

Tarea 2: Miniflow

Rodríguez Hernández, Erick.

erickcarman822@hotmail.com

Centro de Innovación y Desarrollo Tecnológico en Cómputo

Abstract

TensorFlow is one of the most popular open source neural network libraries, built by the team at Google Brain over just the last few years. You'll spend the remainder working with open-source deep learning libraries like TensorFlow and Keras. Backpropagation is the process by which neural networks update the weights of the network over time. Differentiable graphs are graphs where the nodes are differentiable functions. They are also useful as visual aids for understanding and calculating complicated derivatives.

1. Introducción

Una red neuronal es un gráfico de funciones matemáticas tales como combinaciones lineales y funciones de activación. El grafo consiste en nodos y bordes. Los nodos de cada capa realizan funciones matemáticas utilizando entradas de los nodos de las capas anteriores.

De manera similar, cada nodo crea un valor de salida que se puede pasar a los nodos de la siguiente capa. El valor de salida de la capa de salida no se pasa a una capa futura. Las capas entre la capa de entrada y la capa de salida se llaman capas ocultas.

Los bordes en el gráfico describen las conexiones entre los nodos, a lo largo de los cuales los valores fluyen de una capa a la siguiente.

2. Forward Propagation

Al propagar valores desde la primera capa (la capa de entrada) a través de todas las funciones matemáticas representadas por cada nodo, la red genera un valor. Este proceso se denomina paso hacia adelante (forward pass). Observe que la capa de salida realiza una función matemática, además, sobre sus entradas. No hay una capa oculta.

3. Graphs

Los nodos y los bordes crean una estructura gráfica.

Generalmente hay dos pasos para crear redes neuronales:

1. Defina el gráfico de nodos y bordes.
2. Propague los valores a través del gráfico.

MiniFlow funciona de la misma manera. Definirá los nodos y bordes de su red con un método y luego propagará los valores a través del gráfico con otro método.

4. Ejercicio 05

En el ejercicio 5 vemos el funcionamiento de cómo trabaja MiniFlow e implementamos el método de avance (forward) sobre la clase suma (Add), que este a su vez es una subclase del nodo. En la figura [1] vemos la salida en la terminal de la construcción de una red de trabajo que resuelve la ecuación $(x + y) + y$

Código Fuente:

```
from miniflow import *

x, y = Input(), Input()

f = Add(x, y)
e = Add(f, y)

feed_dict = {x: 10, y: 5}

sorted_nodes = topological_sort(feed_dict)
output = forward_pass(e, sorted_nodes)

# the value for x with x.value (same goes for y).
print("{} + {} + {} = {}".format(feed_dict[x].value, feed_dict[y].value, feed_dict[y].value, output))
```

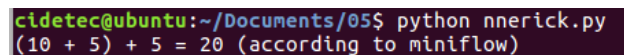


Fig 1. Salida de la terminal.

5. Ejercicio 07

Tal como hemos visto previamente, una neurona calcula la suma ponderada de sus entradas.

Una simple neurona artificial, depende de tres componentes:

1. Entradas, x (vector)
2. Pesos, w (vector)
3. Influencia (bias), b (escalar)

La salida, o , es simplemente la suma ponderada de las entradas más su influencia. Ecuación [1]

$$o = \sum_i x_i w_i + b \quad [1]$$

El aspecto de aprendizaje de las redes neuronales tiene lugar durante un proceso conocido como backpropagation. En backpropagation, la red modifica los pesos para mejorar la precisión de salida de la red.

En este ejercicio construimos una neurona que genera a su salida una versión simplificada de la ecuación [1], en dicha función debe tener una lista de nodos de entrada de longitud n , una lista de pesos de longitud n , y una influencia (bias).

Código fuente de la clase Linear:

```
class Linear(Node):
    def __init__(self, inputs, weights, bias):
        Node.__init__(self, [inputs, weights, bias])
    def forward(self):
        dot = np.dot(self.inbound_nodes[0].value,
                     self.inbound_nodes[1].value)
        self.value = dot + self.inbound_nodes[2].value
```

Código Fuente de la red neuronal:

```
from miniflow import *
inputs, weights, bias = Input(), Input(), Input()
f = Linear(inputs, weights, bias)
feed_dict = {
    inputs: [6, 14, 3],
    weights: [0.5, 0.25, 1.4],
    bias: 2
}
graph = topological_sort(feed_dict)
output = forward_pass(f, graph)
print(output)
```

Cuando realizamos las operaciones, tenemos que el producto de las entradas por los pesos nos da el valor de 10.7, y al sumar el *bias* nos da un valor final de 12.7, tal como lo apreciamos en la figura [2].

```
cidetec@ubuntu:~/Documents/07$ python nn.py
12.7
```

Fig 2. Salida de la terminal.

