Supervised Learning Approach to Inverse Kinematics

Michael Menaker, Leo Ling

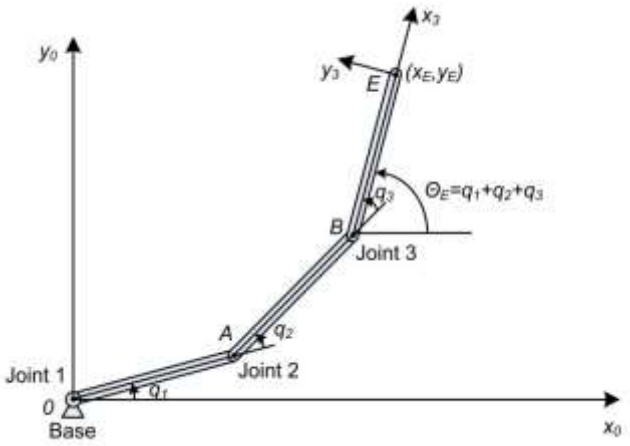
Over the course of this project, we demonstrated the use of neural networks to solve the inverse kinematics problem. The inverse kinematics problem refers to the finding the movements required to reach a certain Cartesian configuration. Or, more formally, for a given joint space Q and a Cartesian space W

$F(Q) = W$ is the 1 to 1 mapping from joint positions/angles to a Cartesian coordinate, aka forward kinematics.

Finding the inverse mapping from W to Q, such that $F^{-1}(W) = Q$, is the inverse kinematics problem.

For this project, we have elected to train a neural networks to approximate the inverse kinematic function for a relatively simple 3-Link System on a 2D plane. For some simple cases, there are analytical solutions or techniques to find the inverse function. Our work here is an example that this class of problem can be approximate using neural networks. This work closely follows a paper by Duka, with the following parameters.

1. $l_1 = l_2 = l_3 = 2$, The links all have the same size
2. $q_1 \in [0, \pi]$, $q_2 \in [-\pi, 0]$, $q_3 \in [-\pi/2, \pi/2]$, Each rotational joint movement is limited



Note that some code has been cut from the report and some images rescaled for brevity.

To generate the training dataset for supervised learning, we constructed a meshgrid over the domain of the joint rotations, $q1, q2, q3$, and found the output Cartesian coordinate, labeled E on the above figure. For supervised training of the neural network, the Cartesian coordinates and the end angle of the system, Θ_E, are the inputs. With the output being the three joint angles that produced this movement.

In [114…

```
## Generate Data for an arbitrary three rotational joint system in 2D
def x_e(q, l=l) :
    sumX = 0
    for ii in range(l.size):
        sumX += l[ii] * np.cos(np.sum(q[:ii, :], axis=0))
    return sumX
def y_e(q, l=l) :
    sumY = 0
    for ii in range(l.size):
        sumY += l[ii] * np.sin(np.sum(q[:ii, :], axis=0))
    return sumY
def theta_e(q):
    return np.sum(q, axis=0)

# Create some training end points by solving the forward kinematics equation
numPoints = 30
# Specifications from paper
q1 = np.linspace(0, np.pi, num=numPoints)
q2 = np.linspace(-np.pi, 0, num=numPoints)
q3 = np.linspace(-np.pi/2, np.pi/2, num=numPoints)
q1v, q2v, q3v = np.meshgrid(q1, q2, q3)
q = np.vstack((q1v.flatten(), q2v.flatten(), q3v.flatten()))

x = x_e(q)
y = y_e(q)
theta = theta_e(q)
```

The code below partially shows the setup for the Neural Network, since the network requires linear activations at the output layer but tanh activations in the hidden layers we did not use the library provided with this class. Our neural network differs from the one proposed by Duka with an additional hidden layer with 50 neurons. Nonetheless, the overall structure is a MLP, Multilayer perceptron with tanh activations for hidden neurons and a linear activation at the output layer.

The inputs are $\{X_E, Y_E, \Theta_E\}$, which describe the Cartesian position and orientation of the end-effector.

The outputs are $\{q_1, q_2, q_3\}$, the joint angles shown in the figure above.

