```
## Load libraries
import pandas as pd
import numpy as np
import sys
import os
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from keras.datasets import mnist
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
plt.style.use('dark_background')
%matplotlib inline
```

```
np.set_printoptions(precision=2)
```

```
import tensorflow as tf
```

```
tf.__version__
```

```
'2.15.0'
```

Mount Google Drive if running in Colab

```
## Mount Google drive folder if running in Colab
if('google.colab' in sys.modules):
    from google.colab import drive
    drive.mount('/content/drive', force_remount = True)
    DIR = '/content/drive/MyDrive/Colab Notebooks/MAHE/MSIS Coursework/EvenSem2024MAHE'
    DATA_DIR = DIR + '/Data/'
    os.chdir(DIR)
else:
    DATA_DIR = 'Data/'
```

Load diabetes data

```
## Load Bengaluru house price data
file = 'diabetes_regression.csv'
df= pd.read_csv(file, header = 0)

print('Diabetes dataset')
print('-----------')
print('Initial number of samples = %d'%(df.shape[0]))
print('Initial number of features = %d\n'%(df.shape[1]))
df.head(5)
```

```
Diabetes dataset
-----------
Initial number of samples = 442
Initial number of features = 11
```

	AGE	GENDER	BMILEVEL	BP	S1	S2	S3	S4	S5	S6	Y
0	59	2	unhealthy	101.0	157	93.2	38.0	4.0	4.8598	87	151
1	48	1	healthy	87.0	183	103.2	70.0	3.0	3.8918	69	75
2	72	2	unhealthy	93.0	156	93.6	41.0	4.0	4.6728	85	141
3	24	1	overweight	84.0	198	131.4	40.0	5.0	4.8903	89	206
4	50	1	healthy	101.0	192	125.4	52.0	4.0	4.2905	80	135

```
## Create lists of ordinal, categorical, and continuous features
categorical_features = (['BMILEVEL', 'GENDER'])
continuous_features = df.drop(categorical_features, axis = 1).columns.tolist()
print(categorical_features)
print(continuous_features)
```

```
['BMILEVEL', 'GENDER']
['AGE', 'BP', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'Y']
```

Assign 'category' datatype to categorical columns

```
## Assign 'category' datatype to ordinal and categorical columns
print(df.dtypes)
df[categorical_features] = df[categorical_features].astype('category')
print('----')
df.dtypes
```

```
AGE          int64
GENDER       int64
BMILEVEL    object
BP         float64
S1           int64
S2         float64
S3         float64
S4         float64
S5         float64
S6           int64
Y            int64
dtype: object
----
AGE            int64
GENDER      category
BMILEVEL    category
BP           float64
S1             int64
S2           float64
S3           float64
S4           float64
S5           float64
S6             int64
Y              int64
dtype: object
```

Remove the target variable column from the list of continuous features

```
## Remove the target variable column from the list of continuous features
continuous_features.remove('Y')
```

```
## Train and test split of the data
X = df.drop('Y', axis = 1)
y = df['Y']
X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)

num_features = X_train.shape[0]
num_samples = X_train.shape[1]

print('Diabetes data set')
print('---------------------')
print('Number of training samples = %d'%(num_samples))
print('Number of features = %d'%(num_features))
```

```
Diabetes data set
--------------------
Number of training samples = 10
Number of features = 353
```

Build pipeline for categorical and continuous features

```
## Build pipeline for categorical and continuous features

# Pipeline object for categorical (features
categorical_transformer = Pipeline(steps = [('onehotenc', OneHotEncoder(handle_unknown = 'ignore'))])

# Pipeline object for continuous features
continuous_transformer = Pipeline(steps = [('scaler', StandardScaler())])

# Create a preprocessor object for all features
preprocessor = ColumnTransformer(transformers = [('continuous',continuous_transformer, continuous_features),
                                                  ('categorical', categorical_transformer, categorical_features)
                                                 ],
                                 remainder = 'passthrough'
                                 )
```

Fit and transform train data using preprocessor followed by transforming test data

```
## Fit and transform train data using preprocessor
X_train_transformed = preprocessor.fit_transform(X_train).T

# Update number of features
num_features = X_train_transformed.shape[0]
# Transform training data using preprocessor
X_test_transformed = preprocessor.transform(X_test).T
# Convert Y_train and Y_test to numpy arrays
Y_train = Y_train.to_numpy()
Y_test = Y_test.to_numpy()
```

