Ain Shams University
Faculty of Engineering

Deep Learning Framework

Sec 4	16W0061	محمد عباده بهاء محمد صرصر
Sec 4	17X0040	محمد حماده محمد محمود
Sec 4	16T0107	مصطفى اشرف محمد عبد العظيم
Sec 3	1600953	غاده رجب عبد النبى اسماعيل
Sec 2	1600361	ايه احمد اسماعيل محمد
Sec 3	1600832	علي سيد علي محمد ابوالعلا
Sec 2	1500241	إسراء جمال زينهم علي الكبير

Supervisor: **Professor Hossam El-Din Hassan Abd El Munim**
Department of Computer Systems Engineering
Faculty of Engineering at Ain Shams University

January 25, 2021

1. Project Overview

- A **deep learning framework** is an interface, library or a tool which allows us to build **deep learning** models more easily and quickly, without getting into the details of underlying algorithms. They provide a clear and concise way for defining models using a collection of pre-built and optimized components.

2. Problem Statement

- Binary classification is the simplest kind of machine learning problem. The goal of binary classification is to categorise data points into one of two buckets: 0 or 1, true or false, to survive or not to survive, blue or no blue eyes, etc.

- Classification problems having multiple classes with imbalanced dataset present a different challenge than a binary classification problem. The skewed distribution makes many conventional machine learning algorithms less effective, especially in predicting minority class examples. In order to do so, let us first understand the problem at hand and then discuss the ways to overcome those.

3. Metrics

- According to the precision and recall concept when we used in binary class classification, will apply this concept into multiclass classification, a typical multi-class classification problem, we need to categorize each sample into 1 of N different classes,Similar to a binary case, we can define precision and recall for each of the classes. So, Accuracy = correct / total size Or = true positive + true negative / dataset size

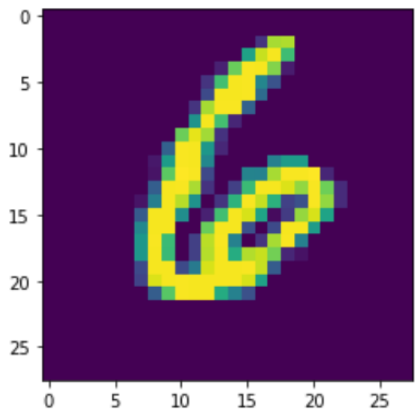
4. Implementation

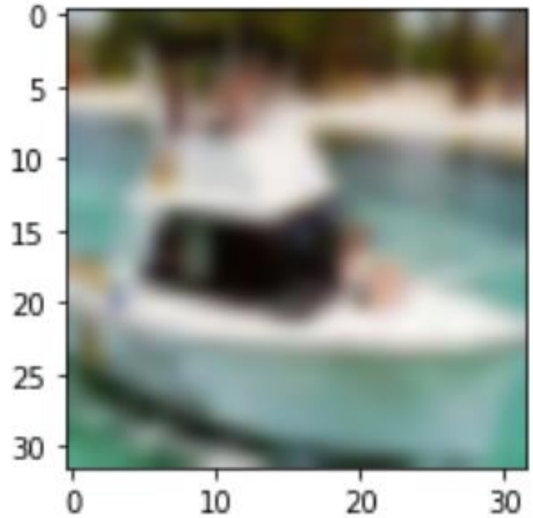
- Our frame was built following the PyTorch school. It’s also implemented on the bases of the software principles such as:

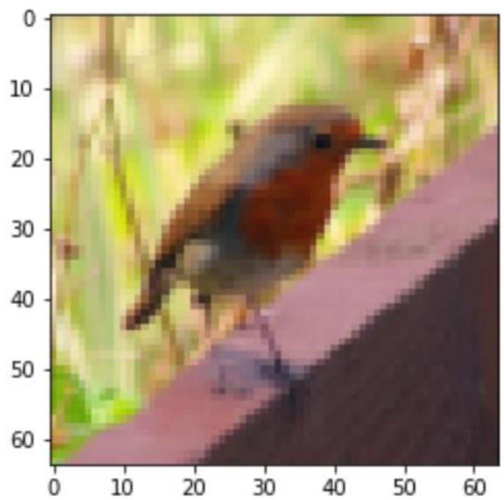
a. Modularity
   - The framework is implemented into modules, classes and functions according to functionality and responsibility
b. Abstraction
   - We have seperated the behavior of the components from their implementation so, when the user use any of the classes or the built in function he doesn’t have to worry with the implementation complexity
c. Incremental Development
   - The framework was developed in small increment at a time