6. Ejercicio 08

Ahora veremos la transformación lineal, el primer cambio que denotaremos, recordando de la ecuación [1], es que cambiaremos la x por X y w por W , debido a que

ahora estaremos trabajando con matrices, y b ahora es un vector en lugar de un escalar.

1. Entradas, X (matriz)
2. Pesos, W (matriz)
3. Influencia (bias), b (vector)

Por lo que redefiniendo la ecuación [1] nos queda la siguiente ecuación [2]

$$Z = XW + b \quad [2]$$

Ahora, dentro de nuestra clase Linear, reescribimos nuestra función Forward para que podamos estar trabajando con matrices.

Código fuente de la clase Linear:

```
class Linear(Node):
    def __init__(self, X, W, b):
        Node.__init__(self, [X, W, b])
    def forward(self):
        X = self.inbound_nodes[0].value
        W = self.inbound_nodes[1].value
        b = self.inbound_nodes[2].value
        Y = np.dot(X, W) + b
        self.value = Y
```

Código Fuente de la red neuronal:

```
import numpy as np
from miniflow import *

X, W, b = Input(), Input(), Input()

f = Linear(X, W, b)

X_ = np.array([[-1., -2.], [-1, -2]])
W_ = np.array([[2., -3], [2., -3]])
b_ = np.array([-3., -5])

feed_dict = {X: X_, W: W_, b: b_}

graph = topological_sort(feed_dict)
output = forward_pass(f, graph)
print(output)
```

Al realizar la multiplicación de la matriz X por W , nos queda $\begin{bmatrix} -6, & 9 \\ -6, & 9 \end{bmatrix}$, al sumar el *bias*, nos queda $\begin{bmatrix} -9, & 4 \\ -9, & 4 \end{bmatrix}$, tal como se aprecia en la figura [3].

```
cidetec@ubuntu:~/Documents/08$ python nn.py
[[ -9.   4.]
 [ -9.   4.]
```

Fig 3. Salida de la terminal.

7. Ejercicio 09

Las transformaciones lineales son excelentes para simplemente cambiar valores, pero las redes neuronales a menudo requieren una transformación más matizada. Por ejemplo, uno de los diseños originales de una neurona artificial, el perceptron, exhibe un comportamiento binario

de salida. Las neuronas Perceptrones comparan una entrada ponderada con un umbral. Cuando la entrada ponderada supera el umbral, el perceptron se activa y su salida da 1, de lo contrario su salida da 0.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad [3]$$

La función sigmoide, ecuación [3], reemplaza el umbral con una curva en forma de S que imita el comportamiento de activación de un perceptron mientras que es diferenciable. Como ventaja, la función sigmoide tiene una derivada muy simple que puede ser calculada a partir de la propia función sigmoide, ecuación [4], donde σ representa la ecuación [3].

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \quad [4]$$

Ahora dentro de este ejercicio crearemos al nodo Sigmoid.

Código fuente de la clase Sigmoid:

```
class Sigmoid(Node):
    def __init__(self, node):
        Node.__init__(self, [node])

    def _sigmoid(self, x):
        return 1/(1+np.exp(-x))

    def forward(self):
        self.value = self._sigmoid(self.inbound_nodes[0].value)
```

Código Fuente de la red neuronal:

```
import numpy as np
from miniflow import *

X, W, b = Input(), Input(), Input()

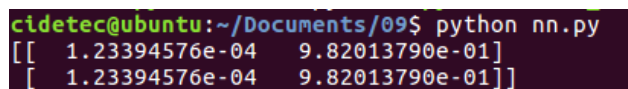
f = Linear(X, W, b)
g = Sigmoid(f)

X_ = np.array([[1., -2.], [-1, -2]])
W_ = np.array([[2., -3], [2., -3]])
b_ = np.array([-3., -5])

feed_dict = {X: X_, W: W_, b: b_}

graph = topological_sort(feed_dict)
output = forward_pass(g, graph)
print(output)
```

Ahora observamos que la salida de la transformación lineal es la entrada que alimenta a la función Sigmoid, dándonos como resultado lo observado en la figura [4].



```
cidedtec@ubuntu:~/Documents/09$ python nn.py
[[ 1.23394576e-04  9.82013790e-01]
 [ 1.23394576e-04  9.82013790e-01]]
```

Fig 4. Salida de la terminal.

