3D Reconstruction with Neural Networks

Michael Melesse

Advisor: Professor Szymon Rusinkiewicz

Second Reader: Professor Olga Russakovsky

Thesis submitted for the degree of

Bachelor of Science in Engineering

Computer Science Department

Princeton University

May 7, 2018

Contents

Acknowledgement

I would like to thank Professor Rusinkiewicz for agreeing to be my advisor and his guidance as I wrote my senior thesis. I would like to thank Professor Russakovsky for agreeing to be the second reader on my thesis. I would like to thank Ragy Markos for looking over my thesis and offering feedback and words of encouragement. I would like to thank professor Michael Berry and his postdoc Jan Homann for the opportunity to be a part of their lab in the summer of my freshman. It is an experience of my time here at Princeton that I will always cherish I would like to thank the University for the resources that it has provided me. I will always be grateful to this place. Finally, I would like to thank my family. My sister and mother who have always been a light in my life.

Abstract

Neural Networks have been successful in tackling problems in computer vision especially over the last few years. Starting with image recognition neural networks, have gone on to produce comparable if not better results in other areas of computer vision such as Image Segmentation and Object Localization. Here we will see an approach in which neural networks can be adapted to deal with another area of computer vision, 3D Reconstruction.

Chapter 1

Introduction

1.1 Motivation

3D reconstruction is the problem of inferring the three dimensional volume of objects given an image or series of images of the same object. This has applications in navigation, robotics and autonomous vehicle etc. There have been various successful approaches to 3D reconstruction [18]. We are less concerned with the problem of 3D reconstruction itself but more concerned with adapting the class of machine learning models known as neural networks to the problem of 3D reconstruction.

1.2 History

In 2012, Krizhevsky et al [12] won the ImageNet Challenge, a competition where different research groups compete to outperform each other on the ImageNet Dataset[16],in the image classification task using a neural network. This lead to a renewed interest in neural networks, which had been around in some form or other since the later half of the last century [17]. The success of AlexNet, the neural network of Krizhevsky et al, lead to the use of neural networks in other standard Computer Vision problems such as Image Segmentation [14] and object detection. AlexNet is a variant of a neural network known as a Convolutional Neural

Network. AlexNet is not the first Convolutional Neural Network that would be LeNet[13] a convolution neural network developed for recognizing hand written digits in the late 1990s by Yann LeCun. A major source for the background knowledge in this thesis will be Goodfellow et al. by far the most complete textbook for contemporary development in neural networks. Finally, for those interested one can find a condensed history of neural networks in the first chapter of Goodfellow et al's Deep Learning Book.[8].

1.3 Objective

The goal of my senior thesis is to implement a neural network capable of 3D reconstruction, followed by experiments on the networks and iterations on the design of the network. We will start by implementing Choy et al [4] and comparing the re-implementation performance with the original. The rational for choosing this particular paper to re-implement has to do with wanting to work on a network that combined to different variants of neural networks. The first of which we have already mentioned, Convolutional Neural Networks which are adept at dealing with patterns in space. The second variant are called Recurrent Neural network which are adept at dealing with patterns in time or sequence in general. We will re-implement Choy et al's network in a different framework. This is so that we will be able to learn what is of fundamental importance in the implementation of the original network.

1.4 Layout

Chapter 2 offers background about neural networks especially the two types of neural network variants already mentioned, that it will possible to understand Choy et al's proposed network and how it relates to neural networks that came before. Chapter 3 is concerned with the implementation details of our network. It makes use of the material developed in chapter 2 and code snippet to explain how our network works. Chapter 4 show the results of experiments with different versions of the network.

Chapter 2

Background

2.1 Notation

Before we start, we will clarify our notation. We will be using vector and sometimes tensor notation. Tensors are a generalizations of vectors and matrices and can be thought of as multidimensional arrays of numbers. One of the properties of a tensor is its rank expressing its dimensionality. A scalar is a tensor with rank 0, scalars are going to be represented with lower case letter with out any indices such as x. A vector is a tensor of rank 1, popular notation for vectors consists of bold lower case letters $\mathbf{x}$ or small case letters with arrows over them $\vec{x}$ and finally the tensor notation for vectors consists of lower case letters with a single subscript indices x_i. The reader should note that this represent an arbitrary component of the vector. A matrix is a tensor of rank 2, popular notation for matrices is usually upper case letters X. Sometimes the matrices are denoted in bold $\mathbf{X}$. The tensor notation for matrices with 2 indices is $x_{i,j}$. The reader should again note that $x_{i,j}$ refers to an arbitrary element of a 2 dimensional tensor. For tensor of rank 3 or more all notation will be restricted to lower case letters with subscript indices equal to the rank of the tensor. For example $x_{i,j,k}$ and $x_{i,j,k,l}$ represents an arbitrary element of a 3 and 4 dimensional tensor respectively.

2.2 Neural Networks

2.2.1 Models of Neurons

There are many different models of neurons, ranging from "...from highly detailed descriptions involving thousands of coupled differential equations to greatly simplified caricatures useful for studying large interconnected networks ..."[5]. We will make use of the latter types of models. An example of a basic neuron model is a function that takes a certain set of inputs, sums them up and then "fires" if the sum exceeds some threshold value, θ. [17]. This model can be further modified by weighing the inputs so that some inputs are bigger than others. These weights are referred to as the parameters of the model. In both cases if the resulting sum exceeds θ then the neuron "fires" by outputting some value, for example the difference between the sum and the threshold otherwise it does nothing. Another modification we can add to our model is by applying some non-linear function, f, referred to as the activation function to the output of the neuron. The non-linearity of the activation function is important for expanding the representational power of the model neurons.

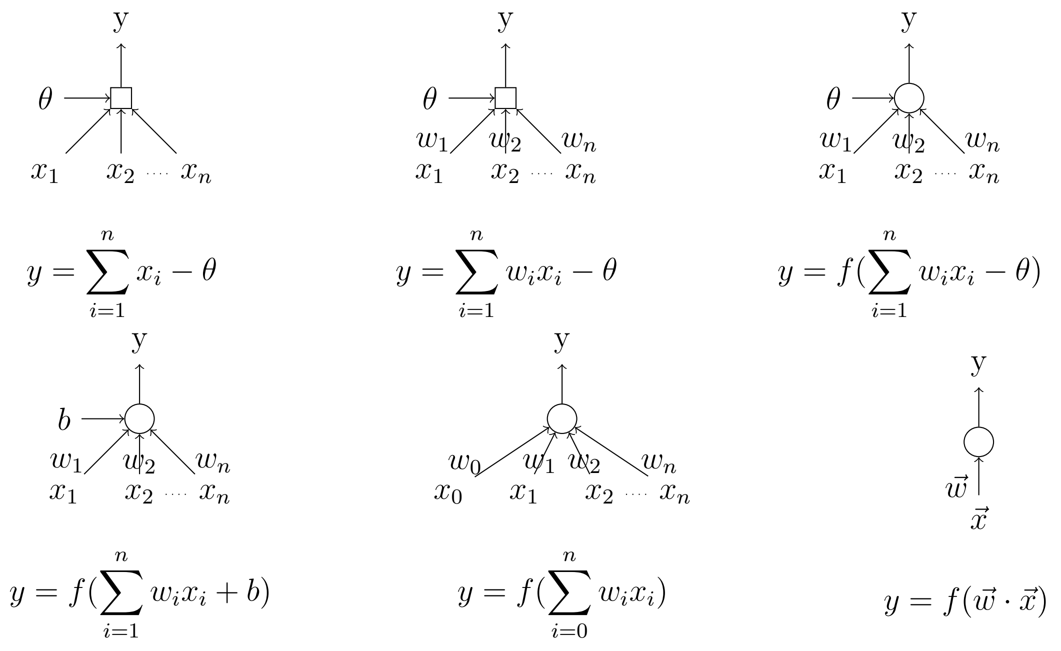


Figure 2.1: Variants of a model Neuron

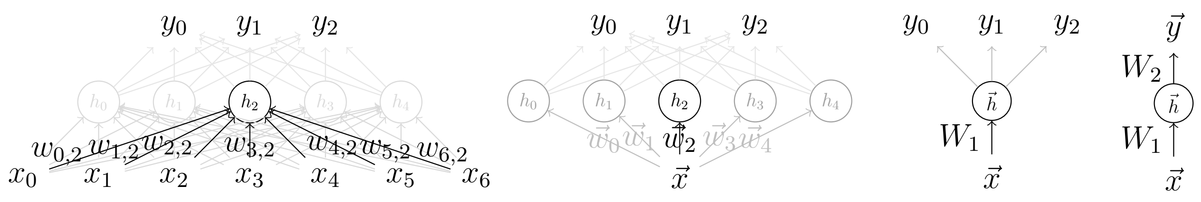
In the case where there is an activation function, the result of the linear operations, $\sum_i^n w_i x_i - \theta$, before the application of f is referred to as the preactivity. The preactivity is usually written as $\sum_i^n w_i x_i + b$, where b is referred to as the bias. It is common for the bias, to not be included in most equation because it is possible to conceive of the bias as just another one of the inputs x_0 with its weight set to $w_0 = 1$. [17] Using all of these above a modern neuron model is written as

