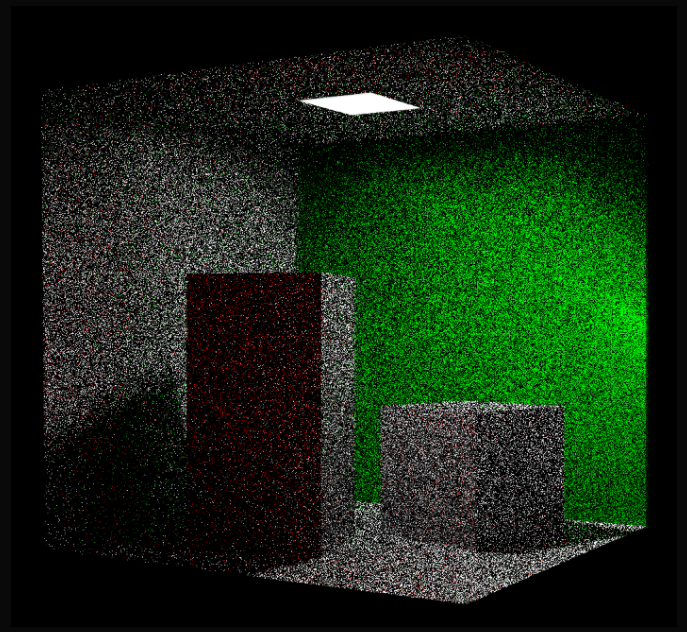


1000 Epoch
36.6 dB

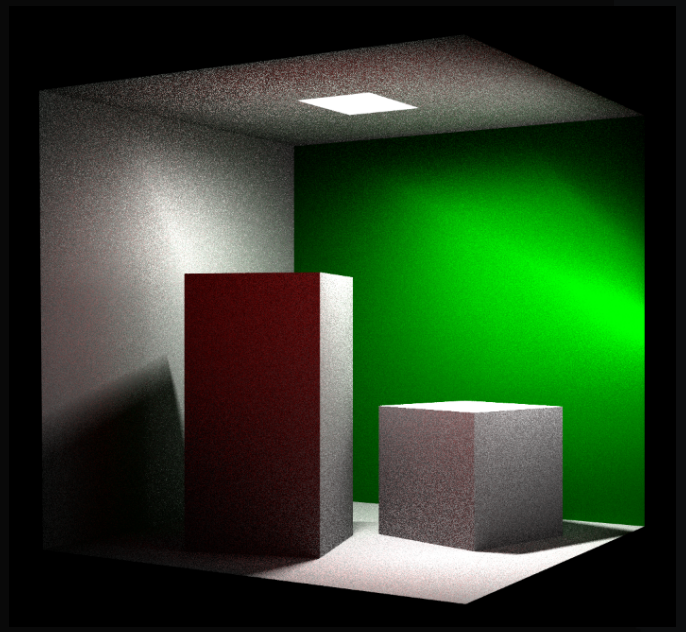
LUMINOUS PRODUCTIONS

Denoising Auto Encoder

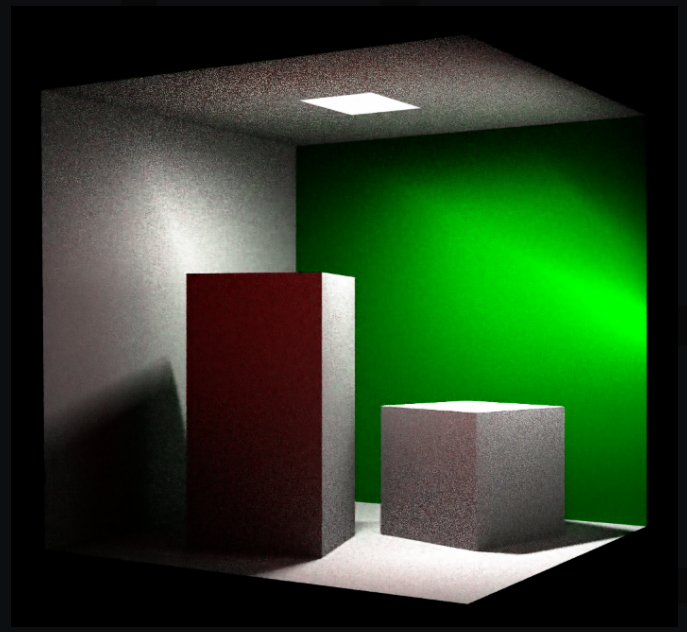
No generalization performance



Input Image
10spp



Ground Truth
100spp

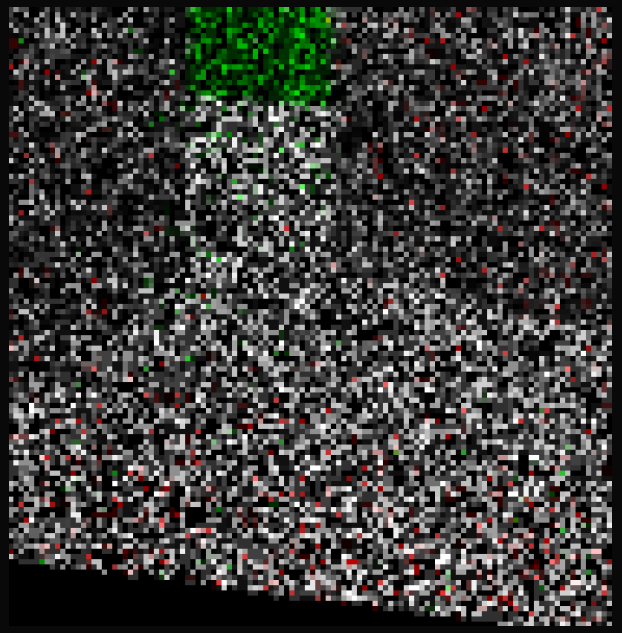


Denoising
Auto Encoder

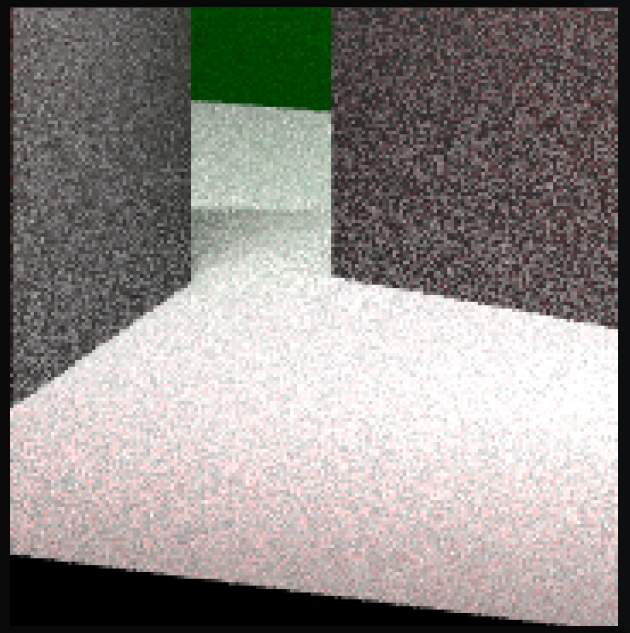
