

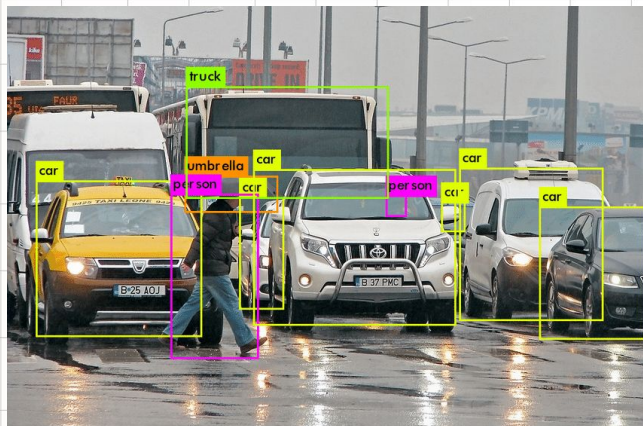


Descenso de gradiente

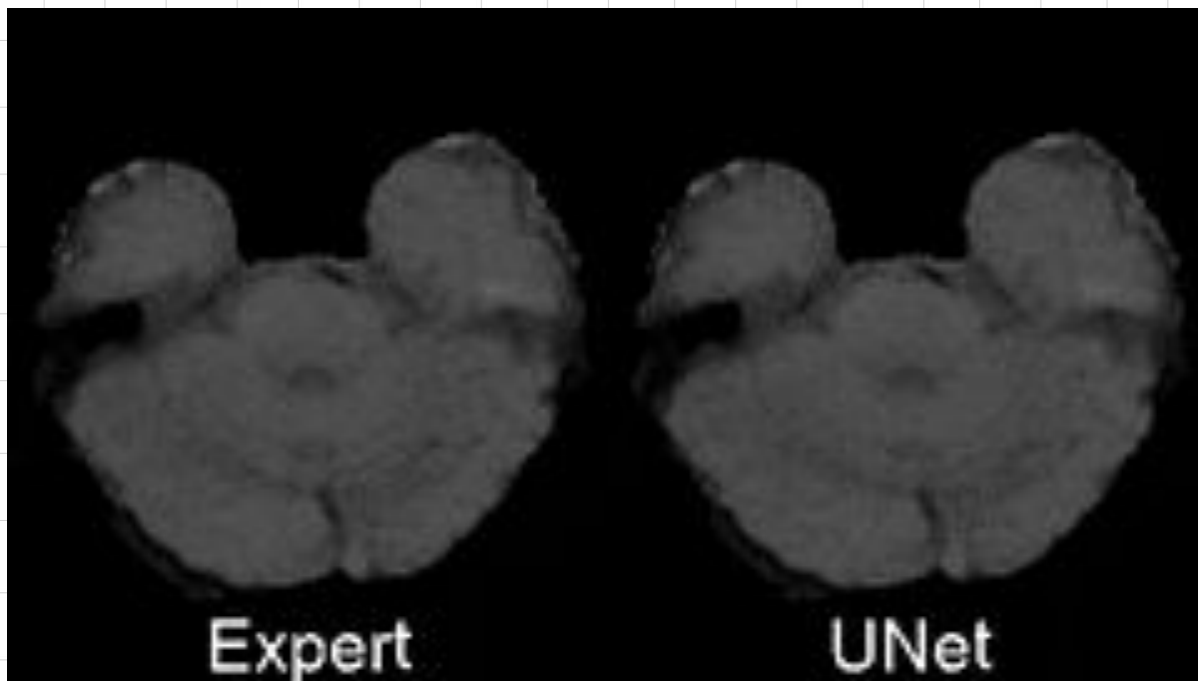
Autor: Sanchez Sauñe, Cristhian Wiki



Aplicaciones



Aplicaciones



Aplicaciones

Source A: gender, age, hair length, glasses, pose



Source B:
everything
else

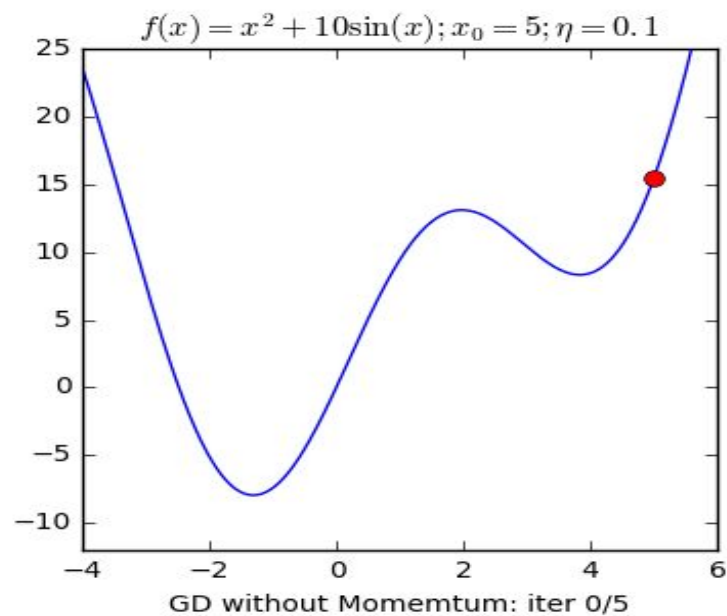


Result of combining A and B

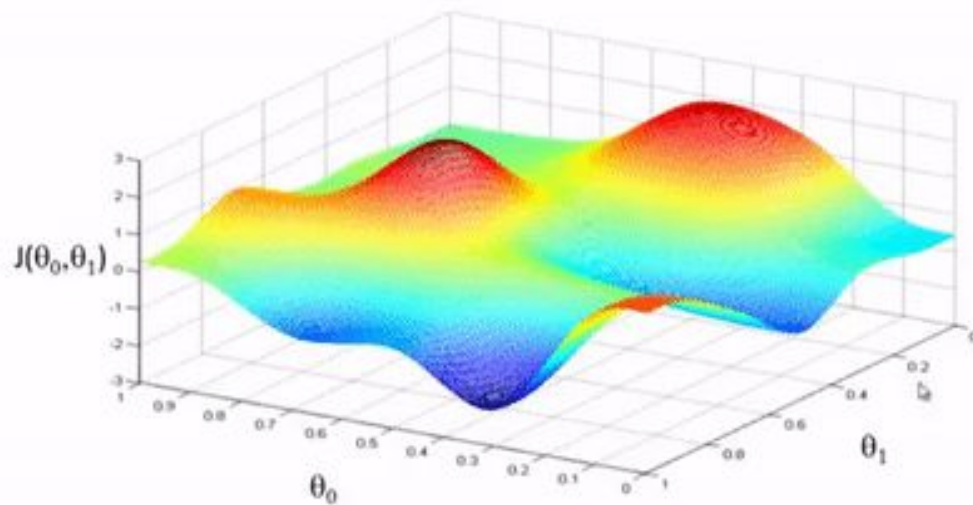
Aplicaciones



Intuición



Descenso de gradiente en acción



Un problema de optimización

Problema:

$$\min_w f(w)$$

Solución Iterativa:

$$w_{k+1} = w_k - \lambda_k \nabla f(w_k)$$

donde,

- w_{k+1} es el valor actualizado luego de k iteraciones
- w_k es el valor inicial antes de la iteración k-ésima,
- λ_k es el tamaño de paso,
- $\nabla f(w_k)$ es el gradiente de f .

