ROB 599 HW 4

Prof. Johnson-Roberson and Prof. Vasudevan

Due 23 Nov, 2017

Submission Details

Use this PDF only as a reference for the questions. You will find a code template for each problem on Cody. You can copy the template from Cody into MATLAB on your personal computer in order to write and test your own code, but *final code submission must be through Cody*.

Problem 1: Model Predictive Control (MPC) [50 points]

In this problem, you will use quadratic programming based MPC to design a controller. Assume you are given the following vehicle model:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} = \begin{bmatrix}0 & 0.310625\\ 0 & 0\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix} + \begin{bmatrix}0\\ 1\end{bmatrix}u. \tag{1}$$

Suppose you are tasked to design a controller, $u : [0, 10] \to [-10, 10]$, that takes the state of the system from the red 'x', $x(0) = \begin{bmatrix}-1\\ 0\end{bmatrix}$, to the green circle, $x(10) = \begin{bmatrix}1\\ 0\end{bmatrix}$, while staying within the road as depicted in Figure 1. Though only the top portion of the road is depicted, below the $y = 0$ axis the road's run vertical. Notice that the optimization problem is not convex due to the road constraints.

An autonomous vehicle engineer named Matt the Eager Junior has determined that the following control input: $u(t) = -0.3175 \sin(\frac{\pi t}{10} - \frac{\pi}{2})$ will take the initial condition to the final condition. An autonomous vehicle sales representative wants you to generalize this controller to take an arbitrary initial condition as close as possible to the green circle by $t = 10$. In particular, you will consider the initial condition $(-1.25, 0)$. To solve this problem, you decide to use quadratic programming-based MPC using error dynamics that are linearized about the nominal trajectory of the system under the control input designed by Matt the Eager Junior (you will be augmenting the nominal control input).

1.1 Euler Discretization [10 points] You decide to represent the linearized dynamics about the trajectory using Euler discretization with a time step $\Delta t = 0.01$. Compute the discrete-time LTI

