Kate Lassiter Homework 2

Problem 1 - Perceptron 15 points

Consider a 2-dimensional data set in which all points with x1 > x2 belong to the positive class, and all points with x1 ≤x2 belong to the negative class. Therefore, the true separator of the two classes is linear hyperplane (line) defined by x1 −x2 = 0. Now create a training data set with 20 points randomly generated inside the unit square in the positive quadrant. Label each point depending on whether or not the first coordinate x1 is greater than its second coordinate x2. Now consider the following loss function for training pair (¯X,y) and weight vector ¯W: L = max{0,a −y(¯W · ¯X)}, where the test instances are predicted as ˆy = sign{ ¯W · ¯X}. For this problem, ¯W = [w1,w2], ¯X = [x1,x2] and ˆy = sign(w1x1 + w2x2). A value of a = 0 corresponds to the perceptron criterion and a value of a = 1 corresponds to hinge-loss.

1. You need to implement the perceptron algorithm without regularization (don't use any existing im-plementation of perceptron), train it on the 20 points above, and test its accuracy on 1000 randomly generated points inside the unit square. Generate the test points using the same procedure as the training points. You need to have your own implementation of the perceptron algorithm. (6)

```
import numpy as np
```

```
np.random.seed(3)
X_train=np.random.uniform(0, 1, size=(20, 2))
y_train=[1 if X_train[x,0]>X_train[x,1] else 0 for x in range(len(X_train))]
X_test=np.random.uniform(0, 1, size=(1000, 2))
y_test=[1 if X_test[x,0]>X_test[x,1] else 0 for x in range(len(X_test))]
```

```
def perceptron(X,y,epochs,learning_rate=0.01,loss_func=None):
    n, d = X.shape
   # np.random.seed(3)
    w0 = np.random.uniform(-3, 3, size=(1, d))
    for epoch in range(epochs):
      for i in range(n):
          y_pred =  np.sign(np.matmul(X[i],np.transpose(w0)))
          y_error = y[i] - y_pred
          if loss_func=="hinge":
                  total_loss=np.mean(list(map(lambda x: max(1-x,0),y_pred)))
                  if y_error == 0 and total_loss > 0:
                    w0 = w0 + (learning_rate * y[i] * X[i])
                  else:
                    w0 = w0 + (learning_rate * y_error * X[i])
          else:
            w0 = w0 + (learning_rate * y_error * X[i])
    return(w0)


def evaluate(X,y,w,loss_func=None):
  y_pred=np.sign(np.matmul(X,np.transpose(w)))[:, 0]
  accuracy=np.mean(np.where(y_pred==y,1,0))
  return(accuracy)


weights=perceptron(X_train,y_train,epochs=100)
print("Accuracy: ",evaluate(X_test,y_test,weights))
```

```
Accuracy:  0.538
```

2. Change the perceptron criterion to hinge-loss in your implementation for training, and repeat the accuracy computation on the same test points above. Regularization is not used. (5)

```
weights2=perceptron(X_train,y_train,epochs=100,loss_func="hinge")
print("Accuracy: ",evaluate(X_test,y_test,weights2))
```

```
Accuracy:  0.505
```

3. In which case do you obtain better accuracy and why? (2)

You obtain better accuracy with the hinge-loss rather than the regular loss on this particular dataset. With hinge loss, if a data point is far from the decison boundary, nothing is added to the loss. This can make it less sensitive to overfitting, converge faster, and be more able to generalize depending on what data it is fed.

4. In which case do you think that the classification of the same 1000 test instances will not change significantly by using a different set of 20 training points? (2)

The hinge loss would not change very dramatically in this case, it is because it seeks to find the maximum margin seperation of the classes and doesn't consider points far from the margin, so as long as they followed the same general pattern the hinge loss perceptron would be less afected.

1. Explain vanishing gradients phenomenon using standard normalization with different values of standard deviation and tanh and sigmoid activation functions.

When a network is many layers, the derivative calculated in backpropagation diminishes and results in a near zero partial derivative of the loss function, rendering the network unable to learn, or at least very slowly, and update weights accordingly. This often occurs due to the choice of activation function and weight intialization scheme.

2) The dying ReLU is a kind of vanishing gradient, which refers to a problem when ReLU neurons become inactive and only output 0 for any input. In the worst case of dying ReLU, ReLU neurons at a certain layer are all dead, i.e., the entire network dies and is referred as the dying ReLU neural networks in Lu et al (reference below). A dying ReLU neural network collapses to a constant function. Show this phenomenon using any one of the three 1-dimensional functions in page 11 of Lu et al. Use a 10-layer ReLU network with width 2 (hidden units per layer). Use minibatch of 64 and draw training data uniformly from Perform 1000 independent training simulations each with 3,000 training points. Out of these 1000 simulations, what fraction resulted in neural network collapse. Is your answer close to over 90% as was reported in Lu et al. ?

```
import matplotlib.pyplot as plt
from tensorflow.keras.optimizers import SGD


X_train = np.random.uniform(-np.sqrt(7),np.sqrt(7),3000).reshape(-1,1)
y_train = abs(X_train.reshape(-1,1))
X_test = np.random.uniform(-np.sqrt(7),np.sqrt(7),500).reshape(-1,1)
y_test = abs(X_test.reshape(-1,1))


model = Sequential([
  Dense(2, input_dim=1,activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(2, activation="relu"),
  Dense(1)
  ])
```

```
model.summary()
opt=SGD(learning_rate=0.01, momentum=0.9 ,nesterov=True, name="SGD")
model.compile(loss="mean_squared_error", optimizer=opt)
```