2.2.2 Learning and Optimization

There are different types of learning. In this instance, we are concerned with a specific type of learning called supervised learning. What makes supervised learning distinct is that learning occurs by using data label pairs with the objective of finding a function that can map each data point to its appropriate label. This is done by using a metric usually referred to as the loss or the error. The loss is a **positive** scalar value that signifies how good a prediction is. A low loss means that the learner was able to make a good prediction. A high loss on the other hand signifies a bad prediction. Given these definition we will conceive of learning as finding parameters that minimize the loss. This approach of framing learning as the problem optimizing parameters is at the core of machine learning and training neural networks.

2.2.3 Networks of model neurons

We first consider the most basic networks we will consider is feed forward networks. Consider the neural network below with consists of a series of neurons with each with their own connection the the input vector $\vec{x}$ hidden layer.



y_0 y_1 y_2
h_0 h_1 h_2 h_3 h_4
$w_{0,2}$ $w_{1,2}$ $w_{2,2}$ $w_{3,2}$ $w_{4,2}$ $w_{5,2}$ $w_{6,2}$
x_0 x_1 x_2 x_3 x_4 x_5 x_6
$\vec{w}_0$ $\vec{w}_1$ $\vec{w}_2$ $\vec{w}_3$ $\vec{w}_4$
$\vec{x}$
$\vec{h}$
W_1
W_2
$\vec{y}$

Figure 2.2: Networks

2.2.4 Back Propagation

Backpropagtion allows us to find out how each parameter contributes to the developing. Let there be some Error function E that takes the target label t and the prediction $\hat{y}$. There are different ways of measuring error but a very common one is the squared error.

$$y = \vec{w} \cdot \vec{x}$$

$$E(t, \hat{y}) = (t - y)^2$$

We can then go about determining how each of the parameters contribute to the over all error. we do this by taking the derivative of the error with respect of the parameters. However we note that the Error function is not a direct function of the parameters but rather is a function of function of those parameters. To determine this we use the chain rule of multivariate calculus

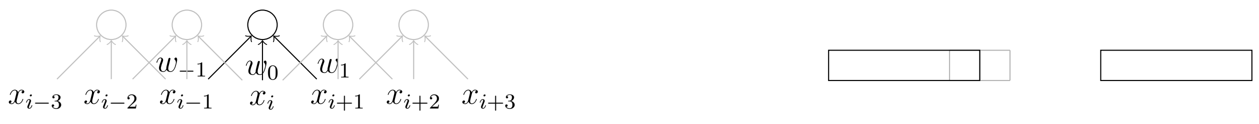
$$\frac{\partial E}{\partial \vec{w}} = \frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial w_i}$$

2.3 Convolutional Neural Networks (CNN)

CNN usually consist of a convolution operation, followed by some non linearity ending with some pooling operation

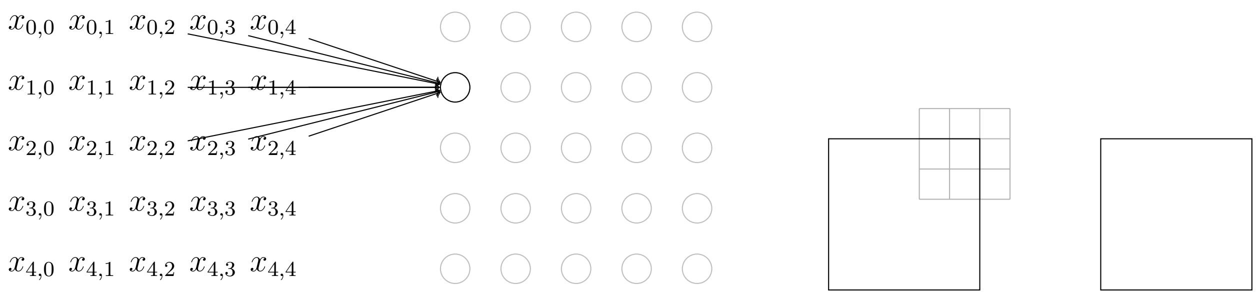
2.3.1 Convolution Operation

Convolution applies in the discrete and also the continuous case but since we are working finite data points will focus on the discrete case.[6] convolution operation can be performed in any number of dimensions, and is usually denoted with an asterisk($*$). Starting with the 1 dimensional case, convolution with a kernel k_m over the input x_i 1 dimension can be expressed as he expression shows 1D Convolution, 2D Convolution and 3D Convolution. There are 2 ways visualizing convolution and both are useful.This project makes us of convlution in 2D and 3D



$$(x * w)_i = \sum_m w_m \cdot x_{i-m}$$

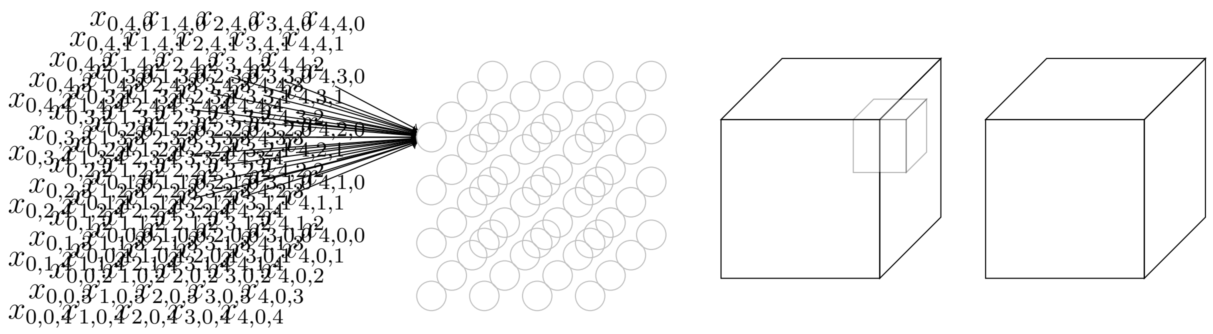
Figure 2.3: 1D Convolution



$x_{0,0}$ $x_{0,1}$ $x_{0,2}$ $x_{0,3}$ $x_{0,4}$
$x_{1,0}$ $x_{1,1}$ $x_{1,2}$ $x_{1,3}$ $x_{1,4}$
$x_{2,0}$ $x_{2,1}$ $x_{2,2}$ $x_{2,3}$ $x_{2,4}$
$x_{3,0}$ $x_{3,1}$ $x_{3,2}$ $x_{3,3}$ $x_{3,4}$
$x_{4,0}$ $x_{4,1}$ $x_{4,2}$ $x_{4,3}$ $x_{4,4}$

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

Figure 2.4: 2D Convolution



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

Figure 2.5: 3D Convolution

2.3.2 Weight updates in CNNs

To perform weight updates on CNNs, We calculate how a single kernel contributes to the overall error in the CNN. Using the gradients we then update the kernels.

$$\frac{\partial E}{\partial w_{i,j}}$$

2.4 Recurrent Neural Networks (RNN)

In addition to CNNS, another type of network we will be working with is the A basic rnn cell can be generate its output by taking in to account both current input its own internal state

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

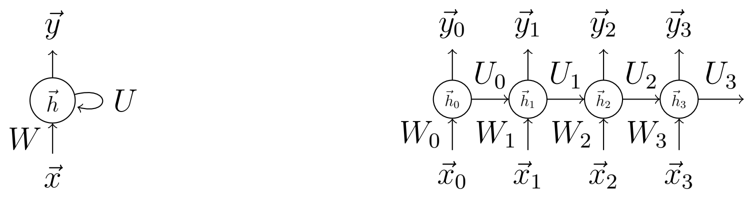

$\vec{y}$
U
$\vec{h}$
W
$\vec{x}$
$\vec{y}_0$ $\vec{y}_1$ $\vec{y}_2$ $\vec{y}_3$
U_0 U_1 U_2 U_3
$\vec{h}_0$ $\vec{h}_1$ $\vec{h}_2$ $\vec{h}_3$
W_0 W_1 W_2 W_3
$\vec{x}_0$ $\vec{x}_1$ $\vec{x}_2$ $\vec{x}_3$

Figure 2.6: Unfolding RNNs

. These models simply consisted of a linear sum gate is a sigmoid layer followed by an element wise product with the previous cell state. A vector with value is the range 0 to 1. With 0 ignore everything and 1 accept everything.

