RED NEURONAL

Estado	IA

Creación de las partes de nuestro algoritmo

Los pasos principales para construir una red neuronal son:

1. Definir la estructura del modelo (como el número de características de entrada)
2. Inicializar los parámetros del modelo
3. Bucle:
    - Calcular la pérdida actual (propagación hacia adelante)
    - Calcular el gradiente actual (propagación hacia atrás)
    - Actualizar los parámetros (descenso del gradiente)

A menudo se construyen los pasos 1-3 por separado y se integran en una función que llamamos `model()`.

¿Para qué sirve `sigmoid`?

La función **sigmoid** convierte **cualquier número** en un valor entre **0 y 1**.

👉 Eso es perfecto porque:

- 0 → "no es gato"
- 1 → "sí es gato"
- valores intermedios → probabilidad

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

En regresión logística:

```
z = wᵀx + b
```

- `x` → una imagen (aplanada)
- `w` → pesos
- `b` → bias
- `z` → un número (o muchos)

La **sigmoid transforma ese número en una probabilidad**.

- `sigmoid` es un **convertidor**
- Convierte números en **probabilidades**
- Es la última capa de la regresión logística

- Sin ella, el modelo no podría decidir “sí” o “no

- Inicialización de parámetros

Se crean los parámetros iniciales del modelo:

- `w` (pesos)
- `b` (bias)

📦 ¿Qué son `w` y `b`?

🔹 `w` (pesos)

- Es un **vector**
- Tiene **un peso por cada píxel** de la imagen
- Si la imagen tiene `dim = 12288` píxeles:

```
w.shape = (12288, 1)
```

Cada número de `w` indica **qué tan importante es ese píxel**.

🔹 `b` (bias)

- Es un **solo número**
- Ayuda a mover la decisión del modelo
- Empieza en `0`

Propagación hacia adelante y hacia atrás

Paso 1: sacar un “puntaje” para cada foto

```
z = np.dot(w.T, X) + b
```

Paso 2: convertir el puntaje a probabilidad (0 a 1)

```
A = sigmoid(z)
```

Ahora la máquina dice:

- **A = 0.90** → "90% seguro que es gato"
- **A = 0.10** → "10% seguro que es gato (casi seguro que no)"

Paso 3: medir qué tan mal lo hizo (costo)

```
cost = (-1/m) * np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))
```

Piensa así:

- `Y` es la respuesta correcta (0 o 1).
- `A` es lo que el modelo predijo (probabilidad).

El **costo** es como una nota:

- costo grande → lo está haciendo mal
- costo pequeño → lo está haciendo bien

📌 El objetivo del entrenamiento es **bajar el costo**.

Backward propagation = "Cómo corregirse"

Aquí la máquina se pregunta:

> "Ok, fallé... ¿qué tengo que cambiar en `w` y `b` para fallar menos?"

Paso 1: calcular el error

```
A - Y
```

Ejemplos:

- Si `Y=1` (sí era gato) y `A=0.2` → error = -0.8
  (predije muy bajo, me equivoqué fuerte)
- Si `Y=0` y `A=0.7` → error = 0.7
  (predije muy alto para un no-gato)

✅ Esto te dice para cada foto: "me pasé" o "me quedé corto".

Paso 2: cuánto cambiar los pesos `w`

```
dw = (1/m) * np.dot(X, (A - Y).T)
```

- `dw` es una lista que dice:

  > "estos píxeles/pedacitos de la imagen están causando errores, ajusta sus pesos"

✅ `dw` te dice **cómo mover** `w`.

Paso 3: cuánto cambiar el sesgo `b`

```
db = (1/m) * np.sum(A - Y)
```

- `db` dice:

  > "en promedio estoy prediciendo muy alto o muy bajo, ajusta b"

✅ `db` te dice **cómo mover** `b`.

¿Qué hace `optimize()`?

