

Redes Neuronales

Este es un proyecto del curso de [Introducción y conceptos básicos](#) de las Redes Neuronales. Comenzamos estudiando [Algoritmos de aprendizaje](#) y lo implementamos en el siguiente código que encontraremos en [Git](#).

El proceso de entrenamiento de una red neuronal implica dos fases principales:

- ***Feedforward (Propagación hacia adelante):** Durante esta fase, los datos de entrenamiento se introducen en la red neuronal, y los valores de activación se propagan a través de las capas de la red, calculando una salida. Esta salida se compara con las etiquetas de entrenamiento reales para calcular un error. El objetivo del entrenamiento es minimizar este error ajustando los pesos y sesgos de la red.
- **Backpropagation:** *Después de la fase de feedforward, se utiliza el algoritmo de retropropagación (backpropagation) para calcular las derivadas parciales del error con respecto a los pesos y sesgos en la red. Estas derivadas se utilizan para ajustar gradualmente los pesos y sesgos en la dirección que minimiza el error. Este proceso se repite iterativamente a lo largo de múltiples ejemplos de entrenamiento hasta que la red neuronal converge hacia un estado en el que el error sea mínimo.*

Los **algoritmos de aprendizaje** específicos se aplican en la fase de **backpropagation** para ajustar los parámetros del modelo y lograr el aprendizaje.

Creación de nuestro entorno

Creamos y entramos al directorio de nuestro proyecto en donde usamos [Comandos de Git](#) para inicializar nuestro nuevo repositorio vacío.

```
[apple ~] annOtoño23 > mkdir annAlgorithms
[apple ~] annOtoño23 > cd annAlgorithms
[apple ~] a/annAlgorithms > ls
[apple ~] a/annAlgorithms > git init
hint: Using 'master' as the name for the initial branch. This default branch name
      is subject to change. To configure the initial branch name to use in all
      of your new repositories, which will suppress this warning, call:
      git config --global init.defaultBranch <name>
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
      'development'. The just-created branch can be renamed via this command:
      git branch -m <name>
Initialized empty Git repository in /Users/vikoluna/Apps/PyCharm/annOtoño23/annAlgorithms/.git/
[apple ~] a/annAlgorithms git master > [green bar]
```

Una vez realizado esto, vemos que en nuestra terminal han aparecido cosas nuevas.

