```
## 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
```

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

```
import tensorflow as tf
```

```
tf.__version__
```

Load MNIST Data

```
## 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 number between 0 and 1
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))
```

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
```

The softmax classifier steps for a batch of comprising b samples represented as the $725 \times b$-matrix (724 pixel values plus the bias feature absorbed as its last row)

$$\mathbf{X} = \left[\,\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(b-1)}\,\right]$$

with one-hot encoded true labels represented as the $10 \times b$-matrix (10 possible categories)

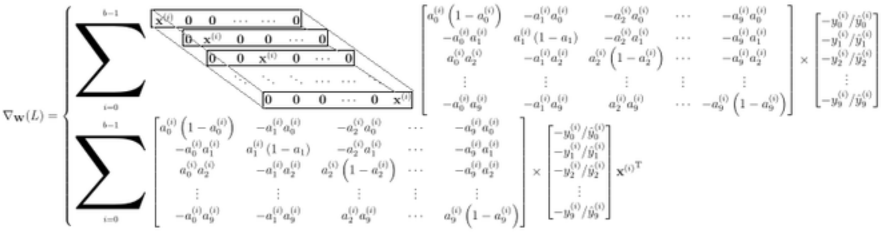
$$\mathbf{Y} = \left[\,\mathbf{y}^{(0)} \quad \ldots \quad \mathbf{y}^{(b-1)}\,\right]$$

using a randomly initialized 10×725-weights matrix $\mathbf{W}$:

1. Calculate $10 \times b$-raw scores matrix :

$$\begin{aligned}\left[\,\mathbf{z}^{(0)} \quad \ldots \quad \mathbf{z}^{(b-1)} \ldots\right] &= \mathbf{W}\left[\,\mathbf{z}^{(0)} \quad \ldots \quad \mathbf{z}^{(b-1)} \ldots\right] \\ &= \left[\,\mathbf{W}\mathbf{z}^{(0)} \quad \ldots \quad \mathbf{W}\mathbf{z}^{(b-1)}\,\right] \\ \Rightarrow \mathbf{Z} &= \mathbf{W}\mathbf{X}.\end{aligned}$$

2. Calculate $10 \times b$-softmax predicted probabilities matrix:
$$\begin{aligned}\left[\mathbf{a}^{(0)} \quad \dots \quad \mathbf{a}^{(b-1)}\right] &= \left[\text{softmax}\left(\mathbf{z}^{(0)}\right) \quad \dots \quad \text{softmax}\left(\mathbf{z}^{(b-1)}\right)\right]\\ \Rightarrow \mathbf{A} &= \text{softmax}(\mathbf{Z}).\end{aligned}$$
3. Predicted probability matrix get a new name: $\hat{\mathbf{Y}} = \mathbf{A}$.
4. The crossentropy (CCE) loss for the ith sample is
$$L_i = \sum_{k=0}^{9} -y^{(i)} \log\left(\hat{y}_k^{(i)}\right) = -\mathbf{y}^{(i)^{\mathrm{T}}} \log\left(\mathbf{y}^{(i)}\right)$$
   which leads to the average crossentropy (CCE) batch loss for the batch as:
$$\begin{aligned}L &= \frac{1}{b}\left[L_0 + \cdots + L_{b-1}\right]\\ &= \frac{1}{b}\left[-\mathbf{y}^{(0)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(0)}\right) + \cdots + -\mathbf{y}^{(b-1)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(b-1)}\right)\right].\end{aligned}.$$
5. The computational graph for the samples in the batch are presented below:
$$\begin{array}{ccc} L_0 & & L_{b-1}\\ \downarrow & & \downarrow\\ \hat{\mathbf{y}}^{(0)} = \mathbf{a}^{(0)} & & \hat{\mathbf{y}}^{(b-1)} = \mathbf{a}^{(b-1)}\\ \downarrow & \dots & \downarrow\\ \mathbf{z}^{(0)} & & \mathbf{z}^{(b-1)}\\ \downarrow & & \downarrow\\ \mathbf{W} & & \mathbf{W}\end{array}$$
6. Calculate the gradient of the average batch loss w.r.t. weights as:
$$\begin{aligned}\Rightarrow \nabla_{\mathbf{W}}(L) &= \frac{1}{b}\left(\underbrace{\left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(0)}\right) \times \nabla_{\mathbf{z}^{(0)}}\left(\hat{\mathbf{y}}^{(0)}\right) \times \nabla_{\hat{\mathbf{y}}^{(0)}}(L_0)\right]}_{\text{sample } 0} + \cdots + \underbrace{\left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(b-1)}\right) \times \nabla_{\mathbf{z}^{(b-1)}}\left(\hat{\mathbf{y}}^{(b-1)}\right) \times \nabla_{\hat{\mathbf{y}}^{(b-1)}}(L_{b-1})\right]}_{\text{sample } b-1}\right)\\ &= \frac{1}{b}\left(\underbrace{\left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(0)}\right) \times \nabla_{\mathbf{z}^{(0)}}\left(\mathbf{a}^{(0)}\right) \times \nabla_{\hat{\mathbf{y}}^{(0)}}(L_0)\right]}_{\text{sample } 0} + \cdots + \underbrace{\left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(b-1)}\right) \times \nabla_{\mathbf{z}^{(b-1)}}\left(\hat{\mathbf{y}}^{(b-1)}\right) \times \nabla_{\hat{\mathbf{y}}^{(b-1)}}(L_{b-1})\right]}_{\text{sample } b-1}\right).\end{aligned}$$
7. The full gradient can be written as $\nabla_{\mathbf{W}}(L) =$



CCE loss and its gradient for the batch samples:
$$\begin{aligned}L &= \frac{1}{b}\left[L_0 + \cdots + L_{b-1}\right]\\ &= \frac{1}{b}\left[-\mathbf{y}^{(0)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(0)}\right) + \cdots + -\mathbf{y}^{(b-1)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(b-1)}\right)\right].\end{aligned}.$$
$$\left[\nabla_{\hat{\mathbf{y}}^{(0)}}(L_0) \quad \dots \quad \nabla_{\hat{\mathbf{y}}^{(b-1)}}(L_{b-1})\right] = \begin{bmatrix} -y_0^{(0)}/\hat{y}_0^{(0)} & \dots & -y_0^{(0)}/\hat{y}_0^{(b-1)}\\ -y_1^{(0)}/\hat{y}_1^{(0)} & \dots & -y_1^{(b-1)}/\hat{y}_1^{(b-1)}\\ -y_2^{(0)}/\hat{y}_2^{(0)} & \dots & -y_2^{(b-1)}/\hat{y}_2^{(b-1)}\\ \vdots & & \\ -y_9^{(0)}/\hat{y}_9^{(0)} & \dots & -y_9^{(b-1)}/\hat{y}_9^{(b-1)}\end{bmatrix}$$

