+ Code + Text

```
## 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
```

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

```
## 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))
```

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

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

  def forward(self, input):
    pass

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

```
## 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)
```

```
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 * self.activation_gradient(self.input))
```

```
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
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]), dtype = np.float64)
    for b in range(softmax_gradient.shape[1]):
      softmax_gradient[:, b] = np.dot((np.identity(self.output.shape[0])-np.atleast_2d(self.output[:, b])) * np.atleast_2d(self.output[
    return(softmax_gradient)
    ## Following is the efficient way of calculating the backward gradient
    #T = np.transpose(np.identity(self.output.shape[0]) - np.atleast_2d(self.output).T[:, np.newaxis, :], (2, 1, 0)) * np.atleast_2d(sel
    #return(np.einsum('jik, ik -> jk', T, output_gradient))
```

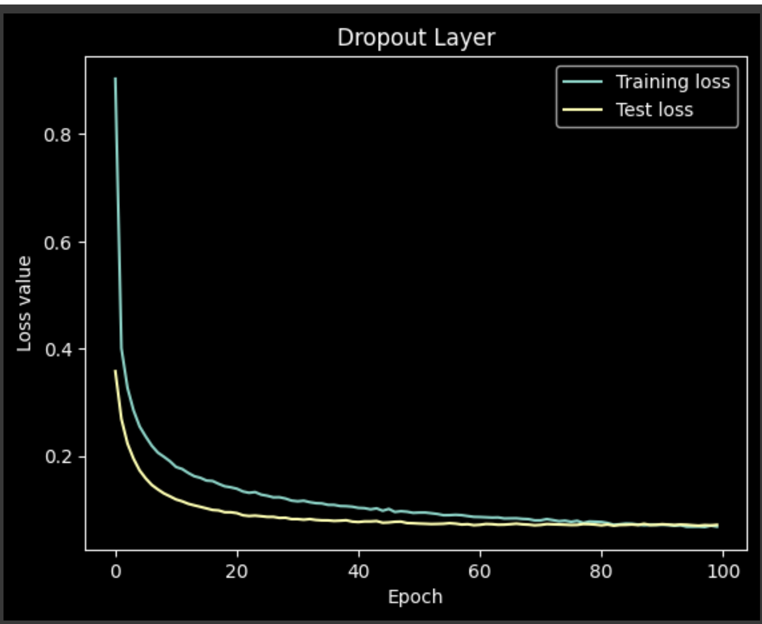
```
# Dropout layer class
class Dropout(Layer):

    def __init__(self, probability_dropout = 0.0):
        self.probability_dropout = probability_dropout
        self.dropout_matrix = None

    def forward(self, input):
        self.dropout_matrix = (np.random.rand(input.shape[0], input.shape[1]) )
        self.dropout_matrix = (self.dropout_matrix < (1 - self.probability_dropout))
        self.output = (input * self.dropout_matrix)/(1 - self.probability_dropout)
        return(self.output)

    def backward(self, output_gradient):
        return(self.dropout_matrix * output_gradient[:-1, :])
```

```
## 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 the regularization loss
        #self.reg_loss = (self.reg_strength)*np.sum((self.weights**2) - (self.weights[:, -1]**2))
        #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 backward gradient
        weights_gradient = (1/output_gradient.shape[1])*np.dot(np.atleast_2d(output_gradient), np.atleast_2d(self.input).T)
        # add regularization gradient here weights should not contain bias weights
        weights_gradient += 2*self.reg_strength*np.hstack([self.weights[:, :-1], np.zeros((self.weights.shape[0], 1))])
        #weights_gradient += (1/output_gradient.shape[1])*np.dot(np.atleast_2d(output_gradient), np.atleast_2d(self.input).T)

        input_gradient = np.dot(self.weights.T, output_gradient)

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

        return(input_gradient)
```

```
## 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)
```

```
learning_rate = 1e-1 # learning rate
batch_size = 100 # batch size
nepochs = 100 # number of epochs
reg_strength = 0
loss_train_epoch = np.empty(nepochs, dtype = np.float64) # create empty array to store training losses over each epoch
loss_test_epoch = np.empty(nepochs, dtype = np.float64) # create empty array to store testing losses over each epoch

dlayer1 = Dense(num_features, 128, reg_strength=0) # define dense layer 1
alayer1 = ReLU() # ReLU activation layer 1
dropout1 = Dropout(probability_dropout=0.5)
dlayer2 = Dense(128, num_labels, reg_strength=0) # 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)):
        #Forward Propagation for training data
        dlayer1.forward(X_train[:, batch_indices[b]]) # forward prop dense layer 1 with batch feature added
        alayer1.forward(dlayer1.output) # forward prop activation layer 1
        dropout1.forward(alayer1.output) # dropout after layer 1
        dlayer2.forward(dropout1.output) # forward prop dense layer 2
        softmax.forward(dlayer2.output) # Softmax activate
        loss += cce(Y_train[:, batch_indices[b]], softmax.output) # calculate training data loss

        # Backward propagation for training data starts here
        grad = cce_gradient(Y_train[:, batch_indices[b]], softmax.output)
        grad = softmax.backward(grad)
        grad = dlayer2.backward(grad, learning_rate)
        grad = dropout1.backward(grad)
        grad = alayer1.backward(grad)
        grad = dlayer1.backward(grad, learning_rate)
    # Calculate average training loss for the current epoch
    loss_train_epoch[epoch] = loss/len(batch_indices)

    # Forward Propagation for test data
    dlayer1.forward(X_test) # 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
    softmax.forward(dlayer2.output) # Softmax activate
    # Calculate test data loss and Add regularization loss
    loss_test_epoch[epoch] = cce(Y_test, softmax.output)

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