Imagina que `w` y `b` son los "ajustes" de una máquina para reconocer gatos.

`optimize()` repite muchas veces este ciclo:

1. **Hace una predicción** con los valores actuales de `w` y `b`
2. **Mide qué tan mal lo hizo** (cost)
3. **Calcula cómo corregirse** (dw, db)
4. **Ajusta** `w` **y** `b` **un poquito** para hacerlo mejor

Eso es "descenso de gradiente".

1) Calcular error y cómo corregirse

```
grads, cost = propagate(w, b, X, Y)
```

Esto devuelve:

2) Sacar `dw` y `db`

```
dw = grads["dw"]
db = grads["db"]
```

Simplemente los extrae del diccionario.

- `cost` = “qué tan mal estoy” (una nota, mientras más baja mejor)
- `grads` = “qué tengo que cambiar”
  - `dw` → cuánto ajustar `w`
  - `db` → cuánto ajustar `b`

3) Actualizar parámetros (aquí aprende)

```
w = w - learning_rate * dw
b = b - learning_rate * db
```

- `dw` te dice **en qué dirección** mover `w`
- `learning_rate` te dice **qué tan grande** es el paso

📌 Si `learning_rate` es muy grande → se puede pasar y no aprende bien

📌 Si es muy pequeño → aprende muy lento

¿Por qué se resta?

Porque queremos **bajar el costo**.

Piensa en una montaña:

- `cost` es la altura
- queremos bajar hacia el valle (mínimo)
- `dw` y `db` son como la “pendiente”
- restar es moverte hacia abajo

🧾 Guardar costos para ver la curva

```
if i %100 ==0:
    costs.append(cost)
```

Cada 100 iteraciones guarda el costo para luego graficar si baja.

🖨️ Imprimir el costo (opcional)

```
if print_costand i %100 ==
0:print("Cost after iterati
on %i: %f" % (i, cost))
```

Si `print_cost=True`, te muestra si va mejorando.

✅ ¿Qué devuelve al final?

🧪 El código de abajo (qué hace)

```
params = {"w": w,"b": b}
grads = {"dw": dw,"db": db}
return params, grads, costs
```

- `params` → los parámetros ya aprendidos (actualizados)
- `grads` → el último gradiente calculado
- `costs` → lista de costos guardados

```
params, grads, costs = opti
mize(w, b, X, Y, num_iterat
ions=100, learning_rate=0.0
09)
```

Eso significa:

- entrenar **100 veces**
- dando pasos de tamaño **0.009**

Luego imprime:

- `w` y `b` finales (ya no serán 0)
- `dw` y `db` finales (última corrección)

1) Inicializa `w` y `b` en cero

```
w, b = initialize_with_zero
s(X_train.shape[0])
```

- `X_train.shape[0]` es el número de "características" (pixeles aplanados).
- Si tus imágenes son 64×64×3, esto vale 12288.

📌 Aquí el modelo todavía no sabe nada. Solo tiene:

- `w` = ceros
- `b` = 0

2) Entrena con descenso de gradiente (aprende)

```
parameters, grads, costs =
optimize(w, b, X_train, Y_t
rain, num_iterations, learn
ing_rate, print_cost)
```

Aquí pasa lo importante:

- Se repite muchas veces (`num_iterations`):
  - el modelo predice
  - calcula error (cost)
  - calcula cómo corregirse (dw, db)
  - actualiza `w` y `b`

✅ Al final, devuelve:

- `parameters` (los `w` y `b` ya aprendidos)
- `grads` (últimos gradientes)
- `costs` (lista del costo cada 100 iteraciones)

3) Recupera los parámetros aprendidos

```
w = parameters["w"]
b = parameters["b"]
```

Ahora `w` y `b` **ya no son 0** (o casi nunca).