LUMINOUS PRODUCTIONS

Denoising Auto Encoder

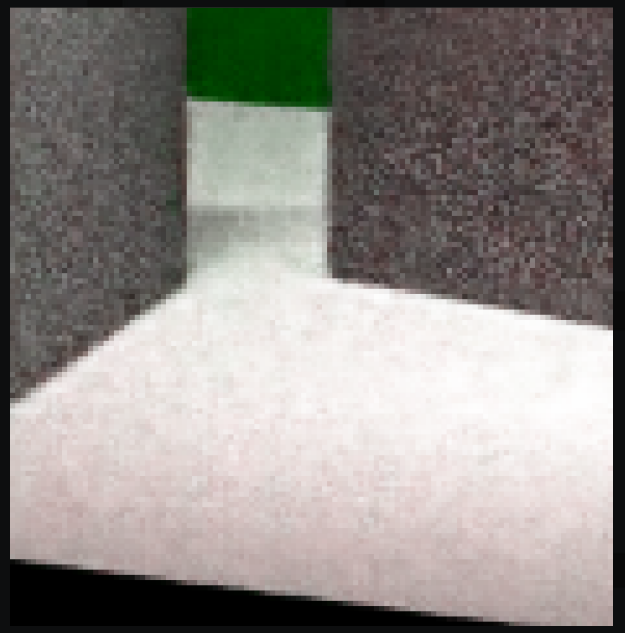
No generalization performance



Input Image
10spp



Ground Truth
100spp



Denoising
Auto Encoder

LUMINOUS PRODUCTIONS

Denoising Auto Encoder

現在は Cornel Box のシーンに限定して過学習させた結果なので注意
この学習を行ったネットワークに別シーンのレンダリング結果を入れたとしても
ここまで綺麗にデノイズは行えない　→ 汎化性能向上が課題

