Transferring Object 6D Pose Estimation From Simulation to Real World

Jacob Hoffman (jacobmh), Ethan Li (lietk12), Antonio Tan-Torres (tantonio)

Motivation & Challenges

A crucial problem in applying deep learning to robotic object manipulation tasks is the high cost, in both time and resources, of collecting enough real-world robotics data to train deep learning systems. On the other hand, while robotic simulators promise a low-cost means of generating large data sets, models trained on simulated data do not perform well when confronted with real-world data. To address this gap, we collaborated with Simon Kalouche to investigate end-to-end deep learning for 3D object pose estimation by training on low-cost simulated data with domain randomization and transferring to a real-world dataset.

Our task involves taking a stock image of our target object, finding that object in a cluttered environment, and returning the object's 3D position and pose. Because our downstream application requires use in real-world scenarios, our challenge was to develop a system which would transfer effectively from simulated training data to real-world test data on previously unseen objects.

Data Sets & Data Collection

Our mentor, Simon, has given us a starting dataset of 1000 labelled scenes, and each scene has 4 different camera perspectives, so our dataset contains 4000 labeled images in total. Because each scene has multiple labeled objects, this provides 58,800 unique examples. This dataset comes from a simulated environment in Unity3D and consists of RGB images of a scene (a table with objects on it) and a corresponding depth map. There is also an object library for 55 objects which consists of an RGB image of the object and a corresponding depth map. Each image is of size 299x299 (with 3 channels for RGB and 1 channel for depth). The corresponding labels are the 3D positions and orientations of the objects (x,y,z,roll,pitch,yaw), and the id of each object in the scene.

Figure 1: An example from the dataset. From left to right: the RGB image of the scene with a boot in it; the depth image of the scene; the RGB image of a query object (the boot); and the depth image of the query object.

We also used the Amazon Picking Challenge's "Shelf & Tote" benchmark dataset (2) of real-world images of objects in a scene to see how our algorithm trained in a simulated environment performs on real-world scene images. This dataset is labeled with the objects and their corresponding 4x4 rigid transformation matrices which we use to extract the x,y,z,roll,pitch,yaw. Extracting the object pose involves transforming the object-to-world transformation matrix to the camera reference system (using the camera extrinsic matrix), and then computing the roll, pitch, yaw from the transformed rotation and the x, y, z from the transformed translation. We filter out object examples as errors due to camera matrix noise where the projected x, y, z into the image falls out of bounds or in the background (using the segmentation images from the dataset). The dataset also provides RGB and RGBD images of the objects in many poses, which we will use as the source for input query stock images (Figure 2). The

dataset provides the object in a scene, and the corresponding object mask. We use this to extract the real-world image and depth of the object in the same format as the simulation dataset (Figure 2).