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_39 (Dense)            (None, 2)                 4         

 dense_40 (Dense)            (None, 2)                 6         

 dense_41 (Dense)            (None, 2)                 6         

 dense_42 (Dense)            (None, 2)                 6         

 dense_43 (Dense)            (None, 2)                 6         

 dense_44 (Dense)            (None, 2)                 6         

 dense_45 (Dense)            (None, 2)                 6         

 dense_46 (Dense)            (None, 2)                 6         

 dense_47 (Dense)            (None, 2)                 6         

 dense_48 (Dense)            (None, 2)                 6         

 dense_49 (Dense)            (None, 1)                 3         

=================================================================
Total params: 61
Trainable params: 61
Non-trainable params: 0
_________________________________________________________________
```

```
def simulation(model_1,X,y,X_val,y_val,n):
  collapse =0
  for x in range(n):
    history = model_1.fit(X,y, epochs=2, batch_size=64, validation_data=(X_val, y_v
    y_pred = model_1.predict(X_val)
    if np.var(y_pred) <= 10e-4:
      collapse = collapse + 1
  return(collapse/n)
```

```
collapsed=simulation(model,X_train,y_train,X_test, y_test,n=1000)
```

```
print("Fraction of collapsed networks: ",collapsed)
```

```
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5760 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5771 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5762 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5767 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5765 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 7ms/step - loss: 0.5763 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5773 - val_lo
16/16 [==============================] - 0s 2ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5770 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5766 - val_lo
16/16 [==============================] - 0s 4ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5765 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5765 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5763 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5769 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5763 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5768 - val_lo
16/16 [==============================] - 0s 4ms/step
Epoch 1/2
47/47 [==============================] - 0s 7ms/step - loss: 0.5770 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5768 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5763 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5775 - val_lo
16/16 [==============================] - 0s 3ms/step
```

```
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5766 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5771 - val_lo
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 0.5766 - val_lo
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 0.5768 - val_lo
16/16 [==============================] - 0s 3ms/step
Fraction of collapsed networks:  1.0
```

The rate of collapsed networks is very high, actually 100% of the networks in my case, similar to what was shown in Lu et al.

3. Instead of ReLU consider Leaky ReLU activation as defined below:Run the 1000 training simulations in part 2 with Leaky ReLU activation and keeping everything else same. Again calculate the fraction of simulations that resulted in neural network collapse. Did Leaky ReLU help in preventing dying neurons ? (10)

```
model = Sequential([
  Dense(2, input_dim=1,activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(2, activation="leaky_relu"),
  Dense(1)
  ])

model.summary()
opt=SGD(learning_rate=0.01, momentum=0.9 ,nesterov=True, name="SGD")
model.compile(loss="mean_squared_error", optimizer=opt)
```

```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_50 (Dense)            (None, 2)                 4

 dense_51 (Dense)            (None, 2)                 6

 dense_52 (Dense)            (None, 2)                 6

 dense_53 (Dense)            (None, 2)                 6

 dense_54 (Dense)            (None, 2)                 6

 dense_55 (Dense)            (None, 2)                 6

 dense_56 (Dense)            (None, 2)                 6

 dense_57 (Dense)            (None, 2)                 6

 dense_58 (Dense)            (None, 2)                 6

 dense_59 (Dense)            (None, 2)                 6

 dense_60 (Dense)            (None, 1)                 3

=================================================================
Total params: 61
Trainable params: 61
Non-trainable params: 0
_________________________________________________________________
```

```
collapsed=simulation(model,X_train,y_train,X_test, y_test,n=1000)
print("Fraction of collapsed networks: ",collapsed)
```

```
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 1.9768e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9764e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 4ms/step - loss: 1.9575e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9635e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 1.9864e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
```

```
47/47 [==============================] - 0s 6ms/step - loss: 1.9828e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 1.9773e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9656e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 1.9800e-07 - va
16/16 [==============================] - 0s 4ms/step
Epoch 1/2
47/47 [==============================] - 0s 7ms/step - loss: 1.9646e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 5ms/step - loss: 1.9741e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9636e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 4ms/step - loss: 1.9710e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9598e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9705e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9611e-07 - va
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9756e-07 - va
16/16 [==============================] - 0s 2ms/step
Epoch 1/2
47/47 [==============================] - 0s 10ms/step - loss: 1.9734e-07 - v
Epoch 2/2
47/47 [==============================] - 0s 7ms/step - loss: 1.9687e-07 - va
16/16 [==============================] - 0s 5ms/step
Epoch 1/2
47/47 [==============================] - 1s 12ms/step - loss: 1.9671e-07 - v
Epoch 2/2
47/47 [==============================] - 0s 6ms/step - loss: 1.9540e-07 - va
16/16 [==============================] - 0s 3ms/step
Epoch 1/2
47/47 [==============================] - 0s 10ms/step - loss: 1.9745e-07 - v
Epoch 2/2
47/47 [==============================] - 0s 7ms/step - loss: 1.9673e-07 - va
16/16 [==============================] - 0s 3ms/step
Fraction of collapsed networks:  0.002
```

It seems in this case utilizing LeakyReLU allowed for a very small portion of the networks to collapse, much less than with reLU.

Problem 3 - Batch Normalization, Dropout, MNIST 25 points

Batch normalization and Dropout are used as effective regularization techniques. However its not clear which one should be preferred and whether their benefits add up when used in conjunction. In this problem we will compare batch normalization, dropout, and their conjunction using MNIST and LeNet-5 (see e.g.,http://yann.lecun.com/exdb/lenet/). LeNet-5 is one of the earliest convolutional neural network developedfor image classification and its implementation in all major framework is available. You can refer to Lecture 3 slides for definition of standardization and batch normalization.

1. Explain the terms co-adaptation and internal covariance-shift. Use examples if needed. You may need o refer to two papers mentioned below to answer this question. (5)

Co-adaptation occurs when hidden units don't learn relavent information on their own, they rely on another hidden units to detect features. This can have a negative impact on the model's ability to generalize and on the efficiency of the network. Regularization is one option in handling co-adaptation. Internal covariance-shift occurs when the distribution of each layers input changes due to the changing parameters of the layer before it. Batch normalization is one technique to reduce this problem by keeping the hidden layers' distributions stable during training.

2. Batch normalization is traditionally used in hidden layers, for input layer standard normalization is used. In standard normalization the mean and standard deviation are calculated using the entire training dataset whereas in batch normalization these statistics are calculated for each mini-batch. Train LeNet-5 with standard normalization of input and batch normalization for hidden layers. What are the learned batch norm parameters for each layer ? (5)