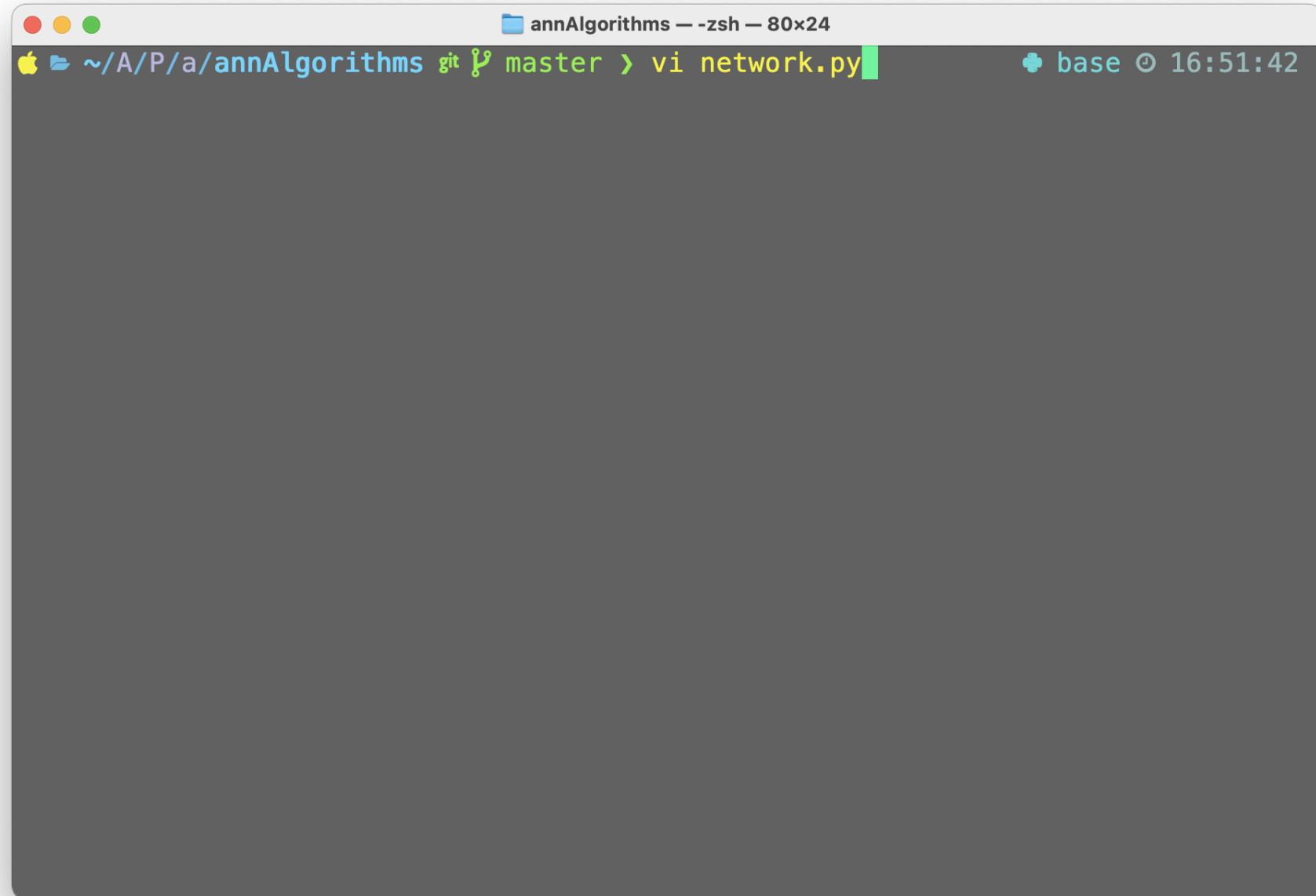
La rama por defecto de Git es la rama `master`, debemos revisar el estado de nuestro repositorio, en este caso como creamos un nuevo proyecto, podemos ver que está vacío.

```
annAlgorithms — -zsh — 80x24
[apple ~] ~ / A / P / annOtoño23 > mkdir annAlgorithms
[apple ~] ~ / A / P / annOtoño23 > cd annAlgorithms
[apple ~] ~ / A / P / a / annAlgorithms > ls
[apple ~] ~ / A / P / a / annAlgorithms > git init
+ base ⌂ 13:37:40 ]
hint: Using 'master' as the name for the initial branch. This default branch name
+ base ⌂ 13:38:45 ]
hint: is subject to change. To configure the initial branch name to use in all
+ base ⌂ 13:38:50 ]
hint: of your new repositories, which will suppress this warning, call:
+ base ⌂ 13:38:54 ]
hint:
hint:     git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:     git branch -m <name>
Initialized empty Git repository in /Users/vikoluna/Apps/PyCharm/annOtoño23/annAlgorithms/.git/
[apple ~] ~ / A / P / a / annAlgorithms git ↵ master > git status
+ base ⌂ 13:38:58 ]
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
[apple ~] ~ / A / P / a / annAlgorithms git ↵ master > █
+ base ⌂ 13:53:56 ]
```

Una vez hecho esto podemos comenzar a trabajar en nuestro proyecto dentro de [Python](#), primero creamos un archivo llamado `network.py` que será donde almacenaremos nuestros algoritmos.



A screenshot of a macOS terminal window titled "annAlgorithms — -zsh — 80x24". The window shows a command-line interface with the following text:

```
apple ~ /A/P/a/annAlgorithms git 🌟 master > vi network.py
```

The status bar at the bottom right indicates "base ⌂ 16:51:42". The main area of the terminal is completely blank, showing only the dark background of the window.

Comencemos analizando nuestro código, así que primero importamos las librerías necesarias para su buen funcionamiento.

```
# Librerias
import random
import numpy as np
```

Una vez que importamos las librerías creamos nuestro *objeto*, así que definimos la clase *Network* y comenzamos inicializando variables que interactúen con las otras funciones de nuestro código. Entonces tenemos lo siguiente

```
class Network(object):
    def __init__(self, sizes):
```

Dentro de la clase *Network* que será nuestra *Red neuronal* comenzamos escribiendo un método *__init__* el cual se llama cada vez que un *objeto* se crea de una *clase*. Éste método permite a la clase *inicializar los atributos del objeto* y se usa únicamente en clases.

Los doble guiones bajos antes y después de " *init* " indican que es un *método especial* en Python.