4.1. Data Exploration and Visualization

- **Data Download**
  - Our DataFrame supports downloading and using (MNIST and CIFAR-10) dataset for training and testing till now. The images are divided into a training set and a validation set
- **Data Preprocessing**
  - After loading the data we have the option to perform some operation on the data. We Pass images to the data loader with batch size as desired, normalize it, convert them into tensors and shuffle it.
- **Data Visualisation**







4.2. Layers

- The dense layer is a neural network layer that is connected deeply, which means each neuron in the dense layer receives input from all neurons of its previous layer. The dense layer is found to be the most commonly used layer in the models.
- The neurons, within each layer of a neural network, perform the same function. They simply calculate the weighted sum of inputs and weights, add the bias and execute an activation function.

4.3 Activation module:

4.3.1. Sigmoid / Logistic

Advantages

- Smooth gradient, preventing “jumps” in output values.
- Output values bound between 0 and 1, normalizing the output of each neuron.
- Clear predictions—For X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions.

Disadvantages

- Vanishing gradient—for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
- Outputs not zero centered.
- Computationally expensive

4.3.2. ReLU (Rectified Linear Unit)

Advantages

- Computationally efficient—allows the network to converge very quickly
- Non-linear—although it looks like a linear function, ReLU has a derivative function and allows for backpropagation

Disadvantages

- The Dying ReLU problem—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

4.3.3. TanH / Hyperbolic Tangent

Advantages

- Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
- Otherwise like the Sigmoid function.

Disadvantages

- Like the Sigmoid function

4.3.4 Softmax

Advantages

- Able to handle multiple classes only one class in other activation functions—normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.
- Useful for output neurons—typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.

```python
import numpy as np
from .Layers import Layer_Dense

class Sigmoid:

    def forwards(self, inputs):
        X = inputs.out
        sig =   1 / (1 + (np.exp(-1 * X)))
        inputs.pass_act('sigmoid',sig)
        return sig

    def backwards(self, inputs):
        s = self.forwards(inputs)
        return s * (1 - s)

class ReLU:

    def forwards(self, inputs):
        rel = np.maximum(0, inputs.out)
        inputs.pass_act('relu',rel)
        return rel

    def Backwards(self, inputs):
         if inputs > 0:
             return 1
         elif inputs <= 0:
             return 0

class Identity:
    def forwards(self, inputs):
        ident =inputs.out
        inputs.pass_act('identity',ident)
        return ident

    def backwards(self, inputs):
        pass

class Tanh:
    def forwards(self, inputs):
        tan = np.tanh(inputs.out)
        inputs.pass_act('tanh',tan)
        return tan


    def backwards(self, inputs):
        a = self.forwards(inputs)
        return 1 - a ** 2
class Softmax:
    def forward(self, inputs):
        exp_x = np.exp(inputs.out)
        probs = exp_x / np.sum(exp_x, axis=1, keepdims=True)
        inputs.pass_act('softmax',probs)
        return probs
```

Fig.1 Activation Functions

4.4. Forward module:

- A node, also called a neuron or Perceptron, is a computational unit that has one or more weighted input connections, a transfer function that combines the inputs in some way, and an output connection.

  Nodes are then organized into layers to comprise a network.

  A single-layer artificial neural network, also called a single-layer, has a single layer of nodes, as its name suggests. Each node in the single layer connects directly to an input variable and contributes to an output variable.

  A single-layer network can be extended to a multiple-layer network, referred to as a Multilayer Perceptron. A Multilayer Perceptron, or MLP for short, is an artificial neural network with more than a single layer.

  It has an input layer that connects to the input variables, one or more hidden layers, and an output layer that produces the output variables.

  We can summarize the types of layers in an MLP as follows:

  Input Layer: Input variables, sometimes called the visible layer.

  Hidden Layers: Layers of nodes between the input and output layers. There may be one or more of these layers.

