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
```

The softmax classifier steps for a batch of comprising b samples represented as the $785 \times b$-matrix (784 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 10×785-weights matrix $\mathbf{W}$:

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

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

2. Calculate $10 \times b$-softmax predicted probabilities matrix:

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

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:

$$
\begin{aligned}
L &= \frac{1}{b}[L_0 + \cdots + L_{b-1}] \\
& \frac{1}{b}\left[\sum_{k=0}^{9} -y^{(0)} \log\left(\hat{y}_k^{(0)}\right) + \cdots + \sum_{k=0}^{9} -y^{(b-1)} \log\left(\hat{y}_k^{(b-1)}\right)\right] \\
&= \frac{1}{b}\left[-\mathbf{y}^{(0)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(0)}\right) + \cdots + -\mathbf{y}^{(b-1)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(b-1)}\right)\right].
\end{aligned}
$$

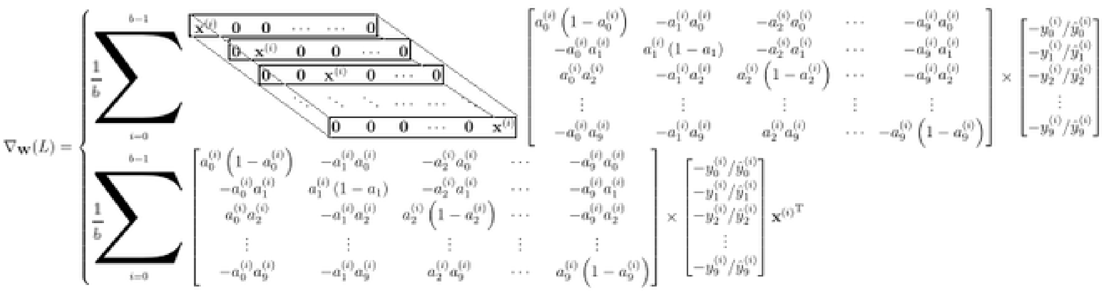
5. The computational graph for the samples in the batch are presented below:

$$
\begin{array}{ccc}
L_0 & & L_{b-1} \\
\downarrow & & \downarrow \\
\hat{\mathbf{y}}^{(0)} = \mathbf{a}^{(0)} & & \hat{\mathbf{y}}^{(b-1)} = \mathbf{a}^{(b-1)} \\
\downarrow & \cdots & \downarrow \\
\mathbf{z}^{(0)} & & \mathbf{z}^{(b-1)} \\
\downarrow & & \downarrow \\
\mathbf{W} & & \mathbf{W}
\end{array}
$$

6. Calculate the gradient of the average batch loss w.r.t. weights as:

$$
\begin{aligned}
\Rightarrow \nabla_{\mathbf{W}}(L) &= \frac{1}{b}[\nabla_{\mathbf{W}}(L_0) + \cdots + \nabla_{\mathbf{W}}(L_{b-1})] \\
&= \frac{1}{b}\left(\underbrace{\left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(0)}\right) \times \nabla_{\mathbf{z}^{(0)}}\left(\hat{\mathbf{y}}^{(0)}\right) \times \nabla_{\hat{\mathbf{y}}^{(0)}}(L_0)\right]}_{\text{sample } 0} + \cdots + \left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(b-1)}\right) \times \nabla\right.\right. \\
&= \frac{1}{b}\left(\underbrace{\left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(0)}\right) \times \nabla_{\mathbf{z}^{(0)}}\left(\mathbf{a}^{(0)}\right) \times \nabla_{\hat{\mathbf{y}}^{(0)}}(L_0)\right]}_{\text{sample } 0} + \cdots + \left[\nabla_{\mathbf{W}}\left(\mathbf{z}^{(b-1)}\right) \times \nabla\right.\right.
\end{aligned}
$$

10. The full gradient can be written as



CCE loss and its gradient for the batch samples:

$$
\begin{aligned}
L &= \frac{1}{b}[L_0 + \cdots + L_{b-1}] \\
&= \frac{1}{b}\left[\sum_{k=0}^{9} -y^{(0)} \log\left(\hat{y}_k^{(0)}\right) + \cdots + \sum_{k=0}^{9} -y^{(b-1)} \log\left(\hat{y}_k^{(b-1)}\right)\right] \\
&= \frac{1}{b}\left[-\mathbf{y}^{(0)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(0)}\right) + \cdots + -\mathbf{y}^{(b-1)^{\mathrm{T}}} \log\left(\hat{\mathbf{y}}^{(b-1)}\right)\right].
\end{aligned}
$$

$$\left[\nabla_{\hat{\mathbf{y}}^{(0)}}(L_0) \quad \ldots \quad \nabla_{\hat{\mathbf{y}}^{(b-1)}}(L_{b-1})\right] = \begin{bmatrix} -y_0^{(0)}/\hat{y}_0^{(0)} & \ldots & -y_0^{(0)}/\hat{y}_0^{(b-1)} \\ -y_1^{(0)}/\hat{y}_1^{(0)} & \ldots & -y_1^{(b-1)}/\hat{y}_1^{(b-1)} \\ -y_2^{(0)}/\hat{y}_2^{(0)} & \ldots & -y_2^{(b-1)}/\hat{y}_2^{(b-1)} \\ \vdots & & \\ -y_9^{(0)}/\hat{y}_9^{(0)} & \ldots & -y_9^{(b-1)}/\hat{y}_9^{(b-1)} \end{bmatrix}$$

In [ ]:
```
Y = np.array([[1, 0, 0],[0, 0, 1], [0, 1, 0]])
print(Y)
Yhat = np.array([[0.8, 0.1, 0.1],[0.05, .5, .4], [.15, .4, 0.5]])
print(Yhat)
np.mean(np.sum(-Y*np.log(Yhat), axis = 0))
-Y/Yhat
```