El parámetro *self* es una convención en Python que *se utiliza para hacer referencia a la instancia actual de la clase*, cuando se define un método en una clase, el primer parámetro del método debe ser *self*.

Esto quiere decir que se hará referencia a un objeto específico de la clase *Network* que sigue las definiciones de la clase, pero que tiene sus propios valores para las variables y puede interactuar con las funciones de la clase.

Además, *sizes* es un parámetro que se espera que se pase al constructor cuando se crea una instancia de la clase. El constructor se utiliza para inicializar los atributos y propiedades de una instancia de la clase cuando se crea.

En este caso, parece que se espera que se pase una *lista* de tamaños (*sizes*) que contiene el número de neuronas en la respectiva capa de la red. Esto es, que si la lista fuera [2, 3, 1], entonces habría una *red de tres capas*. La primera capa contiene *dos* neuronas, la segunda capa contiene *tres* neuronas y la tercera capa solo *una* neurona.

Sesgos y Pesos

Ahora, necesitaremos una variable para *sesgo (bias)* y otra para el *peso (weight)* las cuales las inicializaremos aleatoriamente usando una *distribución Gaussiana* con media 0 y varianza 1. Entonces nuestro método constructor contendrá lo siguiente:

```
def __init__(self, sizes):
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
    self.weights = [np.random.randn(y, x)
                   for x, y in zip(sizes[:-1], sizes[1:])]
```

En la primera capa se asume que debe ser una *capa de entrada* y por convención no ajustaremos ningunos bias para esas neuronas por que *solo son usados* en el cálculo de las salidas de las capas siguientes.

Feedforward

Comenzaremos con el "Feedforward" el cual se refiere a la propagación hacia adelante de los datos a través de la red neuronal, desde la capa de entrada hasta la capa de salida, sin retroalimentación ni ajuste de pesos durante esta fase. Es parte de la fase de inferencia o evaluación del modelo. Por lo que comenzamos definiendo una función de activación.

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a) + b)
```

Este método toma una entrada `a` y realiza la propagación hacia adelante (`feedforward`) a través de la red neuronal utilizando los pesos y sesgos almacenados en los atributos `self.biases` y `self.weights`.

Entonces para cada valor de b y w , se realizará la operación $(w \cdot a) + b$ y el resultado se introduce a una función sigmoide, esto representa la probabilidad la cual se almacenará nuevamente en `a`. Este resultado es el *valor de activación*.

El bucle `for` itera a través de las listas de sesgos b y pesos w simultáneamente utilizando la función `zip`.

La función `sigmoid` es una función de activación comúnmente utilizada en redes neuronales. Toma como entrada $(w \cdot a) + b$ y aplica la función sigmoide para calcular la activación de la neurona.

Finalmente, se actualiza `a` con los valores de activación calculados para la capa actual. Estos valores de activación se utilizarán como entrada para la siguiente capa en la red neuronal.

Ahora implementaremos algunos [Algoritmos de aprendizaje](#) los cuales ajustan automáticamente los pesos " w " y los bias " b " de las redes neuronales.

SGD

El [Stochastic Gradient Descent](#) lo definiremos como un nuevo método en el cual agregaremos el algoritmo, entonces, vayamos por pequeñas partes.

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None)

    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)
```

Primero como ya sabemos definimos nuestro método SGD, le pasamos los atributos que son `training_data`, `epochs`, `mini_batch_size`, `eta` y `test_data`.

Una vez declarado esto lo que haremos será convertir nuestros datos de entrada en una lista y luego se va a registrar cuantos elementos hay y lo almacenaremos en `n`.

El bloque de código con el `if test_data` se utiliza *para verificar si se ha proporcionado un conjunto de datos de prueba* (`test_data`) al método `SGD`. Si se ha proporcionado un conjunto de datos de prueba al método, esta condición *será verdadera* y se ejecutará el bloque de código siguiente. Si no se ha proporcionado un conjunto de datos de prueba y `test_data` es `None`, el bloque no se ejecutará.

Si se proporciona un conjunto de datos de prueba, convertimos nuestros datos de prueba a una lista y luego almacenaremos su número de elementos en `ntest`.

```
for j in range(epochs):
    random.shuffle(training_data)
```

```
mini_batches = [training_data[k:k+mini_batch_size]
               for k in range(0, n, mini_batch_size)]
```

Lo que hace este fragmento de código es realizar varias iteraciones (*épocas*) sobre el conjunto de datos de entrenamiento, aplicando SGD en cada iteración.