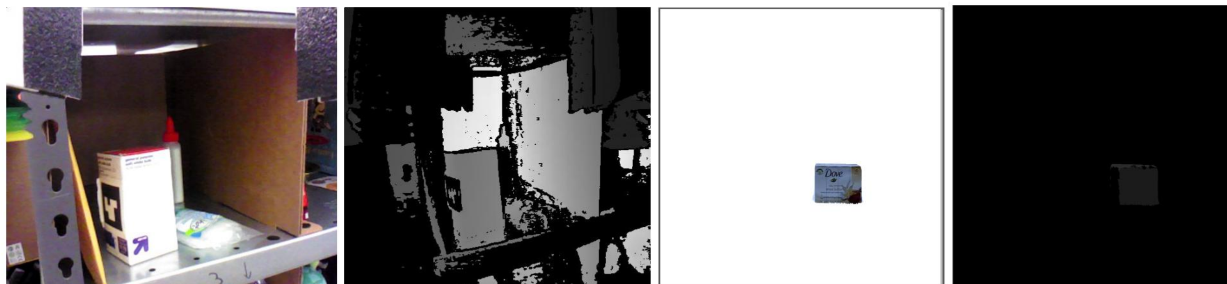


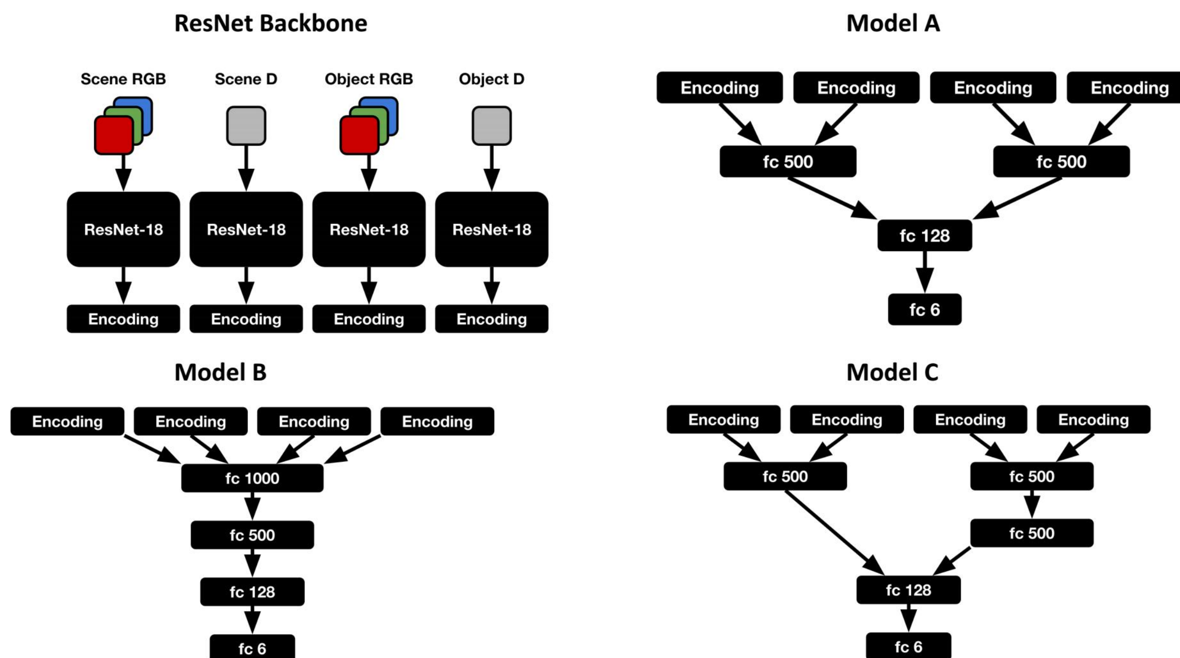
Figure 2: An example from the real dataset. From left to right: the RGB image of the scene; the depth image of the scene; the RGB image of a query object; and the depth image of the query object.

Models

We evaluated five models for this task to test different architectural factors for each model (Figure 3). Because our goal was to learn features from simulated data to predict the 3-D position and orientation of objects in real-world scenes, we chose a transfer learning framework for our model. First, we pass our four input images (scene rgb, scene depth, object rgb, and object depth) through a Res-Net trained in the ImageNet dataset. We froze all the Res-Net's weights, removed the softmax classification layer, and extracted the output of the last fully connected layer of the network, giving us (1000 x 1) vector encodings for each of our images. These four (1000 x 1) encodings serve as the input to the rest of our architecture.

Without prior intuition or literature to inform our downstream architecture for this use of transfer learning to predict 3-D position and pose, we decided to run a series of experiments to test a spectrum of architectures. For simplicity we've labelled our architectures A – E. With architectures A, C, and D we sought to test the approach of learning features from the scene and object independently. For model C we added a second layer to the scene branch to test our hypothesis that learning from the scene required added complexity, while for model D we added another layer after the scene branch and object branch merge. Architecture B was the simplest architecture that concatenated all four image encodings before passing them through a series of fully connected layers.

After constructing architectures A – D, we tested them on a small subset of our dataset to see how well we could get the models to fit to the training data. We found that architecture B performed best and decided to create architecture E to see if we could overfit even more to the training set. While adding more layers generally leads to more overfitting, model E did not overfit more than model B. We think this had to do with our initial mistaken use of an abnormally high dropout rate.