  Output Layer: A layer of nodes that produce the output variables.

```python
import numpy as np
from itertools import count

class Layer_Dense:

    _ids = count(1)
    _params={}
    layer_activations={}
    layers_num_arr = []



    def __init__(self, n_inputs, n_neurons, weight_type="random"):
        '''
        # Initialize weights and biases randomly
        @params :   n_inputs ---> number of features
                    n_neurons --> number of neurons
        '''

        # tracking number of instances
        self.layer_number = next(self._ids)
        self.layers_num_arr.append(self.layer_number)

        # Initializing the weights either random or zeros
        if weight_type == "random":
            self.weights = 0.1 * np.random.randn( n_neurons,n_inputs)
        elif weight_type == "zeros":
```

Fig.2 Forward propagation

4.5. Losses module:

we seek to minimize the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply "*loss*."

The cost or loss function has an important job in that it must faithfully distill all aspects of the model down into a single number in such a way that improvements in that number are a sign of a better model.

In calculating the error of the model during the optimization process, a loss function must be chosen.

This can be a challenging problem as the function must capture the properties of the problem and be motivated by concerns that are important to the project and stakeholders.

4.5.1. Bipolar Perceptron:

```
import numpy as np
from ..forward import Layer_Dense


class BiploarPerceptron(Layer_Dense):
    def __init__(self,pred,lable):
        self.pred = pred
        self.lable = lable
        self.parameters = Layer_Dense._params #parameter
        self.dw={}
    def loss(self):
        return max(0,-(self.lable+self.parameters['Z'+str(len(Layer_Dense.layer_activations.keys()))]))
    def preceptron_loss_grad(self):
        dwm =[]
        L =len(self.lable)
        for l in reversed(range(L)):
            if (self.lable[l]*np.dot(self.parameters['w'+str(l)].shap,self.data[l])) > 0:
              dwm[l]= np.zeros(self.parameters['w'+str(l)].shape)
            else:
              for j in range (0,L):
                  dwm[j] = np.dot(self.lable[l],self.data[j])
            self.dw['dw'+str(l)]= dwm
```

Fig.3 Bipolar Perceptron

4.5.2. Bipolar SVM:

```python

class Biploar_SVM(Layer_Dense) :
    def __init__(self,pred,lable):
        self.pred = pred
        self.lable = lable
        self.parameters = Layer_Dense._params #parameter
        self.dw={}

    def loss(self):
        return max(0,1-(self.lable+self.parameters['Z'+str(len(Layer_Dense.layer_activations.keys()))]))

    def SVM_loss_grad(self):
        dwm =[]
        L = len(self.lable)
        for l in reversed(range(L)):

            if (self.lable[l]*np.dot(self.parameters['W'+str(l)].shap,self.data[l])) > 1:
              dwm[l]= np.zeros(self.parameters['W'+str(l)].shape)

            else :
              for j in range (0,L):
                  dwm[j] = np.dot(self.lable[l],self.data[j])


            self.dw['dW'+str(l)]= dwm
```

Fig.4 Bipolar SVM

4.5.3 SD

18 lines (14 sloc) | 534 Bytes

```python
import numpy as np
from ..forward import Layer_Dense


class SD(Layer_Dense):
    def __init__(self,pred,lable):
        self.pred = pred
        self.lable = lable
        # self.parameters = parameter#Layer_Dense._params #parameter
        self.dw={}

    def loss(self) :
      pred_minus_lable = np.subtract(self.pred , self.lable)
      pred_minus_lable_T = pred_minus_lable.T
      return 0.5 * np.dot(pred_minus_lable_T, pred_minus_lable)

    def SQD_loss_grade(self) :
        return np.subtract(self.pred , self.lable)
```

Fig.5 SD Losses

4.5.4. Multi Class Perceptron:

```python

from ..forward import Sigmoid,ReLU,Tanh,Linear,Identity
import numpy as np

class MultiClassPerceptron:

    def __init__(self , y_out , y_true ):

        self.y_true= y_true-1
        self.y_out=y_out

        self.grads={}


    def loss(self):

        sample_loss=0

        last_layer=len(self.layers_num_arr)
        a_prelast= self._params['A'+ str(last_layer-1)]
        z=self._params['Z'+ str(last_layer)]
        dW = np.zeros((len(z), len(a_prelast)+1))
        dA_prev = np.zeros((len( z), len(a_prelast)))
        a_prelast= np.append( a_prelast ,1).T
        w=self._params['W'+ str(last_layer)]
        act_fc=self.layer_activations[last_layer]
        dl_dz=(np.zeros(len(z))).reshape(len(z) , 1)
```

