Project: 3D Pose Estimation from 2D Projections

Richard (Alex) Showalter-Bucher
Benjamin Yu

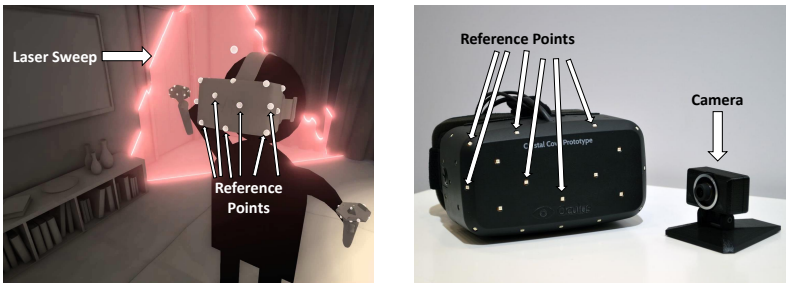
I. Introduction

In this project, we explored the application of machine learning to the problem of estimating an object's 3D position and orientation, known as a pose, from a 2D projection. Being able to accurately perform this estimation has applications in several disciplines including Computer Vision, Photogrammetry, and Robotics [1]. A notable application is object tracking in virtual reality systems (VR), such as the HTC Vive and Oculus Rift. Both of these systems estimate 2D angular locations of reference points on the objects being tracked to determine the object's pose.

For a system like the HTC Vive, this measurement is done directly via time of arrival of laser sweeps on reference points which are photodiodes (Fig 6a). In the case of the Oculus Rift, a camera is used to identify predetermined reference points (IR LEDs) before estimating the 2D angular location. This is done by blinking each LED in a specific pattern which is observed by a camera (Fig 6b). After the reference points are identified they are centroided to determine their angular location on the focal plane. Once the 2D angular locations are estimated a Perspective N-Point method is applied to estimate the object's pose. In the next subsection we discuss what this method is in more detail.

A. Perspective N-Point (PnP) Method

The problem of estimating an object's 3D pose from 2D projections of known reference points is referred to as the "Perspective N-Point" problem (PnP). Common approaches to this problem involve directly or iteratively solving a system of equations [2]. While these approaches can be very precise with the well defined reference points (minimum of 4 non co-planar points for a non-ambiguous solution), there are usually trade-offs between accuracy and compute time. Direct solvers of these system of equations can be performed computationally fast, but may not be robustly accurate. On the other hand, iterative methods tend to be slower but more robust. A well-formed hybrid of these solvers can be both fast and robustly accurate for well defined points. One of these methods was used to baseline our performance to compare with the results of our machine learning algorithms. This baseline method is discussed in more detail in the Baseline Performance EPnP section.



(a) HTC Vive [4] (b) Oculus Rift [5]

Fig. 1: Left: HTC Vive's laser sweep with reference points. Right: Oculus Rift's camera with reference points

B. Motivation for Applying Machine Learning

We wanted to implement and test several machine learning techniques to see how they fair with handling the pose estimation problem. There were three different types of techniques we used: SVM Regression, Feed-Forward Neural Networks (FFNN), and Convolutional Neural Networks(CNN).

We used two different datasets to train our models: raw RGB image data and 2D projections of geometric reference points from a posed object. These data types are similar to the raw and processed data, respectively, of the Oculus Rift system.

Using the reference point data, we were curious if we could match the performance of the PnP method and, if we could obtain similar performance using many points, determine if pose regression using machine learning was robust using less than 4 points (the limitation for non-ambiguous solutions using PnP). Due to the sparsity of this type of dataset, we did not train a CNN with it; we tested this type of data only on the SVM and FFNN.

For the raw image data, we were curious if we could train our models such that they could directly estimate the poses of objects without the need to perform multiple processing steps such as reference point identification and centroiding. If this method worked robustly, then tracked objects may be able to remove the need to explicitly define reference points which would allow for a more cost effective and general solution to the pose estimation problem.

We also note the use of PnP algorithms require that the reference points, no matter how their locations are measured, are only useful if their exact relative geometry to one another is known. That is, the PnP algorithms are given knowledge of what the object looks like. In the case of our machine learning algorithms, the geometry of the objects is not specified, and must be learned during the training phase. The challenge of tracking objects would then be free of the problem of getting exact geometric measurements.

Unfortunately, we found that none of our techniques, as implemented by us, met the performance of the PnP. We were hoping to get comparable results to PnP methods but found that the estimations from all datasets were significantly worse

than the PnP. We discuss this in more detail in each technique's respective section.

In the next section we discuss our dataset and how it was generated.

II. Data Set Generation

To test our machine learning and baseline PnP algorithm we generated two datasets. Each dataset contains versions of a cube, a cone, and a sphere in 500 randomly generated poses. (So each dataset contains 1500 total different samples.) Each pose contains a 3D translation and then yaw, pitch, and roll rotations.

A. Geometric Data Points

The first dataset consists of the projected 2D locations of geometric reference points from the various posed shapes. We defined a set of 3D points on the surfaces of the cube, the cone, and the sphere. These base shapes were centered around the origin of our coordinate system. Then, a randomly generated translation and rotation (a pose) was applied to the points. Finally, the points were projected onto a 2D image using a sixty degree field of view (FOV).

