```
## 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
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
```

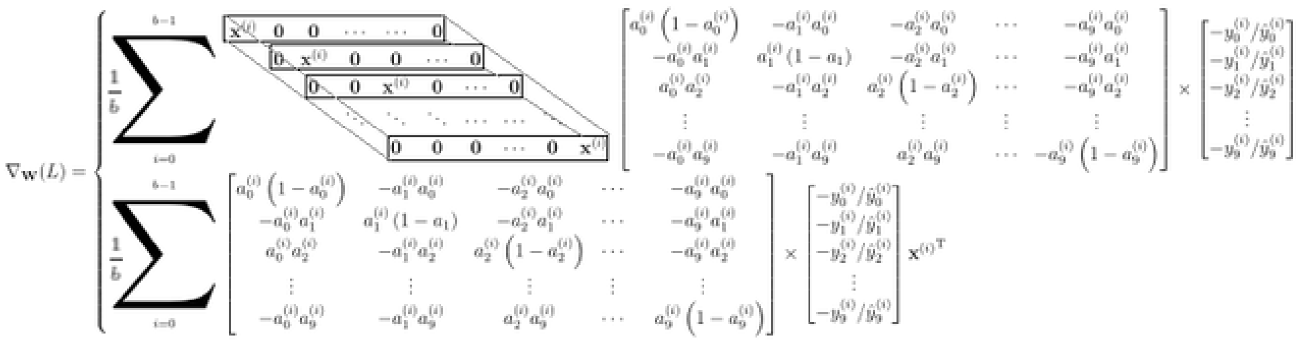
The softmax classifier steps for a batch of comprising b samples represented as the $785\times b$-matrix (784 pixel values plus the bias feature absorbed as its last row) $$\mathbf{X} = \begin{bmatrix}\mathbf{x}^{(0)},\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(b-1)}\end{bmatrix}$$ with one-hot encoded true labels represented as the $10\times b$-matrix (10 possible categories) $$\mathbf{Y}=\begin{bmatrix}\mathbf{y}^{(0)}&\ldots&\mathbf{y}^{(b-1)}\end{bmatrix}$$ using a randomly initialized 10×785-weights matrix $\mathbf{W}$:

1. Calculate $10\times b$-raw scores matrix : $$\begin{align*}\begin{bmatrix}\mathbf{z}^{(0)}&\ldots&\mathbf{z}^{(b-1)}\ldots\end{bmatrix} &= \mathbf{W}\begin{bmatrix}\mathbf{x}^{(0)}&\ldots&\mathbf{x}^{(b-1)}\ldots\end{bmatrix}\\&=\begin{bmatrix}\mathbf{W}\mathbf{x}^{(0)}&\ldots&\mathbf{W}\mathbf{x}^{(b-1)}\end{bmatrix}\\\Rightarrow \mathbf{Z} &= \mathbf{WX}.\end{align*}$$
2. Calculate $10\times b$-softmax predicted probabilities matrix: $$\begin{align*}\begin{bmatrix}\mathbf{a}^{(0)}&\ldots&\mathbf{a}^{(b-1)}\end{bmatrix} &= \begin{bmatrix}\text{softmax}\left(\mathbf{z}^{(0)}\right)&\ldots&\text{softmax}\left(\mathbf{z}^{(b-1)}\right)\end{bmatrix}\\\Rightarrow\mathbf{A} &= \text{softmax}(\mathbf{Z}).\end{align*}$$
3. Predicted probability matrix get a new name: $\hat{\mathbf{Y}} = \mathbf{A}.$
4. The crossentropy (CCE) loss for the ith sample is $$L_i = \sum_{k=0}^9-y^{(i)}\log\left(\hat{y}^{(i)}_k\right) = -{\mathbf{y}^{(i)}}^\mathrm{T}\log\left(\mathbf{y}^{(i)}\right)$$ which leads to the average crossentropy (CCE) batch loss for the batch as: $$\begin{align*}L &=\frac{1}{b}\left[L_0+\cdots+L_{b-1}\right]\\&\frac{1}{b}\left[\sum_{k=0}^9-y^{(0)}\log\left(\hat{y}^{(0)}_k\right)+\cdots+\sum_{k=0}^9-y^{(b-1)}\log\left(\hat{y}^{(b-1)}_k\right)\right]\\&=\frac{1}{b}\left[-{\mathbf{y}^{(0)}}^{\mathrm{T}}\log\left(\hat{\mathbf{y}}^{(0)}\right)+\cdots+-{\mathbf{y}^{(b-1)}}^{\mathrm{T}}\log\left(\hat{\mathbf{y}}^{(b-1)}\right)\right].\end{align*}$$
5. The computational graph for the samples in the batch are presented below:

