CS231a Project Final Report
3D Reconstruction Using Deep Neural Networks

Chafik taiebennefs
tchafik@stanford.edu

Abstract

Over the last several years Deep Neural Networks have been exceptionally at addressing computer vision machine learning tasks such as image classification, object detection, image segmentation, and object tracking. In my project explore few approaches to adapt Deep Neural Networks to the task of 3D object reconstruction.

Introduction

3D reconstruction is defined as the task of inferring the three-dimensional volume of an object given one or several images of that object. 3D reconstruction has wide practical applications in several domains like autonomous driving, and robotic navigation.

In the past, there have been several successful approaches to 3D reconstruction [1]. Typically, single view reconstruction and multi-view reconstruction were treated as separate problems and dealt with using different approaches.

In my project, I intend to implement my own variations of the Choy et al [2] paper solution which can handle both single and multi-view 3D reconstruction. I am mostly concerned with adapting a class of machine learning models known as Deep Neural Networks (DNNs) to the problem of 3D reconstruction.

The reason for my interest in the Choy et al's paper is because the authors created a novel, highly performant solution that combines Convolutional Neural Networks (CNNs), which are good at dealing with Computer Vision tasks, and Recurrent Neural Networks (RNNs) which are good at dealing with time-based sequences. That resulted in an implementation that outperformed the state-of-the art methods for single view 3D reconstruction. In addition, unlike most of the previous work in this space, the Choy et al's implementation does not require any image annotations or object class labels for training and testing.

1. Background/Related Work

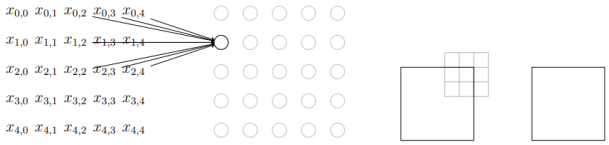
Choy et al's novel, highly performant 3D reconstruction solution leverages two key NN architectures: Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs)

1.1. Convolutional Neural Networks (CNNs)

CNNs consist of a convolution operation, followed by some form of non-linearity and then ending with a pooling operation.

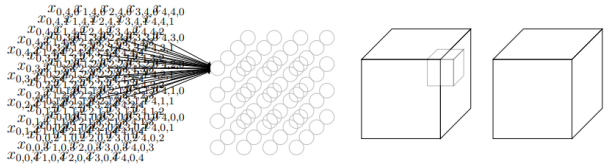
1.1.1 Convolution Operation

The convolution layer (operation denoted by *) takes the input image and abstract it to a feature map, also called an activation map.
In my project I make use of 2D and 3D convolutions.



$$(x * w)_{i,j} = \sum_{m,n} w_{m,n} \cdot x_{i-m,j-n}$$

Figure 1: 2D convolution



$$(x * w)_{i,j,k} = \sum_{m,n} w_{m,n,l} \cdot x_{i+m,j+n,k+l}$$

Figure 2: 3D convolution

1.2. Recurrent Neural Networks (RNN)

Another type of NN I leveraged in my project is RNN.
A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data which makes them good at addressing ordinal or temporal machine learning problems.

An RNN consist of a linear sum gate which is a sigmoid layer followed by an element wise product with the previous cell state.

$$y_t = f(W \cdot \vec{x}_t + U \cdot \vec{h}_{t-1})$$

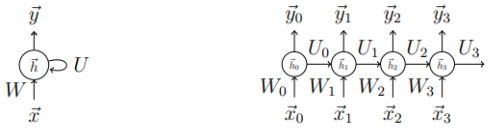


Figure 3: RNN architecture

2. Technical Approach

I will first implement the Choy et al paper solution as a baseline. The model will be fed one or more images of an object (from different viewpoints) and will output a reconstruction of the object in the form of a 3D occupancy grid.

Once my initial baseline model is developed, my goal is to implement and train several different versions of the baseline model to achieve comparable or better performance than Choy et al. My model variations will be constructed using different types of encoders/decoders and RNN units, as well as trained using different hyperparameters and network configurations.