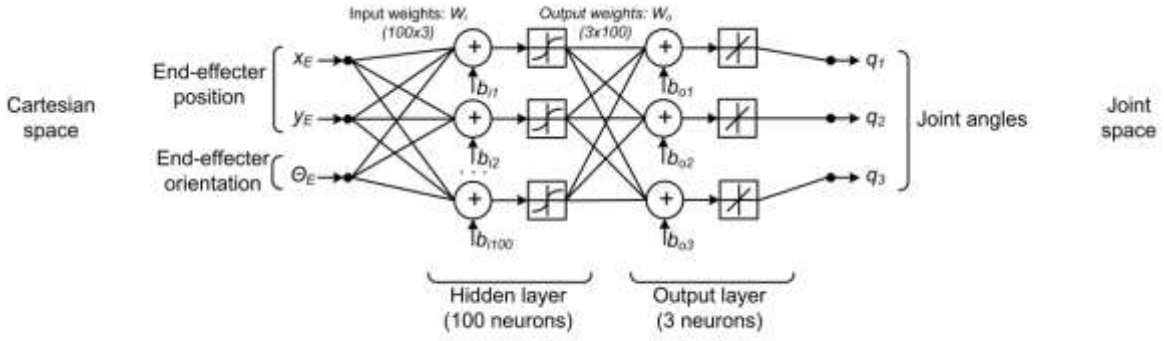
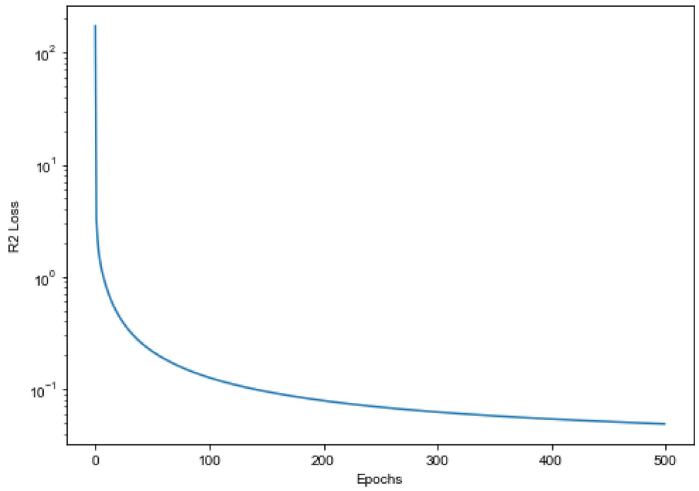


Fig. 4. The structure of the neural network

In [115…

```
# a feature_transforms function for computing
# U_L L layer perceptron units efficiently
def feature_transforms(a, w, activation):
```

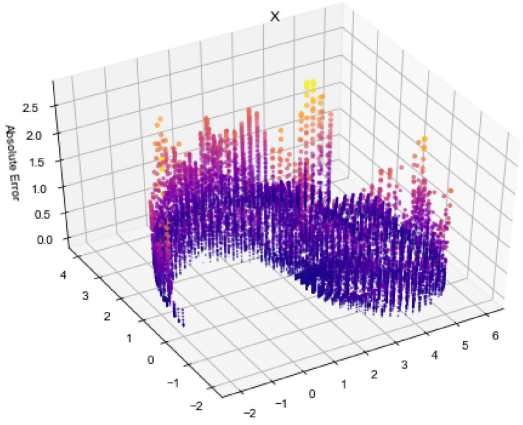
```
    for W in w:
        a = W[0] + np.dot(a.T, W[1:])
        a = activation(a).T
    return a
# an implementation of our model employing a nonlinear feature transformation
def model(x,w, activation):
    # feature transformation
    f = feature_transforms(x,w[0], activation)
    # compute linear combination and return
    a = w[1][0] + np.dot(f.T,w[1][1:])
    return a.T
def nn(w, x):
    return model(x, w, np.tanh)

# In literature, it seems MSE is used, also including L2 normalization term
def nnCost(w, x, y, epsilon=1e-3):
    pred = nn(w, x)
    presum = (y - pred)**2

    return np.sum(presum) / (y.shape[1]) + epsilon * sumW / y.shape[1]

layerSizes = [3, 100, 50, 3]
w = initialize_network_weights(layerSizes, scale=1)
numBatches = 50
for ii in tqdm(range(numIter)):
    alpha = .01/(1 + .01 * ii)
    c[ii] = nnCost(w, xData[:, validIdx], yData[:, validIdx], epsilon=0)

    # Do mini batches, otherwise training set is too large
    for jj in range(numBatches):
        delta = grad(w, xData[:, miniBatchIdxs[jj, :]], yData[:, miniBatchIdxs[jj, :]])

        # Since output is a list, we have to update things elementwise
        for superlayerIdx in range(len(w)):
            for layerIdx in range(len(w[superlayerIdx])):
                w[superlayerIdx][layerIdx] -= alpha * delta[superlayerIdx][layerIdx]
```

