Connor Hughes ME 225ML HW2

For Parts B, C, and D, please see the PDF submission. My collaborators on this assignment were Gabby Villalpando-Torres and Sasha Davydov.

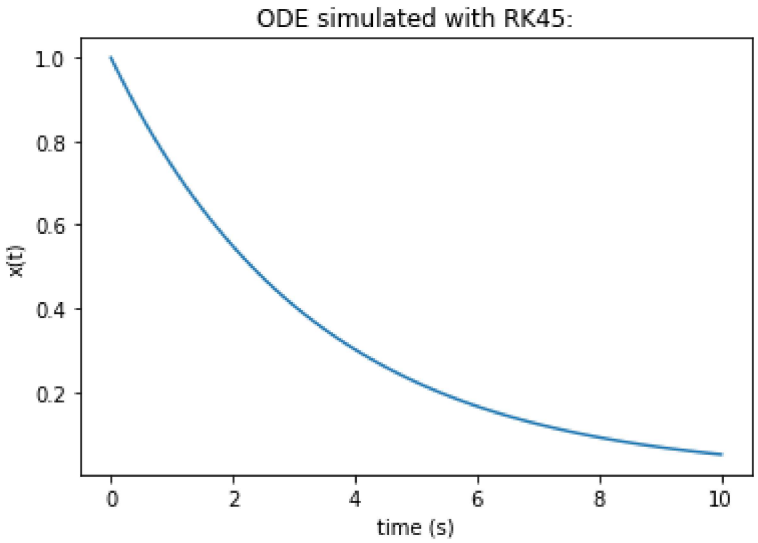
In [ ]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import tensorflow as tf
import scipy.integrate as spi
import numpy as np
import math
import matplotlib.pyplot as plt

tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
```

```
C:\Users\conno\anaconda3\envs\ME225_ML\lib\site-packages\tensorflow\python\client\session.py:1766: UserWarning: An interactive session is already active. This can cause out-of-memory errors in some cases. You must explicitly call `InteractiveSession.close()` to release resources held by the other session(s).
  warnings.warn('An interactive session is already active. This can '
```

Part A:

Simulate the ODE:

In [ ]:
```
# Define ODE:
def dxdt(self, x):
    return -0.3*x[0]

# Initial condition:
x0 = np.array([1])
t_range = np.linspace(0, 10, 100)
t_span = (0, 10)

# Solve the ODE, simulate over desired 10s time interval:
x_traj = spi.solve_ivp(dxdt, t_span, x0, 'RK45', t_range)

plt.plot(x_traj.t, np.transpose(x_traj.y))
plt.xlabel("time (s)")
plt.ylabel("x(t)")
plt.title("ODE simulated with RK45:")
plt.show()
```



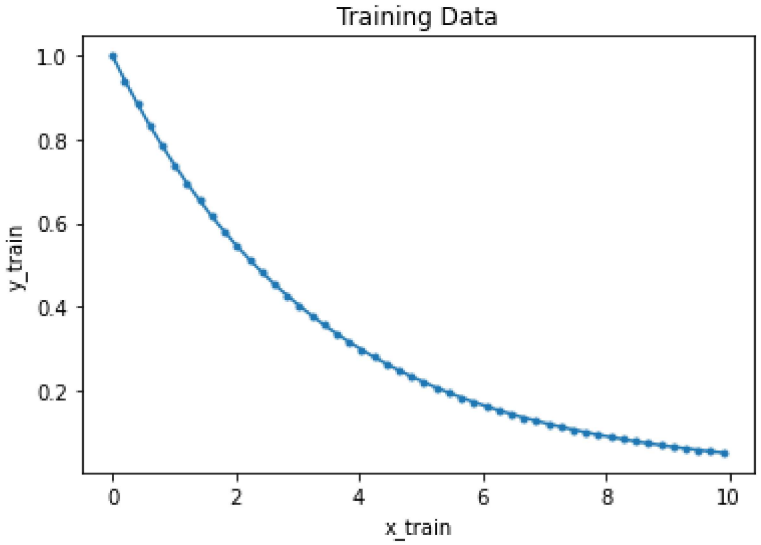
Generate Training and Testing Data:

First, we will split the simulated trajectory from above into 50 data points for training and 50 for testing. The input data points and labels are both 1-dimensional.

In [ ]:
```
# Split the input data points into training and testing data
x_train = x_traj.t[::2]
x_train = np.asarray(x_train,dtype=np.float32)
x_train = x_train.reshape(len(x_train), 1)