```
## Define the loss function and its gradient
def cce(Y, Yhat):
  return(np.mean(np.?(?*?, axis = ?)))

def cce_gradient(Y, Yhat):
  return(?/?)

# TensorFlow in-built function for categorical crossentropy loss
#cce = tf.keras.losses.CategoricalCrossentropy()
```

Softmax activation layer class:

Forward:

$$\begin{aligned}\left[\,\mathbf{a}^{(0)} \quad \ldots \quad \mathbf{a}^{(b-1)}\,\right] &= \left[\,\operatorname{softmax}\left(\mathbf{z}^{(0)}\right) \quad \ldots \quad \operatorname{softmax}\left(\mathbf{z}^{(b-1)}\right)\,\right]\\ \Rightarrow \mathbf{A} &= \operatorname{softmax}(\mathbf{Z}).\end{aligned}$$

Backward:

$$\left[\,\nabla_{\mathbf{z}^{(0)}}(L_0) \quad \ldots \quad \nabla_{\mathbf{z}^{(b-1)}}(L_{b-1})\,\right] = \left[\,\nabla_{\mathbf{z}^{(0)}}\left(\mathbf{a}^{(0)}\right)\times\nabla_{\mathbf{a}^{(0)}}(L_0) \quad \cdots \quad \nabla_{\mathbf{z}^{(b-1)}}\left(\mathbf{a}^{(b-1)}\right)\times\nabla_{\mathbf{a}^{(b-1)}}(L_{b-1})\,\right]$$

$$= \left[\begin{bmatrix} a_0^{(0)}\left(1-a_0^{(0)}\right) & -a_1^{(0)}a_0^{(0)} & -a_2^{(0)}a_0^{(0)} & \cdots & -a_9^{(0)}a_0^{(0)} \\ -a_0^{(0)}a_1^{(0)} & a_1^{(0)}\left(1-a_1\right) & -a_2^{(0)}a_1^{(0)} & \cdots & -a_9^{(0)}a_1^{(0)} \\ a_0^{(0)}a_2^{(0)} & -a_1^{(0)}a_2^{(0)} & a_2^{(0)}\left(1-a_2^{(0)}\right) & \cdots & -a_9^{(0)}a_2^{(0)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -a_0^{(0)}a_9^{(0)} & -a_1^{(0)}a_9^{(0)} & a_2^{(0)}a_9^{(0)} & \cdots & -a_9^{(0)}\left(1-a_9^{(0)}\right) \end{bmatrix} \times \begin{bmatrix} -y_0^{(0)}/\hat{y}_0^{(0)} \\ -y_1^{(0)}/\hat{y}_1^{(0)} \\ -y_2^{(0)}/\hat{y}_2^{(0)} \\ \vdots \\ -y_9^{(0)}/\hat{y}_9^{(0)} \end{bmatrix} \quad \cdots\cdots \quad \begin{bmatrix} a_0^{(b-1)}\left(1-a_0^{(b-1)}\right) & -a_1^{(b-1)}a_0^{(b-1)} & -a_2^{(b-1)}a_0^{(b-1)} & \cdots & -a_9^{(b-1)}a_0^{(b-1)} \\ -a_0^{(b-1)}a_1^{(b-1)} & a_1^{(b-1)}\left(1-a_1\right) & -a_2^{(b-1)}a_1^{(b-1)} & \cdots & -a_9^{(b-1)}a_1^{(b-1)} \\ a_0^{(b-1)}a_2^{(b-1)} & -a_1^{(b-1)}a_2^{(b-1)} & a_2^{(b-1)}\left(1-a_2^{(b-1)}\right) & \cdots & -a_9^{(b-1)}a_2^{(b-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -a_0^{(b-1)}a_9^{(b-1)} & -a_1^{(b-1)}a_9^{(b-1)} & a_2^{(b-1)}a_9^{(b-1)} & \cdots & -a_9^{(b-1)}\left(1-a_9^{(b-1)}\right) \end{bmatrix} \times \begin{bmatrix} -y_0^{(b-1)}/\hat{y}_0^{(b-1)} \\ -y_1^{(b-1)}/\hat{y}_1^{(b-1)} \\ -y_2^{(b-1)}/\hat{y}_2^{(b-1)} \\ \vdots \\ -y_9^{(b-1)}/\hat{y}_9^{(b-1)} \end{bmatrix}\right]$$

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

  def backward(self, output_gradient, learning_rate = None):
    ## Following is the inefficient way of calculating the backward gradient
    softmax_gradient = np.empty((self.input.shape[0], output_gradient.shape[1]), dtype = np.float64)
    for b in range(softmax_gradient.shape[1]):
      softmax_gradient[:, ?] = np.dot((np.identity(self.output.shape[0])-self.?[:, ?].T) * self.output[?, ?], ?[:, b])
    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.output).T[:, np.newaxis, :], (1, 2, 0)) * np.atleast_2d(se
    #return(np.einsum('ijk, ik -> jk', T, output_gradient))
```