Algoritmo completo

Cost Function

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^m [h_{\Theta}(x_i) - y_i]^2$$

↑ ↑
Predicted Value True Value

Gradient Descent

$$\Theta_j = \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1)$$

↑
Learning Rate

Now,

$$\begin{aligned} \frac{\partial}{\partial \Theta} J_{\Theta} &= \frac{\partial}{\partial \Theta} \frac{1}{2m} \sum_{i=1}^m [h_{\Theta}(x_i) - y]^2 \\ &= \frac{1}{m} \sum_{i=1}^m (h_{\Theta}(x_i) - y) \frac{\partial}{\partial \Theta_j} (\Theta x_i - y) \\ &= \frac{1}{m} (h_{\Theta}(x_i) - y) x_i \end{aligned}$$

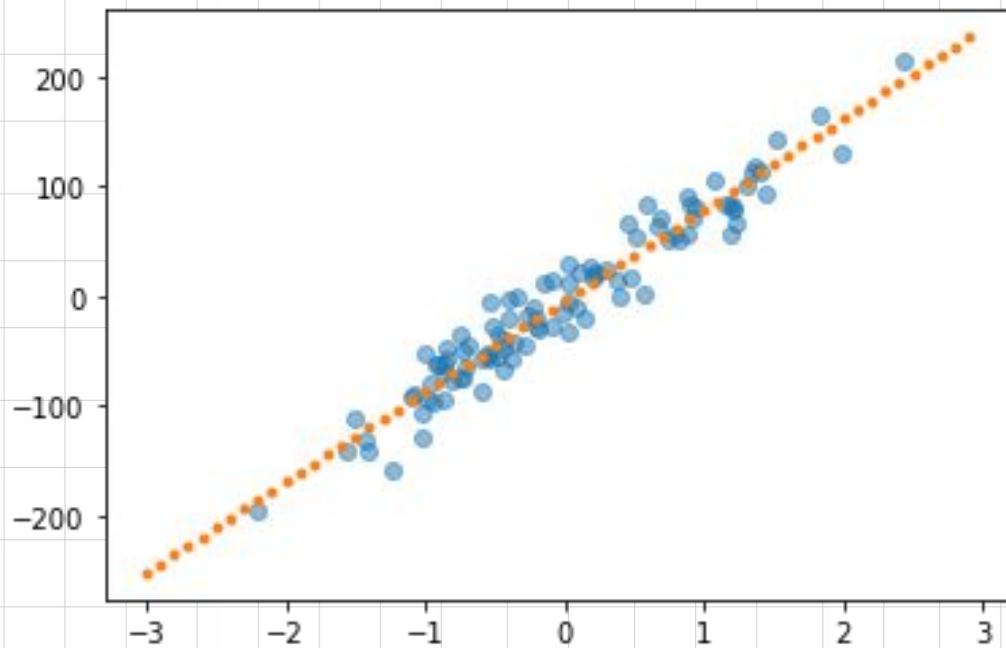
Therefore,

$$\Theta_j := \Theta_j - \frac{\alpha}{m} \sum_{i=1}^m [(h_{\Theta}(x_i) - y) x_i]$$