2.4.1 Weight updates in RNNs

Updating the weights of a recurrent neural network does not require a new algorithm rather we perform standard back propagation on the *unfolded* graph. This is sometime called back propagation in time. However that does not mean that problems do not arise in training neural networks. there are difficult in train recurrent neural networks because of gradient would disappear.

$$\frac{\partial E}{\partial W_t}, \frac{\partial E}{\partial U_t}, \frac{\partial E}{\partial b_t}$$

2.4.2 Variants of RNNs

The issues with effectivley training RNNs was motivated by

Long short term memory (LSTM)

LSTMs are capable of learning long term dependencies. LSTMs have 3 gates referred to as the input, output, forget gates. LSTMs were introduced in 1980s [10].one of the few RNNs that could be effectively trained was LSTM. the long term short term rnn cell, Relevant history. [10]

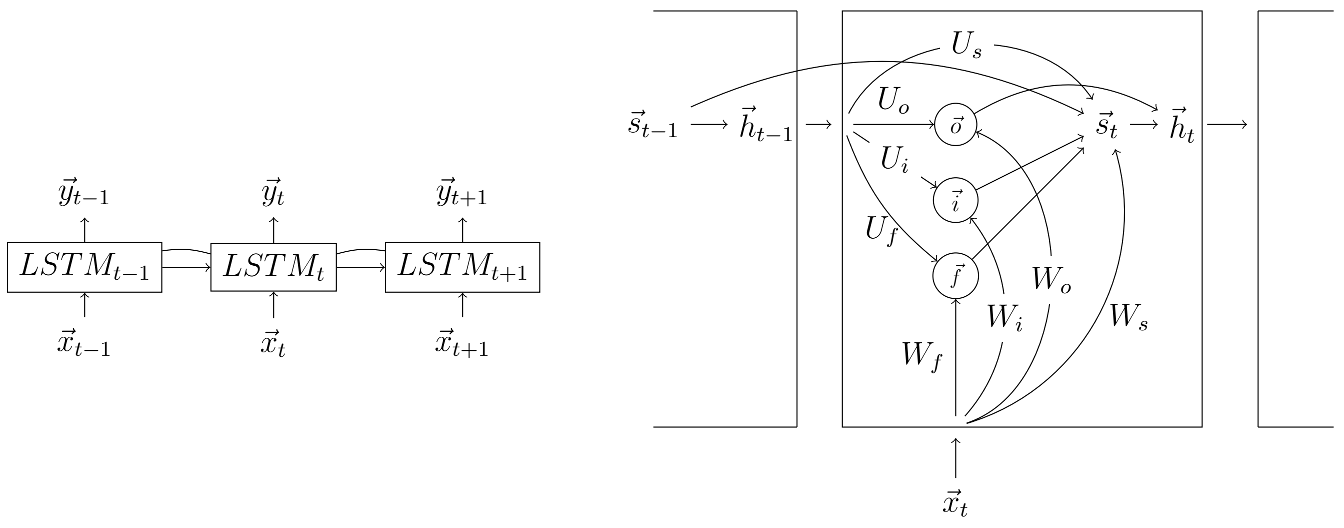


Figure 2.7: Unrolled LSTM cell. Left: High level view of cell. Right: Detailed view of cell

$$i_t = \sigma(W_i \cdot x_t + U_{it-1} + b_i)$$

$$f_t = \sigma(W_f \cdot x_t + U_f \cdot h_{t-1} + b_f)$$

$$o_t = \sigma(W_o \cdot x_t + U_o \cdot h_{t-1} + b_o)$$

$$s_t = f \circ s_{t-1} + i_t \circ \tanh(W_s \cdot x_t + U_s \cdot h_{t-1} + b_s)$$

$$h_t = o_t \circ \tanh(s_t)$$

Gated Recurrent Unit (GRU)

Introduced a few years ago [3], GRUs are a variant of recurrent networks that give comparable performance to LSTM with less parameters. GRUs have 2 gates, which are called the update and reset gate.

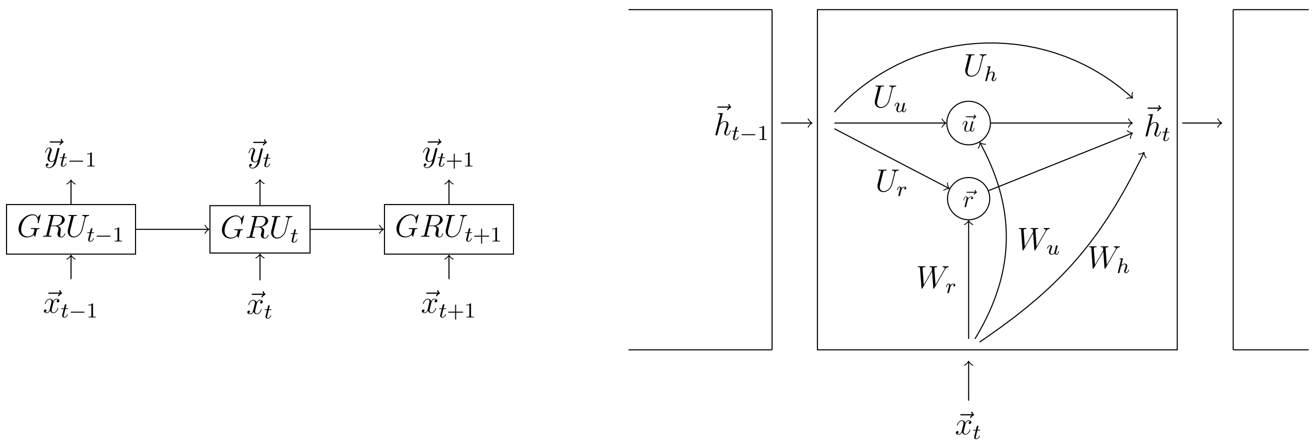


Figure 2.8: Unrolled GRU cell. Left: High level view of cell. Right: Detailed view of cell

$$u_t = \sigma(W_u \cdot x_t + U_u \cdot h_{t-1} + b_u)$$

$$r_t = \sigma(W_r \cdot x_t + U_r \cdot h_{t-1} + b_r)$$

$$s_t = (1 - u_t) \circ s_{t-1} + u_t \circ \tanh(W_s \cdot x_t + U_s * (r_t \circ h_{t-1}) + b_s)$$

Chapter 3

Implementation

3.1 Overview

Choy et al implement their network in theano[15], a deep-learning frame work which is no longer being maintained. We re-implement the network using Tensorflow[1] another deep learning framework developed by Google. Choy et la implement several variants of their network. the most basic of which encodes a sequence of images of the object of interest using a serious of convolutions layers into a sequence of low dimensional vectors. This sequence of low dimensional vectors is then given as input to a recurrent module. This recurrent module, which can be conceived of as a grid of RNN Cells, is the main novelty of the paper.

3.1.1 Tensorflow

Tensorflow is the machine learning frame work that we will be using to perform the reimplementation. To understand how Tensorflow works we will be making use of a toy example given below, which is written to be as simple and as informative as possible.

Before we start we should define some of the types used when working with Tensorflow. First there is the Tensorflow Variable, which are meant to represent anything in the graph that is subject to change. Parameters belong to a subclass of Variables that are called

```
import tensorflow as tf
LEARNING_RATE, EPOCHS = 0.01,100                     # hyper-parameters

# Build Tensorflow Graph
X = tf.constant([[1.0],[2.0],[3.0]])                 # input
T = tf.constant([[3.0],[2.0],[1.0]])                 # target
W = tf.Variable(tf.random_uniform([3,3]))            # parameters
Y = tf.matmul(W,X)                                   # prediction
L = tf.reduce_mean(tf.abs(T-Y))                      # loss(error)

GD = tf.train.GradientDescentOptimizer(LEARNING_RATE)# optimizing algorithm
Gradients = GD.compute_gradients(L)                  # gradient of the error function
TRAIN_STEP = GD.apply_gradients(Gradients)           # update parameters using gradients

# Run Graph
with tf.Session() as sess:                           # start tensorflow session
    tf.global_variables_initializer().run()          # initialize variables
    for i in range(EPOCHS):                          # iterate over the dataset EPOCHS times
        result=sess.run([L,Y,TRAIN_STEP])            # fetch values from graph

print("Loss:", result[0])
print("Prediction:", result[1])
```

Figure 3.1: Toy Tensorflow Program

trainable. These are the Variables hold Tensors that are automatically updated by tensorflow using the computed gradients.

