In [ ]:
```
## Load libraries
import pandas as pd
import numpy as np
import sys
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from keras.datasets import mnist
plt.style.use('dark_background')
%matplotlib inline
```

```
WARNING:tensorflow:From c:\Users\vp140\.conda\envs\pycaretenv\lib\site-packages\k
eras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is depre
cated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

In [ ]:
```
np.set_printoptions(precision=2)
```

In [ ]:
```
import tensorflow as tf
```

In [ ]:
```
tf.__version__
```

Out[ ]:
```
'2.15.0'
```

Load MNIST Data

In [ ]:
```
## Load MNIST data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.transpose(1, 2, 0)
X_test = X_test.transpose(1, 2, 0)
X_train = X_train.reshape(X_train.shape[0]*X_train.shape[1], X_train.shape[2])
X_test = X_test.reshape(X_test.shape[0]*X_test.shape[1], X_test.shape[2])

num_labels = len(np.unique(y_train))
num_features = X_train.shape[0]
num_samples = X_train.shape[1]

# One-hot encode class labels
Y_train = tf.keras.utils.to_categorical(y_train).T
Y_test = tf.keras.utils.to_categorical(y_test).T


# Normalize the samples (images)
xmax = np.amax(X_train)
xmin = np.amin(X_train)
X_train = (X_train - xmin) / (xmax - xmin) # all train features turn into a numb
X_test = (X_test - xmin)/(xmax - xmin)

print('MNIST set')
print('---------------------')
print('Number of training samples = %d'%(num_samples))
print('Number of features = %d'%(num_features))
print('Number of output labels = %d'%(num_labels))
```

```
MNIST set
---------------------
Number of training samples = 60000
Number of features = 784
Number of output labels = 10
```

A generic layer class with forward and backward methods

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

  def forward(self, input):
    pass

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

CCE loss and its gradient

In [ ]:
```
## Define the loss function and its gradient
def cce(Y, Yhat):
  return(np.mean(np.sum(-Y*np.log(Yhat), axis = 0)))
  #TensorFlow in-built function for categorical crossentropy loss
  #cce = tf.keras.losses.CategoricalCrossentropy()
  #return(cce(Y, Yhat).numpy())

def cce_gradient(Y, Yhat):
  return(-Y/Yhat)
```

Generic activation layer class

In [ ]:
```
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

In [ ]:
```
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):
            return 1 - np.tanh(z) ** 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)
```

Softmax activation layer class

In [ ]:
```
## Softmax activation layer class
class Softmax(Layer):
  def forward(self, input):
    self.output = tf.nn.softmax(input, axis = 0).numpy()

  def backward(self, output_gradient, learning_rate = None):
    ## Following is the inefficient way of calculating the backward gradient
    softmax_gradient = np.empty((self.output.shape[0], output_gradient.shape[1])
    for b in range(softmax_gradient.shape[1]):
      softmax_gradient[:, b] = np.dot((np.identity(self.output.shape[0])-np.atle

    # Return gradient w.r.t. input for backward propagation
    return(softmax_gradient)

    ## Following is the efficient of calculating the backward gradient
    #T = np.transpose(np.identity(self.output.shape[0]) - np.atleast_2d(self.out
    #return(np.einsum('ijk, ik -> jk', T, output_gradient))
```

Dense layer class

In [ ]:
```
## Dense layer class
class Dense(Layer):
    def __init__(self, input_size, output_size):
        self.weights = 0.01*np.random.randn(output_size, input_size+1) # bias tr
        self.weights[:, -1] = 0.01 # set all bias values to the same nonzero con

    def forward(self, input):
        self.input = np.vstack([input, np.ones((1, input.shape[1]))]) # bias tri
        self.output= np.dot(self.weights, self.input)

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

        ## Following is the efficient way of calculating the weightsgradient
        #weights_gradient = (1/output_gradient.shape[1])*np.dot(np.atleast_2d(ou

        # Gradient w.r.t. the input
        input_gradient = np.dot(self.weights.T, output_gradient)

        # Update weights using gradient descent step
        self.weights = self.weights + learning_rate * (-weights_gradient)

        # Return gradient w.r.t. input for backward propagation
        return(input_gradient)
```

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

---

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

Train the 1-hidden layer neural network (128 nodes) using batch training with batch size = 100