y_train = np.transpose(x_traj.y)[::2]
y_train = y_train.reshape(len(y_train), 1)


x_test = x_traj.t[1:100:2]
x_test = np.asarray(x_test,dtype=np.float32)
x_test = x_test.reshape(len(x_test), 1)

y_test = np.transpose(x_traj.y)[1:100:2]
y_test = y_test.reshape(len(y_test), 1)

data_feature_dim = 1
label_feature_dim = 1
```

Plot Training Data to sanity check:

In [ ]:
```
plt.plot(x_train, y_train, marker='.')
plt.title('Training Data')
plt.xlabel('x_train')
plt.ylabel('y_train')
plt.show()
```



Define Data Hooks:

In [ ]:
```
X = tf.compat.v1.placeholder(tf.float32,shape=[None,data_feature_dim])
Y = tf.compat.v1.placeholder(tf.float32,shape=[None,label_feature_dim])
```

Define Tensorflow Variables:

Here, we set the number of nodes in the hidden layer to 20. This value was determined by trial and error during repeated training and testing. The weights and biases are initialized using the Xavier Glorot initialization method.

In [ ]:
```
hidden_feature_dim = 20;
```

In [ ]:
```
w0 = tf.Variable(tf.random.truncated_normal(shape=(data_feature_dim,hidden_feature_dim)
                                            stddev = math.sqrt(3.0)/(data_feature_dim*hi
w1 = tf.Variable(tf.random.truncated_normal(shape=(hidden_feature_dim,label_feature_dim
                                            stddev = math.sqrt(3.0)/(data_feature_dim*la


bvec = tf.Variable(tf.random.truncated_normal(shape=(hidden_feature_dim,1),mean=np.rand
                                            stddev = math.sqrt(3.0)/2),dtype=np.float32)
bscalar = tf.Variable(tf.random.truncated_normal(shape=(1,1),mean=np.random.rand(1,1),\
                                            stddev = math.sqrt(3.0)),dtype=np.float32)
```

Define Neural Network Layers:

Here, we define our neural network to have only a single hidden layer, wherein each node uses a ReLU activation function. After this hidden layer, we perform another affine transformation with weights and a bias term (which are to be learned) so that we can generate a scalar output as desired.

In [ ]:
```
z = tf.nn.relu(tf.add(tf.matmul(X,w0),tf.transpose(bvec)))
yhat = tf.add(tf.matmul(z, w1), bscalar)
```

Define Loss Function:

Here, we use the suggested Loss Function, so that our learning algorithm minimizes the maximum 2-norm between the labels and the values estimated for each of the corresponding input data points.

In [ ]:
```
Loss_Function = tf.reduce_max(tf.linalg.norm((yhat)-Y))
```

Initialize Variables and Train the Neural Network:

Below, we set the learning rate to 0.4, which was selected through trial-and-error during numerous training and testing processes. Additionally, we set the number of epochs to 100,000 -- a value which was chosen such that the training loss converged satisfactorily, after having identified effective choices for the number of nodes and the learning rate.

In [ ]:
```
show_every=1000
Learning_Rate = 0.4;
Session_Optimizer = tf.compat.v1.train.AdagradOptimizer(Learning_Rate).minimize(Loss_Fu
result = sess.run(tf.compat.v1.global_variables_initializer())
epochs = 100000
```

In [ ]:
```
for i in range(epochs):
    sess.run(Session_Optimizer,feed_dict={X:x_train,Y:y_train})
    if i % show_every ==0:
        cur_loss_train = sess.run(Loss_Function,feed_dict={X:x_train,Y:y_train});
#        cur_loss_val = sess.run(Loss_Function,feed_dict={X:x_val,y:y_val});
        print("Training loss at Epoch # ", i , "is : ", cur_loss_train/y_train.shape[0]
#        print("Validation loss at Epoch # ", i , "is : ", cur_loss_val/y_val.shape[0])
```