8. Ejercicio 10

Hay muchas técnicas para definir la precisión de una red neuronal, todas las cuales se centran en la capacidad de la red para producir valores que se acercan lo más posible a valores correctos conocidos. La gente usa nombres diferentes para esta medida de exactitud, a menudo denominándola pérdida o costo.

Para calcular el costo usaremos el error de mínimos cuadrados (MSE), dado en la ecuación [5]

$$C(w, b) = \frac{1}{m} \sum_x ||y(x) - a||^2 \quad [5]$$

Aquí denotamos a w como la colección de todos los pesos en la red de trabajo, b todas las influencias, m es el número total de entrenamientos muestra, a es la aproximación de $y(x)$ por la red de trabajo, ambas a y $y(x)$ son vectores del mismo largo.

La colección de pesos es toda la matriz de peso aplanada en vectores y concatenadas a un vector grande. Lo mismo ocurre con la colección de *bias*, excepto que ya son vectores por lo que no hay necesidad de aplanar antes de la concatenación.

Ahora dentro de este ejercicio crearemos al nodo MSE para calcular el costo.

Código fuente de la clase MSE:

```
class MSE(Node):
    def __init__(self, y, a):
        Node.__init__(self, [y, a])

    def forward(self):
        y = self.inbound_nodes[0].value.reshape(-1, 1)
        a = self.inbound_nodes[1].value.reshape(-1, 1)
        # TODO: your code here
        m = len(y)
        div = (1.0 / len(y))
        mse = div * sum((y-a)**2)
        self.value = mse
```

Código Fuente de la red neuronal:

```
import numpy as np
from miniflow import *

y, a = Input(), Input()
cost = MSE(y, a)
```

```
y_ = np.array([1, 2, 3])
a_ = np.array([4.5, 5, 10])
```

```
feed_dict = {y: y_, a: a_}
graph = topological_sort(feed_dict)
# forward pass
forward_pass(graph)
print(cost.value)
```

Calculamos el costo aplicando el error de mínimos cuadrados, quedándonos lo mostrado en la Figura [5].

```
cidetec@ubuntu:~/Documents/10$ python nn.py  
[ 23.41666667]
```

Fig 5. Salida de la terminal.

9. Ejercicio 12

Backpropagation es el proceso por el cual la red ejecuta valores de error hacia atrás.

Durante este proceso, la red calcula la forma en la que los pesos deben cambiar (también llamado gradiente) para reducir el error general de la red. Cambiar los pesos usualmente ocurre a través de una técnica llamada gradiente descendiente.

El proceso de aprendizaje comienza con pesos y *bias* aleatorios.

El Gradiente descendiente trabaja calculando primero la pendiente del plano en el punto actual, que incluye el cálculo de las derivadas parciales de la pérdida con respecto a todos los parámetros. Este conjunto de derivadas parciales se denomina gradiente. A continuación, utiliza el gradiente para modificar los pesos de tal manera que el paso siguiente hacia adelante a través de la red mueva la salida más baja en la hipersuperficie.

¿Cómo podría el descenso gradiente no encontrar la diferencia mínima absoluta entre la salida de la red neuronal y la salida conocida?

La velocidad de aprendizaje (learning rate) determina cuánto se mueve el punto, hará que el punto sobrepase el mínimo absoluto si es que escogemos un valor muy grande.