2.1. Data Set

The primary dataset I will be using is the ShapeNet dataset [3] which is a publicly available repository of 3D objects. The raw ShapeNet dataset needs to be pre-processed to generate rendered images of different viewpoints (model input) as well as voxelized labels (a voxel is the 3D equivalent of a pixel) versions of the 3D model.

2.2. Training

2.2.1 Training batch size

My environment hardware specs are smaller than those used by Choy et al. in their experiments which led me to using smaller batch sizes. Choy et al used batch sizes between 24 and 36 image renderings while in my experiments I use batch sizes between 8 and 16 to fit the RAM of my machine's GPU.

2.2.2 Hyperparameters

My training hyperparameters are:

BATCH SIZE, EPOCH COUNT, OPTIMIZER TYPE, LEARNING RATE, INITIALIZER TYPE, ENCODER MODE, DECODER MODE, RNN HIDDEN SIZE, and RNN CELL COUNT.

Every network trained has its hyperparameters saved in a JSON file called params.json. Below is a snippet of one of a hyperparameter JSON file.


```
"TRAIN": {
        "BATCH_SIZE": 8,
        "EPOCH_COUNT": 10,
        "TIME_STEP_COUNT": "RANDOM",
        "OPTIMIZER": "ADAM",
        "GD_LEARN_RATE": 0.1,
        "ADAM_LEARN_RATE": 0.00001,
        "ADAM_EPSILON": 1e-08,
        "VALIDATION_INTERVAL": 25,
        "SHUFFLE_IMAGE_SEQUENCE": false,
        "INITIALIZER": "XAVIER",
        "ENCODER_MODE": "RESIDUAL",
        "DECODER_MODE": "RESIDUAL",
        "RNN_MODE": "GRU",
        "RNN_HIDDEN_SIZE": 128,
        "RNN_CELL_NUM": 4
    }
```


Figure 4: Hyperparameter JSON

2.3. Model Baseline Architecture

Below is a high-level visualization of my baseline neural network model using a Tensorflow model visualization tool called Tensorboard.

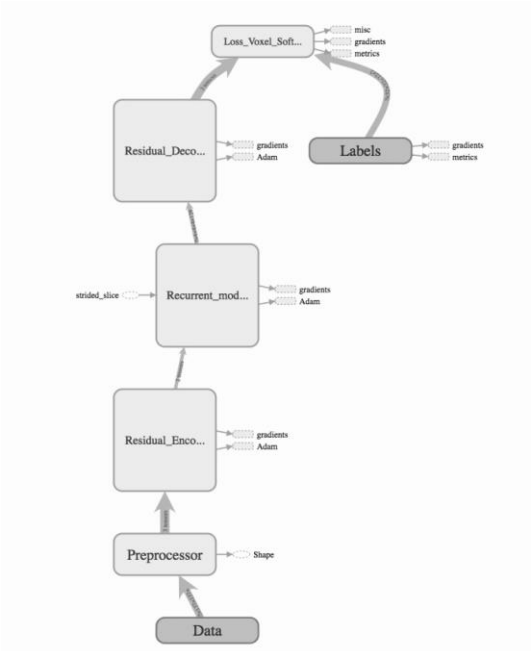


Figure 5: Baseline model architecture

2.3.1 Preprocess Module

Depending on the type of experiment being run the preprocessing step has slightly different steps.
If the experiment being run uses a constant series of time steps, then it is a relatively simple matter of setting n_timesteps to some constant integer. However, if the experiment requires that there the network be shown a sequence of random length then n_timesteps is set to a Tensorflow tensor. Here is a code excerpt:

```
if params["TRAIN"]["TIME_STEP_COUNT"] == "RANDOM":
    n_timesteps = tf.random_uniform([], minval=1, maxval=25, dtype=tf.int32)
elif params["TRAIN"]["TIME_STEPS"] > 0:
    n_timesteps = params["TRAIN"]["TIME_STEPS"]
else:
    n_timesteps = tf.shape(X)[1]
```

If the number of timesteps is not predetermined, then n_timesteps becomes a tensorflow node which means it is going to be evaluated each time the network performs a step. Finally, the actual preprocessing step involves dropping the alpha channel from the 4 colors channel images and cropping the 137X137 image to a 127X127 image at random.

```
n_batchsize = tf.shape(X)[0]
X_dropped_alpha = X[:, :, :, 0:3]
X_cropped = tf.random_crop(X_dropped_alpha, [n_batchsize, n_timesteps, 127, 127, 3])
```