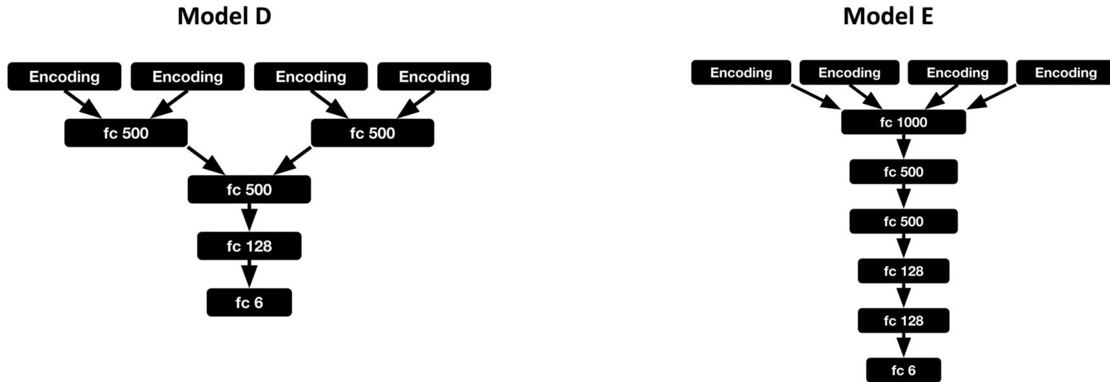


Figure 3: The model architectures evaluated.

We ran these experiments using 3.5% of the full simulated dataset, to determine which model we would train and evaluate on the full dataset (Figure 4). In these experiments, we trained a model for 100 epochs with a mini-batch size of 10 on the simulated train set and evaluated on the simulated train-dev set. The loss is the sum of the position and orientation losses, which are the mean squared errors in position estimation and orientation estimation, respectively. We chose to proceed with Model B because it achieved the lowest bias out of the models on the reduced-size dataset.

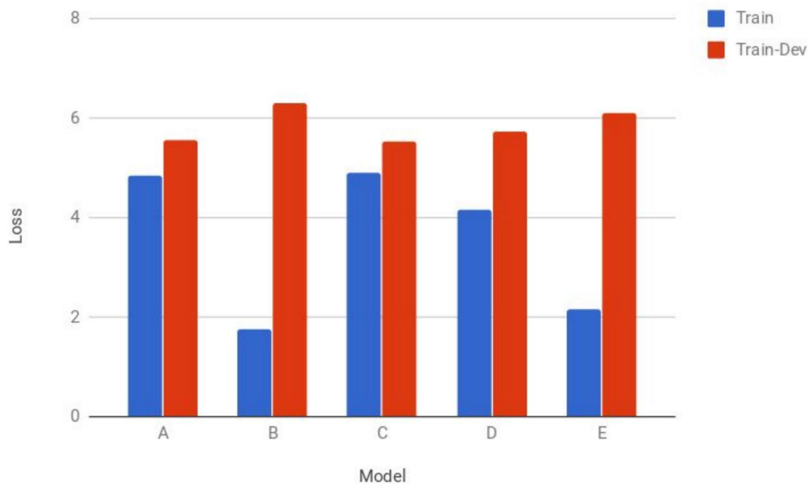


Figure 4: Train and train-dev losses for architecture selection experiment.

Training and Results

With our architecture selected to minimize bias, our next goal was to reduce the variance of this model. To do this, we employed weight decay. We evaluated three values for the weight decay hyperparameter λ (10^{-1}, 10^{-2}, 10^{-3}) on a slightly larger reduced-size dataset, namely 10% of the full dataset (Table 1). In each case, we trained for 50 epochs with a mini-batch size of 40. λ = 0.1 appeared to totally eliminate overfitting on this reduced-size dataset at the expense of fitting to the train set, while λ = 0.01 produced moderate reduction in variance and λ = 0.001 did not appear to reduce variance at all. We chose λ = 0.01 because it seemed to balance bias and variance reasonably well such that training on the full dataset could further reduce variance to acceptable levels.

λ	Train Loss	Train-Dev Loss
0.1	5.14	5.18
0.01	4.22	6.97
0.001	1.74	6.97

Table 1: Train and train-dev losses for weight decay hyperparameter experiment. The value chosen for subsequent experiments, 0.01, is highlighted in bold.

We compared the results of training Model B on the full dataset with weight decay of λ = 0.01 to training with no weight decay (Figure 5). In each case, we trained for 70 epochs with a mini-batch size of 40 on the simulated train set and tested on the simulated train-dev set and the real dev set. Without weight decay, Model B severely overfits to the training set, and train-dev loss increases and dev loss is variable but high. With weight decay, Model B does not seem to fit much to the training set, train-dev loss remains stable, and dev loss is lower and exhibits less variability. Thus, weight decay appears to improve the model's generalization from simulated data to real data.