```
Training loss at Epoch #  0 is :  6.81454345703125
Training loss at Epoch #  1000 is :  0.01219227910041809
Training loss at Epoch #  2000 is :  0.006530978083610535
Training loss at Epoch #  3000 is :  0.004639924764633179
Training loss at Epoch #  4000 is :  0.00371963262557983 4
Training loss at Epoch #  5000 is :  0.0031842872500419616
Training loss at Epoch #  6000 is :  0.0028359419107437133
Training loss at Epoch #  7000 is :  0.0025893589854240418
Training loss at Epoch #  8000 is :  0.002405630499124527
Training loss at Epoch #  9000 is :  0.0022432367503643037
Training loss at Epoch #  10000 is :  0.0021217462420463564
Training loss at Epoch #  11000 is :  0.002011801302433014
Training loss at Epoch #  12000 is :  0.001921193301677704
Training loss at Epoch #  13000 is :  0.0018420229852199555
Training loss at Epoch #  14000 is :  0.0017775264382362366
Training loss at Epoch #  15000 is :  0.001709049493074417
Training loss at Epoch #  16000 is :  0.0016449955105781555
Training loss at Epoch #  17000 is :  0.0016023679077625274
Training loss at Epoch #  18000 is :  0.0015564198791980744
Training loss at Epoch #  19000 is :  0.0015099564194679261
Training loss at Epoch #  20000 is :  0.0014745184779167175
Training loss at Epoch #  21000 is :  0.0014349506795406342
Training loss at Epoch #  22000 is :  0.0013972674310207367
```