潜在変数を増やせば表現力は増すが、全結合時の重みサイズが増加

それでも Denoising Auto Encoder によって、レンダリング結果の品質向上が見られる

- 例えばカットシーンなど限定して適用するといった使い方
- 異なるシーンごとで学習して、重みデータだけ切り替える等

LUMINOUS PRODUCTIONS

Denoising Auto Encoder

Denoising Auto Encoder のネットワークの重みファイルは 約230MB
(今回、諸事情で 720p の Input でネットワークを構成)

例えば 1920 x 1080px の高解像度のデータをニューラルネットワークで扱う場合、
畳み込み処理 (Pooling) を増やして情報を落とす事が必要

Auto Encoder は途中で中間層 (ボトルネック) を作るために全結合を行うため
→ そこでパラメータ数が増加

LUMINOUS PRODUCTIONS

Denoising Auto Encoder

ランタイム上で Convolution と Deconvolution を繰り返し実行すると
メモリアクセス増加と演算数増加で処理負荷が厳しくなる（殆どリアルタイムでは厳しい)

ただ、畳み込みを繰り返さないと全結合した時のパラメータ数が増え
デバイスメモリーにすら乗らないサイズになる可能性

→ 処理負荷とメモリーのバランスの調整

LUMINOUS PRODUCTIONS

Super-Resolution NN

LUMINOUS PRODUCTIONS

Super-Resolution

超解像度処理は効果として分かりやすく、導入しやすいテーマの一つ

4k 解像度、将来的な 8k 解像度を考えた時に、描画結果の NN 超解像度化も選択肢の一つとなる？

畳み込み層を並べただけのシンプルな構造のものもあるため、チャレンジしやすい

→ Filter 枚数の多い畳み込み処理の手法によっては、瞬間的な使用メモリー増加に注意も

LUMINOUS PRODUCTIONS

Super-Resolution

ニューラルネットワークを用いた超解像度の手法は非常に多くの物が提案され
画像処理学会などでも毎年多くの論文が採択

- SRCNN
- FSRCNN
- VDSR
- DRCN
- RED
- ESPCN
- SRGAN

→ ここに上げている手法は少し以前のもの
　2018年、2019年に入ってからも新しい論文が多数発表

LUMINOUS PRODUCTIONS

Super-Resolution

SRCNN のネットワークの構成

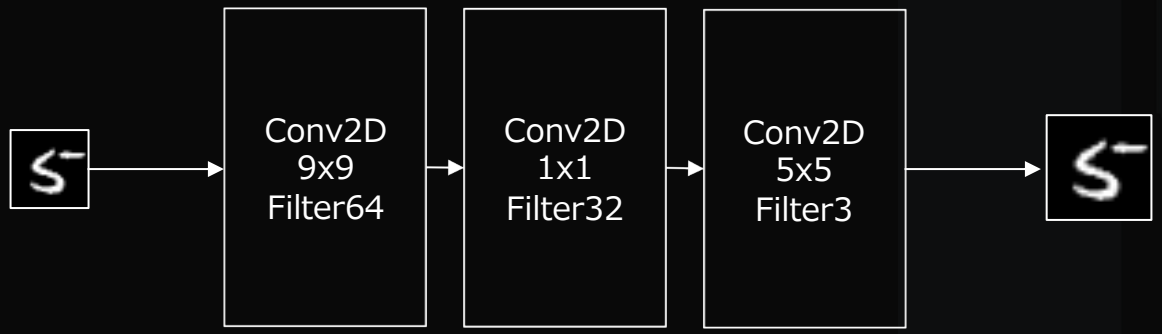


Image Super-Resolution Using Deep Convolutional Networks
https://arxiv.org/pdf/1501.00092.pdf

