my mnist

April 29, 2021

Nous allons essayer de reconnaitre des nombres écrit à la main grace à un réseau de neuronne.

[1]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pickle
```

[2]:
```
(X_train, Y_train), (X_test, Y_test) = tf.keras.datasets.mnist.load_data()

y_train = np.zeros((len(Y_train), 10))
y_train[np.arange(len(Y_train)), Y_train] = 1 # to categorical
y_test = np.zeros((len(Y_test), 10))
y_test[np.arange(len(Y_test)), Y_test] = 1 # to categorical
# cela permet de transformer la sortie en une liste [0, 0, 0, 0, 0, 0, 0, 0 ,0,␣
↪0, 0]
# avec un 1 à l'indice n
# par exemple si le nombre cherché est 2 : [0, 0, 1, 0, 0, 0, 0, 0 ,0, 0, 0]

x_train = X_train.reshape(-1, 28*28)/255 # 28*28 = 784
x_test = X_test.reshape(-1, 28*28)/255
```

1 POO

1.1 Activation

[9]:
```
"""
Fonction activation Sigmoid
"""
def sigmoid(x, derive=False):
    """
    Fonction Sigmoid
    """
    if derive:
        return np.exp(-x) / ((1+np.exp(-x)) ** 2)
    return 1 / (1 + np.exp(-x))

"""
```

```
Fonction activation Softmax
https://levelup.gitconnected.com/
↪killer-combo-softmax-and-cross-entropy-5907442f60ba
"""
def softmax(y, derivative=False):
    result = []
    for x in y:
        exps = np.exp(x - x.max()) # permet d'éviter une exponentielle trop␣
↪grande
        if derivative:
            result.append(exps / np.sum(exps, axis=0) * (1 - exps / np.
↪sum(exps, axis=0)))
        else:
            result.append(exps / np.sum(exps, axis=0))
    return np.array(result)
```

1.2 Layers

[4]:
```
class Layer:
    def __init__(self, input_n=2, output_n=2, lr=0.1, activation=None):
        """
        Crée un layer de n neuronne connecté aux layer de input neuronnes
        """
        # input_n le nombre d'entrée du neuronne
        # output_n le nombre de neuronne de sortie
        self.weight = np.random.randn(input_n, output_n)
        self.input_n = input_n
        self.output_n = output_n
        self.lr = lr # learning rate

        # the name of the layer is 1
        # next one is 2 and previous 0
        self.predicted_output_ = 0
        self.predicted_output  = 0
        self.input_data = 0

        # Fonction d'activation
        self.activation = activation if activation != None else lineaire

    def calculate(self, input_data):
        """
        Calcule la sortie
        """
        self.input_data = input_data
        # self.input_data = np.concatenate((input_data, np.
↪ones((len(input_data), 1))), axis=1)
```

```
        y1 = np.dot(self.input_data, self.weight)
        z1 = self.activation(y1)
        self.predicted_output_ = y1
        self.predicted_output = z1
        return y1, z1

    def learn(self, e_2):
        """
        Permet de mettre à jour les weigths
        """
        e1 = e_2 / self.output_n * self.activation(self.predicted_output_, True)
        # e_0 is for the next layer
        # e_0 = np.dot(e1, self.weight.T)
        e_0 = np.dot(e1, self.weight.T)
        dw1 = np.dot(e1.T, self.input_data)
        self.weight -= dw1.T * self.lr
        return e_0
```

1.3 Loss function

[5]:
```
"""
Mean Square Error function
Je l'utilise mais il serait mieux d'utiliser cross entropy normalement
"""
def mse(predicted_output, target_output, derivate=False):
    if derivate:
        return (predicted_output - target_output) *2
    return ((predicted_output - target_output) ** 2).mean()
```

1.4 Model

[6]:
```
class Model:

    def __init__(self, layers=[], loss_function=None):
        self.layers = layers
        self.loss = []
        self.lr = 0.1
        self.loss_function = loss_function

    def predict(self, input_data):
        predicted_output = input_data  # y_ is predicted data
        for layer in self.layers:
            predicted_output_, predicted_output = layer.
↪calculate(predicted_output) # output
        return predicted_output
```