```
[[1 0 0]
 [0 0 1]
 [0 1 0]]
[[0.8  0.1  0.1 ]
 [0.05 0.5  0.4 ]
 [0.15 0.4  0.5 ]]
```

Out[ ]:
```
array([[-1.25,  0.  ,  0.  ],
       [ 0.  ,  0.  , -2.5 ],
       [ 0.  , -2.5 ,  0.  ]])
```

In [ ]:
```
## Define the loss function and its gradient
def cce(Y, Yhat):
  return(np.mean(np.sum(-Y*np.log(Yhat), axis = 0)))

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

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

Softmax activation layer class:

Forward:

$$\left[\mathbf{a}^{(0)} \quad \ldots \quad \mathbf{a}^{(b-1)}\right] = \left[\text{softmax}\left(\mathbf{z}^{(0)}\right) \quad \ldots \quad \text{softmax}\left(\mathbf{z}^{(b-1)}\right)\right]$$
$$\Rightarrow \mathbf{A} = \text{softmax}(\mathbf{Z}).$$

Backward:

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

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

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

$$\begin{aligned}\nabla_{\mathbf{W}}(L) &= \frac{1}{b}\left[\nabla_{\mathbf{W}}(\mathbf{z}^{(0)}) \times \nabla_{\mathbf{z}^{(0)}}(L) + \cdots + \nabla_{\mathbf{W}}(\mathbf{z}^{(b-1)}) \times \nabla_{\mathbf{z}^{(b-1)}}(L)\right] \\ &= \frac{1}{b}\left[\nabla_{\mathbf{z}^{(0)}}(L_0)\mathbf{x}^{(0)^{\mathrm{T}}} + \cdots + \nabla_{\mathbf{z}^{(b-1)}}(L_{b-1})\mathbf{x}^{(b-1)^{\mathrm{T}}}\right].\end{aligned}$$

In [ ]:
```
## Dense layer class
class Dense(Layer):
    def __init__(self, input_size, output_size):
        self.weights = np.empty((output_size, input_size+1))  # bias trick
        self.weights[:, :-1] = 0.01*np.random.randn(output_size, input_size)
        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 backward gradient
        dense_gradient = np.zeros((self.output.shape[0], self.input.shape[0]), d
        for b in range(output_gradient.shape[1]):
          dense_gradient += np.dot(output_gradient[:, b].reshape(-1, 1), self.ir
        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(outp
        self.weights = self.weights + learning_rate * (-dense_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)
```

Example generation of batch indices

In [ ]:
```
## Example generation of batch indices
num_samples = 64
batch_size = 8
batch_indices = generate_batch_indices(num_samples, batch_size)
print(batch_indices)
```

```
[array([37, 52, 58, 25, 30, 50, 10, 13]), array([ 7, 38,  8, 48, 40, 26, 24, 1
5]), array([14, 62, 55, 34, 23, 21, 11, 44]), array([28, 60, 54,  4, 41,  6, 33,
49]), array([ 3,  2, 45, 35, 36, 27, 16, 51]), array([61, 59, 31, 53, 47, 18, 63,
46]), array([32, 19, 22, 43, 12,  5, 42, 56]), array([ 1, 39,  9, 29,  0, 57, 20,
17])]
```