```
Training loss at Epoch #  23000 is :  0.0013640360534191132
Training loss at Epoch #  24000 is :  0.0013322106003761291
Training loss at Epoch #  25000 is :  0.0013029776513576508
Training loss at Epoch #  26000 is :  0.0012758669257164
Training loss at Epoch #  27000 is :  0.0012512803077697754
Training loss at Epoch #  28000 is :  0.0012278180569410325
Training loss at Epoch #  29000 is :  0.0012059617042541504
Training loss at Epoch #  30000 is :  0.001184849664568901
Training loss at Epoch #  31000 is :  0.001165759339928627
Training loss at Epoch #  32000 is :  0.0011480072140693666
Training loss at Epoch #  33000 is :  0.001130605936050415
Training loss at Epoch #  34000 is :  0.0011145510524511338
Training loss at Epoch #  35000 is :  0.001098928600549698
Training loss at Epoch #  36000 is :  0.001083446741104126
Training loss at Epoch #  37000 is :  0.0010697522759437561
Training loss at Epoch #  38000 is :  0.0010563432425260543
Training loss at Epoch #  39000 is :  0.001044163629412651
Training loss at Epoch #  40000 is :  0.00103131502866745
Training loss at Epoch #  41000 is :  0.0010198959708213806
Training loss at Epoch #  42000 is :  0.0010091472417116166
Training loss at Epoch #  43000 is :  0.0009980086982250213
Training loss at Epoch #  44000 is :  0.0009876183420419694
Training loss at Epoch #  45000 is :  0.000977853387594223
Training loss at Epoch #  46000 is :  0.0009685403853654861
Training loss at Epoch #  47000 is :  0.0009592627733945847
Training loss at Epoch #  48000 is :  0.0009509707987308502
Training loss at Epoch #  49000 is :  0.00094255231320858
Training loss at Epoch #  50000 is :  0.0009340509027242661
Training loss at Epoch #  51000 is :  0.0009260946512222291
Training loss at Epoch #  52000 is :  0.0009185886383056641
Training loss at Epoch #  53000 is :  0.0009113019704818726
Training loss at Epoch #  54000 is :  0.0009041181951761246
Training loss at Epoch #  55000 is :  0.0008974465727806092
Training loss at Epoch #  56000 is :  0.0008907284587621689
Training loss at Epoch #  57000 is :  0.0008843348920345306
Training loss at Epoch #  58000 is :  0.0008780878037214279
Training loss at Epoch #  59000 is :  0.0008720227330923081
Training loss at Epoch #  60000 is :  0.0008661501109600067
Training loss at Epoch #  61000 is :  0.0008605151623487472
Training loss at Epoch #  62000 is :  0.0008549266308546066
Training loss at Epoch #  63000 is :  0.0008496423810720444
Training loss at Epoch #  64000 is :  0.0008444671332836152
Training loss at Epoch #  65000 is :  0.0008394240587949753
Training loss at Epoch #  66000 is :  0.0008345447480678558