The code is divided into 2 main sections. The first section involves building the graph. The data is stored in the Tensorflow constant X and the parameters are stored in a Tensorflow Variable W. We then proceed to create our matrix multiplication operation which takes in as input X and W, outputting its result as Y. We then calculate the error between our prediction and our target value. The loss is defined to be a scalar that decreases as the network become better at making accurate predictions.

3.2 Network

Tensorflow has a tool called Tensorboard for visualizing the structure of the neural networks it creates. Below is a high level view of the network. The network takes the data into a prepossessing module after which it sends it to the encoding module. The encoding module in turn sends its output to the Recurrent module which is at the heart of the network. The recurrent module sends its output to the decoder module. Finally the decoder output is combined with the label information to calculate the loss. This is then used to update the parameters of the network.

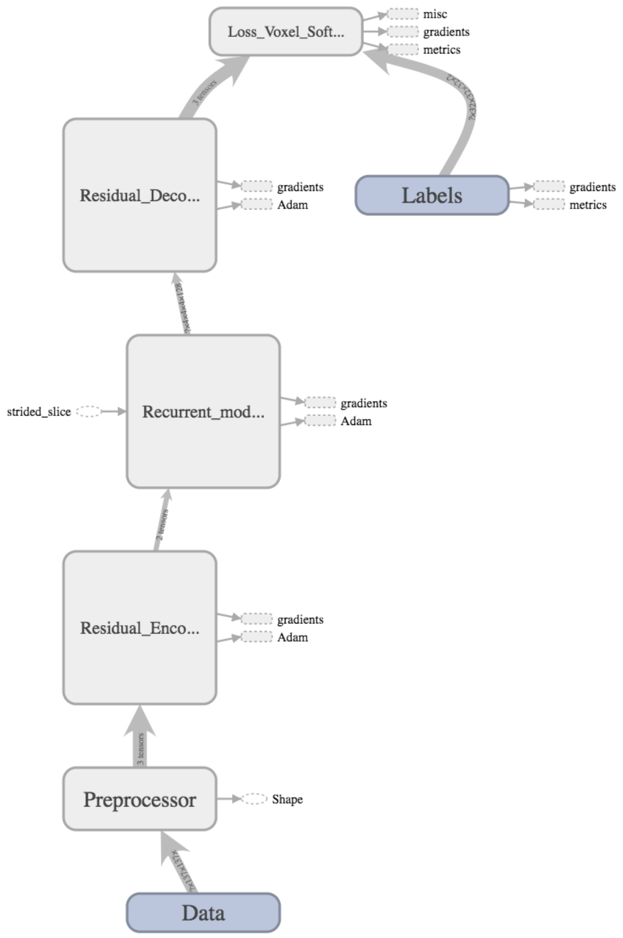


Figure 3.2: High level view of Network

3.3 Dataset

The primary dataset for training is the ShapeNet dataset[2], a repository of 3D models. We use the ShapeNet dataset to generate both the input to the network which is a just images of the 3D models and the labels which are voxelized versions of the original 3D models. A voxel is simply the 3D analog of a pixel. Just as a pixel is location on the plane with a some value a voxel is a location in 3D space with some value.

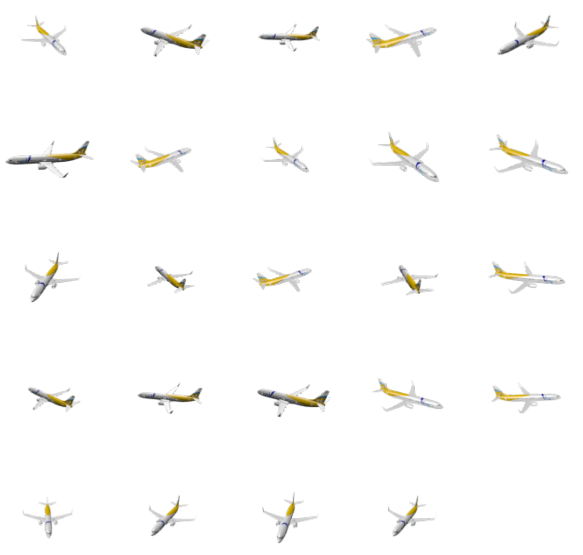


Figure 3.3: Images of ShapeNet 3D model

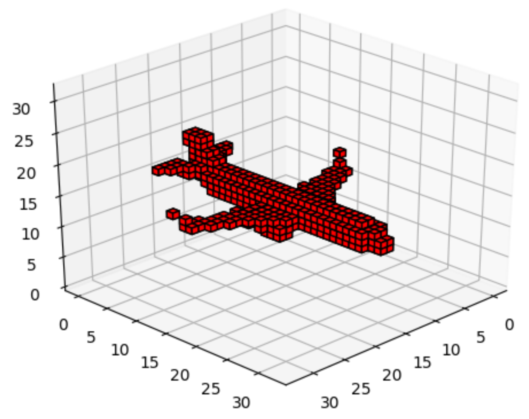


Figure 3.4: Target Voxelized Models

The input to the network is a batch of RGB image sequences of 3D models rendered from different viewpoints. This means that the input has a shape (batch-size,time step,height,width,channels). Below is a code snippet that takes a single 3D model and use a python package trimesh[1] to generate renders along the 3 major axis at arbitrary angles.

[1]https://github.com/mikedh/trimesh

```
mesh_obj = trimesh.load_mesh(mesh_path)
scene = mesh.scene()
for i in range(render_count):
    angle = math.radians(random.randint(15, 30))
    axis = random.choice([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    rotate = trimesh.transformations.rotation_matrix(angle, axis, scene.centroid)
    camera_old, _geometry = scene.graph['camera']
    camera_new = np.dot(camera_old, rotate)
    scene.graph['camera'] = camera_new
```

Despite developing code for generating renders from ShapeNet, we made use of Choy et al's dataset provided with their original paper which consists of renders of ShapeNet and voxelized versions of the 3D model. This was done to simplify the prepossessing step and to focus the project more on the training of the model instead of the data processing step. The results that you will see make use of Choy et al's preprocessed dataset instead of the raw ShapeNet. Choy et al offer their label voxels in .binovx format. We make use of the python package binvox. [2] to load and convert the voxel into a numpy format, which is then converted into a Tensorflow Tensor.

[2]https://github.com/dimatura/binvox-rw-py

3.4 Preprocess

Depending on the type of experiment being run the preprocessing step has slightly different steps. If the experiment being run makes use of 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 sequence of random length then `n_timesteps` must be set to a Tensorflow tensor that is evaluated. This is done with the rational of helping the network Here is a code excerpt below which is close to the original source but cleaned up for readability.

```
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 actually preprocessing step involves just dropping the alpha channel from the 4 color channel images, and crop the 137X137 image to 127X127 image at random. This is done with the rational that a slight resistance to slight perturbation.