LUMINOUS PRODUCTIONS

Super-Resolution

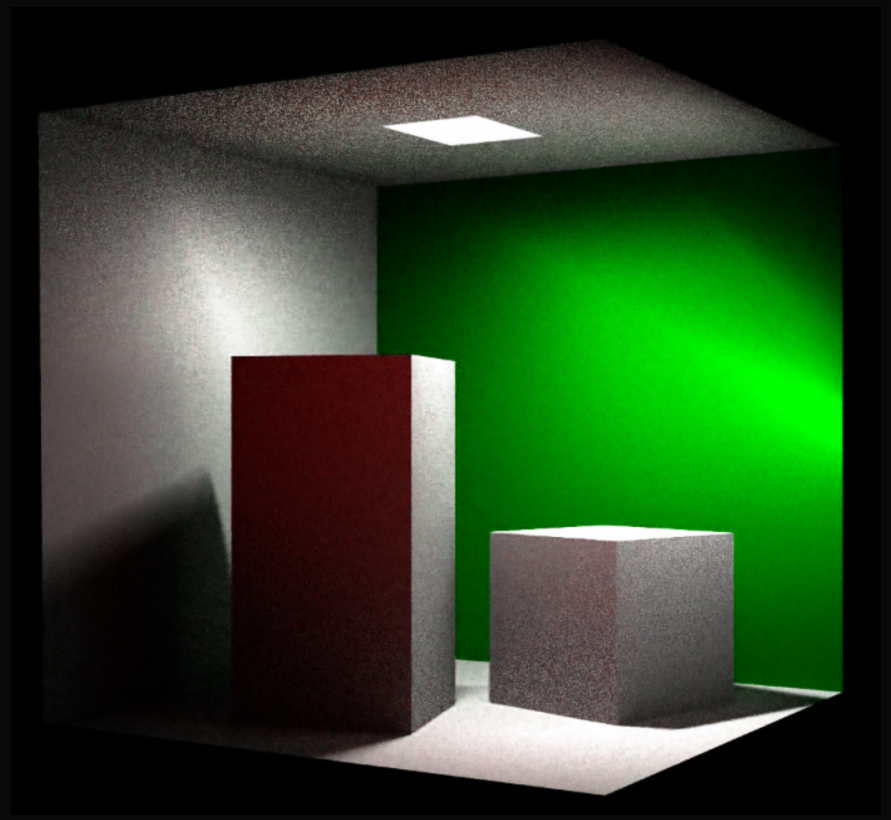
2014年に発表された SRCNNは下記 3層の Convolution のみで構成
ニューラルネットワークを使った超解像度処理の原点

- 低解像度画像の特徴を抽出するフィルター
- 高解像度画像への非線形写像を行うフィルター
- 高解像度画像を再構成するフィルター

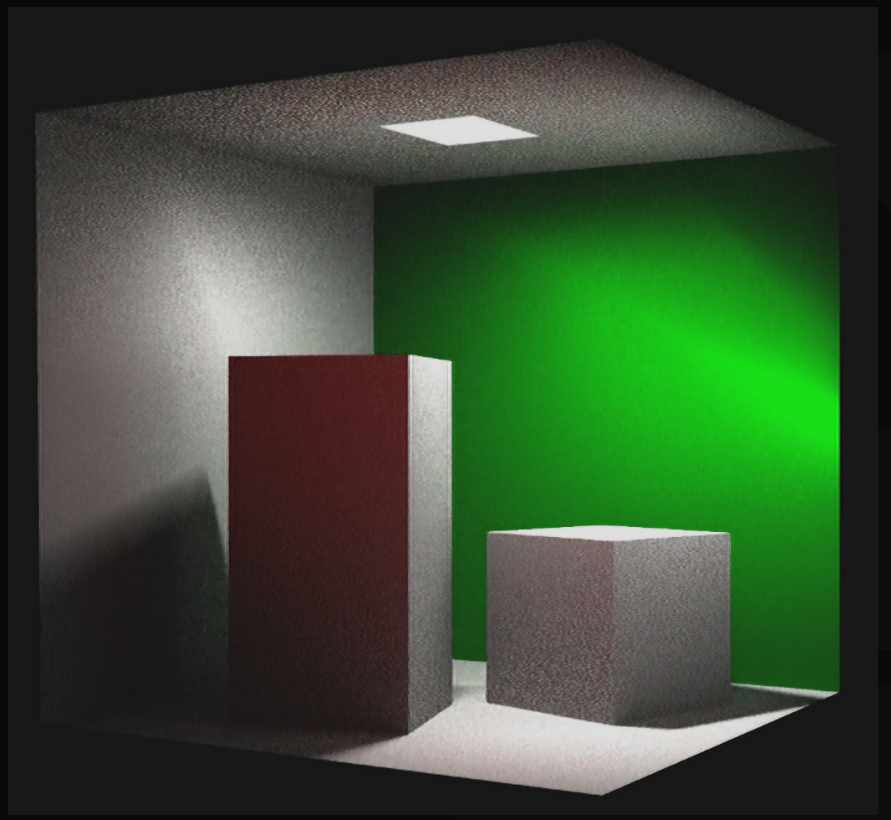
Image Super-Resolution Using Deep Convolutional Networks
https://arxiv.org/pdf/1501.00092.pdf