Training loss at Epoch #  67000 is :  0.0008297897130250931
Training loss at Epoch #  68000 is :  0.0008251898735761642
Training loss at Epoch #  69000 is :  0.0008206401765346528
Training loss at Epoch #  70000 is :  0.0008161614090204239
Training loss at Epoch #  71000 is :  0.0008121225982904434
Training loss at Epoch #  72000 is :  0.0008079788088798523
Training loss at Epoch #  73000 is :  0.0008038050681352615
Training loss at Epoch #  74000 is :  0.0007999186962842942
Training loss at Epoch #  75000 is :  0.0007960494607686996
Training loss at Epoch #  76000 is :  0.0007922417670488358
Training loss at Epoch #  77000 is :  0.0007887548953294754
Training loss at Epoch #  78000 is :  0.0007853662222623826
Training loss at Epoch #  79000 is :  0.0007816825062036514
Training loss at Epoch #  80000 is :  0.0007784733176231384
Training loss at Epoch #  81000 is :  0.0007752604782581329
Training loss at Epoch #  82000 is :  0.0007721143960952759
```

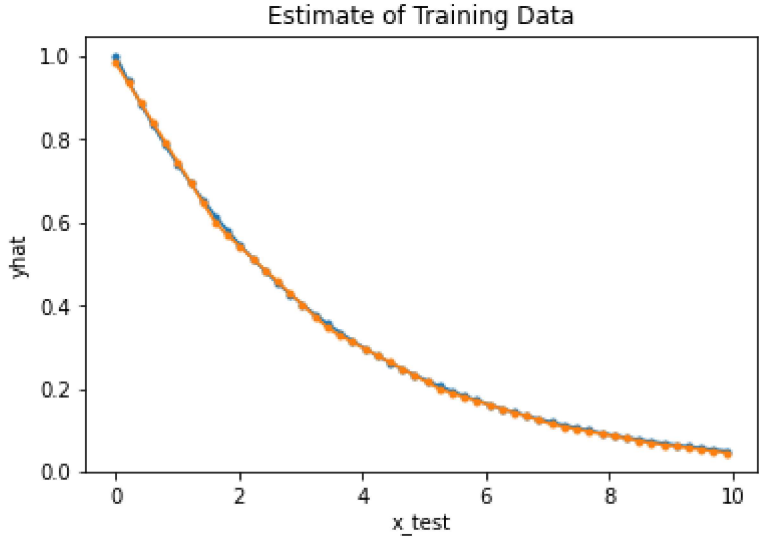
```
Training loss at Epoch #  83000 is :  0.0007691334933042527
Training loss at Epoch #  84000 is :  0.000765763446688652
Training loss at Epoch #  85000 is :  0.0007629180699586869
Training loss at Epoch #  86000 is :  0.0007599437981843948
Training loss at Epoch #  87000 is :  0.0007571236789226532
Training loss at Epoch #  88000 is :  0.0007547151297330856
Training loss at Epoch #  89000 is :  0.0007521117478609085
Training loss at Epoch #  90000 is :  0.0007493650913238526
Training loss at Epoch #  91000 is :  0.0007467395812273026
Training loss at Epoch #  92000 is :  0.0007440874725580215
Training loss at Epoch #  93000 is :  0.0007416794449090958
Training loss at Epoch #  94000 is :  0.0007395369559526444
Training loss at Epoch #  95000 is :  0.0007371197640895843
Training loss at Epoch #  96000 is :  0.000734962522983551
Training loss at Epoch #  97000 is :  0.0007327646762132644
Training loss at Epoch #  98000 is :  0.000730740949511528
Training loss at Epoch #  99000 is :  0.0007284944504499435
```

Plot Fit to Training Data:

Below, we plot the input and labels from the training data on the same axes as the neural network's estimates of the labels, to visualize the performance of the network. The plot shows that the neural network does well to approximate the desired function over the training data set, as expected.

In [ ]:

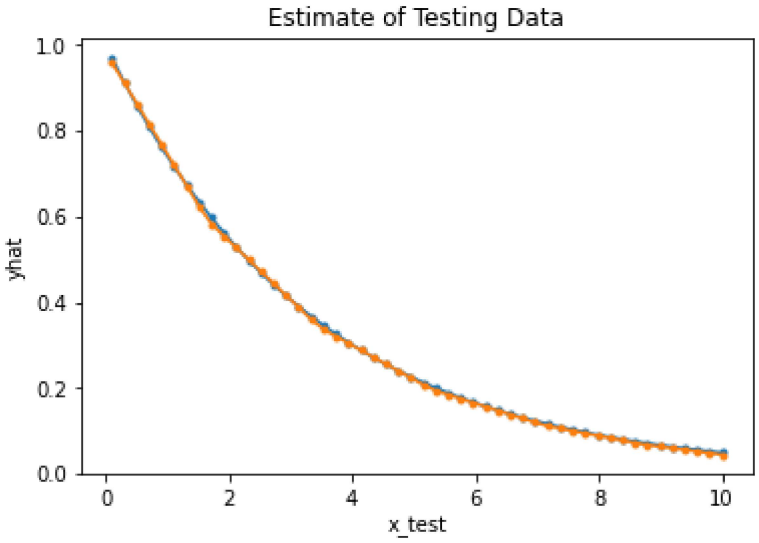
```
yhat_train = yhat.eval(feed_dict={X:x_train,Y:y_train})
plt.plot(x_train, y_train, marker='.')
plt.title('Testing Data')
plt.xlabel('x_train')
plt.ylabel('y_train')