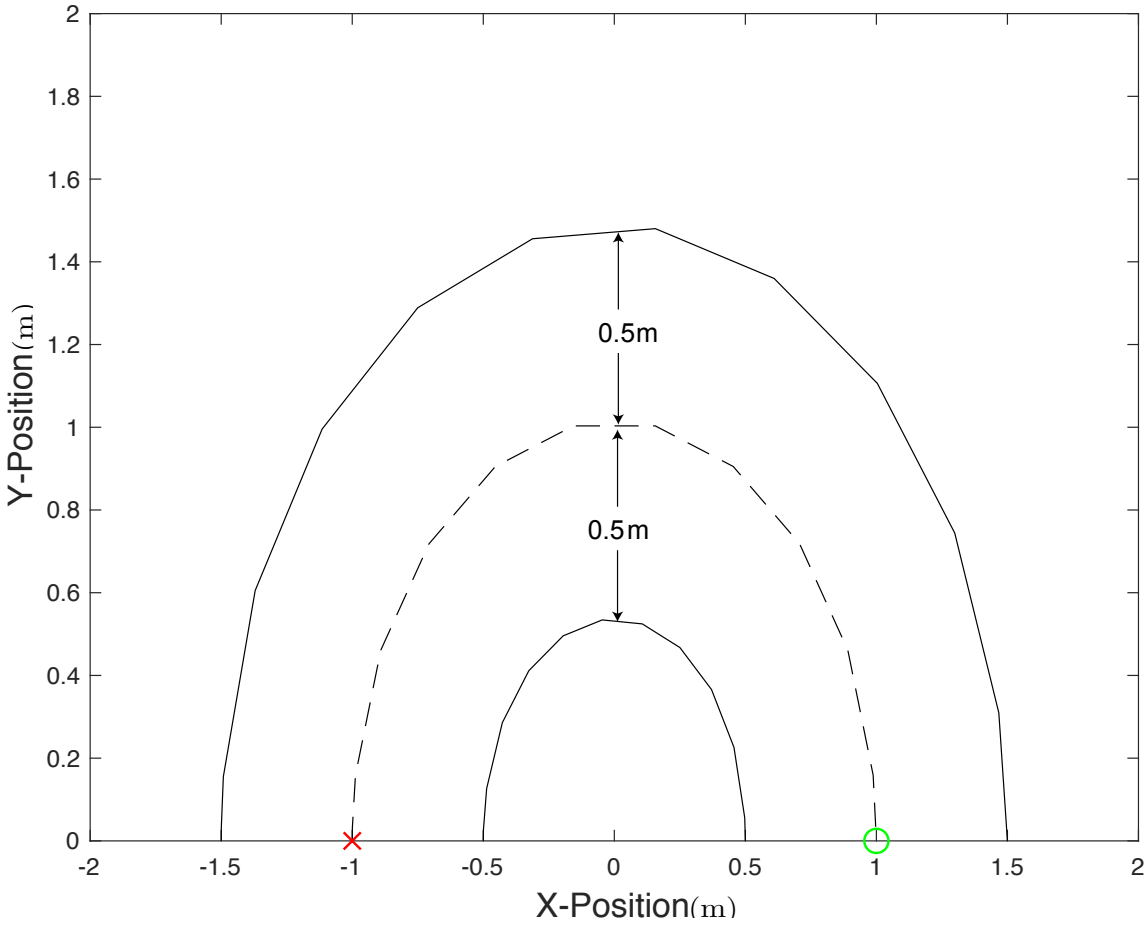


Figure 1: An illustration of the autonomous vehicle control design task.

system representation of the A and B matrices and save them in doubles named `A` and `B`.

1.2 Decision Variables [5 points] Next, you decide to use a planning horizon of 10 time steps, i.e. 0.1 seconds. You will use MPC at each discrete time step k to solve for the next 10 time steps. At each time step, you decide to let the first portion of the decision variable z represent the discrete states of the system from time step k to time step $k+10$ in order. You let the second portion of the decision variable space represent the input of the system from time step k to time step $k+9$ in order. Compute the total size of your decision variable space and save them in a double named `Ndec`.

1.3 Equality Constraints [12.5 points] Next, you decide to solve the MPC using the MATLAB function '`quadprog`,' which represents the problem using an equality and inequality constraint (hint use '`help quadprog`' in MATLAB). You decide to represent the Euler integration using an equality constraint $A_{eq}z = b_{eq}$. The first 2 rows of A_{eq} represents satisfying the initial condition at time step k in the linearized dynamics about the trajectory designed by Matt the Eager Junior. Each subsequent row of A_{eq} represents the Euler integration steps in order from time step k to time step $k+9$. Compute the equality constraint, $A_{eq}z = b_{eq}$ when $k = 0$, and save it in matrix of doubles named `Aeq` and `beq`.
For the autograder, The order of your decision variable, z, should be

$$z = [x(k),\ y(k),\ x(k+1),\ y(k+1)...x(k+10),\ y(k+10),\ u(k),\ u(k+1),\cdots,\ u(k+9)]^T \tag{2}$$

and each equality constraint for the dynamics should be written as

$$A\begin{bmatrix} x(k) \\ y(k) \end{bmatrix} + Bu(k) - \begin{bmatrix} x(k+1) \\ y(k+1) \end{bmatrix} = 0 \tag{3}$$

1.4 Inequality Constraints [12.5 points] Next, you decide to constrain the state of the linearized dynamics about the trajectory designed by Matt the Eager Junior to stay in a box of $[-0.5, 0.5]^2$ centered about the trajectory. You also must ensure that the total control input at each MPC time step obeys the bounds $u(t) \in [-10, 10]$.

To ensure that this constraint is satisfied, you use an inequality constraint, $A_{ineq}z \leq b_{ineq}$. Compute this inequality constraint and save it in a matrix of doubles named `Aineq` and `bineq`. When writing your inequality constraints, enforce all of the upper, then all of the lower bound conditions for your decision variable, z of length `Ndec`. In other words, your constraints for states with upper

and lower bounds $b_u = [b_{u,1}, \cdots, b_{u,\texttt{Ndec}}]^\top$, $b_l = [b_{l,1}, \cdots, b_{l,\texttt{Ndec}}]^\top$ should appear like so:

$$\begin{aligned} \left[A_u\right]\left[z\right] &\leq \left[b_u\right] \\ \left[A_l\right]\left[z\right] &\geq \left[b_l\right] \end{aligned} \tag{4}$$

and

$$\begin{aligned} \left[A_{\text{ineq}}\right] &= \begin{bmatrix} A_u \\ A_l \end{bmatrix} \\ \left[b_{\text{ineq}}\right] &= \begin{bmatrix} b_u \\ b_l \end{bmatrix} \end{aligned} \tag{5}$$

1.5 Generating a Solution [10 points] Next you decide to run the quadratic programming-based MPC with zero penalty on the input at each time step and a quadratic penalty of $Q = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix}$ on the state of the linearized dynamics about the trajectory designed by Matt the Eager Junior at each time step. For each $k > 0$, the initial state $x(k)$ of the system at time step k is generated by applying the Euler integration step to the previous initial state $x(k-1)$ with the input computed using the quadratic programming-based MPC at the previous time step. Run the quadratic programming-based MPC loop until $\Delta tk = 10$ and save the trajectory generated using Euler integration for the overall rather than the linearized dynamics about the trajectory in a variable named `x` where each column of the variable corresponds to each state of the system.