2.3.2 Encoder Module

The input images are then fed to the encoding module which consist of a series of 2D CNN blocks consisting of convolutions, pooling operations, and relu activations and can be thought of as being constituted of several blocks of operations. Each block usually has one down-sampling operation combined with a certain number of convolution operations and relu activation functions.
There are several variants of these blocks Choy et al implements a Simple and Residual Variant. I also implemented the Simple and Residual variant in addition to a Dilated variant which uses dilated convolutions that allow for more context

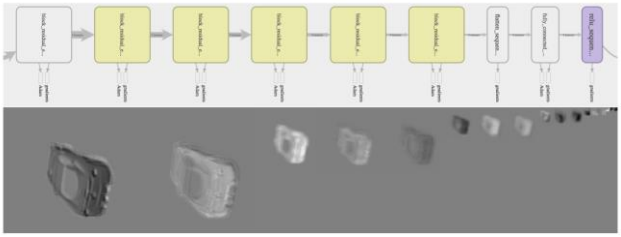


Figure 6: Graph of Encoder and scaled down Feature maps generated by the Encoder portion.

The encoder.py module consists of several functions the most important of which is conv_seqeunce. the following code expert from that function shows how each of the convolution layers is constructed.
Each layer consists of fm_count_in * fm_count_out KxK kernels.

```
kernel = tf.Variable(init([K, K, fm_count_in, fm_count_out]), name="kernel")
bias = tf.Variable(init([fm_count_out]), name="bias")
def conv2d(x): return tf.nn.bias_add(tf.nn.conv2d(
    x, kernel, S, padding=P, dilations=D, name="conv2d"), bias)
ret = tf.map_fn(conv2d, sequence, name="conv2d_map")
```

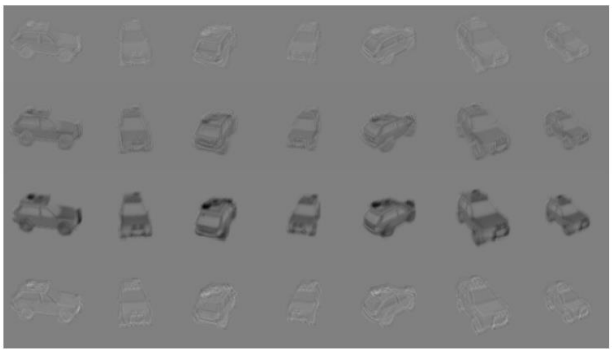


Figure 7: Feature Maps generated from 4 kernels on a sequence of 7 Images

2.3.3 Recurrent Neural Network (RNN) module

The RNN module is the core of my model, it consists of a 3D 4x4x4 grid of RNN cells with each RNN cell responsible for predicting whether a portion of the 3D space is occupied.

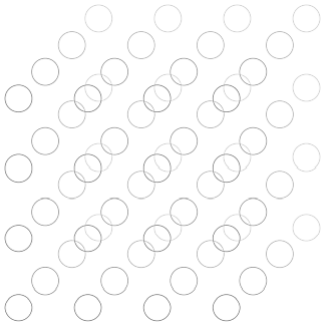


Figure 8: Grid of RNN Cells

Each RNN unit takes an encoded low dimensional feature vector of size 1x1024 which is then fed to the gates in the RNN cells; also, each unit has a hidden state of shape 1x128. The input and the hidden state of the previous time step are combined to calculate a new hidden state.

Each RNN cell can gain access to information from its neighbors via 3D convolutions which convolves the combined hidden state of all the cells to create the new hidden state. The combined hidden state vectors of each cell are 4-dimensional tensors with a 4x4x4x128 shape. Finally, by convolving the hidden state by a 3D kernel of shape 3x3x3, the model can share information between adjacent cells. The final hidden state is then passed to the decoder module.