Fig.6 Multi Class Perceptron

4.5.5. Multi Class SVM:

```python
from ..forward import Sigmoid,ReLU,Tanh,Linear,Identity
import numpy as np


class MultiClassSVM: #(Layer_Dense): #y_true, y_out , a_prelast , w, act_fc , b
    def  __init__ (self , y_out , y_true):
        self.y_true=y_true-1
        self.y_out=y_out
        self.grads={}

    @staticmethod
    def loss(AL,ZL,y_true,layers_num_arr,_params,layer_activations):
        sample_loss=0
        count = 0
        last_layer=len(layers_num_arr)
        a_prelast= _params['A'+ str(last_layer-1)]
        z=_params['Z'+ str(last_layer)]
        dW = np.zeros((len(z), len(a_prelast)+1))
        dA_prev = np.zeros((len( z), len(a_prelast)))
        a_prelast= np.append( a_prelast ,1).T
        w= _params['W'+ str(last_layer)]
        act_fc= layer_activations[ last_layer]
        dl_dz=(np.zeros(len(z))).reshape(len(z) , 1)

        label = y_true

```

Fig.7 Multi Class SVM

4.5.6. Softmax Cross Entropy:

```python
from ..forward import Sigmoid,ReLU,Tanh,Linear,Identity
import numpy as np


class SoftmaxCrossEntropy:
    def  __init__ (self , y_out , y_true):

        self.y_true = y_true-1
        self.y_out = y_out
        self.grads={}

    def loss(self):

        sample_loss=0
        label=self.y_true

        dl_dz=(np.zeros(len(self.y_out))).reshape(len(self.y_out) , 1)
        last_layer=len(self.layers_num_arr)
        #ind = self.CONST[label]
        a_prelast= self._params['A'+ str(last_layer-1)]
        l_inp=len(a_prelast)
        z=self._params['Z'+ str(last_layer)]
        dW = np.zeros((len(z), len(a_prelast)+1))
        dA_prev = np.zeros((len( z), len(a_prelast)))
        a_prelast= np.append( a_prelast ,1)
        a_prelast=a_prelast.reshape(1 ,len(a_prelast))

```

Fig.8 Softmax Cross Entropy

4.6. Backward module:

Back-propagation is the essence of neural net training. It is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.

Backpropagation is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps to calculate the gradient of a loss function with respect to all the weights in the network.

```python
import numpy as np

class backward:  # make inheratnce to loss, and layers
    def __init__(self):
        pass

    def _linear_backward(self,dZ, cache):
        """
        Implement the linear portion of backward propagation for a single layer (layer l)

        Arguments:
        dZ -- Gradient of the cost with respect to the linear output (of current layer l)
        cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the current layer

        Returns:
        dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
        dW -- Gradient of the cost with respect to W (current layer l), same shape as W
        db -- Gradient of the cost with respect to b (current layer l), same shape as b
        """
        A_prev, W, b = cache
        m = A_prev.shape[1]

        dW = (1/m) * np.matmul(dZ,A_prev.T)
        db = (1/m) * np.sum(dZ,axis=1,keepdims=True)
        dA_prev = np.matmul(W.T,dZ)

```

Fig.9 Backward Propagation

1 4.7. Optimization module:

4.7.1. Adam:

- Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data.

```python
import math
import numpy as np

class Adam:
    def __inti__(self):
        pass
    @staticmethod
    def initialize_adam(parameters,layerlen):
        L = layerlen  # number of layers in the neural networks
        v = {}
        s = {}

        for l in range(L):
            v["dW" + str(l+1)] = np.zeros(parameters['W'+str(l+1)].shape)
            v["db" + str(l+1)] = np.zeros(parameters['b'+str(l+1)].shape)
            s["dW" + str(l+1)] = np.zeros(parameters['W'+str(l+1)].shape)
            s["db" + str(l+1)] = np.zeros(parameters['b'+str(l+1)].shape)

        return v, s
    @staticmethod
    def update_parameters(layerlen,parameters, grads, v, s, t=2, learning_rate = 0.01,
                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):

        L = layerlen                 # number of layers in the neural networks
        v_corrected = {}                        # Initializing first moment estimate, python dictionary
        s_corrected = {}                        # Initializing second moment estimate, python dictionary