A generic layer class with forward and backward methods

```
class Layer:
  def __init__(self):
    self.input = None
    self.output = None

  def forward(self, input):
    pass

  def backward(self, output_gradient, learning_rate):
    pass
```

Mean squared error (MSE) loss and its gradient

```
## Define the loss function and its gradient
def mse(Y, Yhat):
  return(np.mean((Y - Yhat)**2))
  #TensorFlow in-built function for mean squared error loss
  #mse = tf.keras.losses.MeanSquaredError()
  #mse(Y, Yhat).numpy()

def mse_gradient(Y, Yhat):
  return(Yhat - Y)
```

Generic activation layer class

```
class Activation(Layer):
    def __init__(self, activation, activation_gradient):
        self.activation = activation
        self.activation_gradient = activation_gradient

    def forward(self, input):
        self.input = input
        self.output = self.activation(self.input)
        return(self.output)

    def backward(self, output_gradient, learning_rate = None):
        return(output_gradient[:-1, :] * self.activation_gradient(self.input))
```

Specific activation layer classes

```
class Sigmoid(Activation):
    def __init__(self):
        def sigmoid(z):
            return 1 / (1 + np.exp(-z))

        def sigmoid_gradient(z):
            a = sigmoid(z)
            return a * (1 - a)

        super().__init__(sigmoid, sigmoid_gradient)

class Tanh(Activation):
    def __init__(self):
        def tanh(z):
            return np.tanh(z)

        def tanh_gradient(z):
            a = np.tanh(z)
            return 1 - a**2

        super().__init__(tanh, tanh_gradient)

class ReLU(Activation):
    def __init__(self):
        def relu(z):
            return z * (z > 0)

        def relu_gradient(z):
            return 1. * (z > 0)

        super().__init__(relu, relu_gradient)
```

Dense layer class

```
## Dense layer class
class Dense(Layer):
    def __init__(self, input_size, output_size, reg_strength):
        self.weights = 0.01*np.random.randn(output_size, input_size+1) # bias trick
        self.weights[:, -1] = 0.01 # set all bias values to the same nonzero constant
        self.reg_strength = reg_strength
        self.reg_loss = None

    def forward(self, input):
        self.input = np.vstack([input, np.ones((1, input.shape[1]))]) # bias trick
        self.output= np.dot(self.weights, self.input)
        # Calculate regularization loss
        self.reg_loss = self.reg_strength * np.sum(self.weights[:, :-1] * self.weights[:, :-1])

    def backward(self, output_gradient, learning_rate):
        ## Following is the inefficient way of calculating the backward gradient
        #weights_gradient = np.zeros((self.output.shape[0], self.input.shape[0]), dtype = np.float64)
        #for b in range(output_gradient.shape[1]):
        #  weights_gradient += np.dot(output_gradient[:, b].reshape(-1, 1), self.input[:, b].reshape(-1, 1).T)
        #weights_gradient = (1/output_gradient.shape[1])*weights_gradient

        ## Following is the efficient way of calculating the weights gradient w.r.t. data
        weights_gradient = (1/output_gradient.shape[1])*np.dot(np.atleast_2d(output_gradient), np.atleast_2d(self.input).T)
        # Add the regularization gradient here
        weights_gradient += 2 * self.reg_strength * np.hstack([self.weights[:, :-1], np.zeros((self.weights.shape[0], 1))])


        input_gradient = np.dot(self.weights.T, output_gradient)
        self.weights = self.weights + learning_rate * (-weights_gradient)

        return(input_gradient)
```

Function to generate sample indices for batch processing according to batch size

```
## Function to generate sample indices for batch processing according to batch size
def generate_batch_indices(num_samples, batch_size):
  # Reorder sample indices
  reordered_sample_indices = np.random.choice(num_samples, num_samples, replace = False)
  # Generate batch indices for batch processing
  batch_indices = np.split(reordered_sample_indices, np.arange(batch_size, len(reordered_sample_indices), batch_size))
  return(batch_indices)
```

Train the 1-hidden layer neural network (128 nodes) using batch training with batch size = 16