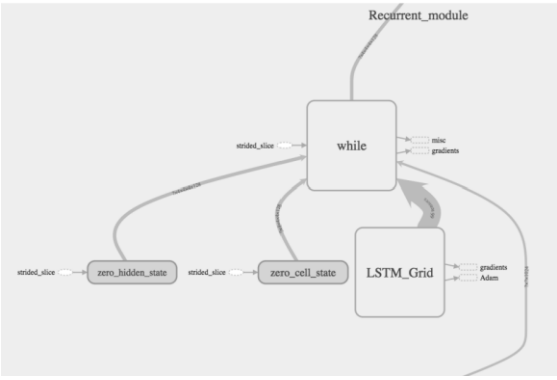


Figure 9: RNN module

2.3.4 Decoder Module

The decoder module consists of a series of 3D convolution and un-pooling layers. The 3D convolutions are implemented like in the 2D case but instead of initialization a 2D kernel, we initialize a 3D kernel. Before generating a prediction, the hidden state of the recurrent unit is de-convolved to a 4-dimensional tensor (shape 32x32x32x2) which is then converted, using the Softmax function, to a final 4-dimensional tensor of the same shape and that represents the probability distribution over each of the voxels.

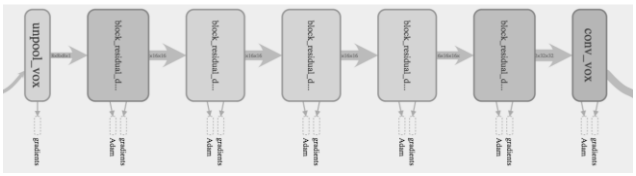
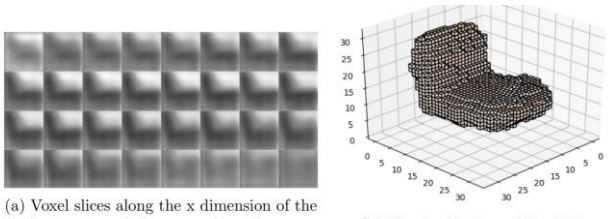


Figure 10: Decoder module



(a) Voxel slices along the x dimension of the final output voxel along one channel

(b) 3D visualization of Prediction

Figure 11: Network output and Visualization

2.3.5 Loss Function

The loss function I use for training my model is the standard Cross Entropy Loss where 'y' is a one hot encoded Voxel label.

$$L(y) = \sum^{m} y \ln p(y) + (1-y) \ln 1 - p(y)$$

3. Experiment

3.1. Development Environment

I set up and configured my ML development environment on AWS including: Tensorflow, numpy, scikit-image, sklearn, keras, and pandas. For training on AWS, I set up a P2 instance with a single NVIDIA Tesla GPU and 12 GB of RAM.

3.2. Data Set Pre-processing

I used the ShapeNet dataset to generate the input to my model which consist of image renderings of the 3D models. The batch of image sequences of 3D models were rendered from different viewpoints and have parameters: batch-size, time step, height, width, and channels.

Figure 12: 3D model sample image viewpoints

Also, I created the code to generate voxelized versions of the original 3D models images (i.e., labels) in '. binovx' format. Finally, the data set examples were shuffled and split into a training and validation batches.

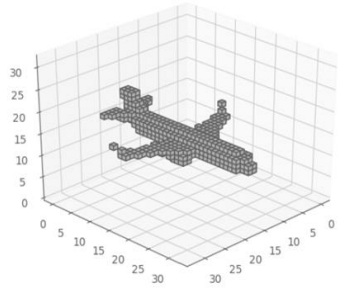


Figure 13: Sample 3D voxelized image rendering

Finally, for easy loading and handling of ground truth labels, I use the python package binvox to serialize the voxelized format into a numpy array that is then converted into a Tensorflow Tensor.

3.3. Training Epochs

To determine what's the optimum number of epochs to train the networks, I performed a training session that went on for 80k iterations. This experiment provided insight into when the training error and the validation error start to diverge, which happens to be around the 30,000's iteration. As such, all subsequent experiments did not go past 30,000 iterations.