Dense layer class:

Forward:

$$\begin{aligned}\left[\,\mathbf{z}^{(0)} \quad \ldots \quad \mathbf{z}^{(b-1)}\ldots\right] &= \mathbf{W}\left[\,\mathbf{z}^{(0)} \quad \ldots \quad \mathbf{z}^{(b-1)}\ldots\right]\\ &= \left[\,\mathbf{W}\mathbf{z}^{(0)} \quad \ldots \quad \mathbf{W}\mathbf{z}^{(b-1)}\,\right]\\ \Rightarrow \mathbf{Z} &= \mathbf{W}\mathbf{X}.\end{aligned}$$

Backward:

$$\begin{aligned}\nabla_{\mathbf{W}}(L) &= \frac{1}{b}\left[\nabla_{\mathbf{W}}(\mathbf{z}^{(0)})\times\nabla_{\mathbf{z}^{(0)}}(L)+\cdots+\nabla_{\mathbf{W}}(\mathbf{z}^{(b-1)})\times\nabla_{\mathbf{z^{(b-1)}}}(L)\right]\\ &= \frac{1}{b}\left[\nabla_{\mathbf{z^{(0)}}}(L)\mathbf{x}^{(0)^{\mathrm{T}}}+\cdots+\nabla_{\mathbf{z^{(b-1)}}}(L)\mathbf{x}^{(b-1)^{\mathrm{T}}}\right].\end{aligned}$$

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

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

    def backward(self, output_gradient, learning_rate):
        ## Following is the inefficient way of calculating the backward gradient
        dense_gradient = np.zeros((self.output.shape[?], self.input.shape[?]), dtype = np.float64)
        for b in range(output_gradient.shape[1]):
          dense_gradient += np.dot(output_gradient[?, b].reshape(-1, 1), self.input[:, ?].reshape(-1, 1).T)
        dense_gradient = (1/output_gradient.shape[1])*dense_gradient
        ## Following is the efficient way of calculating the backward gradient
        #dense_gradient = (1/output_gradient.shape[1])*np.dot(np.atleast_2d(output_gradient), np.atleast_2d(self.input).T)
        self.weights = self.weights + learning_rate * (-dense_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)
```

Example generation of batch indices

```
## Example generation of batch indices
batch_size = 100
batch_indices = generate_batch_indices(num_samples, batch_size)
print(batch_indices)
```

Train the 0-layer neural network using batch training with batch size = 16

```
## Train the 0-layer neural network using batch training with batch size = 16
learning_rate = ? # learning rate
batch_size = ? # batch size
nepochs = ? # number of epochs
loss_epoch = np.empty(nepochs, dtype = np.float32) # create empty array to store losses over each epoch

# Neural network architecture
dlayer = Dense(?, ?) # define dense layer
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)):
    dlayer.forward(?) # forward prop
    softmax.forward(?) # Softmax activate
    loss += cce(?, ?) # calculate loss
    # Backward prop starts here
    grad = cce_gradient(?, ?)
    grad = softmax.backward(?)
    grad = dlayer.backward(?, ?)
  loss_epoch[epoch] = loss/len(batch_indices)
  print('Epoch %d: loss = %f'%(epoch+1, loss_epoch[epoch]))
  epoch = epoch + 1
```

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

```
## Accuracy on test set
dlayer.forward(X_test)
softmax.forward(dlayer.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)
```