```
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])
```

3.5 Encoder

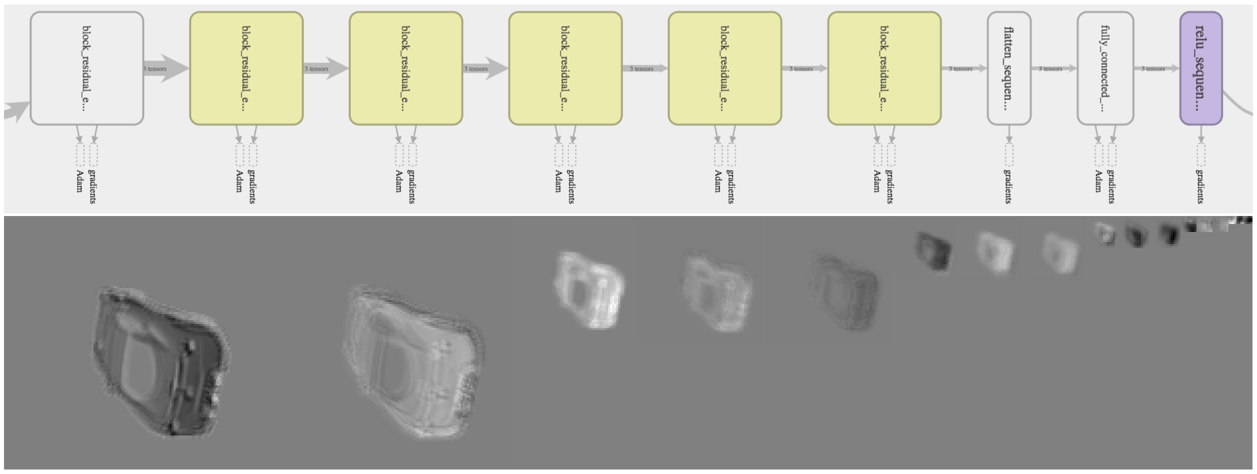


Figure 3.5: High level Graph of Encoder and the scaling down of the Feature maps generated by the encoder portion of the Network.

The input sequence of images are then taken in by the encoding portion of the network. The encoding network is just a serious of 2d convolutions, pooling operations and relu activation functions. but it can be seen as being constituted of several blocks of operations. Each block usually has one down-sampling operation combined with a certain number of convolution operation and relu activation functions. There are several variants of these blocks Choy et al implement a Simple and Residual Variant. We also implement a Simple and Residual variants in addition to a Dilated variant which involve dilated convolutions which are a variant of convolution that allow for more context.[19]

The encoder.py module consists of several functions the most important of which is `conv_seqeunce`. Consider the following code expert from the function which 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")
```

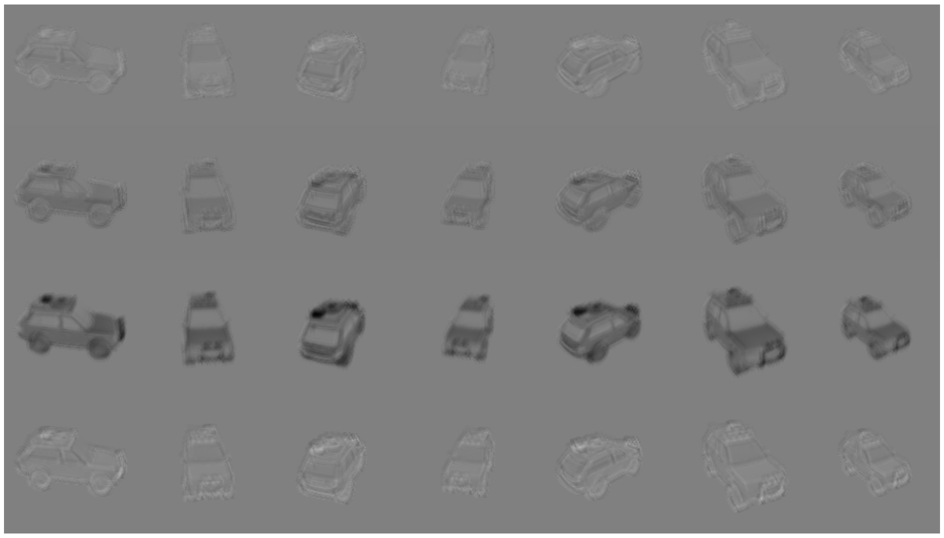


Figure 3.6: Feature Maps generated from 4 kernels on a sequence of 7 Images

3.6 Recurrent module

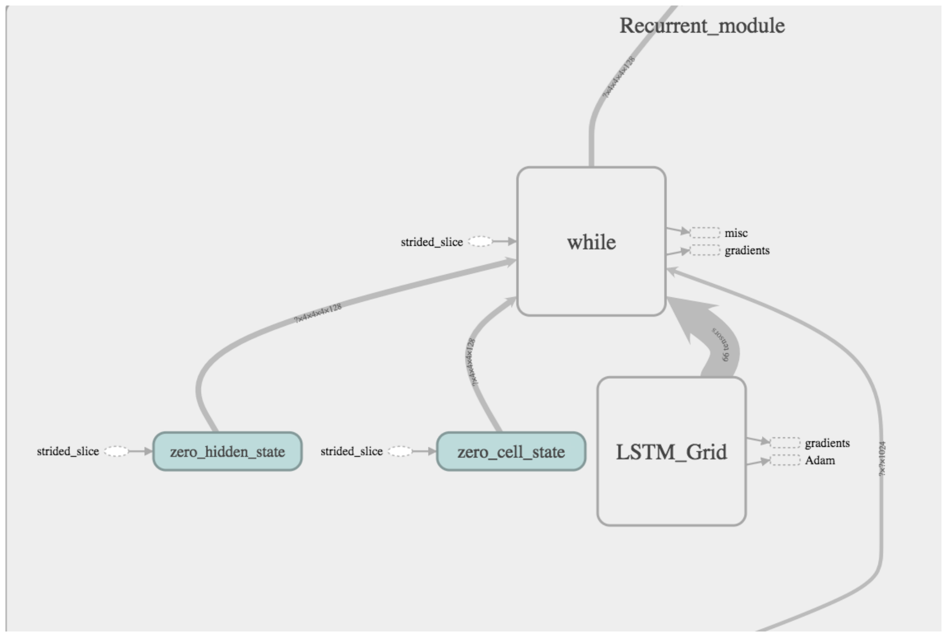


Figure 3.7: High level graph of Recurrent Module

The recurrent module is the core of the network. It is a 3D 4x4x4 grid of RNN cells with each RNN cell responsible for predicting whether a portion of space is occupied.

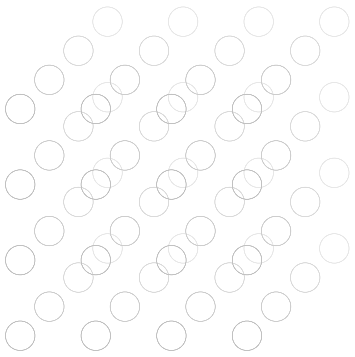


Figure 3.8: Grid of Cells

Although it is useful to visualize the network to develop an intuition of how it works to actually implement the network. We should look at the equations which represent this 3D grid of cells. Each unit takes an encoded low dimensional feature vector of size 1x1024.This input is then feed to the multiple gates in the RNN cells.Each unit each unit has a hidden state of shape 1x128. By combining the input and the hidden state of the previous time step a new hidden state is calculated. Each cell is able to 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 can be considered to be a 4 dimensional tensor with the shape 4x4x4x128.By convolving this hidden state by a 3D kernel of shape 3x3x3, we are able to share information between adjacent cells. This hidden state is then passed on to the decoder network. Below are the equations and their implementations for the LSTM variant of the Recurrent Module.

$$i_t = \sigma(W_i \cdot x_t + U_i * h_{t-1} + b_i)$$

$$f_t = \sigma(W_f \cdot x_t + U_f * h_{t-1} + b_f)$$

$$o_t = \sigma(W_o \cdot x_t + U_o * h_{t-1} + b_o)$$

$$s_t = f \circ s_{t-1} + i_t \circ \tanh(W_s \cdot x_t + U_s * h_{t-1} + b_s)$$

$$h_t = o_t \circ \tanh(s_t)$$

```
f_t = tf.sigmoid(                                                  # forget gate
    self.pre_activity(self.W[0], input_tensor, self.U[0], prev_hidden_state, self.b[0]))
i_t = tf.sigmoid(                                                  # input gate
    self.pre_activity(self.W[1], input_tensor, self.U[1], prev_hidden_state,  self.b[1]))
o_t = tf.sigmoid(                                                  # output gate
    self.pre_activity(self.W[2], input_tensor, self.U[2], prev_hidden_state, self.b[2]))
s_t_1 = f_t * prev_cell_state                                      # cell state
s_t_2 = i_t * tf.tanh(self.pre_activity(self.W[3], input_tensor,
                                  self.U[3], prev_hidden_state, self.b[3]))
s_t = s_t_1 + s_t_2
h_t = o_t*tf.tanh(s_t)
```