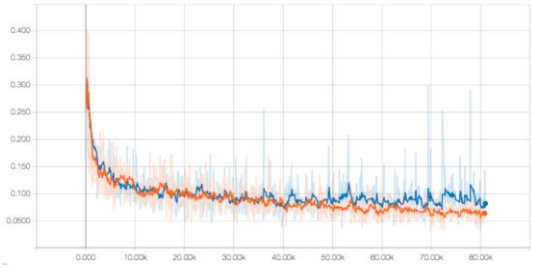


Figure 14: Longest Experiment Run, Training(orange) and Validation(blue)

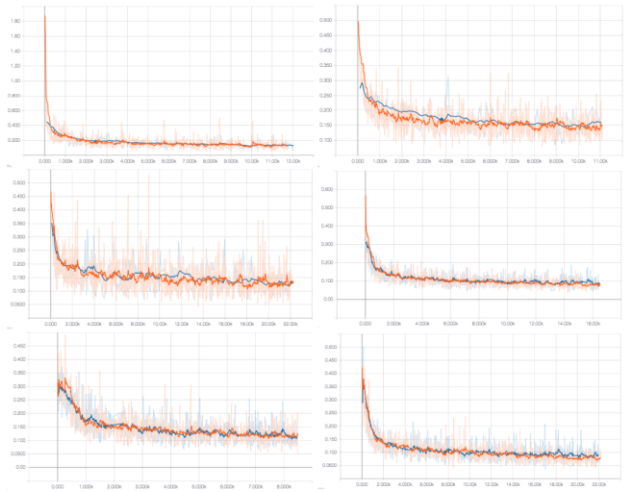


Figure 15: Sample Experiment Runs, Training(orange) and Validation(blue)

3.4. Metrics

To measure the networks performance, I used several performance metrics:

Accuracy

Accuracy can be misleading because it does not consider how confident the network is in its prediction. A prediction can predict a voxel with a probability of 0.6 or 0.9 and the accuracy would still be the same.

Root Mean Squared (RMS)

RMS is usually used to measure error on regression problems. However, I chose to treat the one hot encoded labels as the ground truth and the probability distribution predicted by the network before the final output as the prediction, which made possible to us RMS as a metric. I found that RMS is better metric for training especially as it does not saturate as quickly as the accuracy.

Intersection Over Union (IOU)

This is the metric used by Choy et al. the Intersection over Union (IOU) is defined as the ratio of the intersection of the prediction with the target volume divided by the union of the prediction and target volume.

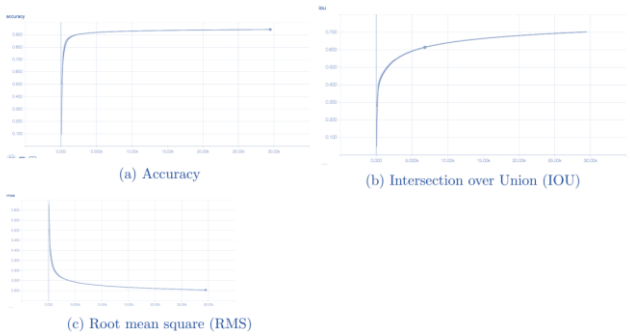


Figure 16: Graphs of performance metrics

3.5. Weight Initialization

When training deep neural networks, the initial network weights can drastically impact the rate at which the network trains and learns.

There are several techniques and approaches for weights initialization in deep neural networks. I chose to use an initialization technique called Xavier initialization because it keeps the variances of the activation of different layers on the same scale.
More formally, for layer *i*, the weights are sampled from the following uniform distribution.

$$W \sim U[\frac{-\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}]$$

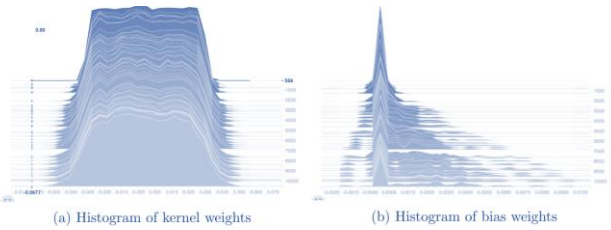


Figure 17: Change of Xavier weight distribution during training

3.6. Optimizers

I used two types of network optimizers in my experiments:

Stochastic Gradient Descent (SGD)

SGD simply consists of calculating the gradient on
A batch of examples and then updating the network parameters using those calculated gradients

$$\theta = \theta_{t-1} - g_t$$

Adaptive Moment Estimation (ADAM)

ADAM is a more sophisticated form of parameter optimization. ADAM is one of several optimization techniques that rely on the "momentum" of the network parameters which can mitigate oscillation.