The geometry of the world was set in a right-handed coordinate system. The camera is at the origin and points along the negative z-axis, so that the x-axis points to the right, and the the y-axis points up (from the point of view of the camera) (Fig 2).

The base geometry of each shape was chosen to to have a characteristic dimension of 1.0; the sphere had a diameter of 1.0, the cone had a diameter of 1.0 at its base, and a height of 1.0, the cube had sides of length 1.0. The rotations were allowed to vary uniformly throughout the space. Yaw varied between 0 and 360 degrees. Pitch varied between positive 90 and negative 90 degrees. Roll varied between 0 and 180 degrees. Those spans are enough to specify any 3D rotation. The x and y of the coordinates of the translation were uniformly distributed between -1.5 and +1.5. The z coordinated varied between -4.33 and -1.7320. This z range placed our shapes at a distance from the camera so that they took up roughly one fifth to a third of the total FOV. We note it was possible for a shape to be half off the image focal plane.

The math for applying our random pose, and then projecting our points onto a 2D image follow the vertex processing of OpenGL 1.0. We start with our points from the base shape. These are converted to homogeneous coordinates, which are 4-vectors of the form:

$$\vec{r} = (x, y, z, 1)^T \tag{1}$$

Where x, y, and z are the traditional 3D coordinates. The benefit of this coordinate system is that both translations and rotations can be written as a matrix multiplication. We create one matrix for the translation, as well as for each of the yaw, pitch, and roll rotations. (T, Y, P, and R matrices.) Finally, we calculate a projection matrix (C) which defines a frustum (Fig 2) that represents the field of view of the camera.

We then calculate the final locations of the points:

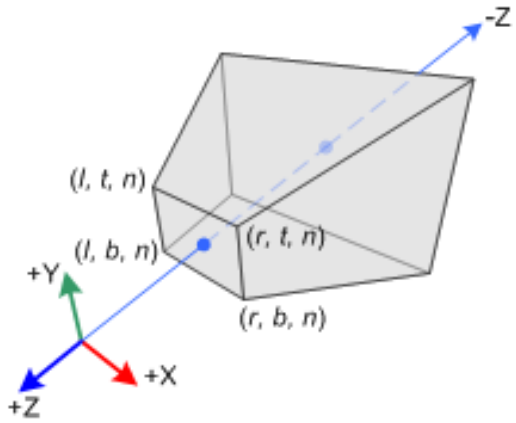


Fig. 2: The frustum defined by the projection matrix, simulating the view of a camera. [3]

$$\vec{r}_{posed} = C \cdot T \cdot Y \cdot P \cdot R \cdot \vec{r} \tag{2}$$

We normalize these coordinates, by dividing each element of each vector by the 4th component of said vector. These normalized coordinates have the property that every point inside that frustum has x, y, and z coordinates in [-1, 1]. Anything else is out of the FOV.

We also calculated the occlusion of each of these points. The final points have z-coordinates, and by defining a set of triangles on the surface of each shape, it is possible to see if a point is blocked by some surface segment of the shape. In the end we record only the location of the projected points on the x-y plane and if the point was occluded.

B. Image Data Set

Our second dataset consisted of full bitmap images of posed cubes, cones, and spheres on a black background. We used the GLUT library to speed our creation of these scenes with OpenGL. Like before, we created 500 random poses of each of the three shapes. The math implemented is the same as in the geometric point case, but of course, OpenGL will render a full scene after transforming the shape vertices.

We applied a texture to our shapes. We chose a high-res version of the Earth from NASA public release. We figured this texture was interesting and also asymmetric. Lack of symmetry is beneficial since, for example, it's possible to determine which face of the cube you are observing; each face will have a unique part of the Earth on it.

We rendered 64x64 24-bit RGB images and saved them in BMP format. The images are saved unnormalized, but we applied normalization in our algorithms. Samples are shown in Fig. 3, 4, and 5.

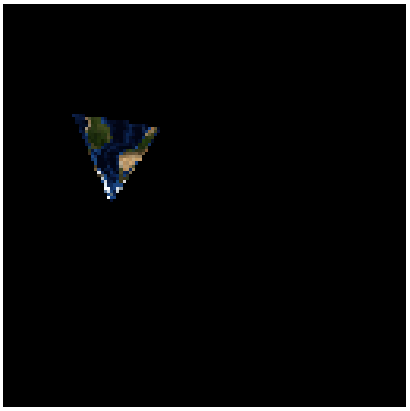


Fig. 3: A sample of a randomly posed cone in our image dataset.

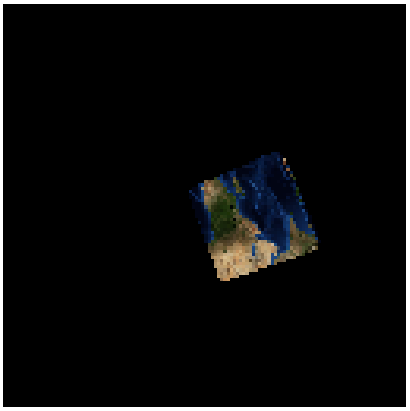


Fig. 4: A sample of a randomly posed cone in our image dataset.