Problem 2: Simple Neural Network

In this problem, we will be examining a simple implementation of a neural network. One of the most basic uses for a neural network is to learn a hidden model based on known input and output data, and make predictions on the outputs given new inputs. The neural network you will build will emulate the linear system in Equation 6.

$$y = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 \tag{6}$$

In this system, $x \in \mathbb{R}^3$ is the input, $y \in \mathbb{R}$ is the output, and $\alpha \in \mathbb{R}^3$ are the system coefficients. With respect to a single input, the neural net will be built in the general form shown in Figure 2.

Note that the hidden layer nodes are fully connected to both the inputs and output. The weights on all of these synapses will be learned through training. You are given the function `simple_nn`, and are tasked with completing the sub-functions that enable the neural network computation.

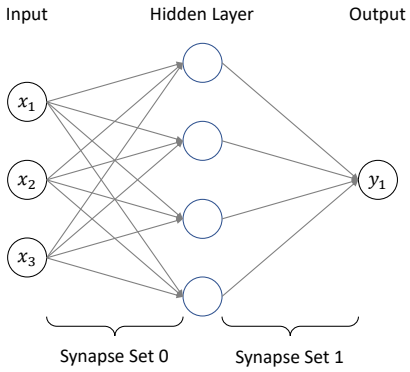


Figure 2: Simple Neural Net Structure

2.1 The Sigmoid Function To ensure the activity of each synapse is limited in magnitude and positive-definite, we will be utilizing the sigmoid activation function, represented in Equation 7.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{7}$$

This function maps $x \in \mathbb{R} \rightarrow \sigma(x) \in (0,1)$, and will be applied to the outputs of each synapse set. To correctly perform backpropagation, we also need the derivative, given in Equation 8.

$$\frac{d\sigma}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = [1-\sigma(x)]\sigma(x) \tag{8}$$

However, given the structure of the forward propagation, when we compute the derivative, we will already have $\sigma(x)$, rather than just x, so when `dodv=1`, the output of the function (in your code) will instead look like:

```
out = (1 - in).*in
```

This is one of the benefits of using sigmoid, in that we can use the outputs to easily compute the derivatives in backpropagation.

The task for this section is to implement the `sigmoid` function in MATLAB, in a way that can output either $\sigma(x)$ or $\frac{d\sigma}{dx}$, depending on the boolean flag `dodv` (perform the derivative if `true`).

2.2 Forward Propagation In this section, you will build the forward propagation function `simple_nn_fwd`. The structure is as follows:

1. Set the initial layer, `l0`, to be the set of inputs.
2. Compute the middle layer, `l1`, in batch form by applying the first set of weights to the initial layer, and pass this through the sigmoid normalization function.

3. Compute the output layer, `l2`, again in batch form, by applying the second set of weights to the middle layer, and pass this through `sigmoid` as well.

4. Finally, compute the error by subtracting the output layer from the true output.

To understand how the synapse weights are applied, treat the synapse interactions in Figure 2 as a set of sums. For example, given a single input $x = [x_1, x_2, x_3]$, the value of the j-th node in the hidden layer (before passing through `sigmoid`), is given by:

$$l_{1j} = \sum_{i=1}^{3} x_i w_{ij} \tag{9}$$

where, in this case, our w-values are from the corresponding synapse array. These operations can be stacked for multiple inputs, and applied through matrix multiplication, allowing us to compute the various layer values for a set of inputs, in batch form. To compute the following layers, treat the outputs of the previous layers as inputs instead, following the structure of connections shown in in Figure 2.

2.3 Backpropagation In this section, you will build the backpropagation function `backprop_simple`. This employs gradient descent, and the structure is as follows:

1. Starting from the end and working your way back, begin by computing the changes for the second set of weights, δ_{l_2}. This is obtained through:

$$\delta_{l_2} = e_{l_2} \left. \frac{d\sigma}{dx} \right|_{x=l_2} \tag{10}$$

   where e_{l_2} is the error in the second layer, and $\left. \frac{d\sigma}{dx} \right|_{x=l_2}$ is the derivative of `sigmoid` about l_2.