Cuando se terminaron de recorrer todos los *datos de entrenamiento* se dice que ha pasado una *época*.

Este bucle `for` ejecuta el entrenamiento durante un número específico de épocas.

Antes de comenzar cada época, el código mezcla aleatoriamente el conjunto de datos de entrenamiento. Esto es importante para garantizar que los ejemplos se presenten en un orden aleatorio en cada época, lo que ayuda a evitar problemas de sesgo y contribuye a una mejor convergencia del algoritmo SGD.

Luego, definimos el conjunto de `mini_batches` dividiendo el *conjunto de datos de entrenamiento* en partes más pequeñas donde `k` es un índice para dividir los datos en `mini_batches`. Notemos que `range(0, n, mini_batch_size)` genera una secuencia de índices desde 0 hasta `n` (la cantidad total de ejemplos de entrenamiento) con un *paso* de `mini_batch_size`. Esto divide efectivamente el conjunto de datos en mini-lotes de tamaño `mini_batch_size`.

```
for mini_batch in mini_batches:
    self.update_mini_batch(mini_batch, eta)
```

Por último, se itera sobre los mini-batches creados previamente y se llama al método `update_mini_batch` para actualizar los parámetros del modelo utilizando cada mini-batch, `mini_batch` es un conjunto pequeño de ejemplos de entrenamiento.

Dentro de cada iteración del bucle, se llama al método `update_mini_batch`, que es responsable de aplicar una actualización a los parámetros del modelo utilizando el mini-lote actual y una tasa de aprendizaje (`eta`).

- Finalmente recapitulemos el código completo:

El algoritmo de aprendizaje va a entrenar la red neuronal usando *mini-batch SGD*. Los datos de entrenamiento (`training_data`) es una lista de tuplas (x, y) que *representan las entradas de entrenamiento*. Si definimos `test_data` entonces la red será evaluada contra los datos de test después de cada época, e imprime el progreso parcial. Esto se hace para dar seguimiento al progreso pero desacelera substancialmente las cosas.

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k+mini_batch_size]
                       for k in range(0, n, mini_batch_size)]

        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)

        if test_data:
```

```

        print("Epoch {} : {} / {}".format(j,
                                             self.evaluate(test_data), n_test))

    else:
        print("Epoch {} complete !! ".format(j))

```

Método update_mini_batch

Ahora veamos la siguiente función que calcula las parciales de la función de costo con respecto a los parámetros del modelo (weights y biases) para un `mini_batch` y luego los actualiza. Recordemos que esto es parte del modelo de entrenamiento de una red neuronal utilizando el algoritmo de [Stochastic Gradient Descent](#) (SGD).

```

def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

```

- `nabla_b` y `nabla_w`: Estas son listas de matrices NumPy que se utilizan para acumular las gradientes con respecto a los sesgos (`nabla_b`) y los pesos (`nabla_w`) de la red neuronal. Estas matrices tienen la misma forma que los sesgos y los pesos de la red.
Se declaran `nabla_b` y `nabla_w` con *matrices* llenas de ceros pero con la misma forma que los sesgos (`self.biases`) y los pesos (`self.weights`) de la red neuronal. La forma de estas matrices se ajusta a la estructura de la red, donde cada elemento de la lista `self.biases` y `self.weights` representa los sesgos y los pesos de una capa de la red.

En la siguiente parte es donde *calculamos gradientes* de la función de costo con respecto a `self.biases` y `self.weights`.

$$\vec{\nabla}C = \left(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \dots, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial b_2}, \dots \right)$$

Para hacer mas rápido el algoritmo, se toma un subconjunto del total de datos de forma aleatoria. A este sub-conjunto se le llama **Mini-batch**. De modo que ahora

$$w_k \rightarrow w_k - \frac{\eta}{m_j} \sum_{X_j}^{m_j} \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_\ell \rightarrow b_\ell - \frac{\eta}{m_j} \sum_{X_j}^{m_j} \frac{\partial C_{X_j}}{\partial b_\ell}$$

m_j es el número de datos en el mini-batch j .

X_j es el conjunto de datos en el mini-batch j .

Los parámetros se actualizan con cada mini-batch $j = 1, 2, \dots$ sin reemplazo, hasta terminar con todos los datos.

Cuando se terminaron todos los datos se dice que ha pasado una **época**.