3.7 Decoder

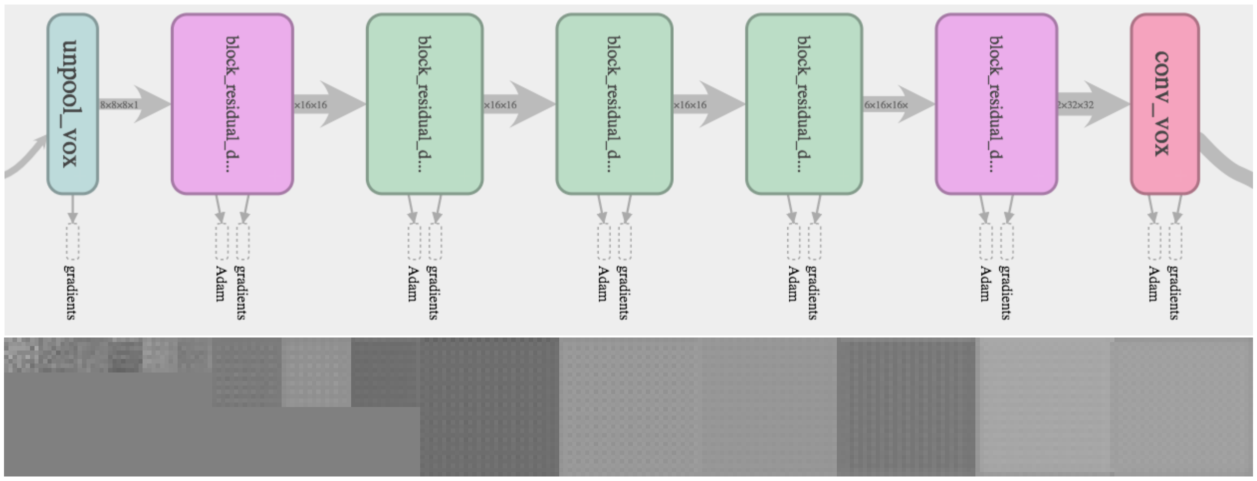
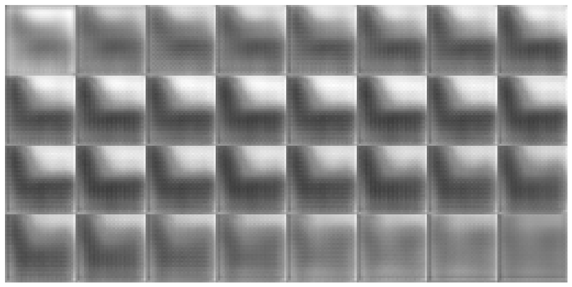


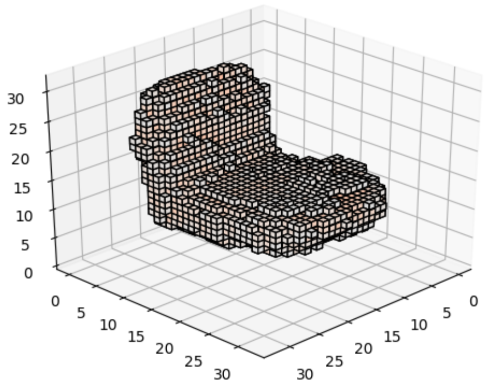
Figure 3.9: High level Graph of Decoder and Scale of Feature Voxel Slices generated by Decoder

The Decoding module is a series of 3D convolutions and unpoolings. 3D convolutions are implemented similar to the 2D case but instead of initialization a 2D kernel you initialize a 3D kernel. The image above shows slice of voxels as the decoder voxels progress through the network.

Before the final prediction, the hidden state of the recurrent unit is deconvolved to a 4 dimensional tensor with shape 32x32x32x2, This tensor is then converted to another 4 dimensional tensor of the same shape which represents the probability distribution over each of the voxels. This is done using the softmax. Let $\vec{x}$ be some vector in d dimensional space, the softmax function compress the vector such that it resemble a probability distribution, which means that each x_i is between 0 and 1 and the total sum of all the elements equals 1.

$$\sigma(x) = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{x_j}}$$


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



(b) 3D visualization of Prediction

Figure 3.10: Network output and Visualization

3.8 Loss

The loss used to train the network is a standard cross entropy loss applied at each voxel. Y is a one hot encoded voxel label.

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

An issue that we encountered when training the network was caused by this problematic piece of code

```
softmax = tf.nn.softmax(cur_tensor)

log_softmax = tf.log(softmax)
```

the issue with this code is if the values in cur_tensor get close enough to zero computer does not have the precision to tell the difference. We are using 32 bit floating point numbers but the but increasing by modifying the code as below we ensure that the values do not get small enough to be 0.

```
epsilon = 1e-10

softmax = tf.clip_by_value(tf.nn.softmax(cur_tensor), epsilon, 1-epsilon)

log_softmax = tf.log(softmax)
```

3.9 Visualization module

The visualization module as the name suggests house the functions that are responsible for generating most of the figure you have seen in this report. The most interesting code expert from that module is the voxel function which is responsible for generating the 3d voxel visualizations. It takes in the tensor representing the Bernoulli distribution of the prediction of voxel, and creates a representation where each voxel assigned a probability above a 0.5 is visualized via color. The higher the probability the redder a voxel gets.

```
vox = np.argmax(y_hat, axis=-1)
color = y_hat[:, :, :, 1]
if color is None or len(np.unique(color)) <= 2:
    color = 'red'
else:
    color_map = plt.get_cmap('coolwarm')
    color = color_map(color)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.voxels(vox, facecolors=color, edgecolor='k')
ax.view_init(30, 45)
```

3.10 Utility module

This is a simple class that serves as repository for varies miscellaneous functions which did not belong in the other modules. Most of them are mainly concerned with convenience and quality of life.

Chapter 4

Results

4.1 Overview

Several different versions of the network were trained achieving comparable performance with Choy et al. The different networks are constructed from different types of encoder, recurrent units,and decoder modules. Encoder and Decoder modules are divided into SIMPLE, DILATED, RESIDUAL. The SIMPLE encoder and decoder module use normal 2D and 3D convolution with pooling and unpooling to encoder the sequence of images and to decode the hidden state of the recurrent module. The DILATED encoder and decoder module use dilated convolutions to perform the encoding and decoding, however this module does still include pooling layers which makes them less interesting. May be in futher work we will construct a network variant with dilated kernels and no pooling operations. The RESIDUAL variant have skip connections which allows us to train much deeper nets.[9] We list the mean validation loss for the different types of network in the table below.

4.2 Setup

The different hyperparameters for each of our training session is stored in JSON files, which can be easily parsed for analysis. This system was not always in place, earlier in the process