```
# Plot training loss as a function of epoch:
plt.title("Dropout Layer")
plt.plot(loss_train_epoch, label = 'Training loss')
plt.plot(loss_test_epoch, label = 'Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss value')
plt.legend()
plt.show()
```



(1) no regularization

```
class Activation1(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))
```

```
class Sigmoid(Activation1):
    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(Activation1):
    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(Activation1):
    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
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)
```

```
learning_rate = 1e-1 # learning rate
batch_size = 100 # batch size
nepochs = 100 # number of epochs
reg_strength = 0
loss_train_epoch_nr = np.empty(nepochs, dtype = np.float64) # create empty array to store training losses over each epoch
loss_test_epoch_nr = np.empty(nepochs, dtype = np.float64) # create empty array to store testing losses over each epoch

dlayer1 = Dense(num_features, 128, reg_strength=0) # define dense layer 1
alayer1 = ReLU() # ReLU activation layer 1

#dropout1 = Dropout(probability_dropout=0.5)
dlayer2 = Dense(128, num_labels, reg_strength=0) # 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)):
        #Forward Propagation for training data
        dlayer1.forward(X_train[:, batch_indices[b]]) # forward prop dense layer 1 with batch feature added
        alayer1.forward(dlayer1.output) # forward prop activation layer 1
        #dropout1.forward(alayer1.output) # dropout after 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 training data loss

        # Backward propagation for training data starts here
        grad = cce_gradient(Y_train[:, batch_indices[b]], softmax.output)
        grad = softmax.backward(grad)
        grad = dlayer2.backward(grad, learning_rate)
        #grad = dropout1.backward(grad)
        grad = alayer1.backward(grad)
        grad = dlayer1.backward(grad, learning_rate)
    # Calculate average training loss for the current epoch
    loss_train_epoch_nr[epoch] = loss/len(batch_indices) + dlayer1.reg_loss

    # Forward Propagation for test data
    dlayer1.forward(X_test) # 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
    softmax.forward(dlayer2.output) # Softmax activate
    # Calculate test data loss and Add regularization loss
    loss_test_epoch_nr[epoch] = cce(Y_test, softmax.output) + dlayer2.reg_loss

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

```
# Plot training loss as a function of epoch:
plt.title("No regularisation")
plt.plot(loss_train_epoch_nr, label = 'Training loss')
plt.plot(loss_test_epoch_nr, label = 'Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss value')
plt.legend()
plt.show()
```

loss-based regularization with reg_strength = 0.1

```
learning_rate = 1e-1 # learning rate
batch_size = 100 # batch size
nepochs = 100 # number of epochs
reg_strength = 0.1
loss_train_epoch_r = np.empty(nepochs, dtype = np.float64) # create empty array to store training losses over each epoch
loss_test_epoch_r = np.empty(nepochs, dtype = np.float64) # create empty array to store testing losses over each epoch

dlayer1 = Dense(num_features, 128, reg_strength=0) # define dense layer 1
alayer1 = ReLU() # ReLU activation layer 1

#dropout1 = Dropout(probability_dropout=0.5)
dlayer2 = Dense(128, num_labels, reg_strength=0) # 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)):
        #Forward Propagation for training data
        dlayer1.forward(X_train[:, batch_indices[b]]) # forward prop dense layer 1 with batch feature added
        alayer1.forward(dlayer1.output) # forward prop activation layer 1
        #dropout1.forward(alayer1.output) # dropout after 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 training data loss

        # Backward propagation for training data starts here
        grad = cce_gradient(Y_train[:, batch_indices[b]], softmax.output)
        grad = softmax.backward(grad)
        grad = dlayer2.backward(grad, learning_rate)
        #grad = dropout1.backward(grad)
        grad = alayer1.backward(grad)
        grad = dlayer1.backward(grad, learning_rate)
    # Calculate average training loss for the current epoch
    loss_train_epoch_r[epoch] = loss/len(batch_indices) + dlayer1.reg_loss

    # Forward Propagation for test data
    dlayer1.forward(X_test) # 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
    softmax.forward(dlayer2.output) # Softmax activate
    # Calculate test data loss and Add regularization loss
    loss_test_epoch_r[epoch] = cce(Y_test, softmax.output) + dlayer2.reg_loss

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

```
# Plot training loss as a function of epoch:
plt.title("With regularisation")
plt.plot(loss_train_epoch_r, label = 'Training loss')
plt.plot(loss_test_epoch_r, label = 'Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss value')
plt.legend()
plt.show()
```