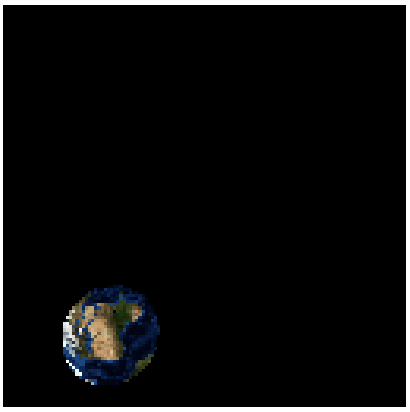


Fig. 5: A sample of a randomly posed cone in our image dataset.

III. Baseline Performance of EPnP

As a baseline to the performance of the 3D estimation, we used a method referred to as Efficient PnP (EPnP). The implementation of the solver used was included as part of a paper by Lepetit et al [2]. This method is a hybrid method between a direct and iterative solver. In the first part of the algorithm, a weighted sum of four virtual points are calculated from N observed reference points. These four virtual points are then used in the direct solver which gives an initial guess of the solution. This guess is then fed into a Gauss-Newton iterative solver which reduces the overall errors in the guess. Overall this implementation's computational cost is linear in terms of the number of reference points.

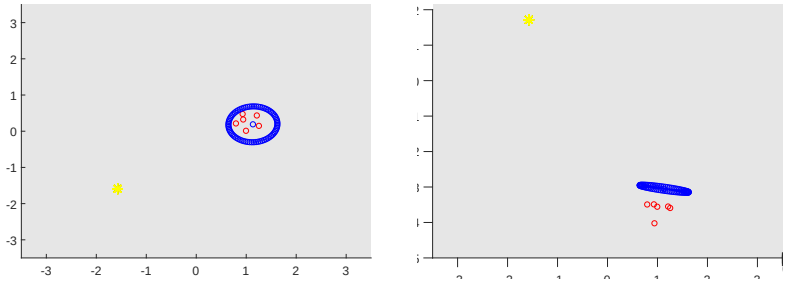
Results of this method for our geometric reference point dataset are shown in Table I. We show the RMS of the norms of the residuals for the collection of points. Overall the performance of this calculation is robust when four non co-planar points are observed. A notable issue with this method is seen in cases where the points are co-planar. This is clearly shown in Figure 6 for the case where the cone's base occludes the non co-planar reference points. The error from this image corresponds to the maximum error observed for the cases where occlusion is included.

Other than these outliers cases this method is great for cases where the reference points are well defined and non-coplanar. The order of magnitude of the errors suggests both the angle and translation estimations are incredibly accurate.

It should be noted that this method also has a strict assumption that the reference points are on a rigid body. This is due to the fact that the 3D geometry of the reference points must be known beforehand to feed to the algorithms.

TABLE I: EPnP Performance

Cone Error				
Points:	All	With Occlusion	4	3
RMS	9.4E-13	1.2	3.8E-1	1.6
MAX	6.5E-12	26.8	6.2	8.1
MIN	1.5E-14	1.0E-14	4.3E-15	4.9E-4
Cube Error				
Points:	All	Non-Occluded	4	3
RMS	2.9E-13	1.9E-13	4.8E-1	1.4
MAX	3.2E-12	1.7E-12	7.7	7.9
MIN	4.9E-15	3.2E-15	2.6E-15	3.5E-3
Sphere Error				
Points:	All	Non-Occluded	4	3
RMS	9.7E-13	5.7E-13	5.2E-1	1.5
MAX	9.1E-12	3.8E-12	7.3	8.1
MIN	6.0E-15	6.3E-15	2.0E-15	2.5E-3



(a) View from Camera Position (b) View From Top

Fig. 6: Figure shows the reference points of the cone which correspond to the max error in the case with occlusion. Blue circles represent non-occluded reference points. Red circles represent reference points occluded from the camera. Yellow star represents the estimated pose. Note that all point estimations are about the same position due to the poor estimation capability for co-planar reference points.

IV. SVM Regression

A. Implementation

We implemented our own version of the regression support vector machine (SVM) in Matlab using its quadratic programming routines. The regression SVM is formulated as the optimization of the ϵ - insensitive loss function:

$$\begin{aligned} min\ \frac{1}{2}||\omega||^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) \\ y_i - f(x_i,\omega) \leq \epsilon + \xi_i^* \\ f(x_i,\omega) - y_i \leq \epsilon + \xi_i \\ \xi_i, \xi_i^* \geq 0\ \ \forall\ \ i \end{aligned} \tag{3}$$

Where y is the vector of true values, $f(x,\omega)$ is the regression, ξ and ξ^* are slack variables, and C, and ϵ are hyperparameters.