LUMINOUS PRODUCTIONS

Super-Resolution



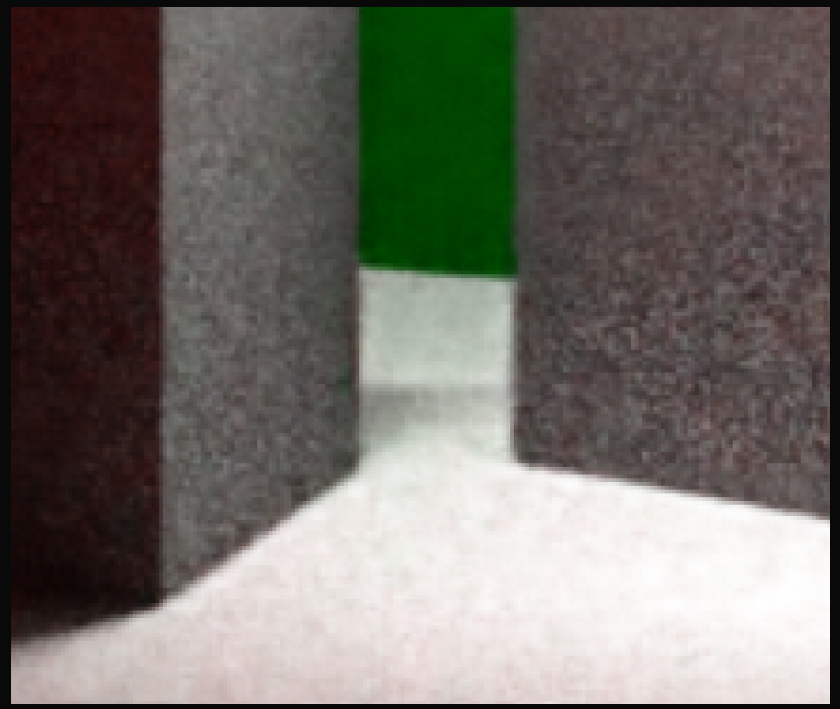
Bilinear



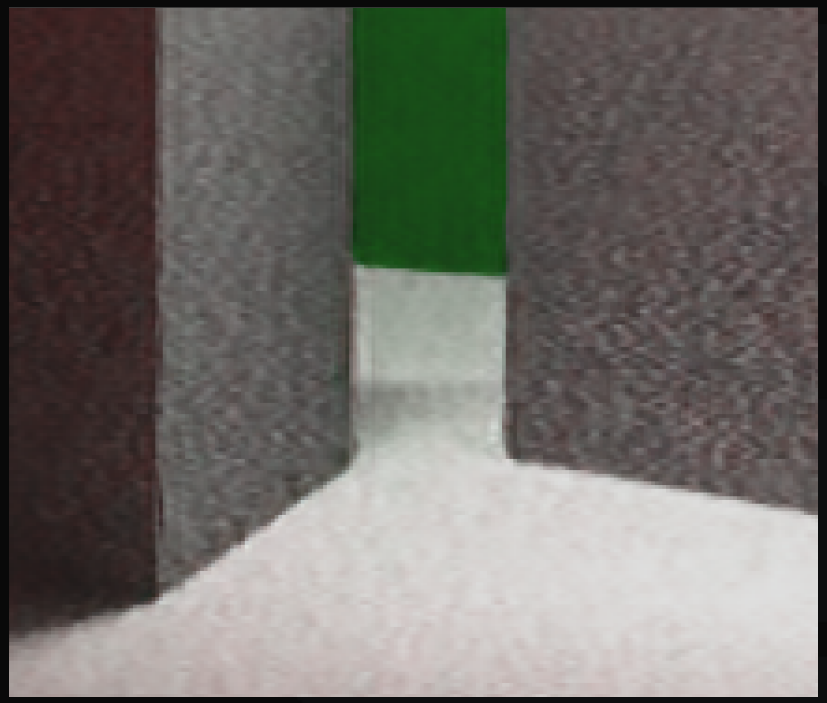
SRCNN
(※色域も変化)