4) Hace predicciones en train y test

```
Y_prediction_test = predict
(w, b, X_test)
Y_prediction_train = predic
t(w, b, X_train)
```

Esto es como decir:

- “Con lo que ya aprendí...”
- “¿Qué digo para cada imagen?”
- devuelve 0 o 1:

✅ `Y_prediction_train` → predicciones sobre entrenamiento

✅ `Y_prediction_test` → predicciones sobre prueba

5) Calcula “accuracy” (porcentaje de aciertos)

```
print("train accuracy: {}
%".format(100 - np.mean(np.
abs(Y_prediction_train - Y_
train)) *100))print("test a
ccuracy: {} %".format(100 -
np.mean(np.abs(Y_prediction
_test - Y_test)) *100))
```

6) Devuelve un diccionario con todo

```
d = {"costs": costs,"Y_pred
iction_test": Y_prediction_
test,"Y_prediction_train":
Y_prediction_train,"w":
w,"b": b,"learning_rate": l
earning_rate,"num_iteration
s": num_iterations}return d
```

Esto hace algo simple:

- compara predicción vs etiqueta real
- si son iguales → bien (error 0)
- si son diferentes → mal (error 1)
- saca el promedio de errores
- lo convierte en porcentaje de aciertos

📌 Si sale:

- train ≈ 100% → aprendió muy bien el entrenamiento
- test ≈ 68% → en fotos nuevas, falla bastante

Esto es para que después puedas:

- graficar `costs`
- revisar predicciones
- reutilizar `w` y `b`

---

1) Lista de tasas de aprendizaje a probar

```
learning_rates = [0.01,0.00
1,0.0001]
```

Aquí dicen: “vamos a entrenar el mismo modelo 3 veces”, cambiando solo α.

2) Entrenar un modelo por cada learning rate

```
models = {}for iin learning
_rates:print ("learning rat
e is: " +str(i))
    models[str(i)] = model
(train_set_x, train_set_y,
test_set_x, test_set_y,

num_iterations =1500, learn
ing_rate = i, print_cost =F
alse)print ('\n' +"-------
---------------------------
--------------------"
+'\n')
```

Qué pasa aquí, sencillo:

- `models = {}`: un diccionario “cajita” para guardar resultados.

- Para cada `i` (0.01, 0.001, 0.0001):
  1. imprime cuál está usando
  2. llama a `model(...)` y **entrena 1500 iteraciones**
  3. guarda el resultado en `models` con clave `"0.01"`, `"0.001"`, etc.

✅ Al final `models["0.01"]` contiene cosas como:

- `costs` (lista del costo)
- `w`, `b`
- predicciones
- learning_rate usado

3) Graficar las curvas de costo

```
for iin learning_rates:
    plt.plot(np.squeeze(mod
els[str(i)]["costs"]), labe
l=str(models[str(i)]["learn
ing_rate"]))
```

Esto dibuja una línea por cada learning rate:

- eje Y = `cost` (error)
- eje X = “pasos/iteraciones” (en realidad, **cada punto es cada 100 iteraciones**, porque en `optimize` guardabas el cost cada 100)

📌 `np.squeeze(...)` solo quita dimensiones extra para que matplotlib lo pueda dibujar bien.

4) Poner etiquetas y leyenda bonita

```
plt.ylabel('cost')
plt.xlabel('iterations')
legend = plt.legend(...)
plt.show()
```

¿Cómo interpretar la gráfica?

Vas a ver 3 líneas (una por α). Lo que buscas:

✅ **Una curva que baje rápido y de forma estable**

❌ Si sube, tiembla mucho, o se vuelve rara → learning rate demasiado grande

❌ Si baja casi nada → learning rate demasiado pequeño

En general:

- **0.01**: suele bajar rápido, pero puede volverse inestable dependiendo del problema.
- **0.001**: suele ser una opción “segura” y estable.
- **0.0001**: casi siempre baja lento.

Tip súper práctico

Una buena tasa de aprendizaje hace que:

- el `cost` disminuya claramente
- y no “rebote” demasiado