plt.plot(x_train, yhat_train, marker='.')
plt.title('Estimate of Training Data')
plt.xlabel('x_test')
plt.ylabel('yhat')

plt.show()
```



Plot Fit to Testing Data:

Below, we compute the neural network's estimates of the labels for the inputs from the testing data set, and plot these alongside the known labels for the testing data set. As shown in the plot below, the network also does well to approximate the desired function over the testing data set.

In [ ]:
```
yhat_test = yhat.eval(feed_dict={X:x_test,Y:y_test})
plt.plot(x_test, y_test, marker='.')
plt.title('Testing Data')
plt.xlabel('x_test')
plt.ylabel('y_test')

plt.plot(x_test, yhat_test, marker='.')
plt.title('Estimate of Testing Data')
plt.xlabel('x_test')
plt.ylabel('yhat')

plt.show()
```



Compute Average Loss Function over Testing Data:

Below, we evaluate the average of the 2-norm of the error between the true testing data labels and the predictions generated by the neural network. The final value is approximately 0.00368, which satisfies our performance benchmark of a maximum 2-norm error of 0.005.

In [ ]:
```
err = yhat_test - y_test
sq_err = np.multiply(err, err)
pwr = np.full((len(sq_err), 1), 0.5)
rtsq_err = np.power(sq_err, pwr)
TestLoss = np.mean(rtsq_err)