```python
    def predict_loss(self, input_data, target_output):  # target_output is␣
↪expected data
        predicted_output = self.predict(input_data)  # y_ is predicted data
        loss = self.loss_function(predicted_output, target_output)
        return predicted_output, loss

    def compute_accuracy(self, x_val, y_val):
        predictions = []
        for x, y in zip(x_val, y_val):
            output = self.predict([x])
            pred = np.argmax(output[0])
            predictions.append(pred == np.argmax(y))
        return np.mean(predictions)

    def backpropagation(self, input_data, target_output, batch=None):
        n = len(input_data)
        if batch is None:
            batch = n
        step = n//batch
        losses = []
        for i in range(step):
            b_input_data = input_data[::step]
            b_target_output = target_output[::step]
            predicted_output, loss = self.predict_loss(b_input_data,␣
↪b_target_output)
            d_loss = self.loss_function(predicted_output, b_target_output,␣
↪True) # dérivé de loss dy_/dy
            # Entrainement des layers
            for i in range(len(self.layers)):
                d_loss = self.layers[-i - 1].learn(d_loss)
            losses.append(loss)
        loss = sum(losses)/len(losses)
        self.loss.append(loss)
        return loss
```

[10]:
```python
# Le model est de taille 784 -> 32 sigmoid -> 16 sigmoid -> 10 softmax
np.random.seed(2) # permet de rendre le programme reproductible
model = Model([
    Layer(784, 32, 0.001, sigmoid),
    Layer(32, 16, 0.001, sigmoid),
    Layer(16, 10, 0.001, softmax),
], mse)
```

[11]:
```python
# Entrainement
for i in range(50):
    loss = model.backpropagation(x_train, y_train)
    acc = model.compute_accuracy(x_test, y_test)
```

```python
    print(f"Epoch : {i} loss : {loss}, acc : {round(acc*100, 2)} %")
# Sur un des test précédement réalisé, après 3000 entrainement, on obtient 70%␣
↪d'accuracy
```

```
Epoch : 0 loss : 0.151419233487184, acc : 11.51 %
Epoch : 1 loss : 0.1157290798817189, acc : 12.01 %
Epoch : 2 loss : 0.1081752170670607, acc : 10.0 %
Epoch : 3 loss : 0.10180303771541105, acc : 10.32 %
Epoch : 4 loss : 0.09722655973188295, acc : 13.24 %
Epoch : 5 loss : 0.09452930048303006, acc : 13.27 %
Epoch : 6 loss : 0.09420079021127374, acc : 13.34 %
Epoch : 7 loss : 0.09388583492810146, acc : 13.67 %
Epoch : 8 loss : 0.09358154596426761, acc : 14.03 %
Epoch : 9 loss : 0.09328117087748627, acc : 14.24 %
Epoch : 10 loss : 0.09298778701802908, acc : 14.84 %
Epoch : 11 loss : 0.0926988624362722, acc : 15.11 %
Epoch : 12 loss : 0.0924154419825512, acc : 15.55 %
Epoch : 13 loss : 0.09213662672953978, acc : 15.87 %
Epoch : 14 loss : 0.09186268533033595, acc : 16.3 %
Epoch : 15 loss : 0.09159327645562816, acc : 16.76 %
Epoch : 16 loss : 0.09132842925791806, acc : 17.22 %
Epoch : 17 loss : 0.09106799753211056, acc : 17.92 %
Epoch : 18 loss : 0.09081196154177407, acc : 18.27 %
Epoch : 19 loss : 0.09056025572738133, acc : 18.81 %
Epoch : 20 loss : 0.09031286496690681, acc : 19.13 %
Epoch : 21 loss : 0.0900697647235585, acc : 19.64 %
Epoch : 22 loss : 0.08983095376259054, acc : 20.1 %
Epoch : 23 loss : 0.08959642973161809, acc : 20.55 %
Epoch : 24 loss : 0.0893662008850668, acc : 20.89 %
Epoch : 25 loss : 0.08914027404235514, acc : 21.29 %
Epoch : 26 loss : 0.08891865882675982, acc : 21.69 %
Epoch : 27 loss : 0.08870136075005296, acc : 22.01 %
Epoch : 28 loss : 0.08848838273501398, acc : 22.24 %
Epoch : 29 loss : 0.08827972085815047, acc : 22.5 %
Epoch : 30 loss : 0.0880753650394452, acc : 22.72 %
Epoch : 31 loss : 0.08787529647848584, acc : 23.06 %
Epoch : 32 loss : 0.08767948828333874, acc : 23.27 %
Epoch : 33 loss : 0.0874879041105026, acc : 23.54 %
Epoch : 34 loss : 0.08730049900195966, acc : 23.72 %
Epoch : 35 loss : 0.08711721888714853, acc : 24.1 %
Epoch : 36 loss : 0.08693800165246961, acc : 24.33 %
Epoch : 37 loss : 0.08676277722056591, acc : 24.55 %
Epoch : 38 loss : 0.08659146876791701, acc : 24.81 %
Epoch : 39 loss : 0.08642399314183263, acc : 25.07 %
Epoch : 40 loss : 0.08626026211216349, acc : 25.3 %
Epoch : 41 loss : 0.08610018293825808, acc : 25.49 %
Epoch : 42 loss : 0.08594365955414401, acc : 25.78 %
```