LUMINOUS PRODUCTIONS

Super-Resolution



Bilinear



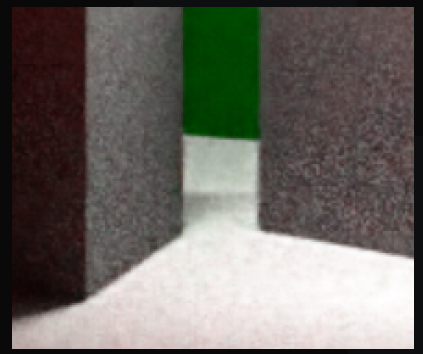
SRCNN

LUMINOUS PRODUCTIONS

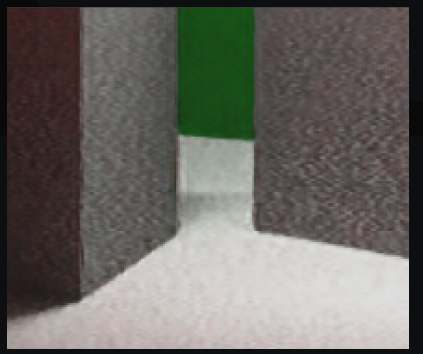
Super-Resolution

単純な Bilinear と比較した時の性能改善は見られたが、ほぼ同じだった

→ CornelBox のレンダリング結果でのみ学習したのがよくなかった
→ 低解像度から高解像度ピクセルへの写像がうまく学習できていない
→ エッジのディティールが復元できた部分もあったが、アーティファクトとなる部分も