```
from keras.datasets import mnist
from keras.utils import np_utils
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import LayerNormalization
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.layers import Input
from tensorflow.random import set_seed
from tensorflow.keras.layers import Dropout
from keras.datasets import mnist
```

```
#load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

X_train = X_train/255
X_test = X_test/255
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datas
11490434/11490434 [==============================] - 0s 0us/step
```

```
#LENet 5 architechture reference
#https://thecleverprogrammer.com/2020/11/05/cnn-architectures-in-machine-learning/h
model = Sequential([
        #c1
        Conv2D(filters=6, kernel_size=(5, 5),padding='same', activation='tanh', inp
        BatchNormalization(trainable=True),
        #S2
        AveragePooling2D(strides=(2, 2)),
        #C3
        Conv2D(filters=16, kernel_size=(5, 5), activation='tanh'),
        BatchNormalization(trainable=True),
        #S4
        AveragePooling2D(strides=(2, 2)),
        #c5
        Conv2D(filters=120, kernel_size=(5, 5), activation='tanh'),
        BatchNormalization(trainable=True),
        Flatten(),
        #F6
        Dense(units=84, activation='tanh'),
        BatchNormalization(trainable=True),
        #out
        Dense(units=10, activation='softmax')
  ])
```

```
# Compile the model
```

```
model.compile(loss=categorical_crossentropy, optimizer='SGD', metrics=['accuracy'])

history = model.fit(x=X_train,y=y_train, epochs=5, batch_size=100, validation_data=

accuracy = model.evaluate(X_test, y_test)
print('Accuracy',accuracy[1] * 100)
```

```
Epoch 1/5
600/600 [==============================] - 54s 88ms/step - loss: 0.2855 - accu
Epoch 2/5
600/600 [==============================] - 52s 87ms/step - loss: 0.1329 - accu
Epoch 3/5
600/600 [==============================] - 63s 105ms/step - loss: 0.0934 - acc
Epoch 4/5
600/600 [==============================] - 58s 97ms/step - loss: 0.0742 - accu
Epoch 5/5
600/600 [==============================] - 52s 87ms/step - loss: 0.0621 - accu
313/313 [==============================] - 4s 13ms/step - loss: 0.0580 - accur
Accuracy 98.3299970626831
```

Batch norms:

```
layer_names=['batch_normalization_19','batch_normalization_20','batch_normalizatior
vars=[]
means=[]
for x in layer_names:
    print("Layer: ",x,"\n",model.get_layer(x).weights)
```

```
       1.0247546, 1.0184675, 1.0317417, 1.0259274, 1.0056797, 1.022772 ,
       1.032289 , 1.0541441, 1.0063851, 1.0402639, 1.020713 , 1.0163351],
      dtype=float32)>, <tf.Variable 'batch_normalization_22/beta:0' shape=(8
 array([-3.4591847e-03,  6.3370843e-03,  1.8992595e-02, -4.3227505e-03,
        -1.1265075e-03,  6.8920278e-03,  1.1990869e-02, -2.1135936e-02,
         2.7094146e-03,  3.0348988e-02, -8.0047650e-03, -3.4754556e-03,
         5.6764850e-04,  8.8920891e-03, -1.4458189e-02, -4.6417024e-03,
         5.8946973e-03, -6.9382787e-03, -6.3201203e-04, -1.6645587e-03,
        -1.4830983e-02,  9.9052545e-03,  1.1317381e-02, -3.1654793e-03,
         4.1103191e-03,  7.5556780e-04,  1.6434144e-02, -1.8535279e-02,
         2.4164099e-02, -2.8911276e-02,  2.9644223e-02,  1.7746236e-02,
        -2.0883126e-02, -4.2022867e-03, -1.5317114e-02,  1.3592099e-02,
        -3.5078912e-03, -2.5804716e-03,  7.9675391e-03, -2.3031717e-03,
         8.4304996e-03,  3.3787817e-02, -1.7250016e-02, -8.1648389e-03,
         8.4398111e-05,  8.7791672e-03, -5.0272967e-04,  4.7200494e-03,
         1.0148158e-02, -1.7366756e-02, -7.0349663e-03, -1.2043510e-02,
         5.6414763e-03,  6.0201110e-03,  2.5109695e-02, -4.1099719e-04,
```