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t = \beta_2 \cdot m_{t-1} + (1 - \beta_2) \cdot g_t^2$$

$$\alpha_t = \alpha \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$$

$$\theta_t = \theta_{t-1} - \alpha_t \cdot m_t/(\sqrt{v_t} + \epsilon)$$

3.7. Results

I run several experiments using several different NN architectures and was able to achieve performance results comparable performance to Choy et al. The NN architectures are constructed using different types of encoders, RNN units, and decoder modules.

I implemented three types of encoder and decoder modules: SIMPLE, DILATED, and RESIDUAL.

The SIMPLE version uses normal 2D and 3D convolution with pooling and unpooling to encode the sequence of images and to decode the hidden state of the recurrent module.

The DILATED version uses dilated convolutions to perform the encoding and decoding.

The RESIDUAL version uses skip connections which allows us to train much deeper networks.

Below is the mean validation loss for the different types of network architectures and configurations

OPTIMIZER	LEARN RATE	NETWORK TYPE	BATCH SIZE	MEAN VAL LOSS
SGD	0.1	SIMPLE-GRU-SIMPLE	16.0	0.185
SGD	0.1	SIMPLE-GRU-SIMPLE	16.0	0.112
SGD	0.1	SIMPLE-GRU-SIMPLE	16.0	0.155
SGD	0.1	SIMPLE-GRU-SIMPLE	16.0	0.098
ADAM	0.0001	SIMPLE-GRU-SIMPLE	16.0	0.112
SGD	0.1	SIMPLE-GRU-SIMPLE	16.0	0.124
SGD	0.1	SIMPLE-GRU-SIMPLE	16.0	0.139
ADAM	1e-05	RESIDUAL-GRU-RESIDUAL	8.0	0.153
ADAM	1e-05	DILATED-GRU-DILATED	16.0	0.169
ADAM	1e-05	RESIDUAL-GRU-RESIDUAL	8.0	0.121
ADAM	1e-05	RESIDUAL-LSTM-RESIDUAL	8.0	0.175

Figure 18: Mean validation loss

3.8. Predictions Quality

Below are sample examples of the 3D reconstruction outputs vs. ground truth (labels). From left to right, the images in the first column show a fairly accurate 3D reconstruction of the airplane model. However, the other samples (middle and last columns) are not as good. Those kinds of flawed reconstructions are similar to the ones Choy et al observed in their work.

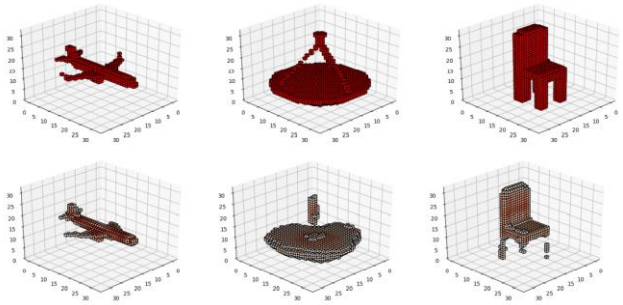


Figure 19: Ground truth voxelized models (Top), Predictions (Bottom)

4. Future Work

There are several ways this project can be further extended. Besides training the networks with more parameters, one could experiment with the size of the feature vectors fed to the recurrent units which could allow the network to capture finer details such as the chair legs.

5. References

[1] Richard Szeliski. Computer vision: algorithms and applications. Springer, 2011, p. 812. isbn: 9781848829350.

[2] Christopher B Choy et al. "3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction". url: https://arxiv.org/pdf/1604.00449.pdf.

[3] Angel X. Chang et al. "ShapeNet: An Information-Rich 3D Model Repository". In: (Dec. 2015). url: http://arxiv.org/abs/1512.03012.

[4] Fisher Yu and Vladlen Koltun. "Multi-scale Context Aggregation By Dilated Convolutions". url: https://arxiv.org/pdf/1511.07122.pdf.