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 4.1: Mean Validation Loss

the number of hyperparameters was quite limited consisting of learning rate, epoch count, and Batch Size. However overtime this number grew giving us a greater degree of control over the nature of the networks trained. This setup allowed us to streamline the multiple experiments that were to be conducted. Every network trained has the details of its hyperparameters saved in a JSON file called params.json. Below is a snippet of one of the JSON files which has the hyperparameters of one of the networks that we trained.

```
"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 4.2: Training Options

In addition to that another thing to note about the setup of the experiments is the hardware used to perform the training. The training sessions were trained on Amazon E2 P2 instance on a single NVIDIA Tesla K80 GPU with 12 GB of VRAM at a cost 0.90 dollars an hour. This difference in the hardware used for training lead to us using a smaller batchsize to that of Choy et al. Choy et al made use of a batch-size of 36 for the simple variant of their network and a batchsize of 24 for the residual variant of their network. Our reimplementation makes use of a batchsize of 8 and 16 for the simple and residual variant of the network respectively. We still achieved comparable results so it does not seem to have had an adverse effects on performance.

4.3 Training

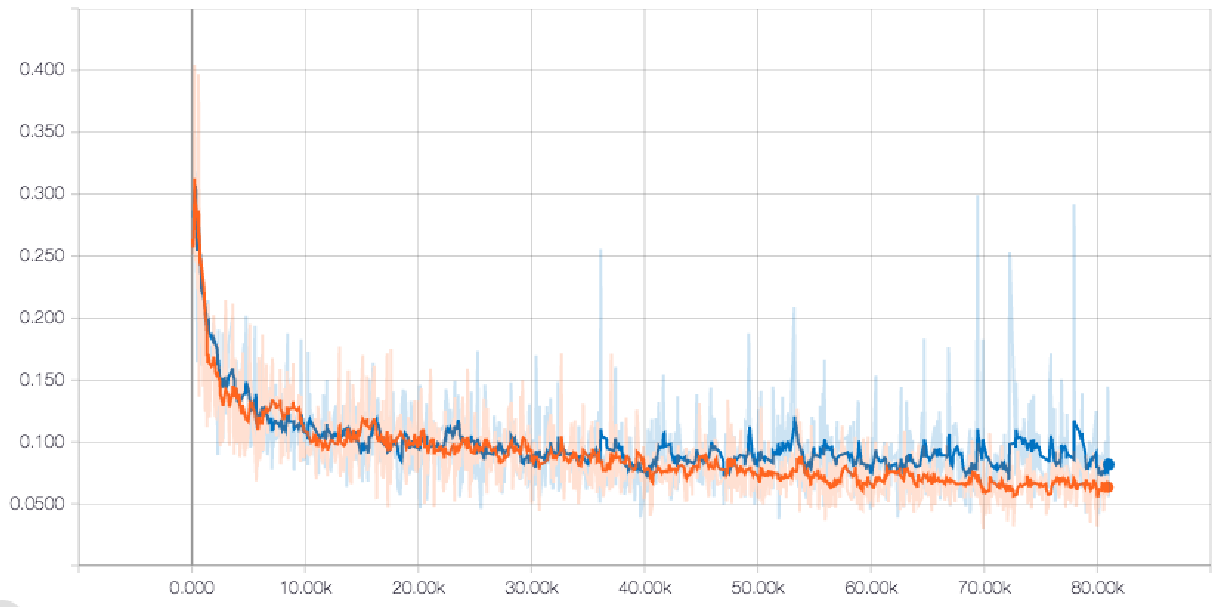


Figure 4.3: Longest Training Session

In order to determine the extent of time to which we should train the network. We performed a training session that went on for 40 Epochs. This took approximately 9 days to complete.This experiment provided insight into when the training error and the validation error start to diverge, which happens after the 30,000's iteration. This meant that subsequent experiments did not go past 30,000 iteration.

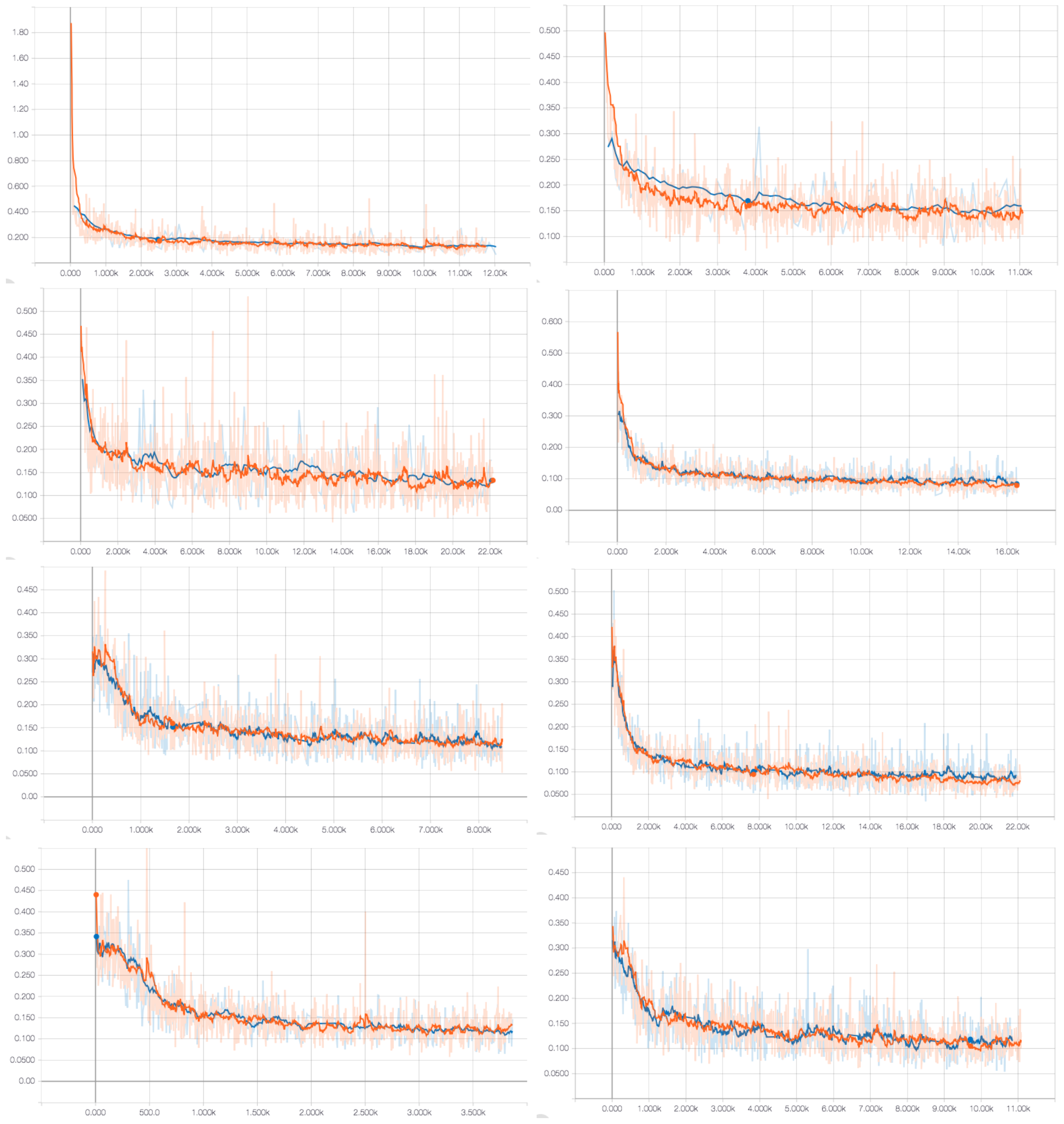
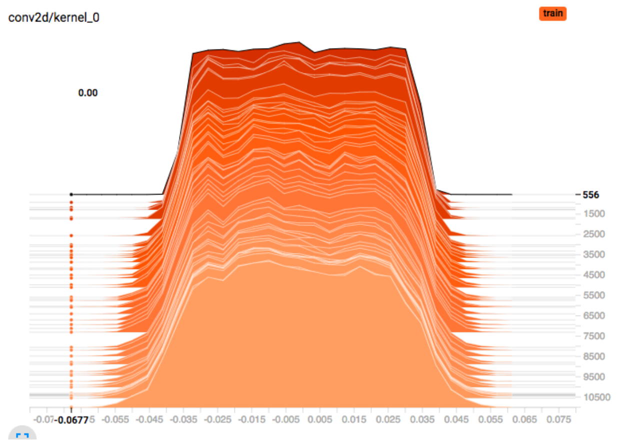


Figure 4.4: Training(orange) and Validation(blue) Loss decreasing during training session

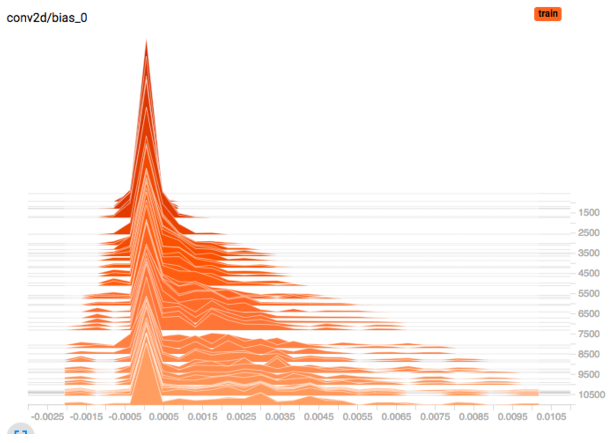
4.4 Initialization

When training deep network the initial weights have an impact on the rate at which the network trains.There are several heuristics,backed by empirical evidence,[7] that are used to initialize the weights of deep neural networks. We make us of a type of initialization called xavier initialization, the justification of which is that it is meant to keep the variances of the activation of different layers on the same scale. More formally, for layer i, we sample the weights from the following Uniform distribution.

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



(a) Histogram of kernel weights



(b) Histogram of bias weights

Figure 4.5: Change of Weight distribution during training

4.5 Optimizer

We make use two types of optimizer. Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (ADAM)[11]. SGD simply consists of calculating the gradient(g_t) on batch of example and then updating the parameters using those gradients (g_t) calculated on the examples.

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

ADAM is a more sophisticated form of parameter optimization that makes use of the gradient in a different way. ADAM is one of several optimization techniques that rely on the "*momentum*" of the parameters. Momentum is used in a similar sense it is used in Newtonian physics meaning that to change the direction of the gradient via an update, it is necessary to over come the momentum of the weights incurred in the previous updates. This is thought to mitigate oscillation.

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

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

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

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

4.6 Predictions

Below are few examples of the reconstruction of the network on examples from the validation set.From left to right, we see on the first column that there is a fairly accurate estimation of the target voxel for the airplane model to the left. That being said we still encounter some of the same problems encountered by Choy et al in the predictions of their network. FOr instance the second column of images shows a table prediction that is flawed in that it relies on a priori knowledge of what a table is in creating its reconstruction, which leads to a prediction of the table that is more inline with what a table looks like. Finally we examine the third column which shows the prediction of chair voxels. We see the network has trouble recreating the legs of the chair.