```
       -1.1846489e-02,  3.1284320e-03, -2.6590266e-04, -1.9162414e-03,
        7.5358665e-03, -8.4754573e-03,  1.6741985e-02, -8.4056100e-04,
        1.2118378e-02,  1.6608343e-02, -2.3060199e-02, -9.9719251e-03,
        1.1269927e-02,  9.4916308e-03,  1.0645518e-02,  1.8355506e-02,
        1.9973962e-02,  9.3932863e-04, -1.5600471e-02, -8.3520710e-03,
       -2.6682662e-03,  4.7757203e-06, -1.0702617e-03,  2.0575395e-02,
       -1.2442287e-02, -1.6916605e-02,  2.8532347e-02, -4.4975844e-03],
      dtype=float32)>, <tf.Variable 'batch_normalization_22/moving_mean:0' s
array([ 0.02166902, -0.00607856,  0.0013245 , -0.01495499,  0.04088008,
       -0.01054503,  0.00387102,  0.00202472, -0.01841165,  0.01896789,
       -0.0051645 , -0.02724126,  0.02698041,  0.00344809,  0.00461331,
        0.05430008,  0.0064806 , -0.00563752, -0.00688195,  0.00334304,
       -0.01100581, -0.02437192, -0.03103832,  0.00024813, -0.0002834 ,
        0.00744437, -0.03882054, -0.03915241,  0.00116583,  0.00143618,
       -0.07987542,  0.01930231, -0.0108349 ,  0.0212036 , -0.04725891,
       -0.01743547,  0.02542983,  0.02355813,  0.02276972,  0.00323018,
       -0.00082608,  0.00751489, -0.02448574,  0.00339798,  0.01163169,
        0.03357039, -0.02000595,  0.0422072 , -0.00627187,  0.00149776,
       -0.00424007,  0.00259522, -0.00616793,  0.01985541,  0.00136933,
        0.01559589, -0.02684539,  0.01061125,  0.02251275, -0.01498563,
        0.01597347,  0.0128989 , -0.00369319,  0.05822136,  0.00283375,
       -0.00409714, -0.01679213,  0.00074452,  0.01118133,  0.02510788,
        0.01455487, -0.03864494, -0.04954318,  0.00358014, -0.0037163 ,
        0.02428715,  0.00142565,  0.01071631,  0.02284299, -0.012973  ,
        0.00841584, -0.03037733, -0.06305598,  0.01334748], dtype=float32)>,
array([0.20754546, 0.41405833, 0.27666324, 0.34097674, 0.38028818,
       0.3361062 , 0.5794933 , 0.5725386 , 0.30949748, 0.3775339 ,
       0.33324254, 0.32918292, 0.43821988, 0.28642556, 0.4070073 ,
       0.50510764, 0.43224156, 0.5175886 , 0.4883332 , 0.3420254 ,
       0.21994913, 0.46894422, 0.4439453 , 0.21519382, 0.4518904 ,
       0.34449106, 0.37711284, 0.53256863, 0.24214949, 0.37963554,
       0.45188335, 0.31405807, 0.5847686 , 0.43879923, 0.5298513 ,
       0.41946945, 0.53371763, 0.3515876 , 0.4986294 , 0.407164  ,
       0.29507118, 0.4960388 , 0.55443317, 0.2952215 , 0.5272664 ,
       0.3493707 , 0.49176022, 0.4864149 , 0.46336067, 0.38928163,
       0.34502614, 0.2834885 , 0.5574469 , 0.58989453, 0.574262  ,
       0.27399424, 0.42937863, 0.38144776, 0.4564133 , 0.23592441,
       0.44715455, 0.34417406, 0.31668922, 0.48398978, 0.42103395,
       0.39985856, 0.41418788, 0.33814022, 0.3277321 , 0.36868066,
       0.31920323, 0.48559114, 0.41784725, 0.29768175, 0.2878993 ,
       0.4567811 , 0.526212  , 0.4140359 , 0.4278888 , 0.32971317,
       0.3912479 , 0.36022416, 0.48112437, 0.41405407], dtype=float32)>]
```

3. Next instead of standard normalization use batch normalization for input layer also and train the network. Plot the distribution of learned batch norm parameters for each layer (including input) using violin plots. Compare the train/test accuracy and loss for the two cases ? Did batch normalization for input layer improve performance ? (5)

```
  #LENet 5 architechture reference
  #https://thecleverprogrammer.com/2020/11/05/cnn-architectures-in-machine-learning

model = Sequential([
        Input(shape=(28,28,1)),
        BatchNormalization(trainable=True),
        #c1
        Conv2D(filters=6, kernel_size=(5, 5),padding='same', activation='tanh'),#,
        BatchNormalization(trainable=True),
        #S2
        AveragePooling2D(strides=(2, 2)),
        #C3
        Conv2D(filters=16, kernel_size=(5, 5), activation='tanh'),
        BatchNormalization(trainable=True),
        #S4
        AveragePooling2D(strides=(2, 2)),
        #c5
        Conv2D(filters=120, kernel_size=(5, 5), activation='tanh'),
        BatchNormalization(trainable=True),
        Flatten(),
        #F6
        Dense(units=84, activation='tanh'),
        BatchNormalization(trainable=True),
        #out
        Dense(units=10, activation='softmax')
  ])