```

for x, y in mini_batch:
    delta_nabla_b, delta_nabla_w = self.backprop(x, y)
    nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
    nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

```

Primero en el `for x, y in mini_batch:`, el bucle `for` itera a través de cada ejemplo (`x, y`) en el `mini_batch`. Donde `x` representa las características de entrada del ejemplo y `y` representa la salida deseada correspondiente.

Luego, `delta_nabla_b, delta_nabla_w = self.backprop(x, y)` hace que en cada iteración del bucle, se llama al método `backprop` con el ejemplo (`x, y`) actual. El método `backprop` se utiliza para calcular las gradientes de la función de costo con respecto a los sesgos (`delta_nabla_b`) y los pesos (`delta_nabla_w`) de la red neuronal mediante el algoritmo de [Backpropagation](#). Estas gradientes representan cuánto debe cambiar cada sesgo y peso para minimizar la función de costo en función del ejemplo (`x, y`) actual.

Luego de calcular las gradientes para el ejemplo actual, se agregan a las matrices acumuladas `nabla_b`. Esto se hace sumando elemento por elemento las gradientes calculadas (`delta_nabla_b`) a las gradientes acumuladas (`nabla_b`). Se utiliza la [Función zip](#) para realizar esta operación.

Este proceso se repite para cada ejemplo en el `mini-batch` de entrenamiento, acumulando así las gradientes de la función de costo en función de todo el `mini-batch`. Estas gradientes acumuladas se utilizarán posteriormente para ajustar los parámetros del modelo (pesos y sesgos) en la dirección que minimiza el error.

```

self.weights = [w - (eta/len(mini_batch))*nw
                for w, nw in zip(self.weights, nabla_w)]
self.biases = [b - (eta / len(mini_batch)) * nb
               for b, nb in zip(self.biases, nabla_b)]

```

Finalmente, la implementación de la actualización de los pesos (`self.weights`) y sesgos (`self.biases`)

Declaramos los nuevos weights y biases donde se itera a través de las matrices de pesos (`self.weights`) y las gradientes de los pesos (`nabla_w`) al mismo tiempo utilizando `zip`.

Vemos que `w` representa una matriz de pesos de una capa de la red y `nw` representa la gradiente de los pesos de la misma capa correspondiente a los datos del `mini-batch` actual.

Código de la primera red network.py

```

# Libraries
import random
import numpy as np

class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                       for x, y in zip(sizes[:-1], sizes[1:])]

```

```

def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = self.sigmoid(np.dot(w, a) + b)

def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k + mini_batch_size]
                       for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print("Epoch {} : {} / {}".format(j, self.evaluate(test_data), n_test))
        else:
            print("Epoch {} complete!!".format(j))

    def update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w - (eta / len(mini_batch)) * nw
                       for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b - (eta / len(mini_batch)) * nb
                      for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # Feedforward
        activation = x
        activations = [x]
        zs = []
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation) + b
            zs.append(z)
            activation = self.sigmoid(z)
        activations.append(activation)

```

```

# Back pass
delta = self.cost_derivative(activations[-1], y) *
self.sigmoid_prime(zs[-1])
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
for l in range(2, self.num_layers):
z = zs[-l]
sp = self.sigmoid_prime(z)
delta = np.dot(self.weights[-l + 1].transpose(), delta) * sp
nabla_b[-l] = delta
nabla_w[-l] = np.dot(delta, activations[-l - 1].transpose())
return nabla_b, nabla_w

def evaluate(self, test_data):
test_results = [(np.argmax(self.feedforward(x)), y)
for (x, y) in test_data]
return sum(int(x == y) for (x, y) in test_results)

def cost_derivative(self, output_activations, y):
return output_activations - y

def sigmoid(z):
return 1.0/(1.0 + np.exp(-z))

def sigmoid_prime(self, z):
return self.sigmoid(z)*(1-self.sigmoid(z))

```

Cargando archivos a nuestro repositorio

Cargamos nuestro código con el modelo de la red neuronal a nuestro repositorio, por lo que vamos a nuestra terminal y tenemos

```
annAlgorithms -- zsh -- 80x24
[apple ~] ~/A/P/a/annAlgorithms git [?] master ?1 > ls
network.py
[apple ~] ~/A/P/a/annAlgorithms git [?] master ?1 > git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    network.py

nothing added to commit but untracked files present (use "git add" to track)
[apple ~] ~/A/P/a/annAlgorithms git [?] master ?1 > git add network.py
[apple ~] ~/A/P/a/annAlgorithms git [?] master +1 > git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   network.py

[apple ~] ~/A/P/a/annAlgorithms git [?] master +1 > [REDACTED]
[apple ~] base ④ 19:05:28 ]
[apple ~] base ④ 19:05:29 ]
[apple ~] base ④ 19:06:47 ]
[apple ~] base ④ 19:07:11 ]
[apple ~] base ④ 19:07:20 ]
```

Para subirlos podemos usar `git commit -m "Primer modelo de red neuronal"`: Sirve para que los archivos pasen a la etapa de subirlos a git-hub.

Ahora para pasarlo a git-hub obtendremos el link de nuestro repositorio el cual lo podemos crear en la página oficial de git-hub. El de este repositorio será <https://github.com/viko09/NeuralNetworkDev>

Para trabajar con git ahora se hacen los pasos anteriores y al momento de colocar el usuario y contraseña, en la contraseña debes generar un token para trabajar de forma remota con github. Para obtener una key abrimos github, luego vamos a '`settings`', luego en '`developer settings`' luego en '`personal access tokens`' luego en '`generate new token`', luego se coloca el nombre, todas las opciones del repositorio y luego en '`generar token`'. Despues de copia la llave y se pega.

En resumen utilizamos los siguientes comandos para subir nuestros archivos al repositorio:

- `git init`
- `git status`
- `git add filename`
- `git commit -m "Primer modelo de red neuronal"`
- `git branch -M main`
- `git remote add origin https://github.com/viko09/NeuralNetworkDev.git`
- `git push -u origin main`

```
annAlg -- zsh -- 80x24
! [rejected]      main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/viko09/NeuralNetworkDev.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
[apple ~] ~/A/P/a/annAlg git > main > git fetch origin          ✨ base ⌂ 22:08:08 ]
[apple ~] ~/A/P/a/annAlg git > main > git rebase origin/main    ✨ base ⌂ 22:08:39 ]
Successfully rebased and updated refs/heads/main.
[apple ~] ~/A/P/a/annAlg git > main > git rebase --continue     ✨ base ⌂ 22:08:46 ]
fatal: No rebase in progress?
[apple ~] ~/A/P/a/annAlg git > main > git push origin main      ✨ base ⌂ 22:08:52 ]
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 1.24 KiB | 1.24 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/viko09/NeuralNetworkDev.git
  18e2fa4..633f7e6  main -> main
[apple ~] ~/A/P/a/annAlg git > main > ls                         ✨ base ⌂ 22:08:58 ]
 README.md          network.py
[apple ~] ~/A/P/a/annAlg git > main > █                         ✨ base ⌂ 22:09:06 ]
```

```
annAlg -- zsh -- 80x24
test.py

nothing added to commit but untracked files present (use "git add" to track)
apple ~ / A / P / a / annAlg git ⌂ main ?3 > git add *           + base ⓧ 22:27:48 ]
apple ~ / A / P / a / annAlg git ⌂ main +3 > git commit -m "Se cargaron los archivos mnist_loader.py y test.py los cuales nos permiten entrenar la red. Tambien se cargaron los datos MNIST"
[main 6717151] Se cargaron los archivos mnist_loader.py y test.py los cuales nos permiten entrenar la red. Tambien se cargaron los datos MNIST
3 files changed, 53 insertions(+)
create mode 100644 mnist.pkl.gz
create mode 100644 mnist_loader.py
create mode 100644 test.py
apple ~ / A / P / a / annAlg git ⌂ main > git push -u origin main           + base ⓧ 22:29:06 ]
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.19 KiB | 607.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/viko09/NeuralNetworkDev.git
  633f7e6..6717151  main -> main
branch 'main' set up to track 'origin/main'.
apple ~ / A / P / a / annAlg cat ⌂ main > + base ⓧ 22:29:24
```

Subimos nuestros archivos a nuestro repositorio.

Implementación de la red

Una vez que hemos terminado de revisar el código de nuestra red neuronal, es momento de ponerla a trabajar. Para esto usaremos una base de datos para identificación de dígitos llamada [MNIST](#).

Corremos el archivo `test.py` y tenemos lo siguiente:

[apple ~] annAlg -- zsh -- 80x24
[apple ~] A/P/a/annAlg main > python3 test.py [base 22:30:57]