Código fuente del gradiente descendiente:

```
def gradient_descent_update(x, gradx, learning_rate):  
    # TODO: Implement gradient descent.  
    update = learning_rate * gradx  
    x = x - update  
    # Return the new value for x  
    return x
```

Código Fuente de la red neuronal:

```
import random  
from gd import gradient_descent_update  
def f(x):  
    return x**2 + 5  
  
def df(x):  
    return 2*x  
  
# Random number between 0 and 10,000. Feel free to set x whatever  
# you like.  
x = random.randint(0, 10000)  
# TODO: Set the learning rate  
learning_rate = .1  
epochs = 100  
  
for i in range(epochs+1):
```

```
cost = f(x)  
gradx = df(x)  
print("EPOCH {}: Cost = {:.3f}, x = {:.3f}".format(i, cost, gradx))  
x = gradient_descent_update(x, gradx, learning_rate)
```

A continuación veremos que al dar un learning rate de 0.1, la cantidad de épocas (iteraciones) necesarias para que alcance un costo = 5 y un x = 0 es de 74 épocas. Figura[6]

```
cidetec@ubuntu:~/Documents/12$ python f.py  
EPOCH 0: Cost = 13329806.000, x = 7302.000  
EPOCH 1: Cost = 8531077.640, x = 5841.600  
EPOCH 2: Cost = 5459891.490, x = 4673.280  
EPOCH 3: Cost = 3494332.353, x = 3738.624  
EPOCH 4: Cost = 2236374.506, x = 2990.899  
EPOCH 5: Cost = 1431281.484, x = 2392.719  
EPOCH 6: Cost = 916021.950, x = 1914.175  
EPOCH 7: Cost = 586255.848, x = 1531.340  
EPOCH 8: Cost = 375285.543, x = 1225.072  
EPOCH 9: Cost = 240133.347, x = 980.058  
EPOCH 10: Cost = 153687.142, x = 784.046  
EPOCH 11: Cost = 98361.571, x = 627.237  
EPOCH 12: Cost = 62953.205, x = 501.790  
EPOCH 13: Cost = 40291.851, x = 401.432  
EPOCH 14: Cost = 25788.585, x = 321.145  
EPOCH 15: Cost = 16506.494, x = 256.916  
EPOCH 16: Cost = 10565.956, x = 205.533  
EPOCH 17: Cost = 6764.012, x = 164.426  
EPOCH 18: Cost = 4330.768, x = 131.541  
EPOCH 19: Cost = 2773.491, x = 105.233  
EPOCH 20: Cost = 1776.834, x = 84.106  
EPOCH 21: Cost = 1138.974, x = 67.349  
EPOCH 22: Cost = 730.743, x = 53.879  
EPOCH 23: Cost = 469.476, x = 43.183  
EPOCH 24: Cost = 302.264, x = 34.483  
EPOCH 25: Cost = 195.249, x = 27.586  
EPOCH 26: Cost = 126.760, x = 22.069  
EPOCH 27: Cost = 82.926, x = 17.655  
EPOCH 28: Cost = 54.873, x = 14.124  
EPOCH 29: Cost = 36.919, x = 11.299  
EPOCH 30: Cost = 25.428, x = 9.039  
EPOCH 31: Cost = 18.074, x = 7.232  
EPOCH 32: Cost = 13.367, x = 5.785  
EPOCH 33: Cost = 10.355, x = 4.628  
EPOCH 34: Cost = 8.427, x = 3.703  
EPOCH 35: Cost = 7.193, x = 2.962  
EPOCH 36: Cost = 6.484, x = 2.378  
EPOCH 37: Cost = 5.898, x = 1.896  
EPOCH 38: Cost = 5.575, x = 1.517  
EPOCH 39: Cost = 5.368, x = 1.213  
EPOCH 40: Cost = 5.236, x = 0.971  
EPOCH 41: Cost = 5.151, x = 0.776  
EPOCH 42: Cost = 5.096, x = 0.621  
EPOCH 43: Cost = 5.062, x = 0.497  
EPOCH 44: Cost = 5.040, x = 0.398  
EPOCH 45: Cost = 5.025, x = 0.318  
EPOCH 46: Cost = 5.010, x = 0.254  
EPOCH 47: Cost = 5.010, x = 0.204  
EPOCH 48: Cost = 5.007, x = 0.163  
EPOCH 49: Cost = 5.004, x = 0.130  
EPOCH 50: Cost = 5.003, x = 0.104  
EPOCH 51: Cost = 5.002, x = 0.083  
EPOCH 52: Cost = 5.001, x = 0.067  
EPOCH 53: Cost = 5.001, x = 0.053  
EPOCH 54: Cost = 5.000, x = 0.043  
EPOCH 55: Cost = 5.000, x = 0.034
```

Fig. 6. Salida de la terminal.

10. Ejercicio 13

Con el fin de averiguar cómo debemos alterar un parámetro para minimizar el costo, primero debemos averiguar qué efecto tiene ese parámetro en el costo.

Eso tiene sentido ya que no podemos simplemente cambiar los valores de los parámetros y obtener resultados significativos. El gradiente tiene en cuenta el efecto que tiene cada parámetro en el costo, así que esta es la manera cómo encontramos la dirección del ascenso más empinado.

¿Cómo determinamos el efecto que tiene un parámetro sobre el costo? Esta técnica es conocida como backpropagation o diferenciación en modo inverso. Simplemente calculamos la derivada del costo con

respecto a cada parámetro en la red. El gradiente es un vector de todas estas derivadas.

Código fuente de la clase Sigmoid:

```
class Sigmoid(Node):
    def __init__(self, node):
        # The base class constructor.
        Node.__init__(self, [node])

    def _sigmoid(self, x):
        """
        This method is separate from `forward` because it
        will be used with `backward` as well.

        `x`: A numpy array-like object.
        """
        return 1. / (1. + np.exp(-x))

    def forward(self):
        input_value = self.inbound_nodes[0].value
        self.value = self._sigmoid(input_value)

    def backward(self):
        # Initialize the gradients to 0.
        self.gradients = {n: np.zeros_like(n.value) for n in
                           self.inbound_nodes}
        # Cycle through the outputs. The gradient will change depending
        # on each output, so the gradients are summed over all outputs.
        for n in self.outbound_nodes:
            # Get the partial of the cost with respect to this node.
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += self.value*(1-self.value)
            * grad_cost
```

Código Fuente de la red neuronal:

```
import numpy as np
from miniflow import *

X, W, b = Input(), Input(), Input()
y = Input()
f = Linear(X, W, b)
a = Sigmoid(f)
cost = MSE(y, a)

X_ = np.array([[1., -2.], [-1, -2]])
W_ = np.array([[2.], [3.]])
b_ = np.array([-3.])
y_ = np.array([1, 2])

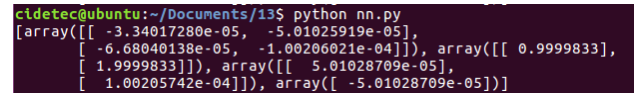
feed_dict = {
    X: X_,
    y: y_,
    W: W_,
    b: b_,
}

graph = topological_sort(feed_dict)
forward_and_backward(graph)
# return the gradients for each Input
gradients = [t.gradients[t] for t in [X, y, W, b]]
print(gradients)
```

Esto también se puede escribir como una composición de funciones:

$MSE(Linear(Sigmoid(Linear(X, W1, b1)), W2, b2), y)$.

Nuestro objetivo es ajustar los pesos y *bias* representados por los nodos de entrada $W1, b1, W2, b2$, de modo que el costo se minimice. Tal como se aprecia en el resultado de la figura [7].



```
cidetec@ubuntu:~/Documents/13$ python nn.py
[array([[ -3.34017280e-05,  -5.01025919e-05],
        [ -6.68040138e-05,  -1.00206021e-04]])], array([[ 0.9999833],
        [ 1.9999833]])], array([[ 5.01028709e-05],
        [ 1.00205742e-04]])], array([ -5.01028709e-05]))
```

Fig. 7. Salida de la terminal.

11. Ejercicio 14

El Gradiente Descendiente Estocástico (SGD por sus siglas en inglés) es una versión del Gradiente Descendiente donde en cada paso de avance (forward pass) un lote de datos es muestreado aleatoriamente del conjunto total de datos. Lo ideal sería que el conjunto de datos se alimentara en la red neuronal en cada paso hacia adelante, pero en la práctica, no es práctico debido a las restricciones de memoria. SGD es una aproximación de Gradiente Descendiente, cuanto más lotes procesados por la red neural, mejor es la aproximación.

Una implementación nativa de SGD implica:

1. Muestrear aleatoriamente un lote de datos del conjunto de datos total.
2. Ejecución de la red hacia delante (forward) y hacia atrás (backward) para calcular el gradiente (con datos de (1)).
3. Aplicar la actualización del Gradiente Descendiente
4. Repita los pasos 1-3 hasta que la convergencia o el bucle sea detenido por otro mecanismo (es decir, el número de épocas).