In [ ]:
```
## Train the 1-hidden layer neural network (128 nodes)
## using batch training with batch size = 100
learning_rate = 1e-3 # learning rate
batch_size = 100 # batch size
nepochs = 200 # number of epochs
loss_epoch = np.empty(nepochs, dtype = np.float64) # create empty array to store

# Neural network architecture

dlayer1 = Dense(num_features, 128) # define dense layer 1
```

```
alayer1 = ReLU() # ReLU activation layer 1
dlayer2 = Dense(128, num_labels) # define dense layer 2
softmax = Softmax() # define softmax activation layer

# 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)):
    dlayer1.forward(X_train[:, batch_indices[b]]) # forward prop dense layer 1 w
    alayer1.forward(dlayer1.output) # forward prop activation layer 1
    dlayer2.forward(alayer1.output) # forward prop dense layer 2
    softmax.forward(dlayer2.output) # Softmax activate
    loss += cce(Y_train[:, batch_indices[b]], softmax.output) # calculate loss
    # Backward prop starts here
    grad = cce_gradient(Y_train[:, batch_indices[b]], softmax.output)
    grad = softmax.backward(grad)
    grad = dlayer2.backward(grad, learning_rate)
    grad = alayer1.backward(grad)
    grad = dlayer1.backward(grad, learning_rate)
  loss_epoch[epoch] = loss/len(batch_indices)
  print('Epoch %d: loss = %f'%(epoch+1, loss_epoch[epoch]))
  epoch = epoch + 1
```

```
Epoch 1: loss = 2.298796
Epoch 2: loss = 2.290666
Epoch 3: loss = 2.278866
Epoch 4: loss = 2.260469
Epoch 5: loss = 2.231436
Epoch 6: loss = 2.186450
Epoch 7: loss = 2.119613
Epoch 8: loss = 2.026019
Epoch 9: loss = 1.904079
Epoch 10: loss = 1.758383
Epoch 11: loss = 1.600563
Epoch 12: loss = 1.445367
Epoch 13: loss = 1.304079
Epoch 14: loss = 1.181739
Epoch 15: loss = 1.078536
Epoch 16: loss = 0.992365
Epoch 17: loss = 0.920443
Epoch 18: loss = 0.860121
Epoch 19: loss = 0.809190
Epoch 20: loss = 0.765806
Epoch 21: loss = 0.728518
Epoch 22: loss = 0.696218
Epoch 23: loss = 0.667946
Epoch 24: loss = 0.643071
Epoch 25: loss = 0.620950
Epoch 26: loss = 0.601229
Epoch 27: loss = 0.583466
Epoch 28: loss = 0.567419
Epoch 29: loss = 0.552848
Epoch 30: loss = 0.539558
Epoch 31: loss = 0.527372
Epoch 32: loss = 0.516182
Epoch 33: loss = 0.505854
Epoch 34: loss = 0.496290
Epoch 35: loss = 0.487452
Epoch 36: loss = 0.479217
Epoch 37: loss = 0.471535
Epoch 38: loss = 0.464388
Epoch 39: loss = 0.457676
Epoch 40: loss = 0.451380
Epoch 41: loss = 0.445471
Epoch 42: loss = 0.439917
Epoch 43: loss = 0.434687
Epoch 44: loss = 0.429714
Epoch 45: loss = 0.425027
Epoch 46: loss = 0.420613
Epoch 47: loss = 0.416385
Epoch 48: loss = 0.412408
Epoch 49: loss = 0.408594
Epoch 50: loss = 0.404990
Epoch 51: loss = 0.401515
Epoch 52: loss = 0.398215
Epoch 53: loss = 0.395049
Epoch 54: loss = 0.392032
Epoch 55: loss = 0.389137
Epoch 56: loss = 0.386319
Epoch 57: loss = 0.383674
Epoch 58: loss = 0.381091
Epoch 59: loss = 0.378608
Epoch 60: loss = 0.376200
```

```
Epoch 61: loss = 0.373904
Epoch 62: loss = 0.371687
Epoch 63: loss = 0.369507
Epoch 64: loss = 0.367431
Epoch 65: loss = 0.365408
Epoch 66: loss = 0.363457
Epoch 67: loss = 0.361545
Epoch 68: loss = 0.359706
Epoch 69: loss = 0.357909
Epoch 70: loss = 0.356151
Epoch 71: loss = 0.354478
Epoch 72: loss = 0.352796
Epoch 73: loss = 0.351214
Epoch 74: loss = 0.349646
Epoch 75: loss = 0.348095
Epoch 76: loss = 0.346602
Epoch 77: loss = 0.345152
Epoch 78: loss = 0.343730
Epoch 79: loss = 0.342323
Epoch 80: loss = 0.340955
Epoch 81: loss = 0.339630
Epoch 82: loss = 0.338322
Epoch 83: loss = 0.337028
Epoch 84: loss = 0.335787
Epoch 85: loss = 0.334531
Epoch 86: loss = 0.333314
Epoch 87: loss = 0.332135
Epoch 88: loss = 0.330967
Epoch 89: loss = 0.329803
Epoch 90: loss = 0.328660
Epoch 91: loss = 0.327571
Epoch 92: loss = 0.326490
Epoch 93: loss = 0.325395
Epoch 94: loss = 0.324328
Epoch 95: loss = 0.323273
Epoch 96: loss = 0.322270
Epoch 97: loss = 0.321230
Epoch 98: loss = 0.320227
Epoch 99: loss = 0.319253
Epoch 100: loss = 0.318266
Epoch 101: loss = 0.317312
Epoch 102: loss = 0.316366
Epoch 103: loss = 0.315414
Epoch 104: loss = 0.314482
Epoch 105: loss = 0.313578
Epoch 106: loss = 0.312667
Epoch 107: loss = 0.311766
Epoch 108: loss = 0.310879
Epoch 109: loss = 0.310000
Epoch 110: loss = 0.309133
Epoch 111: loss = 0.308274
Epoch 112: loss = 0.307422
Epoch 113: loss = 0.306571
Epoch 114: loss = 0.305750
Epoch 115: loss = 0.304939
Epoch 116: loss = 0.304109
Epoch 117: loss = 0.303307
Epoch 118: loss = 0.302498
Epoch 119: loss = 0.301716
Epoch 120: loss = 0.300912
```