```
Epoch : 43 loss : 0.0857905931531849, acc : 26.09 %
Epoch : 44 loss : 0.08564088323736645, acc : 26.24 %
Epoch : 45 loss : 0.08549442813863024, acc : 26.35 %
Epoch : 46 loss : 0.08535112589631406, acc : 26.65 %
Epoch : 47 loss : 0.085210874675363, acc : 26.84 %
Epoch : 48 loss : 0.08507357346336095, acc : 27.01 %
Epoch : 49 loss : 0.08493912237497195, acc : 27.11 %
```

[12]:
```python
# sauvegarde du model
pickle.dump( model, open( "demo.p", "wb" ) )
```

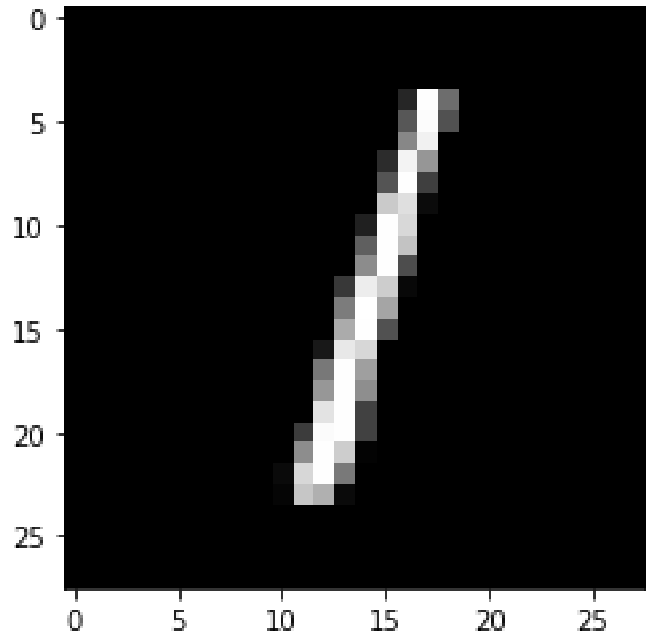
2 Test sur le model

[13]:
```python
model = pickle.load( open( "demo.p", "rb" ) )
```

[14]:
```python
# L'accuracy n'est pas le meme selon les nombres
for i in range(10):
    indexer = (Y_test == i)
    acc = model.compute_accuracy(x_test[indexer], y_test[indexer])
    print(f"For {i} accuracy is {round(acc * 100, 2)}")
```

```
For 0 accuracy is 28.06
For 1 accuracy is 80.0
For 2 accuracy is 6.49
For 3 accuracy is 7.52
For 4 accuracy is 14.05
For 5 accuracy is 6.95
For 6 accuracy is 47.08
For 7 accuracy is 39.2
For 8 accuracy is 9.34
For 9 accuracy is 23.79
```

[30]:
```python
plt.imshow(X_test[2], cmap="gray")
plt.show()
```



[16]:
```
model.predict([x_test[2]]).argmax()
```

[16]: 1