print(TestLoss)
```

```
0.0036846914873479154
```

Examine the Learned Weights and Biases:

Below, we print out the learned weight vectors and biases which enabled the neural network to accurately approximate the desired function.

In [ ]:
```
w0.eval()
```

Out[ ]:
```
array([[-0.07879049, -0.10509172, -0.04424781, -0.06509826, -0.03407151,
         0.16842546, -0.16891085, -0.08701259,  0.02117357, -0.33708465,
         0.06882526,  0.13377638,  0.00232994,  0.17267834,  0.05517942,
         0.05118841, -0.03458065, -0.37818533,  0.10841689,  0.08589365]],
      dtype=float32)
```

In [ ]:
```
w1.eval()
```

Out[ ]:
```
array([[ 2.0812938e-01],
       [ 1.4236367e+00],
       [ 1.4268253e+00],
       [ 2.1429482e+00],
       [-2.9077837e-01],
       [-2.5730661e-01],
       [ 2.0045754e-01],
       [ 1.6488566e+00],
       [ 2.5966363e+00],
       [ 1.7165309e-01],
       [-1.0067239e+00],
       [ 1.3535878e-01],
       [ 1.3753768e-03],
       [ 3.4606472e-02],
       [ 2.2997262e+00],
       [-4.9234916e-02],
       [ 2.5199144e+00],
       [ 2.6335305e-01],
       [ 2.6856115e-02],
       [-1.7494133e+00]], dtype=float32)
```

In [ ]:
```
bvec.eval()
```

Out[ ]:
```
array([[-5.3601801e-02],
       [-7.4867558e-01],
       [-3.8137019e-02],
       [-1.0557363e+00],
       [-4.9971342e-01],
       [ 1.1868802e+00],
       [ 8.9358628e-01],
       [-3.7020725e-01],
       [-4.1343197e-01],
       [ 1.1667192e+00],
       [-8.2408446e-01],
       [-9.5189452e-01],
       [-1.9466095e-02],
       [-1.5107638e+00],
       [-6.5293813e-01],
       [ 3.0222535e-04],
       [-1.1720126e+00],
       [ 6.2455076e-01],
       [-9.2741472e-01],
       [-1.1333020e+00]], dtype=float32)
```

In [ ]:
```
bscalar.eval()
```

Out[ ]:
```
array([[0.74507546]], dtype=float32)
```

Conclusion:

In summary, this was another friendly introduction to the process of training a neural network. We generated training and testing data, selected an architecture for the neural network, utilized Tensorflow to train the network, and evaluated the performance after training. The most challenging aspect was simply experimenting with numerous options for the network architecture, and tuning hyperparameters including the learning rate and the number of epochs such that the desired performance was achieved.

Connor Hughes
2/3/22
ME 225 ML HW 2

Part B:

List and argue sufficient conditions such that the solution $x(t)$ to the system

$$\frac{dx}{dt} = Ax + Bu$$

can be approximated by a neural network.

Solution:

Here we interpret the question to ask when the solution $x(t)$ can be approximated arbitrarily well by a shallow neural network, so we will make our selection of sufficient conditions with the aim of applying the Universal Approximation Theorem.

In class, we were given the following two forms of the Universal Approximation Theorem:

> **Theorem 1:** If the activn fn Ψ is unbounded and nonconstant, then $\mathcal{N}$ is dense in $L^p(\mu)$ for all finite measures μ on $\mathbb{R}^k$.
>
> **Theorem 2:** If the activn fn Ψ is continuous, bounded, and non-constant, then $\mathcal{N}$ is dense in the space of all continuous functions $C(X)$ on all compact subsets $X \subset \mathbb{R}^k$.

We have seen examples of activation functions which are unbounded and non-constant (ReLU, the identity fn, etc) as well as those which are continuous, bounded, and nonconstant (logistic fn, sigmoid fn, etc) so it is trivial that we can find activation functions Ψ which satisfy the requirements of either Thm 1 or Thm 2 above.

Since $\mathcal{N}$ is the set of functions which can be represented exactly by a shallow neural network with a sufficient number of nodes, Thm 1 states that a NN can approximate any function $f \in L^p(\mu)$ arbitrarily well (for some finite measure μ). Similarly, Thm 2 states that a NN can approximate any continuous function arbitrarily well over any compact subset of the input domain $\mathbb{R}^k$.

Connor Hughes
2/3/22
ME 225ML HW2

Part B (cont.):

So, we will provide conditions guaranteeing $x(t) \in L^p(\mu)$ for some finite measure $\mu(t)$, then apply Thm 1 to show $x(t)$ can be approximated arbitrarily well by a NN.

To have $x(t) \in L^p(\mu)$ we need:

$$\|x(t)\|_{L^p} = \left(\int_{\mathbb{R}} (x(t))^p \, \mu(t) \, dt \right)^{1/p} < \infty$$

$$\Leftrightarrow \int_{\mathbb{R}} (x(t))^p \, \mu(t) \, dt < \infty$$

$$\Leftrightarrow \lim_{t \to \pm\infty} (x(t))^p \mu(t) = 0$$

and $(x(t))^p \mu(t) \xrightarrow[t \to \pm\infty]{} 0$ at a rate faster than $\frac{1}{t}$ ⊛

To provide a comprehensive set of sufficient conditions on A, B, and μ guaranteeing ⊛ holds is a tremendous task. We list and argue below a reasonably broad set of such conditions, and will not concern ourselves with ensuring the set is comprehensive.