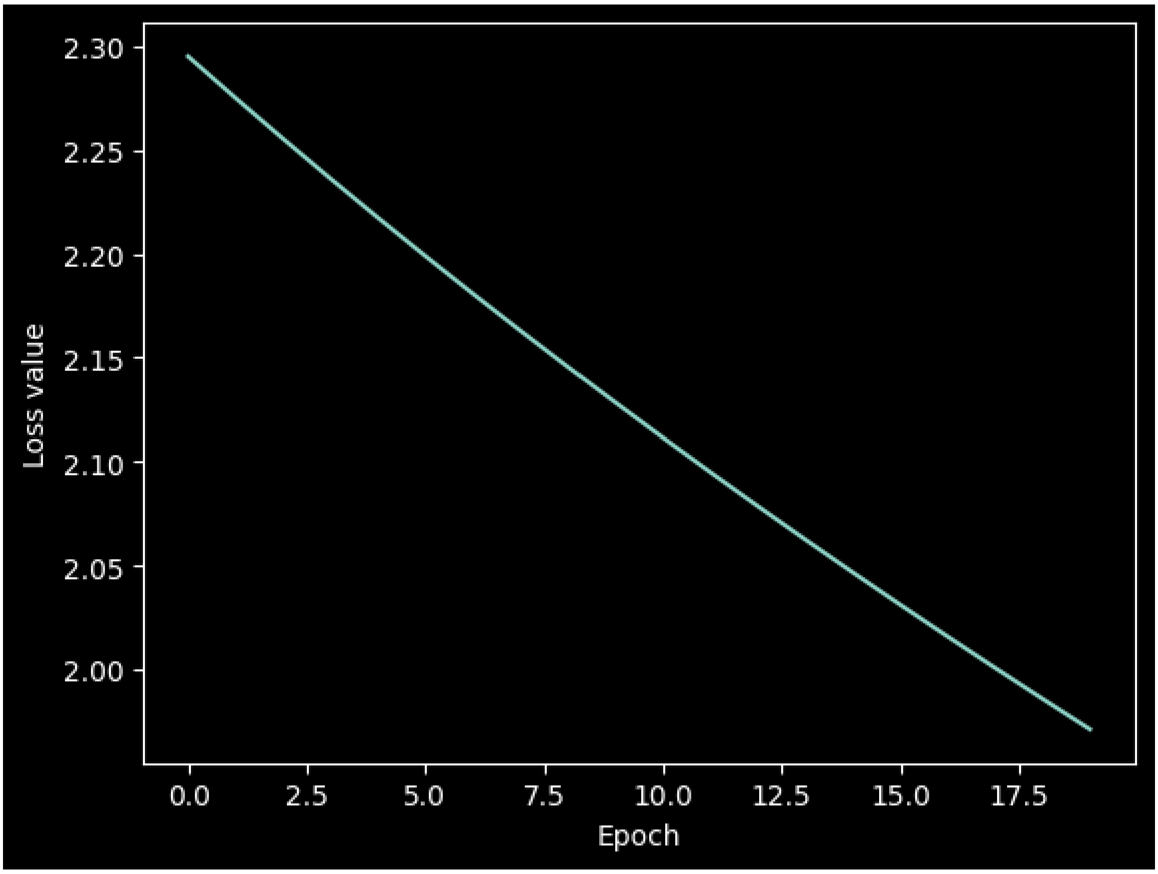
Train the 0-layer neural network using batch training with batch size = 16

In [ ]:
```
## Train the 0-layer neural network using batch training with batch size = 16
learning_rate = 1e-2 # learning rate
batch_size = 200 # batch size
nepochs = 20 # number of epochs
loss_epoch = np.empty(nepochs, dtype = np.float32) # create empty array to store

# Neural network architecture
dlayer = Dense(num_features, num_labels) # 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(X_train[:, batch_indices[b]]) # forward prop
    softmax.forward(dlayer.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 = dlayer.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 = 2.295188
Epoch 2: loss = 2.275155
Epoch 3: loss = 2.255539
Epoch 4: loss = 2.236321
Epoch 5: loss = 2.217483
Epoch 6: loss = 2.199008
Epoch 7: loss = 2.180882
Epoch 8: loss = 2.163089
Epoch 9: loss = 2.145615
Epoch 10: loss = 2.128447
Epoch 11: loss = 2.111573
Epoch 12: loss = 2.094982
Epoch 13: loss = 2.078661
Epoch 14: loss = 2.062600
Epoch 15: loss = 2.046791
Epoch 16: loss = 2.031222
Epoch 17: loss = 2.015887
Epoch 18: loss = 2.000776
Epoch 19: loss = 1.985882
Epoch 20: loss = 1.971197
```

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



In [ ]:
```
## 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)
```

```
[9 0 1 ... 9 0 0]
[7 2 1 ... 4 5 6]
```

Out[ ]: 0.4146