TC 5033

Advanced Machine Learning

José Antonio Cantoral Ceballos, Ph.D.

Team Members:

- A01200230 - Armando Bringas Corpus

Activity 1a: Implementing a Multilayer Fully Connected Network using Numpy

Non-graded activity (0 points)

- Objective

The primary objective of this activity is to deepen your understanding of Fully Connected Networks by implementing a multilayer network using only Numpy. You are given the follosing starter code that solves the MNIST dataset problem. Your task is to read, understand, and then apply this knowledge to solve classification problems on other datasets such as the Kaggle ASL dataset (Starter code will be provided separately for that activity).

- Instructions

  Read and Understand the following Code: The provided starter code outlines the architecture of a Fully Connected Network designed to classify MNIST images. Go through the code to understand how each function and class is used to implement the network.

Understand the Math: Make sure you understand the math operations implemented in the code, especially during the forward and backward passes. This will involve matrix multiplications, activation functions, loss computations, and backpropagation.

- Experiment You are encouraged to play with the code, change any hyperparameters and train the model, you should be able to achieve over 95% accuracy on the test set without problems.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Import Images

In [2]:
```
from get_images import get_images
```

In [3]:
```
# MNIST path
mnist_path = 'data/mnist_raw/'
x_train_num, y_train_num, x_test_num, y_test_num = get_images(mnist_path)

x_train = x_train_num[:50000].reshape(50000, -1).astype(float)
y_train = y_train_num[:50000].reshape(50000, 1)

x_val = x_train_num[50000:].reshape(10000, -1).astype(float)
y_val = y_train_num[50000:].reshape(10000, 1)

x_test = x_test_num.copy().reshape(10000, -1).astype(float)
y_test = y_test_num.copy().reshape(10000, 1)
```

In [4]:
```
x_train.mean(), x_train.std(), x_train.min()
```

Out[4]:
```
(33.39512885204082, 78.6661972212754, 0.0)
```

In [5]:
```
def normalise(x_mean, x_std, x_data):
    return (x_data - x_mean) / x_std
```

In [6]:
```
x_mean = x_train.mean()
x_std = x_train.std()

x_train = normalise(x_mean, x_std, x_train)
x_val = normalise(x_mean, x_std, x_val)
x_test = normalise(x_mean, x_std, x_test)
```

In [7]:
```
x_train.mean(), x_train.std()
```

Out[7]:
```
(-9.646705203355238e-18, 0.9999999999999997)
```

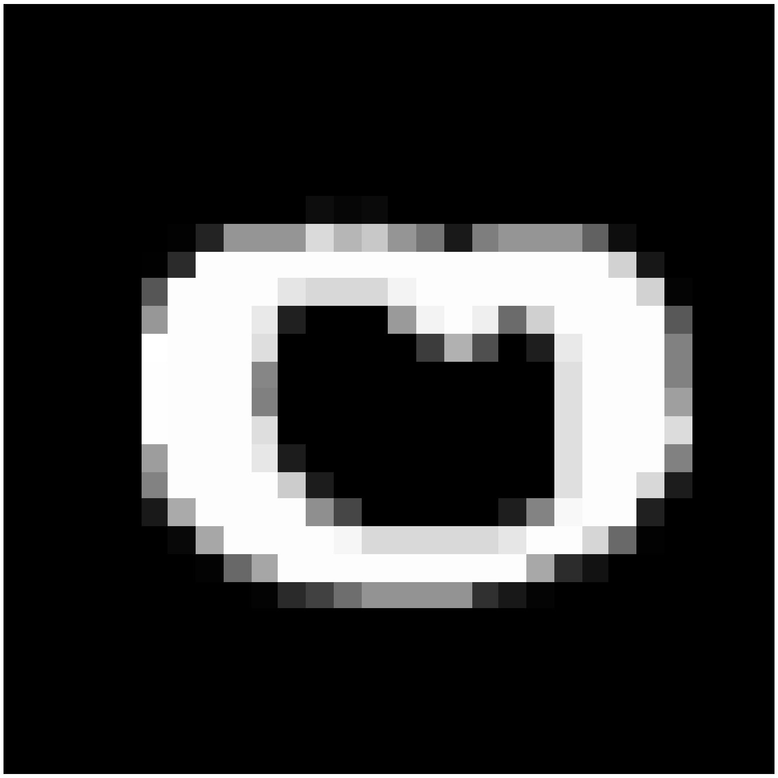
Plot samples

In [8]:
```
def plot_number(image):
    plt.figure(figsize=(5,5))
```

```
    plt.imshow(image.squeeze(), cmap=plt.get_cmap('gray'))
    plt.axis('off')
    plt.show()
```