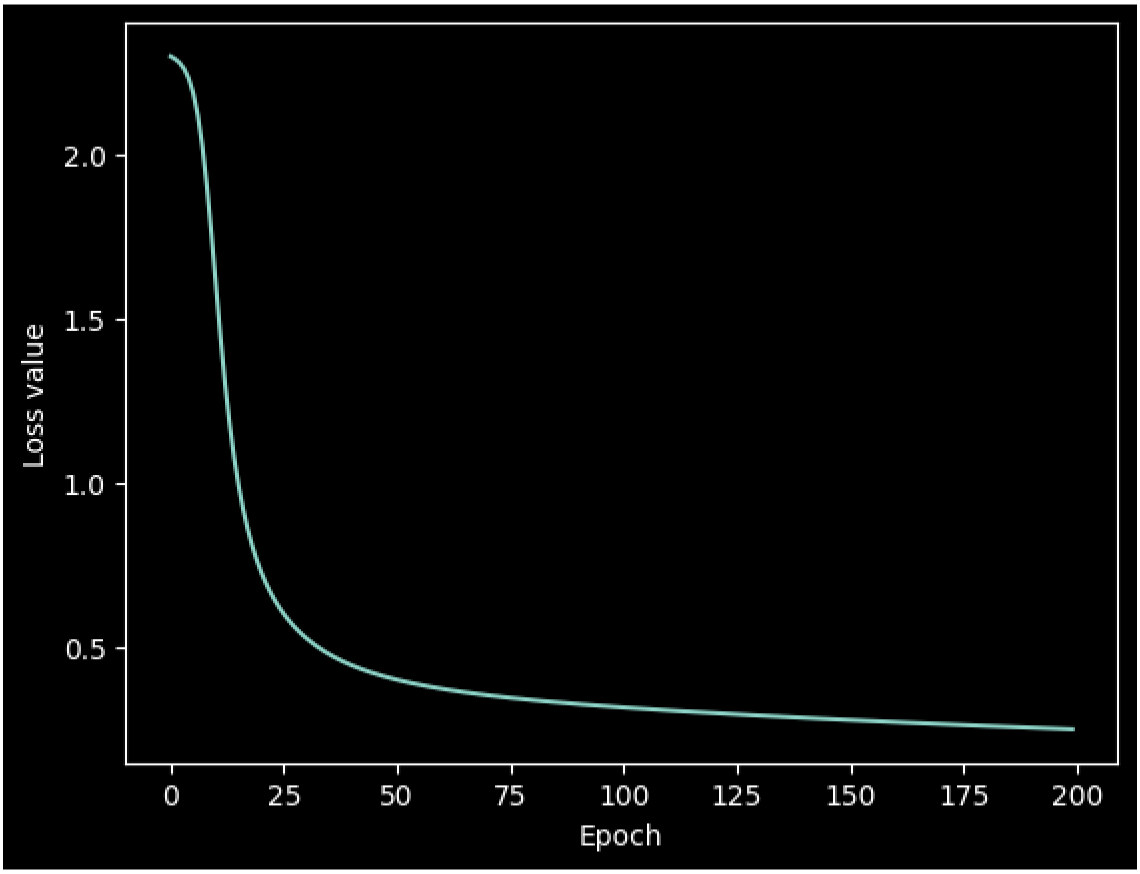
```
Epoch 121: loss = 0.300138
Epoch 122: loss = 0.299354
Epoch 123: loss = 0.298585
Epoch 124: loss = 0.297826
Epoch 125: loss = 0.297065
Epoch 126: loss = 0.296320
Epoch 127: loss = 0.295556
Epoch 128: loss = 0.294837
Epoch 129: loss = 0.294093
Epoch 130: loss = 0.293351
Epoch 131: loss = 0.292623
Epoch 132: loss = 0.291910
Epoch 133: loss = 0.291186
Epoch 134: loss = 0.290481
Epoch 135: loss = 0.289774
Epoch 136: loss = 0.289066
Epoch 137: loss = 0.288379
Epoch 138: loss = 0.287672
Epoch 139: loss = 0.286978
Epoch 140: loss = 0.286311
Epoch 141: loss = 0.285612
Epoch 142: loss = 0.284897
Epoch 143: loss = 0.284253
Epoch 144: loss = 0.283598
Epoch 145: loss = 0.282926
Epoch 146: loss = 0.282264
Epoch 147: loss = 0.281599
Epoch 148: loss = 0.280949
Epoch 149: loss = 0.280286
Epoch 150: loss = 0.279642
Epoch 151: loss = 0.278997
Epoch 152: loss = 0.278335
Epoch 153: loss = 0.277712
Epoch 154: loss = 0.277075
Epoch 155: loss = 0.276441
Epoch 156: loss = 0.275797
Epoch 157: loss = 0.275193
Epoch 158: loss = 0.274570
Epoch 159: loss = 0.273931
Epoch 160: loss = 0.273330
Epoch 161: loss = 0.272713
Epoch 162: loss = 0.272083
Epoch 163: loss = 0.271514
Epoch 164: loss = 0.270899
Epoch 165: loss = 0.270297
Epoch 166: loss = 0.269698
Epoch 167: loss = 0.269099
Epoch 168: loss = 0.268503
Epoch 169: loss = 0.267937