The Lagrangian dual formulation of this problem is

$$\begin{aligned} max -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}(\alpha_n - \alpha_n^*)(\alpha_m - \alpha_m^*)k(x_n,x_m) \\ -\epsilon\sum_{n=1}^{N}(\alpha_n + \alpha_n^*) + \sum_{n=1}^{N}(\alpha_n - \alpha_n^*)y_n \\ 0 \leq \alpha_i, \alpha_i^* \leq C \end{aligned} \tag{4}$$

We used the quadprog routine of Matlab to optimize this problem. Matlab optimizes:

$$\begin{aligned} \min_x \frac{1}{2}x^T H x + f^T \\ Ax \leq b \\ A_{eq}x = b_{eq} \\ b_{lower} \leq x \leq b_{upper} \end{aligned} \tag{5}$$

And we set:

$$H = \begin{bmatrix} K & -K \\ -K & K \end{bmatrix} \tag{6}$$

$$f = \begin{bmatrix} \epsilon - y \\ \epsilon + y \end{bmatrix} \tag{7}$$

$$A_{eq} = \begin{bmatrix} 1 \\ \dots \\ -1 \\ \dots \end{bmatrix} \tag{8}$$

$$b_{eq} = 0 \tag{9}$$

$$b_{lower} = 0\ \ b_{upper} = C \tag{10}$$

Where K is the Gram matrix.

We made the slightly surprising discovery that Matlab quadratic programming doesn't seem to to support optimizing over multiple dimensions at once. Thus, for our investigations into the regression SVM, we actually created 6 SVMs (X translation, Y translation, Z translation, Yaw, Pitch, Roll) for each of the 3 different test shapes.

We used a Gaussian Radial Basis kernel for both our geometric reference point and the image dataset. The bitmaps were simply changed into a 64x64x3 length vector (which we normalized to $[-0.5, 0.5]$) and passed into the kernel as usual. For the N points of each shape in the reference point dataset, we formed a vector of $2N$ dimensions. (The X, and Y coordinates of each 2D projected point.) For example, we had 14 reference points in our cube poses, and thus formed each 2D projection into a length 28 vector.

This formulation on the geometric reference point set ignores the effects of occlusion. Hypothetically, to best match the problem, we should not provide information from points that are occluded. We did attempt to define a kernel based on the GRBK which only considered points that were not occluded in both samples. (All other entries corresponded to points occluded in either sample were set to zero.) Unfortunately, Matlab quickly informed us using this "kernel" made the problem non-convex, and we suspect this operation is not actually positive semi-definite.

We divided our 500 poses from each shape into a training set of 250 samples, a validation set of 125 samples, and a testing set of 125 samples. Our figure of merit was the root mean square value of the residuals between the real translation or rotation coordinates of our data, and those predicted by our SVM. (Recall, we trained a different SVM for each of the 6 dimensions of the pose.)

For both our datasets we have three hyperparameters to optimize for: C, ϵ, and σ (the bandwidth of the kernel). We optimized our hyperparameters by using an empirically based coordinate descent algorithm. We varied our hyperparameters over:

$$\begin{aligned} C \in [10^6 ... 10^{10}] \\ \sigma \in [10^{-2} ... 10^2] \\ \epsilon \in [10^{-3} ... 10^3] \end{aligned}$$

where 100 test points were in each parameter sweep, spread evenly logarithmically.

In turn, we optimized each of the hyperparameters keeping the others constant. We trained an SVM for each of the 100 possible values for the hyperparameter in question, and then set it to the value that produced the lowest validation error. We then continued on with the next hyperparameter, cycling through until validation error stayed constant through a cycle of all the hyperparameters.

B. Performance on Reference Point Data

TABLE II: SVM Performance on the Reference Point Dataset

Shape	Dimension	Train E.	Val E.	Test E
Cone	X	0.028562	0.097226	1.203973
Cone	Y	0.017899	0.096197	1.324511
Cone	Z	0.074570	0.240741	0.673912
Cone	Yaw	65.554484	64.610879	143.902763
Cone	Pitch	9.679785	19.332789	67.931383
Cone	Roll	22.610235	28.204543	63.173690
Cube	X	0.006369	0.051618	1.246132
Cube	Y	0.010817	0.038766	1.148181
Cube	Z	0.050916	0.115780	0.719099
Cube	Yaw	54.671545	69.127465	128.892843
Cube	Pitch	11.516737	15.104284	70.884934
Cube	Roll	22.665185	27.860259	64.804574
Sphere	X	0.031519	0.078171	1.208821
Sphere	Y	0.019581	0.069591	1.313238
Sphere	Z	0.085546	0.170599	0.715436
Sphere	Yaw	61.834910	76.257994	124.865239
Sphere	Pitch	15.551304	16.981590	67.641860
Sphere	Roll	15.323147	36.774045	62.371281

Table II shows our RMS error for pose estimation using the geometric reference points (angles in degrees) with the optimized hyperparameters we found by coordinate descent. The first thing we note, of course, is that these values are terrible! We appear to be doing better than random chance, but not by much. We see the largest error in Yaw, which we think might be due to the ambiguous discontinuity near 360 / 0 degrees. The other dimensions do not have a wrap around point as they were only varied 180 degrees.