In [9]:
```
rnd_idx = np.random.randint(len(y_test))
print(f'La imagen muestreada representa un: {y_test[rnd_idx, 0]}')
plot_number(x_test_num[rnd_idx])
```

```
La imagen muestreada representa un: 0
```



Equations

$$z^1 = W^1X + b^1$$

$$a^1 = ReLU(z^1)$$

$$z^2 = W^2a^1 + b^2$$

$$\hat{y} = \frac{e^{z^2{}_k}}{\sum_j e^{z_j}}$$

$$\mathcal{L}(\hat{y}^i, y^i) = -y^i \ln(\hat{y}^i) = -\ln(\hat{y}^i)$$

$$\mathcal{J}(w, b) = \frac{1}{num\_samples} \sum_{i=1}^{num\_samples} -\ln(\hat{y}^i)$$

Helper functions

Create Mini batches

In [10]:
```
def create_minibatches(mb_size, x, y, shuffle = True):
    '''
    x  #muestras, 784
    y #muestras, 1
    '''
    assert x.shape[0] == y.shape[0], 'Error en cantidad de muestras'
    total_data = x.shape[0]
    if shuffle:
        idxs = np.arange(total_data)
        np.random.shuffle(idxs)
        x = x[idxs]
        y = y[idxs]
    return ((x[i:i+mb_size], y[i:i+mb_size]) for i in range(0, total_data, mb_size))
```

Linear, ReLU and Sequential classes

In [11]:
```
class np_tensor(np.ndarray): pass
```

In [12]:
```
a = np.array([0, 0])
b = a.view(np_tensor)
```

In [13]:
```
type(a)
```

Out[13]:
```
numpy.ndarray
```

In [14]:
```
type(b)
```

Out[14]:
```
__main__.np_tensor
```

In [15]:
```
a == b
```

Out[15]:
```
np_tensor([ True,  True])
```

In [16]:
```
a is b
```

Out[16]:
```
False
```

Linear class

In [17]:
```
class Linear():
    def __init__(self, input_size, output_size):
        '''
        Init parameters utilizando Kaiming He
        '''
        self.W = (np.random.randn(output_size, input_size) / np.sqrt(input_size/2))
        self.b = (np.zeros((output_size, 1))).view(np_tensor)
    def __call__(self, X): # esta el foward de la clase lineal
        Z = self.W @ X + self.b
```

```
        return Z
    def backward(self, X, Z):
        X.grad = self.W.T @ Z.grad
        self.W.grad = Z.grad @ X.T
        self.b.grad = np.sum(Z.grad, axis = 1, keepdims=True)
```

ReLU class

In [18]:
```
class ReLU():
    def __call__(self, Z):
        return np.maximum(0, Z)
    def backward(self, Z, A):
        Z.grad = A.grad.copy()
        Z.grad[Z <= 0] = 0
```

Sequential class

In [19]:
```
class Sequential_layers():
    def __init__(self, layers):
        '''
        layers - lista que contiene objetos de tipo Linear, ReLU
        '''
        self.layers = layers
        self.x = None
        self.outputs = {}
    def __call__(self, X):
        self.x = X
        self.outputs['l0'] = self.x
        for i, layer in enumerate(self.layers, 1):
            self.x = layer(self.x)
            self.outputs['l'+str(i)]=self.x
        return self.x
    def backward(self):
        for i in reversed(range(len(self.layers))):
            self.layers[i].backward(self.outputs['l'+str(i)], self.outputs['l'+str(
    def update(self, learning_rate = 1e-3):
        for layer in self.layers:
            if isinstance(layer, ReLU): continue
            layer.W = layer.W - learning_rate * layer.W.grad
            layer.b = layer.b - learning_rate * layer.b.grad
    def predict(self, X):
        return np.argmax(self.__call__(X))
```

Cost Function

In [20]:
```
def softmaxXEntropy(x, y):
    batch_size = x.shape[1]
    exp_scores = np.exp(x)
    probs = exp_scores / exp_scores.sum(axis = 0)
    preds = probs.copy()
    # Costo
```

```
    y_hat = probs[y.squeeze(), np.arange(batch_size)]
    cost = np.sum(-np.log(y_hat)) / batch_size
    # Calcular gradientes
    probs[y.squeeze(), np.arange(batch_size)] -= 1 #dl/dx
    x.grad = probs.copy()

    return preds, cost
```