Bilinear



SRCNN

LUMINOUS PRODUCTIONS

Super-Resolution

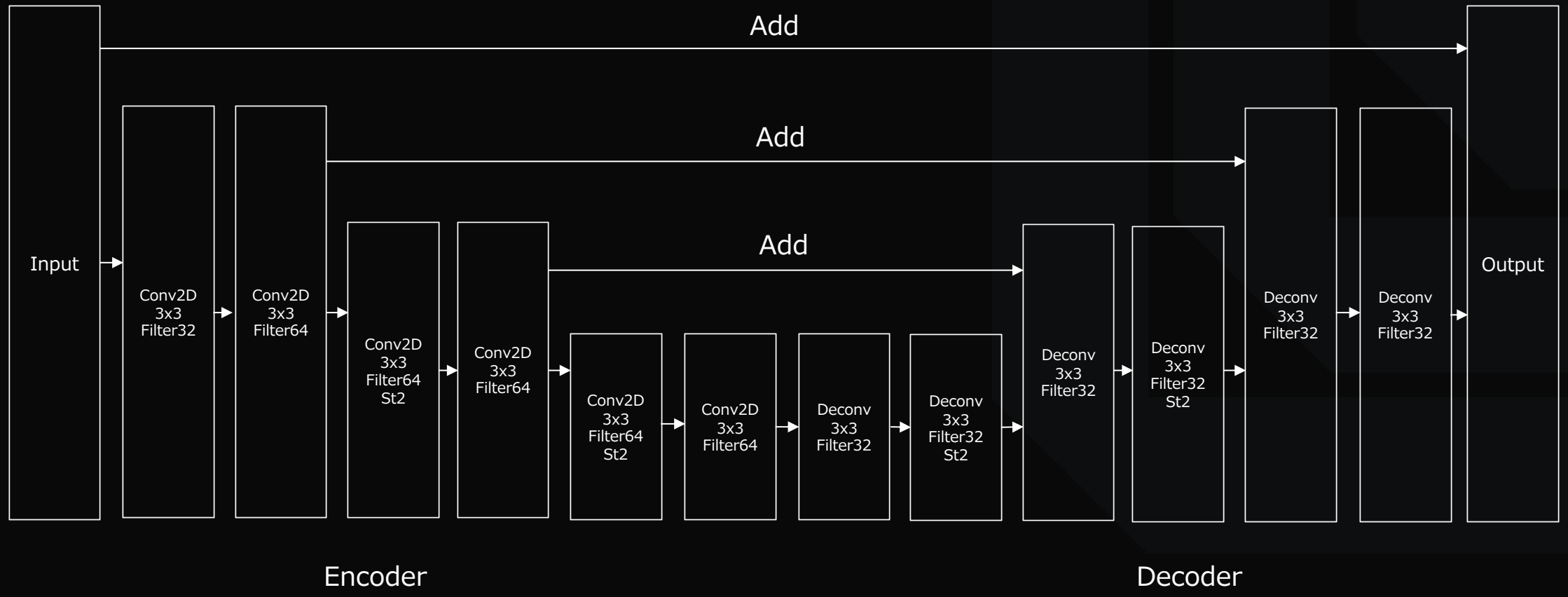
SRCNN から性能の改善のための手法も幾つか提案されている

VDSR や RED と呼ばれる超解像度の仕組みでは、
ネットワークの層の間に Skip Connection と呼ばれる接続を持った構造

Image Restoration Using Convolutional Auto-encoders with Symmetric Skip Connections
https://arxiv.org/abs/1606.08921