We note a very interesting trend in the relative differences between train, test, and validation error. Generally, our training error was less than our validation error, suggesting overfitting, but we also note many times the validation error was significantly less than the test error! The biggest examples of this effect can be seen in the translation dimensions. Our training and validation errors are small, on the order of 10^{-2} or 10^{-1}, but the test error is over 1. We theorize that perhaps our 500 samples are too sparse in the dimensionality of the problem, such that learning results near the validation set, does not mean areas near the test set have been learned. We did check to see if there were some correlations in the test set to coverage over the feature space, but the poses do seem to be indeed randomly distributed.

Table III shows what our optimized hyperparameters were. We note a few general trends. First, the angle dimensions have much larger epsilons, which intuitively makes sense as they span over larger order of magnitudes than the translation components. C was also, in general, higher for angle dimensions than translational, with the notable exceptions of the Y dimensions for both the sphere and the cube.

TABLE III: SVM Optimized Parameters on the Reference Point Dataset

Shape	Dim	C	σ	ϵ
Cone	X	65.793322	29.836472	0.012328
Cone	Y	151.991108	35.938137	0.006136
Cone	Z	132.194115	27.185882	0.043288
Cone	Yaw	8697.490026	83.021757	4.328761
Cone	Pitch	1629.750835	18.738174	2.848036
Cone	Roll	10000.000000	100.000000	4.328761
Cube	X	21.544347	4.641589	0.001748
Cube	Y	26560.877829	62.802914	0.006136
Cube	Z	65.793322	8.111308	0.018738
Cube	Yaw	1873.817423	6.734151	0.001000
Cube	Pitch	5722.367659	17.073526	3.764936
Cube	Roll	2848.035868	14.174742	0.533670
Sphere	X	9.326033	15.556761	0.009326
Sphere	Y	70548.023107	100.000000	0.012328
Sphere	Z	1873.817423	100.000000	0.049770
Sphere	Yaw	20092.330026	100.000000	0.464159
Sphere	Pitch	3274.549163	52.140083	0.001000
Sphere	Roll	705.480231	8.111308	1.417474

C. Performance on Raw Image Data

TABLE IV: SVM Performance on the Image Dataset

Shape	Dimension	Train E.	Val E.	Test E
Cone	X	0.929743	0.810887	0.856581
Cone	Y	0.984332	0.818735	0.886178
Cone	Z	0.490626	0.496716	0.519886
Cone	Yaw	108.586440	98.178370	102.914391
Cone	Pitch	60.378916	51.779552	54.047725
Cone	Roll	52.591000	50.459018	47.921814
Cube	X	0.897220	0.884600	0.885430
Cube	Y	0.883796	0.779065	0.925021
Cube	Z	0.512376	0.520755	0.508106
Cube	Yaw	99.024495	101.423926	98.324162
Cube	Pitch	57.957975	48.915493	50.332767
Cube	Roll	53.386639	50.835804	53.885835
Sphere	X	0.879085	0.951192	0.906496
Sphere	Y	0.958136	0.792612	0.912514
Sphere	Z	0.474422	0.491515	0.509054
Sphere	Yaw	104.293443	101.257011	100.387393
Sphere	Pitch	58.114635	52.445369	57.654654
Sphere	Roll	55.780784	53.296742	59.528568

Table IV shows the performance of our SVM regression using the full bitmap data. We note that the test data performance is marginally better than our reference point dataset, but is still far inferior to the performance of EPnP. It is notable, that the large differences between test error and validation error have disappeared. Training error is also much more in line with validation and test error. In fact, often times the test dataset out-performed the training dataset.

Our optimized parameters from the image dataset are shown in Table V. We note again the higher values of ϵ showed up in the angle dimensions.

TABLE V: SVM Optimized Parameters on Image Dataset

Shape	Dim	C	σ	ϵ
Cone	X	464158.883361	100.000000	0.100000
Cone	Y	4641.588834	14.677993	0.001000
Cone	Z	46415.888336	0.100000	0.001778
Cone	Yaw	215.443469	0.010000	0.056234
Cone	Pitch	21544.346900	0.146780	5.623413
Cone	Roll	0.215443	100.000000	0.001000
Cube	X	21544.346900	21.544347	0.031623
Cube	Y	0.010000	0.010000	0.001000
Cube	Z	46415.888336	0.010000	0.001000
Cube	Yaw	21544.346900	0.068129	0.056234
Cube	Pitch	464158.883361	68.129207	1.000000
Cube	Roll	4641.588834	0.010000	0.316228
Sphere	X	0.010000	0.010000	0.001000
Sphere	Y	10000.000000	68.129207	0.003162
Sphere	Z	4641.588834	0.068129	0.001000
Sphere	Yaw	0.100000	0.010000	0.001000
Sphere	Pitch	464158.883361	0.146780	0.003162
Sphere	Roll	0.464159	0.010000	0.010000

D. Sparsity of SVM Solutions

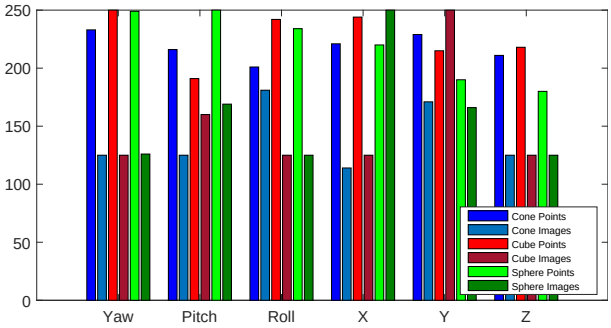


Fig. 7: The number of support vectors for each dimension in both the reference point and bitmap datasets. The image dataset created a much sparser solution (in terms of numbers of support vectors) than the bitmap dataset.