# Compile the model
model.compile(loss=categorical_crossentropy, optimizer='SGD', metrics=['accuracy'])

history = model.fit(x=X_train,y=y_train, epochs=5, batch_size=100, validation_data=

accuracy = model.evaluate(X_test, y_test)
print('Accuracy',accuracy[1] * 100)
```

```
Epoch 1/5
600/600 [==============================] - 60s 98ms/step - loss: 0.2539 - accu
Epoch 2/5
600/600 [==============================] - 58s 97ms/step - loss: 0.1023 - accu
Epoch 3/5
600/600 [==============================] - 58s 97ms/step - loss: 0.0733 - accu
Epoch 4/5
600/600 [==============================] - 56s 93ms/step - loss: 0.0597 - accu
Epoch 5/5
600/600 [==============================] - 58s 97ms/step - loss: 0.0512 - accu
313/313 [==============================] - 5s 17ms/step - loss: 0.0474 - accur
Accuracy 98.71000051498413
```

The training and testing accuracy and loss are better for the network with batch normalization on the input layer.

4. Train the network without batch normalization but this time use dropout. For hidden layers use dropout probability of 0.5 and for input layer take it to be 0.2 Compare test accuracy using dropout to test accuracy obtained using batch normalization in part 2 and 3. (5)

```
  #LENet 5 architechture reference
  #https://thecleverprogrammer.com/2020/11/05/cnn-architectures-in-machine-learning

model = Sequential([
        Input(shape=(28,28,1)),
        Dropout(0.2),
        #c1
        Conv2D(filters=6, kernel_size=(5, 5),padding='same', activation='tanh'),
        Dropout(0.5),
        #S2
        AveragePooling2D(strides=(2, 2)),
        #C3
        Conv2D(filters=16, kernel_size=(5, 5), activation='tanh'),
        Dropout(0.5),
        #S4
        AveragePooling2D(strides=(2, 2)),
        #c5
        Conv2D(filters=120, kernel_size=(5, 5), activation='tanh'),
        Dropout(0.5),
```

```
        Flatten(),
        #F6
        Dense(units=84, activation='tanh'),
        Dropout(0.5),
        #out
        Dense(units=10, activation='softmax')
  ])


# Compile the model
model.compile(loss=categorical_crossentropy, optimizer='SGD', metrics=['accuracy'])

history = model.fit(x=X_train,y=y_train, epochs=5, batch_size=100, validation_data=

accuracy = model.evaluate(X_test, y_test)
print('Accuracy',accuracy[1] * 100)
```

```
    Epoch 1/5
    600/600 [==============================] - 44s 72ms/step - loss: 1.4186 - accu
    Epoch 2/5
    600/600 [==============================] - 48s 80ms/step - loss: 0.7145 - accu
    Epoch 3/5
    600/600 [==============================] - 44s 73ms/step - loss: 0.6049 - accu
    Epoch 4/5
    600/600 [==============================] - 42s 71ms/step - loss: 0.5484 - accu
    Epoch 5/5
    600/600 [==============================] - 43s 72ms/step - loss: 0.5104 - accu
    313/313 [==============================] - 4s 11ms/step - loss: 0.2790 - accur
    Accuracy 91.89000129699707
```

The network with only dropout performs worse than those with batch normalization, having a lower accuracy and higher loss.

5. Now train the network using both batch normalization and dropout. How does the performance (test accuracy) of the network compare with the cases with dropout alone and with batch normalization alone ? (5)

```
  #LENet 5 architechture reference
  #https://thecleverprogrammer.com/2020/11/05/cnn-architectures-in-machine-learni

model = Sequential([
```

```
        Input(shape=(28,28,1)),
        Dropout(0.2),
        BatchNormalization(trainable=True),
        #c1
        Conv2D(filters=6, kernel_size=(5, 5),padding='same', activation='tanh'),#
        Dropout(0.5),
        BatchNormalization(trainable=True),
        #S2
        AveragePooling2D(strides=(2, 2)),
        #C3
        Conv2D(filters=16, kernel_size=(5, 5), activation='tanh'),
        Dropout(0.5),
        BatchNormalization(trainable=True),
        #S4
        AveragePooling2D(strides=(2, 2)),
        #c5
        Conv2D(filters=120, kernel_size=(5, 5), activation='tanh'),
        Dropout(0.5),
        BatchNormalization(trainable=True),
        Flatten(),
        #F6
        Dense(units=84, activation='tanh'),
        Dropout(0.5),
        BatchNormalization(trainable=True),
        #out
        Dense(units=10, activation='softmax')
  ])


# Compile the model
model.compile(loss=categorical_crossentropy, optimizer='SGD', metrics=['accuracy'

history = model.fit(x=X_train,y=y_train, epochs=5, batch_size=100, validation_dat

accuracy = model.evaluate(X_test, y_test)
print('Accuracy',accuracy[1] * 100)
```