Epoch 170: loss = 0.267346
Epoch 171: loss = 0.266752
Epoch 172: loss = 0.266177
Epoch 173: loss = 0.265606
Epoch 174: loss = 0.265012
Epoch 175: loss = 0.264461
Epoch 176: loss = 0.263890
Epoch 177: loss = 0.263322
Epoch 178: loss = 0.262761
Epoch 179: loss = 0.262185
Epoch 180: loss = 0.261618
```

```
Epoch 181: loss = 0.261066
Epoch 182: loss = 0.260530
Epoch 183: loss = 0.259976
Epoch 184: loss = 0.259423
Epoch 185: loss = 0.258878
Epoch 186: loss = 0.258333
Epoch 187: loss = 0.257789
Epoch 188: loss = 0.257250
Epoch 189: loss = 0.256730
Epoch 190: loss = 0.256184
Epoch 191: loss = 0.255640
Epoch 192: loss = 0.255129
Epoch 193: loss = 0.254594
Epoch 194: loss = 0.254066
Epoch 195: loss = 0.253542
Epoch 196: loss = 0.253010
Epoch 197: loss = 0.252497
Epoch 198: loss = 0.251966
Epoch 199: loss = 0.251462
Epoch 200: loss = 0.250942
```

Plot training loss vs. epoch

In [ ]:

```
# Plot training loss as a function of epoch:
plt.plot(loss_epoch)
plt.xlabel('Epoch')
plt.ylabel('Loss value')
plt.show()
```



Test performance on test data

In [ ]:
```
dlayer1.forward(X_test)
alayer1.forward(dlayer1.output)
dlayer2.forward(alayer1.output)
softmax.forward(dlayer2.output)
ypred = np.argmax(softmax.output.T, axis = 1)
print(ypred)
ytrue = np.argmax(Y_test.T, axis = 1)
print(ytrue)
np.mean(ytrue == ypred)
```

```
[7 2 1 ... 4 5 6]
[7 2 1 ... 4 5 6]
```

Out[ ]:
```
0.9305
```