Fig 7 shows the number of support vectors generated in each of our SVMs. Generally, we found that the SVMs trained to work on full bitmaps generated a sparser solution than those SVMs trained on the geometric point datasets. In several cases, we see that there was no reduction in solution size at all. This suggests we are not sufficiently filling the input feature space with enough training examples; there's no "redundant" samples to ignore. That does raise the question of why the image SVMs made sparser solutions; they performed better but the input data was of a much higher dimensionality?

V. Feed-Forward Network (FFNN)

We implemented a feed forward neural network in MATLAB (based on our code from HW3). We used this network to train and predict on the geometric reference point dataset. (The fully connected nature and the large data size of the bitmaps made running on a MATLAB based code too slow for practicality.)

We used ReLUs for our activation functions in the hidden layers. All of our weights were initialized to a normal random variable with variance $1/\sqrt{m}$ where m is the number of neurons in the previous layer. Biases were initialized to 0. This initialization scheme means that the magnitude of the input to a neuron should not depend on how many neurons are in the previous layer. Our output layer was simply a linear output, as we were performing regression and not classification. We used squared loss as our loss function. Unlike the SVM, the neural network was able to output a complete prediction for all dimensions of the pose, as opposed to our SVM case where we had different regressions for each dimension.

TABLE VI: Optimal hyperparameters for Feed-Forward Neural Networks

Shape	L	η
Cube	[32, 256, 256]	.01
Cone	[128, 128]	.01
Sphere	[128, 128]	0.01

Instead of outputting a 6 dimensional pose, we opted to convert the poses to a 3 dimensional translation concatenated to a 9 dimensional direction cosine matrix. The direction cosine matrix (DCM) is a 3x3 matrix where each column is an axis in the body frame of the shape, with the rotation of the pose applied.

$$\hat{x} = R \cdot \vec{x} \quad \hat{y} = R \cdot \vec{y} \quad \hat{z} = R \cdot \vec{z}$$
$$DCM = \begin{bmatrix} \hat{x} & \hat{y} & \hat{z} \end{bmatrix} \tag{11}$$

Where $\vec{x}$, $\vec{y}$, and $\vec{z}$ are the unit vectors for the coordinate system, and R is the rotation matrix of the pose.

There are two benefits of this system. The first is that there are no ambiguities with wrap-around (like the with the yaw coordinate that we saw with the SVMs). The second is that now the dimensions of the prediction are all approximately around the same order of magnitude. Previously, angular components varied all the way to 360, but now, all components are approximately around magnitude 1. This will help prevent the neural network from favoring one coordinate over the other. We, of course, also converted our true pose parameters into this 12-vector form as well.

Like the SVMs, we used a coordinate descent empirical algorithm to search for an optimum configuration. For each training, we trained with a batch size of 1, and terminated when the validation error (the RMS of the norms between the predicted 12-vectors and the true 12-vectors) did not improve for 10 epochs (an epoch being presenting 250 randomly drawn training samples). We varied 6 hyperparameters, the number of neurons in 5 layers ($L_1 \dots L_5$), and the learning rate η.

$$\eta \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-8}\} \tag{12}$$
$$L_i \in \{0, 2, 8, 32, 128, 256\} \tag{13}$$

(An L_i of 0 indicated we just dropped the layer, so we varied different total number of layers in the architecture.) As there is some randomness in the final validation accuracy due to initialization, we couldn't rely on a monotonically decreasing error and it's eventually shallowing out to determine when to terminate. We thus used a termination of 10 full cycles of optimizing every hyperparameter.

TABLE VII: Feed-Forward Neural Network Performance

Shape	Train Error	Val Error	Test Error	Test Trans. Err.	Test Ang. Err.
Cube	0.0019	0.1222	0.1462	.2307	6.3224
Cone	0.0110	0.1645	0.1566	.2091	6.5724
Sphere	0.0164	0.1347	0.1696	.2053	7.3250

Table VI shows the optimized hyperparameters that our coordinate descent algorithm found for each shape. Two and three layer networks were the best, with large amount of neurons in. η seems to have converged on 0.01 as the best step size.

Table VII shows the performance of our optimal neural networks on the point dataset. Validation and test error are the RMS values of the norm of the vector residual between the real 12-vector and the predicted 12-vector. To get a more intuitive measure of how the regression was performing, we also for the test set, listed the average translational error (norm of the residual between translation vectors) and the average angle between the body axes of the predicted rotation versus the real rotation (in degrees).

We note that first, we have performed much better than the SVM. Translational error is smaller, and the average angular error has sunk to less than 10 degrees. This is still no where close to the accuracy, of the EPnP algorithm, but is much closer.

We note that based on the training error vs. the validation error and the test error, that we are overfitting. There are a number of ways we could have tried to remove this overfitting if we had more time to study the problem, including noising up the point locations, or implementing drop-out. It's also possible, that we simply need to provide more samples during the training.