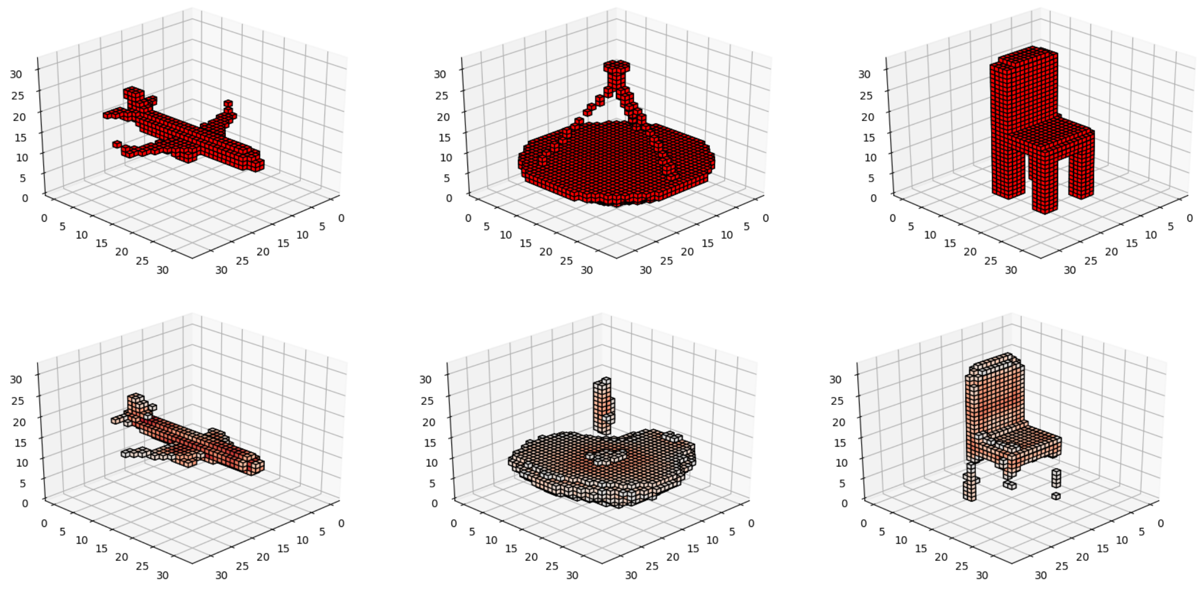
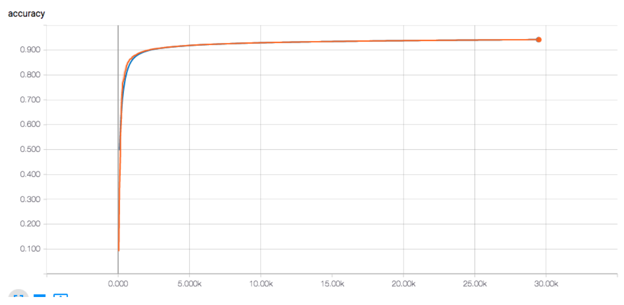


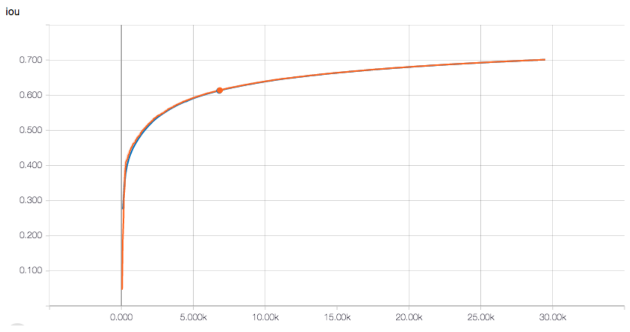
Figure 4.6: Target Voxelized Models and Network Prediction

4.7 Metrics

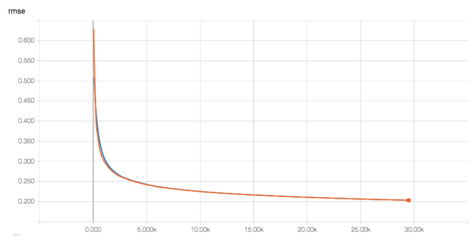
To measure the performance of our network, we make use of several metrics. The first one of which is the **accuracy**. Even though Accuracy is high but it can be slightly misleading metric because accuracy does not take into account how confident the network is in its prediction. A prediction can predict a voxel with a probability of 0.6 or 0.9 and the accuracy would still be the same. Next metric we use is the root mean squared(**RMS**) which is a metric usually used to measure error on regression problems. However if we choose to treat the one hot encoded label as the target and the probability distribution predicted by the network before the final prediction then it is possible to us RMS as a metric, and we have found that it a better metric for viewing training as it does not saturate as quickly as the



(a) Accuracy



(b) Intersection over Union (IOU)



(c) Root mean square (RMS)

Figure 4.7: Plots of the different metrics

accuracy or loss. Finally we consider the metric used by Choy et al, the Intersection over Union (**IOU**), which 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. Another way of looking at this is in-terms of as the ratio of true positives, i.e instances where the prediction and label overlap to the total sum of true positive, false positive and false negative.

4.8 Concluding remarks and Future work

There are several ways this project can be extended. There is the natural route of training the networks with more parameters. A point of experimentation could be the size of the feature vectors feed to the recurrent unit. This would may be allow the network to capture finer details such as the chair legs.

This project is further demonstration of the malleability of neural networks and their ability to adeptly handle problems in various domains.

Bibliography

[1] Martfffdfffdn Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: (). URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

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

[3] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoderfffdfffdfffd-Decoder for Statistical Machine Translation". In: (). URL: https://arxiv.org/pdf/1406.1078v3.pdf.

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

[5] Peter. Dayan and L. F. Abbott. *Theoretical neuroscience : computational and mathematical modeling of neural systems*. Massachusetts Institute of Technology Press, 2001, p. 460. ISBN: 9780262041997.

[6] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: (Mar. 2016). URL: http://arxiv.org/abs/1603.07285.

[7] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: (). URL: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: (Dec. 2015). URL: `https://arxiv.org/abs/1512.03385`.

[10] Sepp Hochreiter and Jj Urgen Schmidhuber. “Long Short-term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. URL: `http://www7.informatik.tu-muenchen.de/~hochreit%20http://www7.informatik.tu-muenchen.de/~hochreit`.

[11] Diederik P Kingma and Jimmy Lei Ba. “ADAM: A Method for Stochastic Optimization”. In: (). URL: `https://arxiv.org/pdf/1412.6980.pdf`.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: (). URL: `https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf`.

[13] Yann Lecun et al. “Gradient-Based Learning Applied to Document Recognition RS-SVM Reduced-set support vector method. SDNN Space displacement neural network. SVM Support vector method. TDNN Time delay neural network. V-SVM Virtual support vector method”. In: *PROC. OF THE IEEE* (1998). URL: `http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf`.

[14] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: (). URL: `https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf`.

[15] Rami Al-Rfou et al. “Theano: A Python framework for fast computation of mathematical expressions”. In: (2016). URL: `https://arxiv.org/pdf/1605.02688.pdf`.

[16] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: (Sept. 2014). URL: `https://arxiv.org/abs/1409.0575`.

[17] Stuart J. (Stuart Jonathan) Russell, Peter. Norvig, and Ernest. Davis. *Artificial intelligence : a modern approach*. Prentice Hall, 2010, p. 1132. ISBN: 0136042597.

[18] Richard Szeliski. *Computer vision : algorithms and applications.* Springer, 2011, p. 812. ISBN: 9781848829350.

[19] Fisher Yu and Vladlen Koltun. "Multi-scale Context Aggregation By Dilated Convolutions". In: (). URL: `https://arxiv.org/pdf/1511.07122.pdf`.