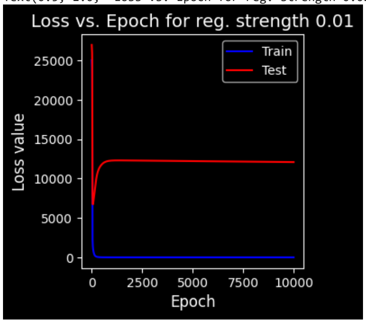
```
## Train the 2-hidden layer neural network (8 nodes, 8 nodes followed by 1 node)
## using batch training with batch size = 100
learning_rate = 1e-03 # learning rate
batch_size = 16 # batch size
nepochs = 10000 # number of epochs
reg_strength = 0.01 # regularization strength
# Create empty array to store training losses over each epoch
loss_train_epoch = np.empty(nepochs, dtype = np.float64)
# Create empty array to store test losses over each epoch
loss_test_epoch = np.empty(nepochs, dtype = np.float64)


# Neural network architecture

dlayer1 = Dense(num_features, 8, reg_strength) # define dense layer 1
alayer1 = ReLU() # ReLU activation layer 1
dlayer2 = Dense(8, 1, reg_strength) # define dense layer 2

# Steps: run over each sample in the batch, calculate loss, gradient of loss,
# and update weights.

epoch = 0
while epoch < nepochs:
  batch_indices = generate_batch_indices(num_samples, batch_size)
  loss = 0
  for b in range(len(batch_indices)):
    # Forward propagation for training data
    dlayer1.forward(X_train_transformed[:, batch_indices[b]]) # forward prop dense layer 1 with batch feature added
    alayer1.forward(dlayer1.output) # forward prop activation layer 1
    dlayer2.forward(alayer1.output) # forward prop dense layer 2
    # Calculate training data loss
    loss += mse(Y_train[batch_indices[b]], dlayer2.output)
    # Add the regularization losses
    loss += dlayer1.reg_loss + dlayer2.reg_loss

    # Backward prop starts here
    grad = mse_gradient(Y_train[batch_indices[b]], dlayer2.output)
    grad = dlayer2.backward(grad, learning_rate)
    grad = alayer1.backward(grad)
    grad = dlayer1.backward(grad, learning_rate)
  # Calculate the average training loss for the current epoch
  loss_train_epoch[epoch] = loss/len(batch_indices)

  # Forward propagation for test data
  dlayer1.forward(X_test_transformed)
  alayer1.forward(dlayer1.output)
  dlayer2.forward(alayer1.output)

  # Calculate test data loss plus regularization loss
  loss_test_epoch[epoch] =  mse(Y_test,dlayer2.output) + dlayer1.reg_loss + dlayer2.reg_loss

  print('Epoch %d: train loss = %f, test loss = %f'%(epoch+1, loss_train_epoch[epoch], loss_test_epoch[epoch]))
  epoch = epoch + 1
```

```
Epoch 1: train loss = 25006.592098, test loss = 26951.593589
Epoch 2: train loss = 24963.896749, test loss = 26908.592798
Epoch 3: train loss = 24921.284867, test loss = 26865.654206
Epoch 4: train loss = 24878.724171, test loss = 26822.745696
Epoch 5: train loss = 24836.168405, test loss = 26779.834900
Epoch 6: train loss = 24793.579059, test loss = 26736.870399
Epoch 7: train loss = 24750.876388, test loss = 26693.774422
Epoch 8: train loss = 24707.950703, test loss = 26650.442419
Epoch 9: train loss = 24664.658532, test loss = 26606.724132
Epoch 10: train loss = 24620.793232, test loss = 26562.380615
Epoch 11: train loss = 24576.027963, test loss = 26517.084479
Epoch 12: train loss = 24529.902410, test loss = 26470.340589
Epoch 13: train loss = 24481.729431, test loss = 26421.419360
Epoch 14: train loss = 24430.497108, test loss = 26369.258154
Epoch 15: train loss = 24374.716351, test loss = 26312.281289
Epoch 16: train loss = 24312.198582, test loss = 26248.190689
Epoch 17: train loss = 24239.726420, test loss = 26173.425013
Epoch 18: train loss = 24152.304258, test loss = 26083.064644
Epoch 19: train loss = 24043.102013, test loss = 25969.872440
Epoch 20: train loss = 23901.989474, test loss = 25822.928006
Epoch 21: train loss = 23713.834341, test loss = 25627.440283
Epoch 22: train loss = 23457.984164, test loss = 25362.346153
Epoch 23: train loss = 23105.224717, test loss = 24999.099774
Epoch 24: train loss = 22615.944063, test loss = 24500.641625
Epoch 25: train loss = 21938.841229, test loss = 23822.249937
Epoch 26: train loss = 21012.294236, test loss = 22916.562439
Epoch 27: train loss = 19771.764664, test loss = 21745.539524
Epoch 28: train loss = 18167.244199, test loss = 20301.272044
Epoch 29: train loss = 16193.494685, test loss = 18630.927340
Epoch 30: train loss = 13926.504890, test loss = 16849.503083
Epoch 31: train loss = 11541.414087, test loss = 15067.732717
Epoch 32: train loss = 9221.071177, test loss = 13461.785935
Epoch 33: train loss = 7246.091919, test loss = 12118.390987
Epoch 34: train loss = 5733.214607, test loss = 11038.931730
Epoch 35: train loss = 4658.407816, test loss = 10175.568558
Epoch 36: train loss = 3918.273402, test loss = 9489.168111
Epoch 37: train loss = 3404.555683, test loss = 8938.125870
Epoch 38: train loss = 3038.579712, test loss = 8499.872252
Epoch 39: train loss = 2768.435823, test loss = 8151.315376
Epoch 40: train loss = 2561.503467, test loss = 7873.126346
Epoch 41: train loss = 2396.978634, test loss = 7649.804626
Epoch 42: train loss = 2261.410798, test loss = 7469.231088
Epoch 43: train loss = 2146.718665, test loss = 7322.586076
Epoch 44: train loss = 2047.722797, test loss = 7202.514647
Epoch 45: train loss = 1959.648557, test loss = 7103.788650
Epoch 46: train loss = 1880.125281, test loss = 7022.467432
Epoch 47: train loss = 1808.161068, test loss = 6955.706196
Epoch 48: train loss = 1741.973534, test loss = 6901.255987
Epoch 49: train loss = 1680.950710, test loss = 6857.222801
Epoch 50: train loss = 1624.624489, test loss = 6822.105059
Epoch 51: train loss = 1572.262963, test loss = 6794.553219
Epoch 52: train loss = 1523.449304, test loss = 6773.743484
Epoch 53: train loss = 1477.920651, test loss = 6758.823505
Epoch 54: train loss = 1435.501295, test loss = 6748.671571
Epoch 55: train loss = 1395.767573, test loss = 6739.170557
Epoch 56: train loss = 1358.377005, test loss = 6732.407001
Epoch 57: train loss = 1323.219141, test loss = 6729.480724
Epoch 58: train loss = 1290.214928, test loss = 6730.046457
```