Super-Resolution

RED ネットワークの構成



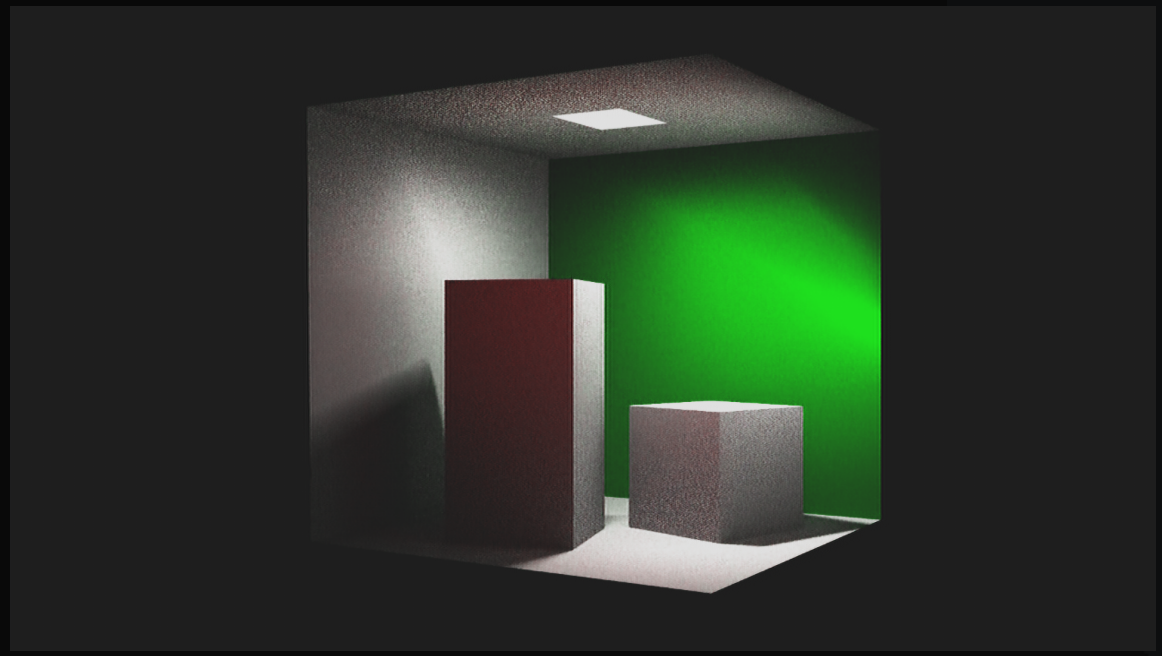
LUMINOUS PRODUCTIONS

Super-Resolution

見た目にはほぼ差が無いが PSNR 値は SRCN から若干の改善は確認

SRCNN 同様レンダリング結果のみを学習させたのは良くなかった

→ SR 処理に関しては、現実世界の多彩な特徴を持つ画像を入力して特徴を抽出しないといけない



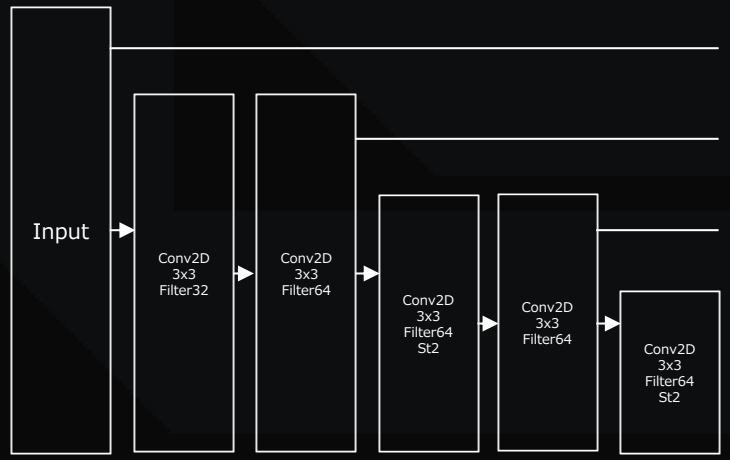
RED Result

Super-Resolution

なぜ Skip Connection のような構造が有効か？

Encoder-Decoder モデルは層が深くなり圧縮される情報が増えるほど、
入力の情報が欠落してしまい出力層にまで届かなくなる

Input や Encoder の値を直接 Decoder へ接続することによって情報を補完している
→ 超解像度での処理では Skip Connection は効果的



LUMINOUS PRODUCTIONS

今後の課題

レンダリングにニューラルネットワークを適用する場合は CNN の実行速度が全て
今回の独自実装の CNN は速度が非常にネックだった
→ DirectML Super Res (Compute Shader) での実行サンプルよりも数倍遅い結果

CNN 重ねすぎ問題
UAV へのアクセス回数と、単体バッファで大きいサイズが必要
論文の内容のそのままの実装ではなく、Conv, Deconv 回数が少ないモデルの研究開発

LUMINOUS PRODUCTIONS

今後の課題

メモリー使用量は大きい

Denosing ネットワークと SR 合わせて、学習データは float32 250MB ほど
→ 留意はするが、CNN ネットワークの場合は学習データは大きく問題にはならない
→ float16 の使用や、力技だが Texture に詰め込んで DDS 圧縮など減らせる可能性は有り

Convolution 適用時は、フィルター枚数へ制限が必要
レンダリングターゲットへの枚数の多い Filter の適用は、非常に大きいメモリー使用量
出力リソース、一時計算用のバッファ、逆伝播用の入力保持などで、非常に大きいなメモリ使用量

まとめ

Conclusion.

LUMINOUS PRODUCTIONS

Conclusion

ニューラルネットワークを画像処理的に応用し、描画品質の向上は可能

今後も、応用事例は増えるだろうと予想

リアルタイム向けでは無いが、毎年新しい超解像度技術なども出ているため、性能改善には期待

ネイティブ高品質レンダリングした時と比較して、

速度面、品質面で優位性のあるニューラルネットワークが構築できるかが課題

LUMINOUS PRODUCTIONS

Conclusion

DirectML を使うか、独自に実装するか？

プラットフォーム提供のフレームワークがあれば使用を推奨

しかし学習機能が無かったり、最新の機能追加への対応が行われない事もある

→ 既存フレームワークを補完する形での独自実装が必要となる

→ 仕組み上、DirectML が提供する機能として Training が入る可能性は高く無い

LUMINOUS PRODUCTIONS

Conclusion

グラフィックス処理へのニューラルネットワークの活用は始まったばかり

・現在はまだ実行速度には大きな課題があるが、DirectML や AI Core の登場など、
リアルタイムへのニューラルネットワーク活用への可能性は広がっている

LUMINOUS PRODUCTIONS

ご清聴ありがとうございました。

Kento Masuno
masukent@luminous-productions.com

Reference

- Building Autoencoders in Keras
  https://blog.keras.io/building-autoencoders-in-keras.html
- TsnsorFlow 開発入門　Keras による深層学習モデル構築手法
- ゼロから作るdeep learning
- 機械学習＆ディープラーニングの仕組みと技術がしっかりわかる教科書
- Trains a denoising autoencoder on MNIST dataset.
  https://keras.io/examples/mnist_denoising_autoencoder/

LUMINOUS PRODUCTIONS

Copyrights

DIRECTX並びにWINDOWSはマイクロソフト コーポレイションの商標または登録商標です。
TENSORFLOWはGoogle LLCの商標または登録商標です。
GOOGLEはGoogle LLCの商標または登録商標です。
APPLE並びにMETALはアップル インコーポレイテッドの商標または登録商標です。
その他掲載されている会社名、商品名は、各社の商標または登録商標です。