```
Epoch 1/5
600/600 [==============================] - 62s 102ms/step - loss: 0.9527 - acc
Epoch 2/5
600/600 [==============================] - 62s 104ms/step - loss: 0.5990 - acc
Epoch 3/5
600/600 [==============================] - 62s 103ms/step - loss: 0.4893 - acc
Epoch 4/5
600/600 [==============================] - 62s 103ms/step - loss: 0.4188 - acc
Epoch 5/5
600/600 [==============================] - 62s 103ms/step - loss: 0.3674 - acc
313/313 [==============================] - 4s 13ms/step - loss: 0.1341 - accu
Accuracy 95.74000239372253
```

The accuracy and loss of this network are better than applying only dropout to a network, but it is still outperformed by the use of batch normalization alone. With dropout, training is much slower but generalization is better. But considering the network was only trained for 5 epochs, it is the worse model than batch normalization only.

Problem 4 - Universal Approximators: Depth Vs. Width 30 points Multilayer layer feedforward network, with as little as two layers and sufficiently large hidden units can approximate any arbitrary function. Thus one can tradeoff between deep and shallow networks for the same problem. In this problem we will study this tradeoff using the Eggholder function. Let y(x1,x2) = f(x1,x2) + N(0,0.3) be the function that we want to learn from a neural network through regression with $-512 \leq x1 \leq 512$ and $-512 \leq x2 \leq 512$. Draw a dataset of 100K points from this function (uniformly sampling in the range of x1 and x2) and do a 80/20 training/test split

1. Assume that total budget for number of hidden units we can have in the network is 512. Train a 1, 2, and 3 hidden layers feedforward neural network to learn the regression function. For each neural network you can consider a different number of hidden units per hidden layer so that the total number of hidden units does not exceed 512. We would recommend to work with 16, 32, 64, 128, 256, 512, hidden units per layer. So if there is only one hidden layer you can have at most 512 units in that layer. If there are two hidden layers, you can have any combination of hidden units in each layer, e.g., 16 and 256, 64 and 128, etc. such that the total is less than 512. Plot the RMSE (Root Mean Square Error) on test set for networks with different number of hidden layers as a function of total number of hidden units. If there are more than one network with the same number of hidden units (say a two hidden layer with 16 in first layer and 128 in second layer and another network with 128 in first layer and 16 in second) you will use the average RMSE. So you will have a figure with three curves, one each for 1, 2, and 3 layer networks, with x-axis being the total number of hidden units. Also plot another curve but with the x-axis being the number of parameters (weights) that you need to learn in the network. (20)

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.optimizers import SGD
from timeit import default_timer as timer
from tensorflow.keras.callbacks import Callback


def my_equation(x1,x2):
    result= -((x2+47)*np.sin(np.sqrt(abs((x1/2)+(x2+47)))))-(x1*np.sin(np.sqrt(ab
    return(result+np.random.normal(0, 0.3, 1))
```

```
np.random.seed(0) # seed random number generator
X= np.random.uniform(low=-512, high=512, size=[100000,2])
y=np.array(list(map(lambda x: my_equation(X[x,0],X[x,1]),range(len(X)) )))
X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
y_train_scaled = scaler.fit_transform(y_train)
y_test_scaled = scaler.transform(y_test)



#time the training
class TimingCallback(Callback):
    def __init__(self, logs={}):
        self.logs=[]
    def on_epoch_begin(self, epoch, logs={}):
        self.starttime = timer()
    def on_epoch_end(self, epoch, logs={}):
        self.logs.append(timer()-self.starttime)

cb = TimingCallback()


opt=SGD(learning_rate=0.01, momentum=0.9 ,nesterov=True, name="SGD")


#training one layer w/ different # hidden units
histories=[]
mse_s=[]
params=[]
times=[]
for x in [16,32,64,128,256,400]:
  print("x  ",x)
  model = Sequential([
  Dense(x, input_dim=2,activation="relu"),
  Dense(1)
  ])
  params_count=model.count_params()
  params=params+[params_count]
  model.summary()
  model.compile(loss="mean_squared_error", optimizer=opt)
  history = model.fit(X_train_scaled, y_train_scaled, epochs=2000,batch_size=1000,
```

```
  time=sum(cb.logs)
  times=times+[time]
  histories=histories+[history]
  mse_test = model.evaluate(X_test_scaled, y_test_scaled)
  mse_s=mse_s+[mse_test]


#training two layers w/ different # hidden units
hidden_layer_combos=np.array([[128, 16],[32,16],[256,128],[16, 16],[128,32]])
histories_2=[]
mse_s_2=[]
params_2=[]
times_2=[]
for x in hidden_layer_combos:
  print("x  ",x)
  model = Sequential([
  Dense(x[0], input_dim=2,activation="relu"),
  BatchNormalization(trainable=True),
  Dense(x[1], activation="relu"),
  BatchNormalization(trainable=True),
  Dense(1)
  ])
  params_count=model.count_params()
  params_2=params_2+[params_count]
  model.summary()
  model.compile(loss="mean_squared_error", optimizer=opt)
  history = model.fit(X_train_scaled, y_train_scaled, epochs=2000,batch_size=1000,
  time=sum(cb.logs)
  times_2=times_2+[time]
  histories_2=histories_2+[history]
  mse_test = model.evaluate(X_test_scaled, y_test_scaled)
  mse_s_2=mse_s_2+[mse_test]
```

```
Epoch 1972/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0469
Epoch 1973/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0469
Epoch 1974/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0476
Epoch 1975/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0451
Epoch 1976/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0473
Epoch 1977/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0449
Epoch 1978/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0455
```

```
Epoch 1979/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0461
Epoch 1980/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0461
Epoch 1981/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0476
Epoch 1982/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0488
Epoch 1983/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0480
Epoch 1984/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0498
Epoch 1985/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0456
Epoch 1986/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0458
Epoch 1987/2000
80/80 [==============================] - 1s 15ms/step - loss: 0.0487
Epoch 1988/2000
80/80 [==============================] - 1s 15ms/step - loss: 0.0479
Epoch 1989/2000
80/80 [==============================] - 1s 14ms/step - loss: 0.0455
Epoch 1990/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0462
Epoch 1991/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0447
Epoch 1992/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0491
Epoch 1993/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0474
Epoch 1994/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0442
Epoch 1995/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0436
Epoch 1996/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0448
Epoch 1997/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0466
Epoch 1998/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0472
Epoch 1999/2000
80/80 [==============================] - 1s 11ms/step - loss: 0.0456
Epoch 2000/2000
80/80 [==============================] - 1s 12ms/step - loss: 0.0484
625/625 [==============================] - 2s 3ms/step - loss: 0.0452
```

```
#get total number hidden units in each layer
num_hidden_2=[]
for x in np.array([[128, 16],[32,16],[256,128],[16, 16],[128,32]]):
  num_hidden_2=num_hidden_2+[sum(x)]


#sort by x-axis ascending
sort_num_hidden, sort_mse_num_hidden = (list(t) for t in zip(*sorted(zip(num_hidd
sort_params_mse, sort_mse = (list(t) for t in zip(*sorted(zip(params_2, mse_s_2))
sort_params_time, sort_times = (list(t) for t in zip(*sorted(zip(params_2, times_


#training three layers w/ different # hidden units
hidden_layer_combos3=np.array([[128, 16,16],[32,16,32],[128,128,64]]) #,[128,32,1
histories_3=[]
mse_s_3=[]
params_3=[]
times_3=[]
for x in hidden_layer_combos3:
  print("x  ",x)
  model = Sequential([
  Dense(x[0], input_dim=2,activation="relu"),
  BatchNormalization(trainable=True),
  Dense(x[1], activation="relu"),
  BatchNormalization(trainable=True),
  Dense(x[2], activation="relu"),
  BatchNormalization(trainable=True),
  Dense(1)
  ])
  params_count=model.count_params()
  params_3=params_3+[params_count]
  model.summary()
  model.compile(loss="mean_squared_error", optimizer=opt)
  history = model.fit(X_train_scaled, y_train_scaled, epochs=2000,batch_size=1000
  time=sum(cb.logs)
  times_3=times_3+[time]
  histories_3=histories_3+[history]
  mse_test = model.evaluate(X_test_scaled, y_test_scaled)
  mse_s_3=mse_s_3+[mse_test]
```

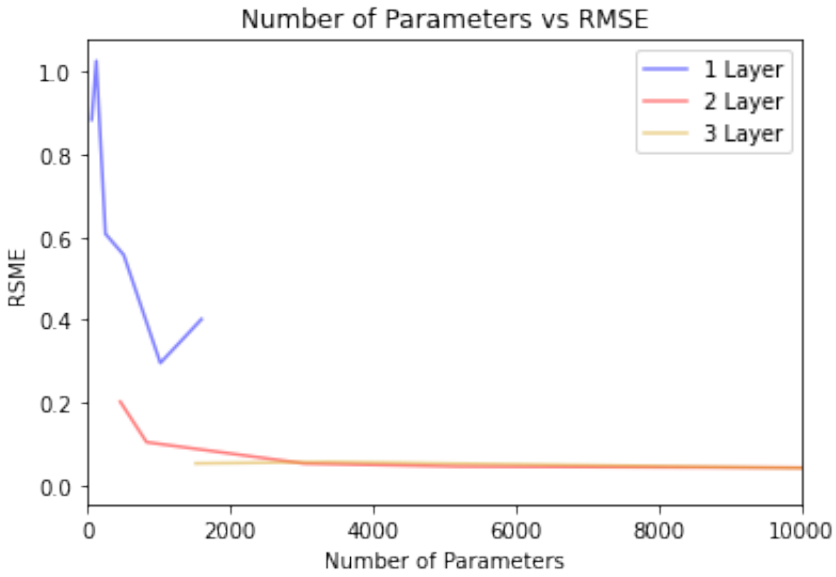
```
Epoch 1956/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0083
Epoch 1957/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0084
Epoch 1958/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0078
Epoch 1959/2000
```

```
80/80 [==============================] - 2s 22ms/step - loss: 0.0080
Epoch 1960/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0079
Epoch 1961/2000
80/80 [==============================] - 2s 24ms/step - loss: 0.0078
Epoch 1962/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0074
Epoch 1963/2000
80/80 [==============================] - 2s 23ms/step - loss: 0.0077
Epoch 1964/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0081
Epoch 1965/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0079
Epoch 1966/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0075
Epoch 1967/2000
80/80 [==============================] - 2s 24ms/step - loss: 0.0075
Epoch 1968/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0077
Epoch 1969/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0075
Epoch 1970/2000
80/80 [==============================] - 2s 24ms/step - loss: 0.0077
Epoch 1971/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0077
Epoch 1972/2000
80/80 [==============================] - 2s 25ms/step - loss: 0.0078
Epoch 1973/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0075
Epoch 1974/2000
80/80 [==============================] - 2s 23ms/step - loss: 0.0073
Epoch 1975/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0081
Epoch 1976/2000
80/80 [==============================] - 2s 23ms/step - loss: 0.0076
Epoch 1977/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0075
Epoch 1978/2000
80/80 [==============================] - 2s 23ms/step - loss: 0.0078
Epoch 1979/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0080
Epoch 1980/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0081
Epoch 1981/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0079
Epoch 1982/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0077
Epoch 1983/2000
80/80 [==============================] - 2s 22ms/step - loss: 0.0075
Epoch 1984/2000
```

```
80/80 [==============================] - 2s 21ms/step - loss: 0.0070
Epoch 1985/2000
80/80 [==============================] - 2s 21ms/step - loss: 0.0076
```