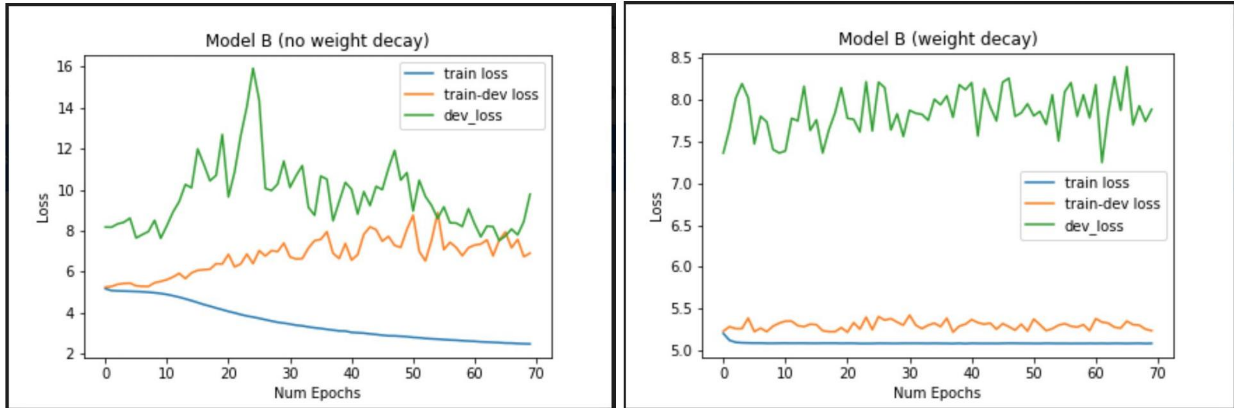


Figure 5: Loss curves for training on the full dataset without weight decay (left) and with weight decay of λ = 0.01 (right).

We then examined the final losses and corresponding root mean squared errors of position and orientation predictions from Model B trained on the full set with weight decay (Table 2). Position estimation errors increase between 50% and 100% with transfer to the real dataset, while orientation estimation errors increase much less. These differences in pose estimation errors suggest that the model is unable to generalize to previously unseen objects. Overall, these errors are too large to be practically useful for either simulated or real data.

λ	Position Loss	Orientation Loss	Total Loss	Position RMSE (m)	Orientation RMSE (deg)
Train	0.783	4.15	4.94	0.885	117
Train-Dev	0.845	4.42	5.27	0.919	120
Dev	2.34	5.99	8.33	1.53	140
Test	2.17	5.96	8.13	1.47	140

Table 2: Position and orientation losses for final model.

Analysis & Conclusion

Our results suggest two key areas of challenges and adjustments required to successfully train an end-to-end network for object pose estimation. Most importantly, the high losses reflect the difficulty of our specified task and indicate that our approach of training a straightforward feedforward neural network may not be appropriate.

First, allowing only a single viewpoint of a query object may not provide sufficient information about the geometry of the object. For example, the boot in Figure 1 is shown from a frontal view in the query object image, while it is visible from a side view in the scene image. A more promising approach may be to provide a full range of viewpoints of the query object in order to allow the network to learn to interpolate between orientations and to recognize objects from different viewpoints in order to identify them correctly.

Second, the end-to-end pose estimation task may require more sophisticated or multi-stage architectures. For example, (3) describes a specialized deep neural network architecture trained on three separate tasks as part of a pipeline, and whose predictions are refined with the Iterative Closest Point algorithm. An open question from our project remains as to whether transfer learning by applying a fixed CNN pre-trained on ImageNet to our RGBD images can be usefully applied to simplify the design of such networks.

Contributions

Whole Team: coming up with the model architectures to experiment with, code debugging, error analysis
Jacob Hoffman: coding up the models and building improved data parser
Antonio Tan-Torres: coding the amazon data parser and preparing the real-world dataset by extracting labels
Ethan Li: running the experiments and implementing changes to models
Simon Kalouche: Provided the simulation dataset and initial data parser, and served as a project mentor

Code Repo

You can find our code repository here: https://github.com/jacobmh1177/CS230_Visuomotor_Learning

References

1. J Tobin, R Fong, A Ray, J Schneider, W Zaremba, P Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. IROS 2017.
2. A Zeng, KT Yu, S Song, D Suo, E Walker Jr., A Rodriguez, J Xiao. Multi-view Self-supervised Deep Learning for 6D Pose Estimation in the Amazon Picking Challenge. ICRA 2017.
3. Y Xiang, T Schmidt, V Narayanan, D Fox. PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes