

Github <https://github.com/JDgomez2002/deep-learning.git>

1. Sí, existe una diferencia. Al inicializar los pesos con cero, todas las neuronas en cada capa reciben la misma señal, y esto provoca que se realicen las mismas operaciones en cada iteración durante el entrenamiento del modelo. Esto no sucede solo con 0, si no con cualquier valor constante, y la clave está en romper la simetría, usualmente usando distribuciones aleatorias, como la distribución normal o la uniforme.

Al inicializar los parámetros en cero, se obtienen los siguientes resultados:

<pre>1 def initialize_parameters(n_x, n_h, n_y): 2 W1 = np.zeros((n_h, n_x)) 3 b1 = np.zeros((n_h, 1)) 4 W2 = np.zeros((n_y, n_h)) 5 b2 = np.zeros((n_y, 1)) 6 7 parameters = { 8 "W1": W1, 9 "b1" : b1, 10 "W2": W2, 11 "b2" : b2 12 } 13 return parameters</pre>	<pre>Cost after iteration# 0: 0.693147 Cost after iteration# 100: 0.693147 Cost after iteration# 200: 0.693147 Cost after iteration# 300: 0.693147 Cost after iteration# 400: 0.693147 Cost after iteration# 500: 0.693147 Cost after iteration# 600: 0.693147 Cost after iteration# 700: 0.693147 Cost after iteration# 800: 0.693147 Cost after iteration# 900: 0.693147 Cost after iteration# 1000: 0.693147</pre>
--	---

El hecho de que el costo no disminuya después de cada iteración es un indicador de que el modelo no está aprendiendo. Esto significa que no estamos logrando minimizar la función de costo. Esto tiene sentido, ya que no estamos sacando provecho de las capas ocultas: todas realizan las mismas operaciones debido a la inicialización simétrica, lo que impide que el modelo aprenda correctamente durante la retropropagación.

En cambio, utilizando parámetros aleatorios podemos observar que se reduce la función de costo:

```

1 # Inicializacion de los parametros
2 def initialize_parameters(n_x, n_h, n_y):
3     W1 = np.random.randn(n_h, n_x)
4     b1 = np.random.randn(n_h, 1)
5     W2 = np.random.randn(n_y, n_h)
6     b2 = np.random.randn(n_y, 1)
7
8     parameters = {
9         "W1": W1,
10        "b1": b1,
11        "W2": W2,
12        "b2": b2
13    }
14    return parameters
15

```

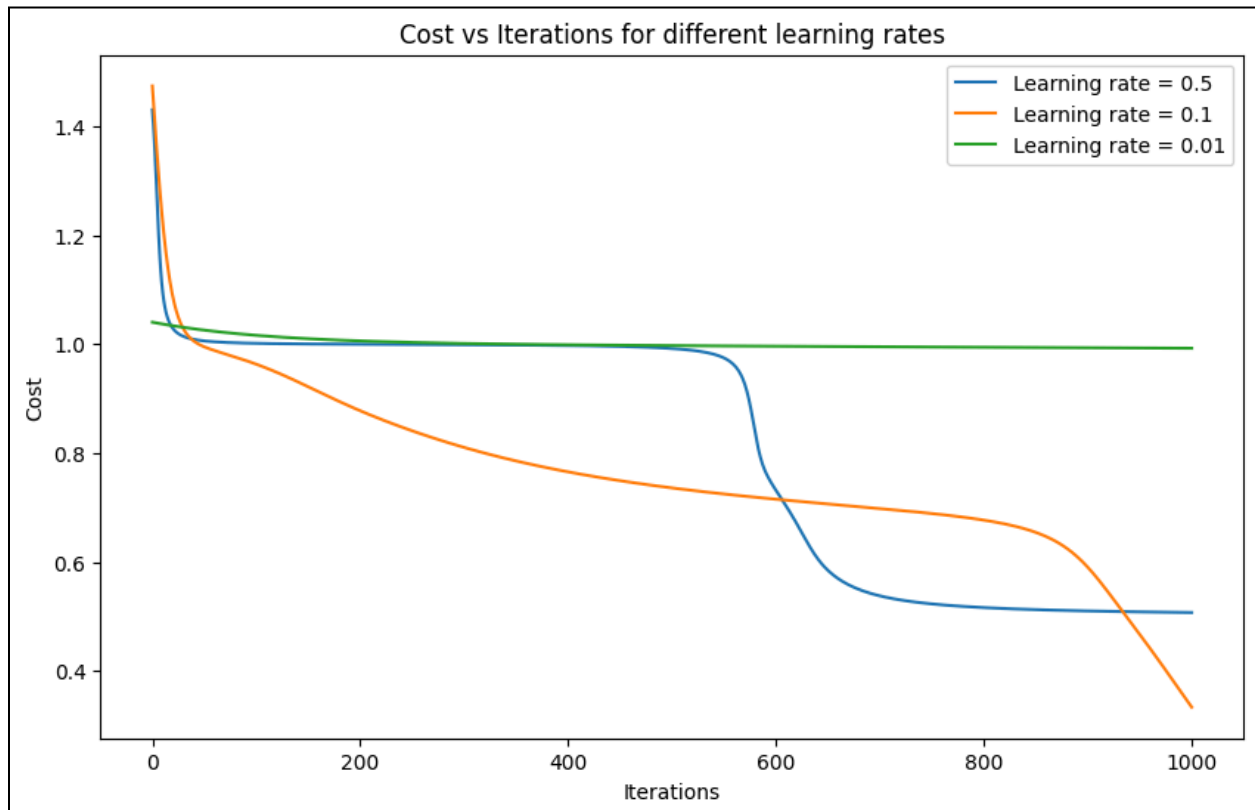
```

Cost after iteration# 0: 1.052558
Cost after iteration# 100: 0.695402
Cost after iteration# 200: 0.693668
Cost after iteration# 300: 0.693206
Cost after iteration# 400: 0.692966
Cost after iteration# 500: 0.692779
Cost after iteration# 600: 0.692587
Cost after iteration# 700: 0.692352
Cost after iteration# 800: 0.692030
Cost after iteration# 900: 0.691539
Cost after iteration# 1000: 0.690679

```

Si bien el modelo todavía no es muy bueno, es una mejora.


2.



Al observar los datos, nos damos cuenta de que un aprendizaje de 0.1 produjo los mejores resultados. Escoger un aprendizaje muy bajo significa que el modelo aprenderá de manera muy lenta, lo que puede llevar a un proceso de entrenamiento extremadamente largo y a la posibilidad de quedarse atrapado en mínimos locales. Por otro lado, uno muy alto puede provocar que el modelo oscile alrededor del mínimo de la función de costo sin converger, o incluso que diverja completamente. Por lo tanto, tiene sentido que el de 0.1 diera los mejores resultados, pues es un buen balance entre los dos.

3.

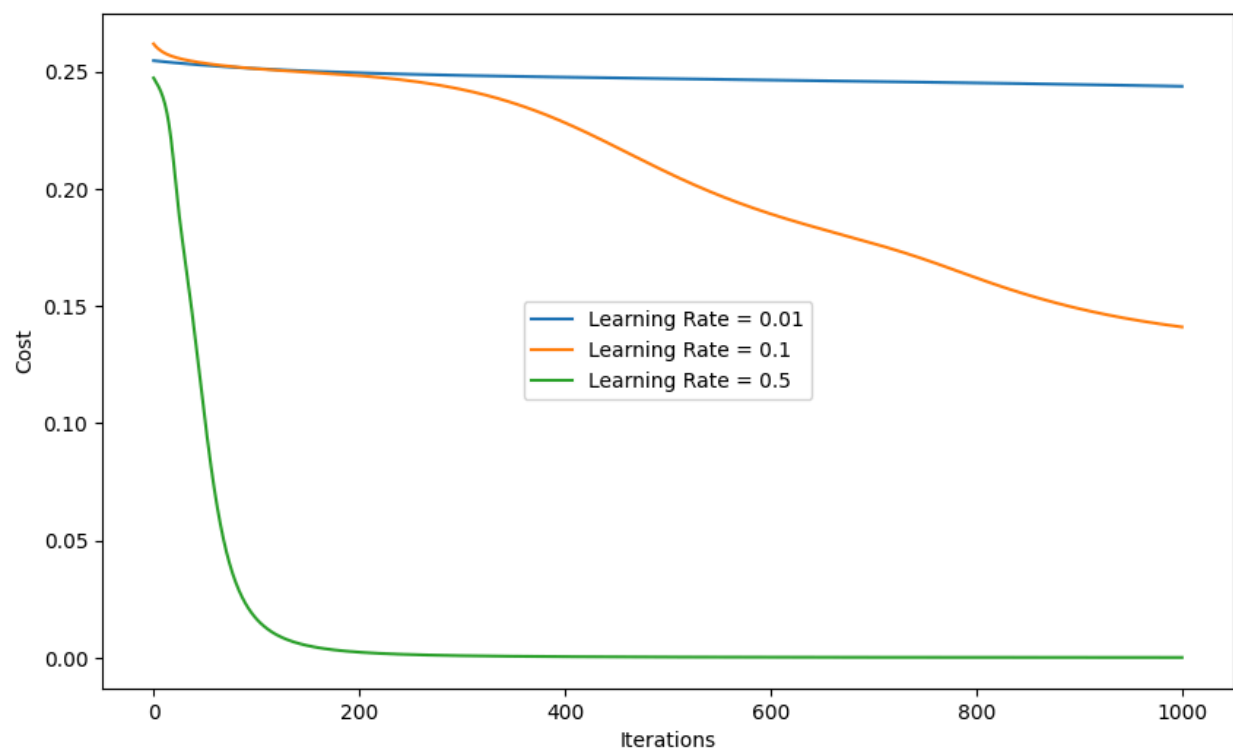
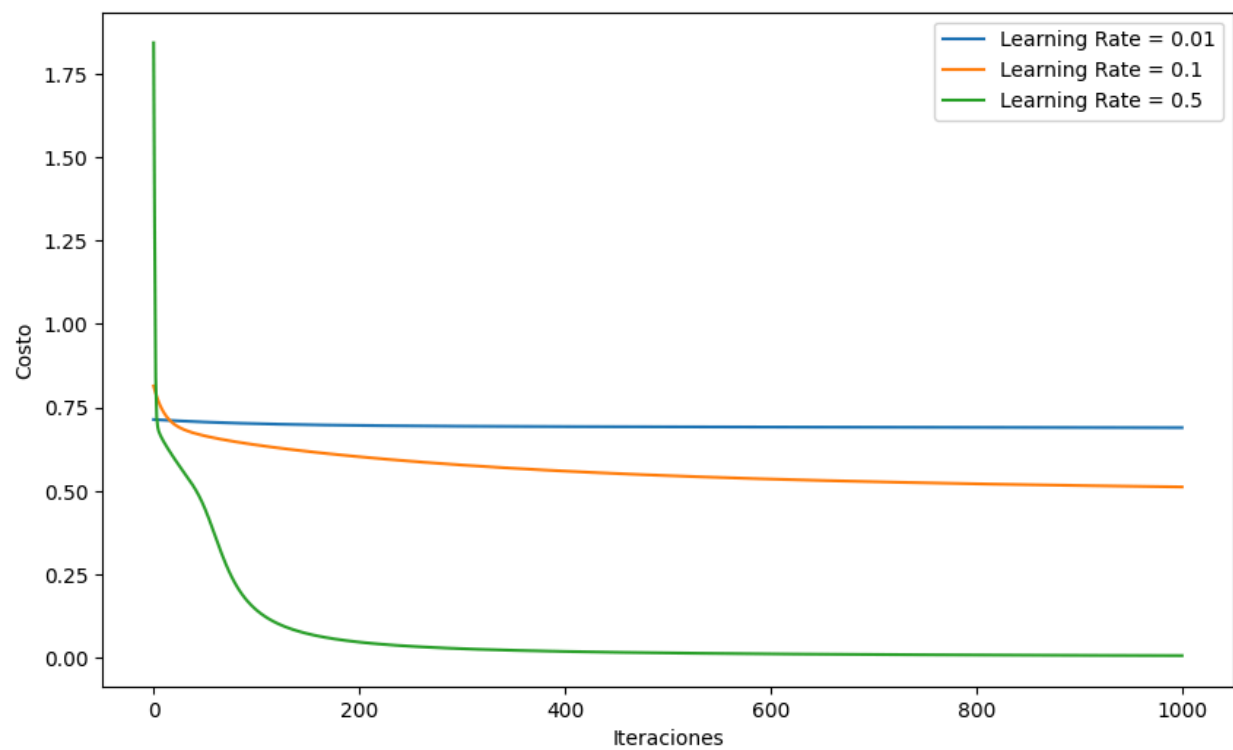
La función de costo ahora es:

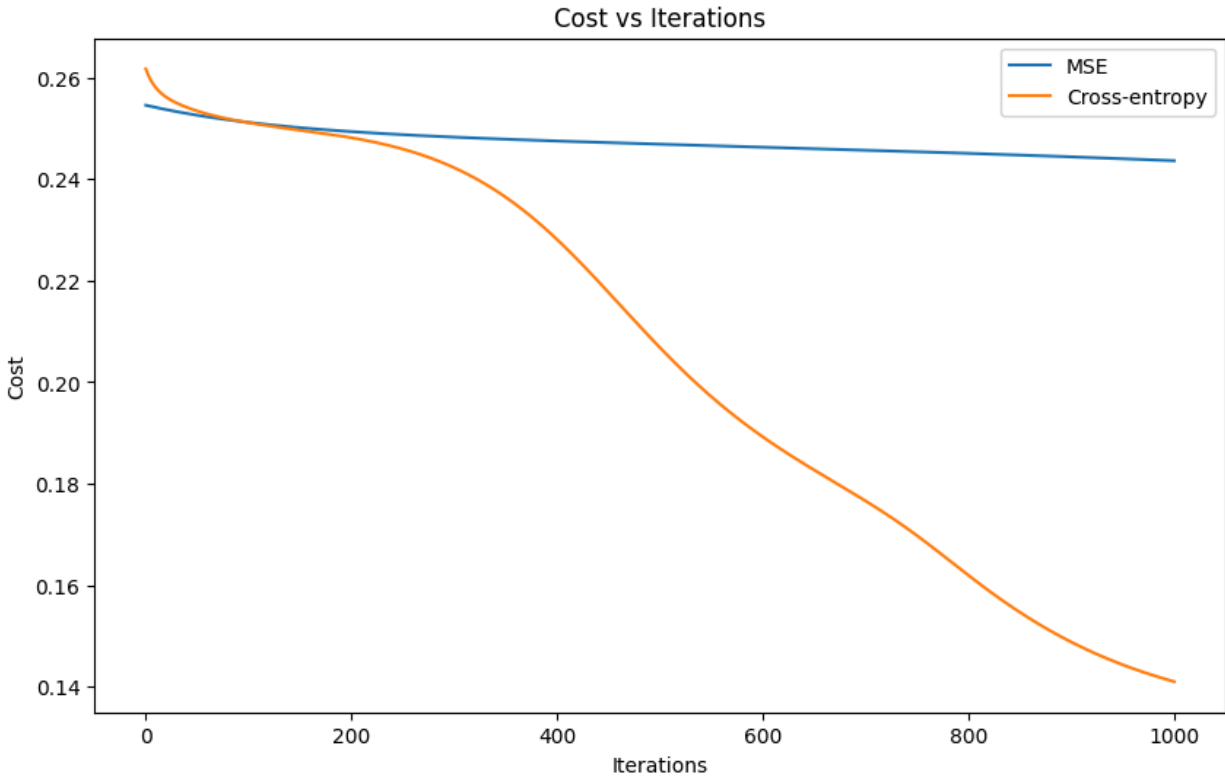


```
1 def loss_function_mse(A2, Y):
2     cost = np.sum((A2 - Y)**2)
3     cost = np.squeeze(cost)
4
5     return cost
```

Es simplemente la media de los cuadrados de las diferencias entre las predicciones y los valores reales. Usando el mismo learning rate, y la misma *seed* aleatoria inicial, se producen los siguientes resultados:

```
Cost after iteration# 0: 1.430116
Cost after iteration# 100: 1.430116
Cost after iteration# 200: 1.430116
Cost after iteration# 300: 1.430116
Cost after iteration# 400: 1.430116
Cost after iteration# 500: 1.430116
Cost after iteration# 600: 1.430116
Cost after iteration# 700: 1.430116
Cost after iteration# 800: 1.430116
Cost after iteration# 900: 1.430116
Cost after iteration# 1000: 1.430116
```





Para este caso, se ve que Cross-entropy presenta una mejor reducción de costo inicial que MSE. Esto quiere decir que muy probablemente la función de pérdida original pueda ser la mejor opción para esta configuración de MSE. Sin embargo, en MSE parece tener resultados similares para las primeras 200 iteraciones, sin embargo, posteriormente, la función de Cross entropy desciende bruscamente el costo inicial. En este caso, se puede confiar más en la función de Cross entropy, se debe seguir realizando más configuraciones de MSE para poder explorar diferentes resultados.