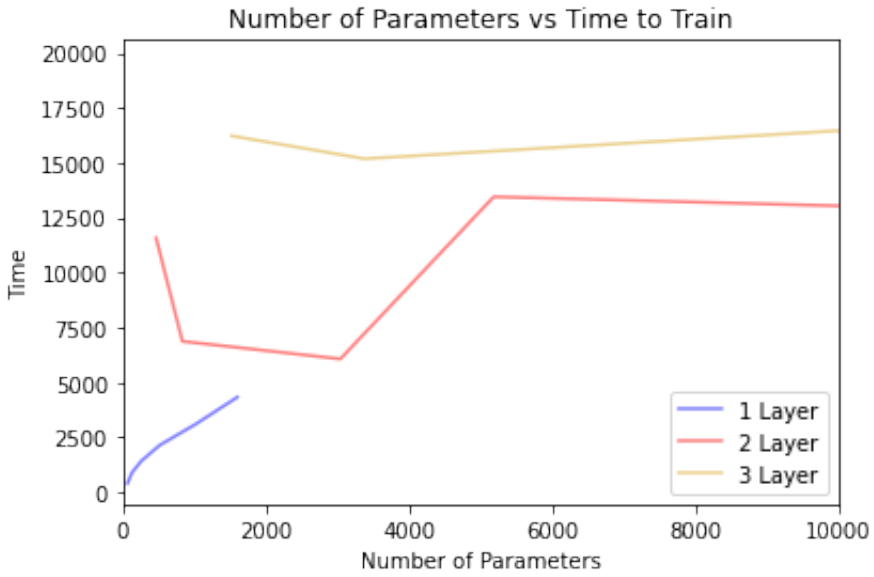
```
hidden_layer_combos3=np.array([[128, 16,16],[32,16,32],[128,128,64]])
num_hidden_3=[]
for x in hidden_layer_combos3:
  num_hidden_3=num_hidden_3+[sum(x)]


#sort by x-axis ascending
sort_num_hidden3, sort_mse_num_hidden3 = (list(t) for t in zip(*sorted(zip(num_hidc
sort_params_mse3, sort_mse3 = (list(t) for t in zip(*sorted(zip(params_3, mse_s_3))
sort_params_time3, sort_times3 = (list(t) for t in zip(*sorted(zip(params_3, times_


import matplotlib.pyplot as plt
fig, ax = plt.subplots()
plt.plot( [16,32,64,128,256,512],mse_s ,  c="blue", label='1 Layer',alpha=0.5)
plt.plot( sort_num_hidden,sort_mse_num_hidden ,  c="red", label='2 Layer',alpha=0.5
plt.plot( sort_num_hidden3,sort_mse_num_hidden3 ,  c="goldenrod", label='3 Layer',a
plt.legend()
plt.title('Hidden units vs RMSE')
plt.xlabel("Hidden Units")
plt.ylabel("RSME")
plt.show()
```

```
fig, ax = plt.subplots()
plt.plot( params,mse_s ,  c="blue", label='1 Layer',alpha=0.5)
plt.plot( sort_params_mse,sort_mse ,  c="red", label='2 Layer',alpha=0.5)
plt.plot( sort_params_mse3,sort_mse3 ,  c="goldenrod", label='3 Layer',alpha=0.5)
plt.legend()
plt.xlim(0,10000)
plt.title('Number of Parameters vs RMSE')
plt.xlabel("Number of Parameters")
plt.ylabel("RSME")
plt.show()
```



2. Comment on the tradeoff between number of parameters and RMSE as you go from deeper (3 hidden layers) to shallow networks (1 hidden layer). Also measure the wall clock time for training each config-uration and plot training time vs number of parameters. Do you see a similar tradeoff in training time? (10)

```
fig, ax = plt.subplots()
plt.plot( params,times ,  c="blue", label='1 Layer',alpha=0.5)
plt.plot( sort_params_time,sort_times ,  c="red", label='2 Layer',alpha=0.5)
plt.plot( sort_params_time3,sort_times3 ,  c="goldenrod", label='3 Layer',alpha=0.5
plt.legend()
plt.xlim(0,10000)
plt.title('Number of Parameters vs Time to Train')
plt.xlabel("Number of Parameters")
plt.ylabel("Time")
plt.show()
```



The deeper the network becomes, the longer it takes to train because there are significantly many more parameters to update. With just 3 layers it took several hours to train 3 models. In contrast, 5 shallow 1 layer networks could be trained in just 2 hours. However, the RMSE is much lower for the deeper networks, showing that while they may take longer to train, they result in more accurate models that generalize better than a shallow network using the same number of hidden units.

Colab paid products - Cancel contracts here

✓ 0s completed at 7:31 PM