Training Loop

In [21]:
```
def train(model, epochs, mb_size=128, learning_rate = 1e-3):
    for epoch in range(epochs):
        for i, (x, y) in enumerate(create_minibatches(mb_size, x_train, y_train)):
            scores = model(x.T.view(np_tensor))
            _, cost = softmaxXEntropy(scores, y)
            model.backward()
            model.update(learning_rate)
        print(f'costo: {cost}, accuracy: {accuracy(x_val, y_val, mb_size)}')
```

In [22]:
```
def accuracy(x, y, mb_size):
    correct = 0
    total = 0
    for i, (x, y) in enumerate(create_minibatches(mb_size, x, y)):
        pred = model(x.T.view(np_tensor))
        correct += np.sum(np.argmax(pred, axis=0) == y.squeeze())
        total += pred.shape[1]
    return correct/total
```

In [23]:
```
model = Sequential_layers([Linear(784, 200), ReLU(), Linear(200, 200), ReLU(), Line
mb_size = 512
learning_rate = 1e-4
epochs = 20
```

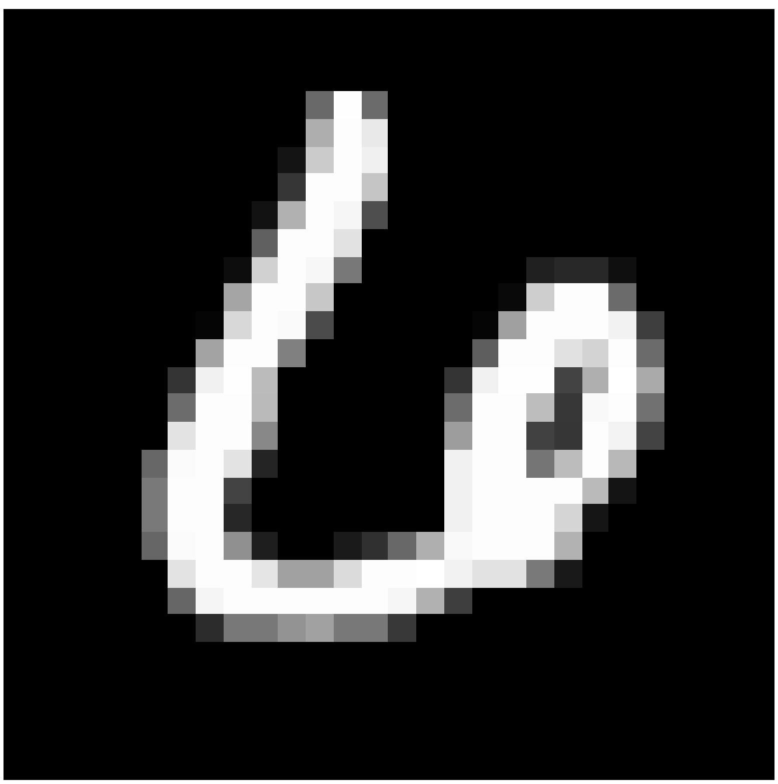
In [24]:
```
train(model, epochs, mb_size, learning_rate)
```

```
costo: 0.28137976011586663, accuracy: 0.919
costo: 0.26266943308309926, accuracy: 0.9344
costo: 0.26462165253048947, accuracy: 0.9473
costo: 0.2524000909635484, accuracy: 0.9535
costo: 0.18227676428083384, accuracy: 0.9562
costo: 0.11340337427622472, accuracy: 0.9615
costo: 0.13783988954581272, accuracy: 0.9607
costo: 0.16329686813083844, accuracy: 0.9638
costo: 0.08037089634142147, accuracy: 0.965
costo: 0.07035645625164537, accuracy: 0.9665
costo: 0.08448773959808355, accuracy: 0.9672
costo: 0.09362133181999231, accuracy: 0.9678
costo: 0.10413137898980851, accuracy: 0.969
costo: 0.08633836027346868, accuracy: 0.9693
costo: 0.07342279793265465, accuracy: 0.9699
costo: 0.05305521496215485, accuracy: 0.9704
costo: 0.045319071185451296, accuracy: 0.9711
costo: 0.03288242310244632, accuracy: 0.9717
costo: 0.05266148593261098, accuracy: 0.971
costo: 0.06736438891699519, accuracy: 0.9722
```

In [25]:
```python
print(accuracy(x_test, y_test, mb_size))
```

```
0.9706
```

In [26]:
```python
idx = np.random.randint(len(y_test))
plot_number(x_test_num[idx])
pred = model.predict(x_test[idx].reshape(-1, 1))
print(f'Predicted value is: {pred}, Real value is:{y_test[idx][0]}')
```



```
Predicted value is: 6, Real value is:6
```