VI. Convolutional Neural Network (CNN)

We explored using a convolutional neural network on our image data only. This is because CNNs have the ability to relate features locally which could be beneficial for image data. We again note that due to the low dimensionality of the reference point dataset, we did not test the CNN's performance with that data.

A. Implementation

We implemented our CNN using a GPU version of TensorFlow. Similar to homework 3, we had an input layer, multiple convolution and pooling layers, multiple hidden layers, and an output layer (Fig 8). We initialized our weights using the TensorFlow truncated normal operation with a standard deviation of 0.1.

For the input layer we took each channel (RGB) of the image and normalized the pixel values to $[-0.5, 0.5]$. The size of our input layer corresponded to the number of pixels times the number of channels. In our cases, all of our runs used a 64x64 pixel image with three channels that corresponded to red, blue, and green. For the output layer, we used 12 neurons; three neurons that represented the 3D position of the object and nine neurons that represented the objects orientation as a direction cosine matrix. We applied a linear output layer since we were attempting to perform regression and not classification.

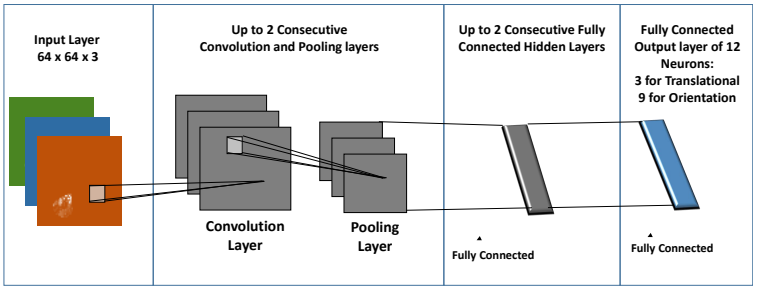


Fig. 8: Graphical Representation of our CNN Setup

The number of layers, nodes, filter size and stride were tunable hyperparameters for each convolution or pooling layer. For the standard hidden layers, the number of layers and nodes were tunable parameters. The convolutional and hidden layers used ReLU activation functions, while the pooling layers used maximum operation as activation functions.

B. Training

For training the networks, we used a standard L2 Loss function as well the same gradient descent method used in homework 3 (the gradient descent optimizer built into TensorFlow). We used the first 300 samples as a training set. The remaining were left as validation data.

Since we were not totally sure the best hyperparameters to use for the dataset, we decided to implement an empirical version of coordinate descent (as in the previous sections) to attempt to minimize validation error. In each iteration of coordinate descent we ran through the following list of hyperparameters:

$$
\begin{aligned}
Train.\ Batch\ Size &\in [100] \\
Number\ of\ Training\ Steps &\in [1000] \\
Learning\ Rate &\in [10^{-2},\ 10^{-4},\ 10^{-8}] \\
Conv.\ Filter\ Size &\in [3,\ 5,\ 8] \\
Conv.\ Stride &\in [1,\ 2,\ 4] \\
Conv.\ Depth &\in [8,\ 16] \\
Conv.\ Layer\ Pooling &\in [ON,\ OFF] \\
Pooling\ Filter\ Size &\in [3,\ 5,\ 8] \\
Pooling\ Stride &\in [1,\ 2,\ 4] \\
Weight\ Penalty\ Filter\ Size &\in [3,\ 5,\ 8] \\
Num\ Hidden\ Layer\ Neurons &\in [16,\ 32,\ 64,\ 128]
\end{aligned}
$$

We originally were considering including the number of convolution, pooling, and hidden layers as part of the hyperparameter sweep that we performed coordinate descent on. However, we were concerned that in doing so, the coordinate descent may not correctly choose the number of layers due to hyperparameter dependencies (for example, varying the stride of convolutional layer 3 produces identical validation

TABLE VIII: Optimal Architecture for Convolutional Neural Network

Shape	Train RMS Error	Val RMS Error	Num Hidden	Num Conv.
Cube	4.5070	4.6018	2	1
Cone	4.4347	4.0840	1	2
Sphere	4.5627	4.5696	2	0

TABLE IX: Optimal Hyperparameters for Convolutional Neural Network

	Cube	Cone	Sphere
Weight Penalty	0.1	0	0
Learning Rate	10^{-4}	10^{-4}	10^{-4}
Neurons in Hidden Layer 1	16	16	32
Neurons in Hidden Layer 2	32	*	16
Conv. Layer 1 Filter Size	3	8	*
Conv. Layer 2 Filter Size	*	3	*
Conv. Layer 1 Stride	4	2	*
Conv. Layer 2 Stride	*	2	*
Conv. Layer 1 Depth	8	8	*
Conv. Layer 2 Depth	*	8	*
Pooling	ON	ON	*
Pooling Layer 1 Filter Size	8	5	*
Pooling Layer 2 Filter Size	*	3	*
Pooling Layer 1 Stride	41	4	*
Pooling Layer 2 Stride	*	1	*