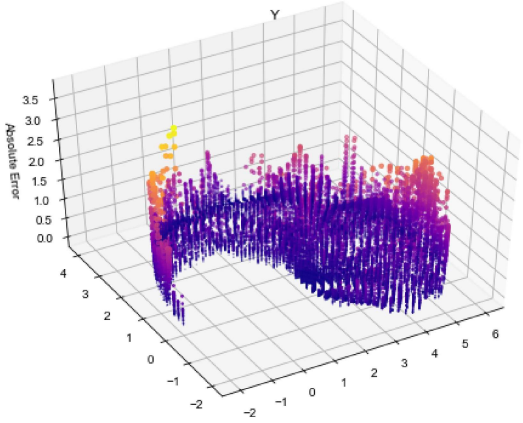


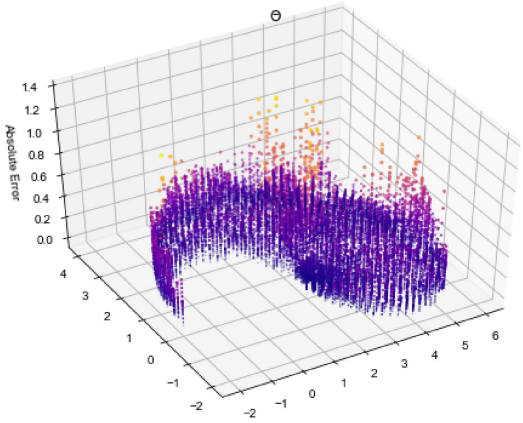
The network shows convergence with decreasing R2 loss as training progresses. Due to the large number of generated data points, we used 50 Minibatches per epoch with a total of 500 epochs. As well as decreasing training rate.

In [ ]:

```
# Calculate Z which is R2 error from expected
pred = nn(w, xData) * np.pi
# Check how close the proposed solution is to the original input xData
xPred = x_e(pred)
yPred = y_e(pred)
thetaPred = theta_e(pred)

inputPredNorm = np.vstack(((xPred-np.mean(x))/np.std(x),
                           (yPred-np.mean(y))/np.std(y),
                           thetaPred/np.pi))
presum = np.abs(inputPredNorm - xData)

# Plot meshgrid on countour plot
for ii in range(3):
    maxPresum = np.amax(presum[ii,:])
    cs = ax.scatter(x, y, presum[ii, :],
                    s=presum[ii,:]*5,
                    c=colormap(presum[ii,:]/maxPresum))
```



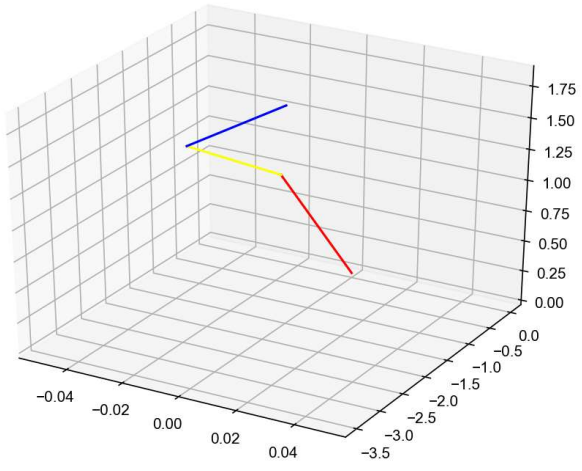




This figure shows the deviation from the inputs $\{X_E, Y_E, \Theta_E\}$ for the generated outputs. Since we are able to trivially solve the forward kinematics equations, we can map the output in the joint space to the Cartesian space to measure the error of the proposed solution of the neural network. As we can see the Neural network performs more poorly at the edge of it's output range, but overall, there is little error in the proposed Cartesian output.

In [ ]:

```
#visualize manipulator
import matplotlib.animation
plt.rcParams["animation.html"] = "jshtml"
plt.rcParams['figure.dpi'] = 150
plt.ioff()
theta = np.array([0,0,0])
link1 = np.array([[0,0,0],[0, l1*np.cos(theta[0]), l1*np.sin(theta[0])]])
link2 = np.array([link1[1],[0, link1[1,1] + l2*np.cos(theta[0]+theta[1]), link1[1,2]+l2
link3 = np.array([link2[1],[0, link2[1,1] + l3*np.cos(theta[0]+theta[1]+theta[2]), link
fig = plt.figure()
ax = plt.axes(projection='3d')
line1 = ax.plot([link1[0,0], link1[1,0]], [link1[0,1], link1[1,1]], zs = [link1[0,2], l
line2 = ax.plot([link2[0,0], link2[1,0]], [link2[0,1], link2[1,1]], zs = [link2[0,2], l
line3 = ax.plot([link3[0,0], link3[1,0]], [link3[0,1], link3[1,1]], zs = [link3[0,2], l
z_desired = np.linspace(1,3,100)

def animate(i):
    ax.cla()
    theta = nn(w, np.array([0,1.5,z_desired[i]])) * np.pi
    link1 = np.array([[0,0,0],[0, l1*np.cos(theta[0]), l1*np.sin(theta[0])]])
    link2 = np.array([link1[1],[0, link1[1,1] + l2*np.cos(theta[0]+theta[1]), link1[1,2
    link3 = np.array([link2[1],[0, link2[1,1] + l3*np.cos(theta[0]+theta[1]+theta[2]),
    line1 = ax.plot([link1[0,0], link1[1,0]], [link1[0,1], link1[1,1]], zs = [link1[0,2
    line2 = ax.plot([link2[0,0], link2[1,0]], [link2[0,1], link2[1,1]], zs = [link2[0,2
    line3 = ax.plot([link3[0,0], link3[1,0]], [link3[0,1], link3[1,1]], zs = [link3[0,2

matplotlib.animation.FuncAnimation(fig, animate, frames=100)
```



This code visualizes the link positions over the Cartesian space for an given set of joint angles.