```

Fig.10 Adam Optimization

4.7.2. Momentum:

A very popular technique that is used along with SGD is called Momentum. Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the past steps to determine the direction to go.

```python
import numpy as np
class Momentum:
    def __init__(self,paramterts,lr=0.1,beta=0.9):
        self.params = paramterts
        self.lr = lr
        self.beta = beta
        self.v = {}

    def initialize_velocity(self):
        L = len(self.params) // 2
        v = {}
        for l in range(L):
            v['dW'+str(l+1)] = np.zeros(self.params['W'+str(l+1)].shape)
            v['db'+str(l+1)] = np.zeros(self.params['b'+str(l+1)].shape)
        self.v = v
    def update(self):
        L = len(self.params) // 2
        for l in range(L):
            self.v["dW" + str(l+1)] = (beta*self.v["dW" + str(l+1)]) + ((1-beta)*self.params['W'+str(l+1)])
            self.v["db" + str(l+1)] = (beta*self.v["db" + str(l+1)]) + ((1-beta)*self.params['b'+str(l+1)])
            self.params["W" + str(l+1)] = self.params["W" + str(l+1)] - (self.lr*self.v["dW" + str(l+1)])
            self.params["b" + str(l+1)] = self.params["b" + str(l+1)] - (self.lr*self.v["db" + str(l+1)])
    def __repr__(self):
        self.initialize_velocity()
        self.update()
        return {'velocity':self.v,
                    'parameters':self.params}
```

Fig.11 Momentum Optimization

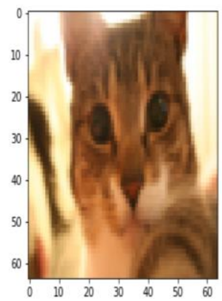
5. Testing

In [25]:
```
prob = net.Prediction(test_set_x,test_y,parameter=net.Parameters())
print(prob)
```

```
Accuracy: 0.62
[[0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1.
  1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1.
  0. 1.]]
```

In [26]:
```
index = 11
plt.imshow(test_x_orig[index])
print ("y = " + str(test_y[0,index]) + ". It's a " + classes[test_y[0,index]].decode("utf-8") +  " picture.")
```

```
y = 1. It's a cat picture.
```



In [27]:
```
index = 11
my_image = test_x_orig[index]
my_label_y = test_y[0,index]
my_image = my_image.reshape(64*64*3,1)
my_image = my_image/255
```


Fig.12 Unit Testing

6. Installation and Usage

- ## Usage

MNIST_Test.py > ...

```python
from DLFrameWork.forward import NetWork
from DLFrameWork.dataset import FashionMNIST,DataLoader

if __name__ == '__main__':
    FMNIST = FashionMNIST(path='MNIST_Data',download=True,train=True)

    dLoader = DataLoader(FMNIST,batchsize=500,shuffling=True,normalization={'Transform':True})

    net = NetWork((784,256,128,64,10),('ReLU','ReLU','ReLU','SoftMax'),optimType={'Adam':True})
    print(net)
    costs = []
    print_cost = True
    epochs = 10
    for i in range(epochs):
        cost = 0.0
        for j,(images,labels) in enumerate(dLoader):
            ourimages = images.T
            ourlabel = labels.T
            innercost = net.fit(ourimages,ourlabel,learning_rate =0.1)
            cost += innercost
            # print('iteration num {},inner cost is {}'.format(j, innercost))
        if print_cost:# and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i, cost/120))
            print('-'*10)


    # print(dLoader)
```

According to the previous example:

- At first the user needs to download & load the data whether it's MNIST or CIFAR So, he can use either the FashionMNIST or the CIFAR-10 class. Both classes allow the user to select the path he wants for the downloaded data, If he wants to download it or no and select which data he wants to use whether it's the train or test data

  FMNIST = FashionMNIST(path='MNIST_Data',download=True,train=True)

- Pass the chosen data to the data loader by passing the return of the FashionMNIST or the CIFAR-10 class then select his batch size to work on, choose if he wants to normalize the data or shuffle. Now the data is ready to be passed to the next stage which is entering the designed dl neural network

  dLoader = DataLoader(FMNIST,batchsize=500,shuffling=True,normalization={'Transform':True})

- After preprocessing the data the user needs to build his neural network and select numbers of inputs.

  creating the dense layers he needs and within he can select the number of hidden layers.

  By Network class the user can select a number of neurons in each layer and the desired activation function of all layers(Sigmoid, TanH, RelU, Softmax).Finally, selecting optimization methods(Adam, momentum).

- Once the user is done with loading the data and building his neural network what's left to do is to start training his model. This is done by selecting the number of the iterations then feeding the model with the data we got from the data loader and calculating the loss.

- Project Github repo: https://github.com/Mostafa-ashraf19/TourchPIP