Plot training loss vs. epoch

```
# Plot train and test loss as a function of epoch:
fig, ax = plt.subplots(1, 1, figsize = (4, 4))
fig.tight_layout(pad = 4.0)
ax.plot(loss_train_epoch, 'b', label = 'Train')
ax.plot(loss_test_epoch, 'r', label = 'Test')
ax.set_xlabel('Epoch', fontsize = 12)
ax.set_ylabel('Loss value', fontsize = 12)
ax.legend()
ax.set_title('Loss vs. Epoch for reg. strength 0.01', fontsize = 14)
```

```
Text(0.5, 1.0, 'Loss vs. Epoch for reg. strength 0.01')
```



Test performance on test data

```
dlayer1.forward(X_test_transformed)
alayer1.forward(dlayer1.output)
dlayer2.forward(alayer1.output)
ypred = dlayer2.output.flatten()
ytrue = Y_test
np.column_stack((ytrue, ypred))
```

```
array([[ 78.  ,  13.25],
       [152.  ,  71.77],
       [200.  , 357.32],
       [ 59.  ,  55.69],
       [311.  , 223.85],
       [178.  , 179.17],
       [332.  , 244.3 ],
       [132.  ,  33.72],
       [156.  ,  53.48],
       [135.  , 127.89],
       [220.  , 354.81],
       [233.  , 321.54],
       [ 91.  ,  14.01],
       [ 51.  ,  19.72],
       [195.  , 325.9 ],
       [109.  , 340.44],
       [217.  , 127.  ],
       [ 94.  ,  59.07],
       [ 89.  , 151.58],
       [111.  , 228.39],
       [129.  , 275.9 ],
       [181.  ,  73.46],
       [168.  , 285.51],
```