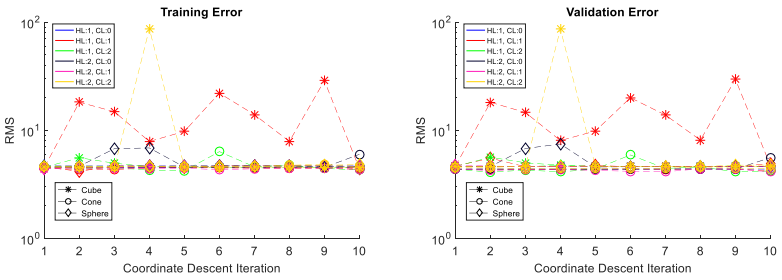
errors if layer 3 is off). We did solve this problem in the feed-forward networks of the previous section, but that only worked because there was only one type of layer-specific hyperparameter. We addressed this problem by choosing a set of fixed layer architectures to run the coordinate descent with and did a comparison between them. We ran our coordinate descent for 10 iterations (an iteration being a complete cycle through every hyperparameter) on the following permutations:

$$Num.\ HiddenLayers \in [1,\ 2]$$
$$Num.\ Conv./Pooling\ Layers \in [0,\ 1,\ 2]$$

C. Performance on Raw Image Data

Performance on the raw image datasets over coordinate descent is shown in Fig 9. Overall the performance of the CNN did not fair well for the image data. It is clear that in some cases coordinate descent improved over each iteration while others it seemed to have performance that bounced around. The inherent variability in training a neural net was greater than our coordinate descent algorithm could handle. We also note, that for our best performing layer architectures, the minimum of validation error was quickly met, and that changes in hyperparameters did not cause any significant improvement. No architecture that we explored brought the RMS error of the norms of the 12-vector residuals below 4. We note that number is very much higher than the RMS errors we saw in the feed-forward networks applied to the geometric reference point dataset. We took the hyperparameters of the best performing iteration of each layer architecture. Those parameters are shown in Table IX, and their performance is shown in Table VIII.

Unlike the SVM case, where training on the full bitmap images produced better performance than just using the points,



(a) Training data RMS (b) Validation data RMS

Fig. 9: Figure shows RMS of the validation and training data after each coordinate descent iteration for each shape.

for the neural network case, the point dataset far exceeded the performance of the image dataset. It's possible we have some sort of misconfiguration in our CNNs that is bounding our performance, but we don't have any smoking guns. We do note, that although performance quickly plateaued, we did see large increases in performance for the first few hundred training steps, so there isn't any incredibly fatal error in our implementation.

It's also possible that the images simply do not provide enough, or accessible information to estimate the problem, and that the performance increase for images over points with the SVM, is simply because the SVM has such horrible point dataset performance to begin with.

VII. Summary

We found that our machine learning algorithms could not match the performance of the deterministic EPnP algorithm. We note that ML algorithms are starting from a disadvantage, as the EPnP algorithm is provided the geometry of the objects. We found that neural networks out-performed SVMs, at least when using the point dataset with the neural network. The neural network is a more flexible architechture, and can learn relationships on its own. Our choice of specific kernels imparts some bias and information on the SVM. It is possible, that the GRBK is simply very ill-suited for the problem, and that the information exploited by the neural network is not well-exploited by the GRBK.

We also found that neural networks using a reduce point dataset, performed better than neural networks using a full bitmap image dataset. We note that the point dataset is given an inherent advantage, as the locating of specific points is exact, and they are positively identified by their indexing in the input vector. A neural network working with just image data, must not only estimate a pose, but must first identify non-ambiguous features in the image. There might be a configuration of a CNN that can do this, but we did not find it.

We did plan to evaluate the introduction of noise in both datasets, but as we could not come close to the performance of EPnP even with perfect data, it seemed prudent to focus on improving non-noise performance. We realize that given the amount of over-fitting we did see, noise may have actually improved the situation, at least in terms of over-fitting.

Neural networks seem more promising than SVMs. We note the approximately 6 degrees of rotation error is probably in

the realm of what a human can perform. There might be more fruitful results by reformulating the output of a neural network. EPnP estimates the locations of virtual points, and then pose can be derived from that estimate. A neural network could be trained to estimate these virtual points rather than a full pose.

VIII. Division of Labor

TABLE X: Project Division Of Labor

Task	Person
Data Set Generation	Ben
EPnP Baseline Work	Alex
SVM Regression	Ben
FFNN	Ben
CNN	Alex
Report Writing	Alex and Ben

References

[1] S. Li, C. Xu and M. Xie. *A Robust O(n) Solution to the Perspective-n-Point Problem*, in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, p. 1444-1450, 2012.

[2] V. Lepetit, F. Moreno-Noguer and P. Fua. *EPnP: An Accurate O(n) Solution to the PnP Problem*, in International Journal Of Computer Vision, vol. 81, p. 155-166, 2009.

[3] Image Source: `http://www.songho.ca/opengl/files/g_projectionmatrix01.png`

[4] Image Source: `https://i.ytimg.com/vi/J54dotTt7k0/maxresdefault.jpg`

[5] Image Source: `http://www.roadtovr.com/wp-content/uploads/2014/01/oculus-rift-crystal-cove.jpeg`