$\hspace{1.5in}\begin{align*}L_0\\{\color{yellow}\downarrow}\\ \hat{\mathbf{y}}^{(0)} &= \mathbf{a}^{(0)}\\{\color{yellow}\downarrow}\\\mathbf{z}^{(0)}\\

{\color{yellow}\downarrow}\\\mathbf{W}\end{align*}$$\qquad\cdots\qquad$$\begin{align*} L_{b-1}\\{\color{yellow}\downarrow}\\ \hat{\mathbf{y}}^{(b-1)} &= \mathbf{a}^{(b-1)}\\ {\color{yellow}\downarrow}\\\mathbf{z}^{(b-1)}\\ {\color{yellow}\downarrow}\\\mathbf{W}\end{align*}$

6. Calculate the gradient of the average batch loss w.r.t. weights as: $$\begin{align*}\Rightarrow \nabla_\mathbf{W}(L) &= \frac{1}{b}\left[\nabla_\mathbf{W}\left(L_0\right)+\cdots+\nabla_\mathbf{W}\left(L_{b-1}\right)\right]\\&= \frac{1}{b}\left(\underbrace{\left[\nabla_\mathbf{W}\left(\mathbf{z}^{(0)}\right) \times\nabla_{\mathbf{z}^{(0)}}\left(\hat{\mathbf{y}}^{(0)}\right)\times\nabla_{\hat{\mathbf{y}}^{(0)}}(L_0)\right]}_{\text{sample}\,0}+\cdots+\underbrace{\left[\nabla_\mathbf{W}\left(\mathbf{z}^{(b-1)}\right) \times\nabla_{\mathbf{z}^{(b-1)}}\left(\hat{\mathbf{y}}^{(b-1)}\right)\times\nabla_{\hat{\mathbf{y}}^{(b-1)}}(L_{b-1})\right]}_{\text{sample}\,b-1}\right)\\&=\frac{1}{b}\left(\underbrace{\left[\nabla_\mathbf{W}\left(\mathbf{z}^{(0)}\right) \times\nabla_{\mathbf{z}^{(0)}}\left({\mathbf{a}}^{(0)}\right)\times\nabla_{\hat{\mathbf{y}}^{(0)}}(L_0)\right]}_{\text{sample}\,0}+\cdots+\underbrace{\left[\nabla_\mathbf{W}\left(\mathbf{z}^{(b-1)}\right) \times\nabla_{\mathbf{z}^{(b-1)}}\left({\mathbf{a}}^{(b-1)}\right)\times\nabla_{\hat{\mathbf{y}}^{(b-1)}}(L_{b-1})\right]}_{\text{sample}\,b-1}\right).\end{align*}$$
7. The full gradient can be written as



CCE loss and its gradient for the batch samples:

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

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

```
## 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: $$\begin{align*}\begin{bmatrix}\mathbf{a}^{(0)}&\ldots&\mathbf{a}^{(b-1)}\end{bmatrix} &= \begin{bmatrix}\text{softmax}\left(\mathbf{z}^{(0)}\right)&\ldots&\text{softmax}\left(\mathbf{z}^{(b-1)}\right)\end{bmatrix}\\\Rightarrow\mathbf{A} &= \text{softmax}(\mathbf{Z}).\end{align*}$$$$\begin{align*}\begin{bmatrix}\mathbf{a}^{(0)}&\ldots&\mathbf{a}^{(b-1)}\end{bmatrix} &= \begin{bmatrix}\text{softmax}\left(\mathbf{z}^{(0)}\right)&\ldots&\text{softmax}\left(\mathbf{z}^{(b-1)}\right)\end{bmatrix}\\\Rightarrow\mathbf{A} &= \text{softmax}(\mathbf{Z}).\end{align*}$$

Backward: $$\begin{align*}\begin{bmatrix}\nabla_{\mathbf{z}^{(0)}}(L_0)&\ldots&\nabla_{\mathbf{z}^{(b-1)}}(L_{b-1})\end{bmatrix} &= \begin{bmatrix}\nabla_{\mathbf{z}^{(0)}}\left({\mathbf{a}}^{(0)}\right)\times\nabla_{\mathbf{a}^{(0)}}(L_0)&\cdots&\nabla_{\mathbf{z}^{(b-1)}}\left({\mathbf{a}}^{(b-1)}\right)\times\nabla_{\mathbf{a}^{(b-1)}}(L_{b-1})\end{bmatrix}\end{align*}$$$$\begin{align*}\begin{bmatrix}\nabla_{\mathbf{z}^{(0)}}(L_0)&\ldots&\nabla_{\mathbf{z}^{(b-1)}}(L_{b-1})\end{bmatrix} &= \begin{bmatrix}\nabla_{\mathbf{z}^{(0)}}\left({\mathbf{a}}^{(0)}\right)\times\nabla_{\mathbf{a}^{(

0)}}(L_0)&\cdots&\nabla_{\mathbf{z}^{(b-1)}}\left({\mathbf{a}}^{(b-1)}\right)\times\nabla_{\mathbf{a}^{(b-1)}}(L_{b-1})\end{bmatrix}\end{align*}$$

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

```
## 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
    for b in range(softmax_gradient.shape[1]):
      softmax_gradient[:, ?] = np.dot((np.identity(self.output.shape[0])-self.?[:, ?].resha
    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[:,
    #return(np.einsum('ijk, ik -> jk', T, output_gradient))
```

Dense layer class:

Forward: $$$$$$$$\begin{align*}\begin{bmatrix}\mathbf{z}^{(0)}&\ldots&\mathbf{z}^{(b-1)}\ldots\end{bmatrix} &= \mathbf{W}\begin{bmatrix}\mathbf{z}^{(0)}&\ldots&\mathbf{z}^{(b-1)}\ldots\end{bmatrix}\\&=\begin{bmatrix}\mathbf{W}\mathbf{z}^{(0)}&\ldots&\mathbf{W}\mathbf{z}^{(b-1)}\end{bmatrix}\\\Rightarrow \mathbf{Z} &= \mathbf{WX}.\end{align*}\begin{align*}\begin{bmatrix}\mathbf{z}^{(0)}&\ldots&\mathbf{z}^{(b-1)}\ldots\end{bmatrix} &= \mathbf{W}\begin{bmatrix}\mathbf{z}^{(0)}&\ldots&\mathbf{z}^{(b-1)}\ldots\end{bmatrix}\\&=\begin{bmatrix}\mathbf{W}\mathbf{z}^{(0)}&\ldots&\mathbf{W}\mathbf{z}^{(b-1)}\end{bmatrix}\\\Rightarrow \mathbf{Z} &= \mathbf{WX}.\end{align*}$$$$$$$$

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

```
## 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.f
        for b in range(output_gradient.shape[1]):
          dense_gradient += np.dot(output_gradient[?, b].reshape(-1, 1), self.input[:, ?].r
        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
        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_sa
  return(batch_indices)
```

Example generation of batch indices

```
## Example generation of batch indices
num_samples = 64
batch_size = 16
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 ove

# 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
```

```
## 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)
```