2. Use δ_{l_2} to compute the esimated error in l_1 using:

$$e_{l_1} = \delta_{l_2} w^{(2)} \tag{11}$$

   where $w^{(2)}$ is the second set of weights (`syn1`).

3. Perform a similar computation as before to obtain δ_{l_1}, this time using e_{l_1} and the l_1 values.

4. Here is where you will the apply gradient descent correction. Compute the new set of weights **in batch form**:

$$\begin{cases} w^{(0)} = w^{(0)} + r_1 l_0 \cdot \delta_{l_1} \\ w^{(1)} = w^{(1)} + r_2 l_1 \cdot \delta_{l_2} \end{cases} \tag{12}$$

   where r_i is the i-th learning rate scalar multiplier. Note the structures of l_0, l_1, δ_{l_1}, and δ_{l_2}, when batch processing to understand how the dot products are applied here, as their products must have dimensions that exactly match those of $w^{(0)}$ and $w^{(1)}$.

2.4 Applying the Neural Network With the sub-functions complete, you can now move on to running the neural network. Experiment with different learning rates and numbers of epochs, to get a feel for how more aggressive learning can affect the stability of the learning process, and submit the code to Cody. An example of what you should see is shown in Figure 3.

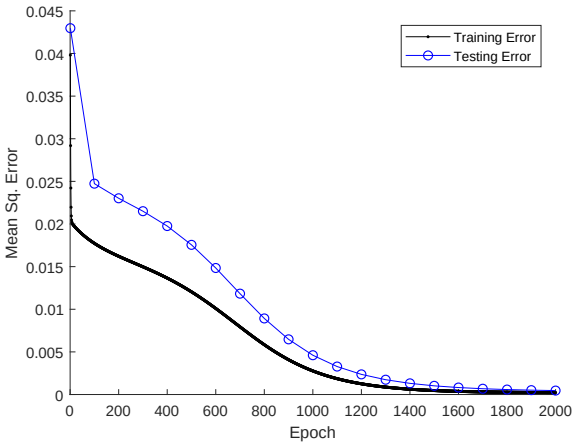


Figure 3: Neural Net Error Example

Problem 3: Simple Convolutional Neural Network

In this problem, we explore a more complex version of a neural network, the type of which is regularly used in image processing. One common use for convolutional neural nets is for handwriting recognition used in OCR methods. However, these methods tend to be much more computationally intensive than most neural nets, so to simplify things you will not be implementing a full handwriting classifier, but rather a classifier that detects whether an image is an "X" or an "O". You are given the function `simple_cnn`, and are tasked with completing it and its sub-functions to enable the convolutional neural network. Note: This convolutional neural net uses a set of pre-defined kernels, rather than learning the kernels themselves, which decreases computational complexity.

Compared to real-world examples, this will be a very small convolutional neural net implementation. The training set consists of 5 representations each of "X" and "O", with four of those five affected by either translation, rotation, scaling, or added lineweight, shown in Figure 4a. The testing set consists of 3 representations each of "X" and "O", each affected by either a different rotation, a different translation, or dissociation (breaking up the shape), shown in Figure 4b. This problem is mainly an exercise to demonstrate how a convolutional neural net is structured.

(a) Training images. (b) Testing images.

Figure 4

3.1 ReLu Layer In the previous problem, you implemented the sigmoid activation function. Here you will implement the Rectified Linear Unit (ReLu) activation function. This one is much simpler than sigmoid, as it merely raises any negative value to zero. Your task is to apply this in a function named `relu`, which has only one input and one output.

3.2 Max Pooling Layer A max pooling layer serves as one of the abstracting layers of a CNN by keeping only the portions of a layer with the highest activation, shrinking the size of the layer. In our case, we will be implementing pooling that has the same step size as the width of the pooling window. Your task for this section will be to implement the function `pool`, that has two inputs (an array, and a scalar p, the pool window/step size), and outputs the pooled layer. Method:

1. If necessary, pad the input array with zeros on the right and/or bottom sides to ensure that the pooling window/step sizes will not cause indexing errors.
2. Slide the pooling window of size $p \times p$ across the (potentially) padded image, with step size p, and extract only the largest values from each window.

For reference, both a 9×9 and 10×10 array, when pooled with this method, should yield a 5×5.