We assume that the given system is time-invariant, as is loosely implied by the notation used in the problem. Then, by the variation of constants formula, we have:

$$x(t) = e^{A(t-t_0)} x(t_0) + \int_{t_0}^{t} e^{A(\tau - t_0)} B u(\tau) \, d\tau$$

We can see from this formula that $x(t)$ cannot blow up in finite time unless $u(t)$ blows up in finite time. If this occurs, so long as $u(t)$ does not contain any Dirac delta functions, $x(t)$ will still be continuous over all compact subsets of t whose upper bound is less than the t-value at which $\|x(t)\|$ asymptotically approaches ∞. Thus, by Thm 2 we can approximate $x(t)$ arbitrarily accurately over any of these compact subsets of t.

Connor Hughes
2/3/22
ME 225 ML HW2

Part B (cont.):

If, alternatively, $u(t)$ does not blow up in finite time, then we know $x(t)$ does not blow up in finite time. Still, $x(t)$ could blow up as $t \to \pm\infty$. The rate at which this occurs is partially governed by $u(t)$. If the spectral abscissa of A is positive, then even for bounded $u(t)$ it's possible that $\|x(t)\|$ could grow exponentially fast. If $u(t)$ either decays with t, is constant, or grows at a rate no faster than exponential growth, then by the variation of constants formula, we see that $\|x(t)\| \leq \beta e^{\alpha t}$ for some sufficiently large $\beta, \alpha \in \mathbb{R}_{\geq 0}$.

For simplicity, we will not consider the case wherein $u(t)$ grows faster than exponentially as $t \to \infty$.

So then, we need only to find a measure $\mu(t)$ such that $\beta e^{\alpha t}\mu(t) \xrightarrow[\text{as } t \to \pm\infty]{} 0$ faster than $\frac{1}{t}$. We can see that this is clearly the case if $\mu(t) = \gamma e^{-\zeta t^2}$, where $\gamma, \zeta \in \mathbb{R}$ and $\zeta > \alpha$. Such a measure is similar to a Gaussian measure. Since we need only to find some measure satisfying ⊛, we see that if $u(t)$ does not grow faster than exponentially, $x(t) \in L^p(\mu)$ for some μ as described above, and thus $x(t)$ can be approximated arbitrarily well by a NN over all $t \in \mathbb{R}$. Note ⊛ holds trivially if $\|x(t)\|$ is bounded $\forall t$, so we need not make any restrictions on A and B except that their entries be finite in order to apply Thm 1.

Connor Hughes
2/3/22
ME 225 ML HW 2

Part C: List and argue sufficient conditions such that the solution $x(t)$ to the system

$$\frac{dx}{dt} = f(x)$$

can be approximated by a neural network.

Solution:

The same assumptions about the intent of the question described in Part B are made again here, so we again seek to apply one of the 2 given forms of the Universal Approximation Theorem.

In general, the system can have any number of discontinuities for which $\|x(t)\| \to \infty$ in finite time. In this case, as long as we restrict our attention to compact subsets of $t \in \mathbb{R}$ s.t. $\|x(t)\| < \infty$, Thm 2 ensures we can approximate $x(t)$ arbitrarily accurately with a NN. Similarly to part B, we here must assume $f(x)$ does not contain Dirac delta functions to ensure continuity of $x(t)$.

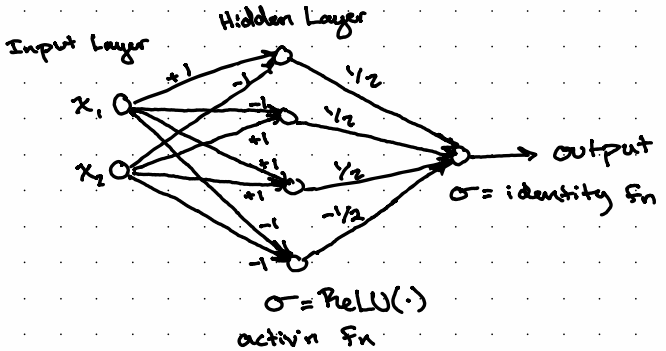
If the system does not contain any such discontinuities, then either $x(t)$ is bounded $\forall t$ or $\|x(t)\| \to \infty$ as $t \to \pm\infty$. The discussion here becomes the same as for Part B. In general, $\|x(t)\|$ could grow faster than exponentially, but for simplicity we assume this is not the case, and we are able to apply Thm 1 in the same manner as in part B.

Connor Hughes
2/3/22
ME 225 ML HW2

Part D:

We will show that the function $f(x_1, x_2) = \max\{x_1, x_2\}$ can be written exactly as the output of a shallow ReLU network with 4 nodes by construction, that is, by providing an example of such a network.

Consider the following neural network:



With the weights and activation functions shown above, we can see that the action of the neural network can be expressed as follows:

Hidden Unit #:	Precursor	Neuron Output
1	$x_1 - x_2$	$\text{ReLU}(x_1 - x_2) = \begin{cases} x_1 - x_2 & \text{if } x_1 > x_2 \\ 0 & \text{if } x_2 \leq x_1 \end{cases}$
2	$x_2 - x_1$	$\text{ReLU}(x_2 - x_1) = \begin{cases} x_2 - x_1 & \text{if } x_2 > x_1 \\ 0 & \text{if } x_2 \leq x_1 \end{cases}$
3	$x_1 + x_2$	$\text{ReLU}(x_1 + x_2) = \begin{cases} x_1 + x_2 & \text{if } x_1 + x_2 > 0 \\ 0 & \text{if } x_1 + x_2 \leq 0 \end{cases}$
4	$-x_1 - x_2$	$\text{ReLU}(-x_1 - x_2) = \begin{cases} -x_1 - x_2 & \text{if } x_1 + x_2 < 0 \\ 0 & \text{if } x_1 + x_2 \geq 0 \end{cases}$

So, we see that the output of the NN is given by:

$$\text{output} = \begin{cases} \frac{1}{2}(x_1 + x_2) + \frac{1}{2}(x_1 - x_2) = x_1 & \text{if } x_1 \geq x_2 \\ \frac{1}{2}(x_1 + x_2) + \frac{1}{2}(x_2 - x_1) = x_2 & \text{if } x_1 < x_2 \end{cases}$$

Thus, the NN represents the function $f(x_1, x_2) = \max\{x_1, x_2\}$ exactly.