3.3 CNN Layering In this section, you will implement the layers of the CNN that lead up to the Fully Connected Layer, in the function `cnn_fwd`, using the sub-functions you have defined previously. `cnn_fwd` takes an image, the provided cell array of kernels, and pooling parameter p, and outputs the Fully Connected Layer representation of the image. Specifically, this is referred to as a 3-layer convolutional neural net.
The layer structure is as follows:

```
for i = 1:number_of_kernels
    convolutionLayer(kernel(i))
```

```
    reluLayer
    maxPoolingLayer

    convolutionLayer(kernel(i))
    reluLayer
    maxPoolingLayer

    convolutionLayer(kernel(i))
    reluLayer
    maxPoolingLayer

    normalizationLayer
endfor
combineNormalizedLayers
```

Notes on implementation:

- For each `convolutionLayer`, use any convolution function you prefer to apply the given kernels (`conv2` and `imfilter` are two options), and ensure that the outputs are the same dimension as what was put in. The pooling layers will handle the dimension manipulations.
- `normalizationLayer` does what you expect: it normalizes the sublayer to sum to 1. Caution: ReLu can sometimes zero out a layer if all the values happen to be negative, so just pass the layer along, instead of normalizing, if this happens.
- You will be passing the image through this set of layers for each of the convolution kernels, and combining the normalized representations at the end, hence the `combineNormalizedLayers` after the loop. Effectively, after passing through all the layers in the `for` loop, the input image will be converted into several distinct, smaller dimension sublayers, one for each kernel.
- `combineNormalizedLayers` takes the resulant sublayers, vectorizes them (column-first), and concatenates them all into one large vector. This is the Fully Connected Layer.

3.4 Partial CNN Backpropagation For the sake of simplicity, the backpropagation for this problem only changes the weights on the Fully Connected Layer. As we are performing classification this time, the weight structure is slightly different from the previous problem: the weights now have two columns, one each for "X" and "O". This lets us compute guesses for "X" and "O" independently, and allows for a more nuanced error estimate than "correct" or "incorrect".

Instead, the error is represented by how sure the network is about the given image being either an "X" or an "O". For example, if we pass the network an "X", the true classification is $[1,0]$. If

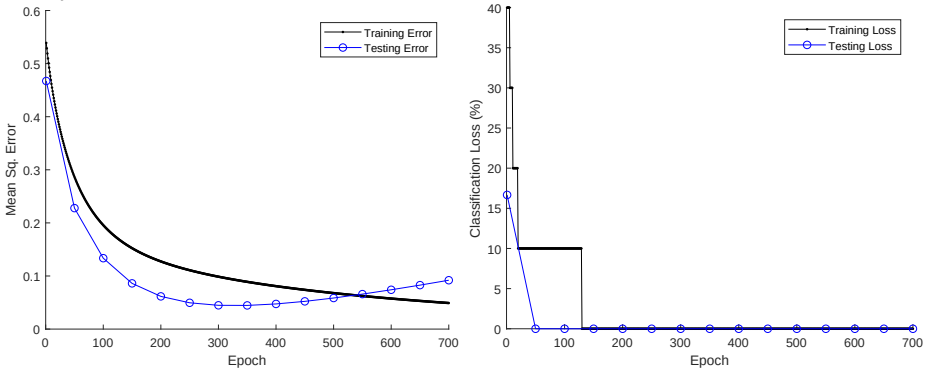
the network outputs an estimate of $[0.88, 0.64]$, the error is then $[0.12, -0.64]$, which can be used to manipulate the weights with respect to "X" and "O" separately.

You will implement the function `backprop_cnn`, in the following way:

1. Compute the initial guess, and subtract this from the true value for the error.
2. Modify the weights, which for a single column would be in the form of $w_{new} = w_{old} + rel_{fcn}$. r is the learning rate, e is the error, l_{fcn} is the fully-connected layer.
3. Compute the new error by recomputing the guess with the new weights.

3.5 Evaluate Classification Accuracy Implement a function `eval_class` that takes the actual and guessed classifications, returns a 1 if the classification is correct, or a 0 otherwise. To determine the classification from a guess, take the maximum value of the guess to be the classification. We are not considering the more flexible error estimate here.

3.6 Applying the Convolutional Neural Network Fill in the final pieces of `simple_cnn` where indicated. This should be relatively clear given everything you have completed up to this point. To visualize the effects of the training, uncomment the plot section at the end of `simple_cnn`. You should notice that after a certain point, while the classification performance does not degrade, the mean-squared error in the testing set begins to rise. This indicates the network has gone through over-fitting, due to the set of training examples being too sparse. For examples of what you should see in terms of error and classification loss, refer to Figures 5a and 5b, respectively.



(a) Sample error vs. epoch. (b) Sample classification loss vs. epoch.

Figure 5