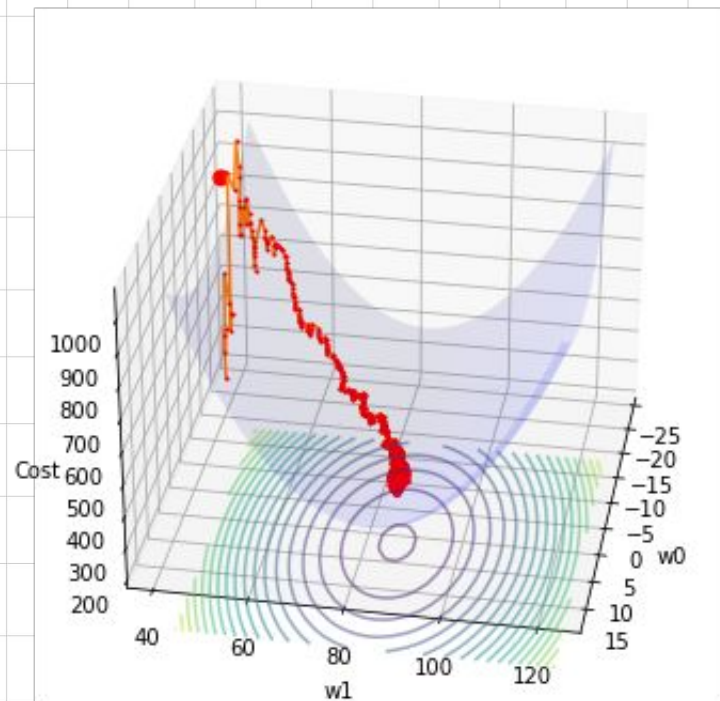
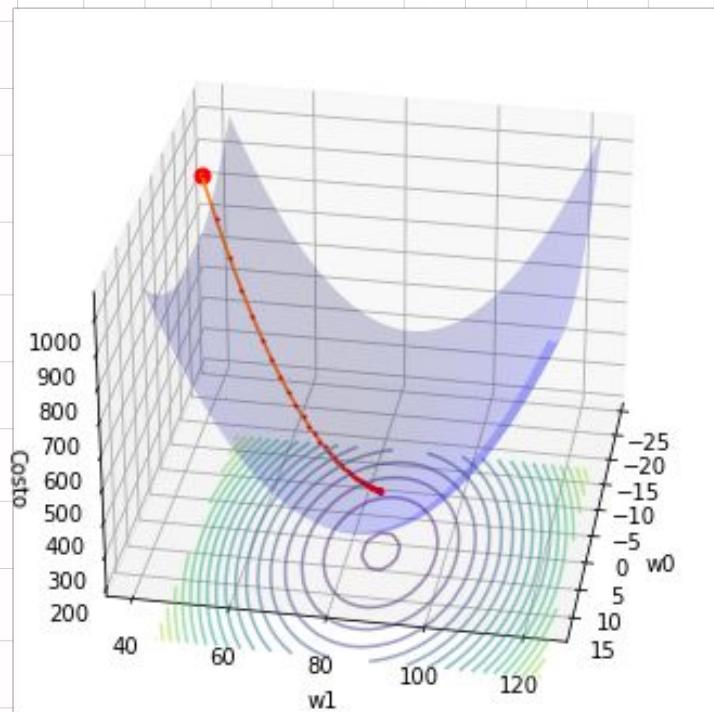
Código

```
def gradient_descent(X,y,theta,learning_rate=0.01,iterations=100):  
    '''  
    X      = Matriz de X con unidades de sesgo agregadas  
    y      = Vector de Y  
    theta=Vector de thetas np.random.randn(j,1)  
    learning_rate  
    iterations = número de iteraciones  
  
    Devuelve el vector theta final y la matriz del historial de  
    costos sobre el número de iteraciones  
    '''  
    m = len(y)  
    cost_history = np.zeros(iterations)  
    theta_history = np.zeros((iterations,2))  
    for it in range(iterations):  
  
        prediction = np.dot(X,theta)  
  
        theta = theta -(1/m)*learning_rate*( X.T.dot((prediction - y)))  
        theta_history[it,:] =theta.T  
        cost_history[it] = cal_cost(theta,X,y)  
  
    return theta, cost_history, theta_history
```

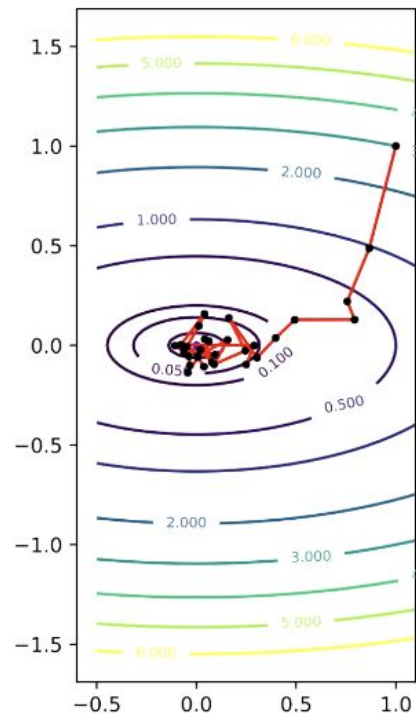
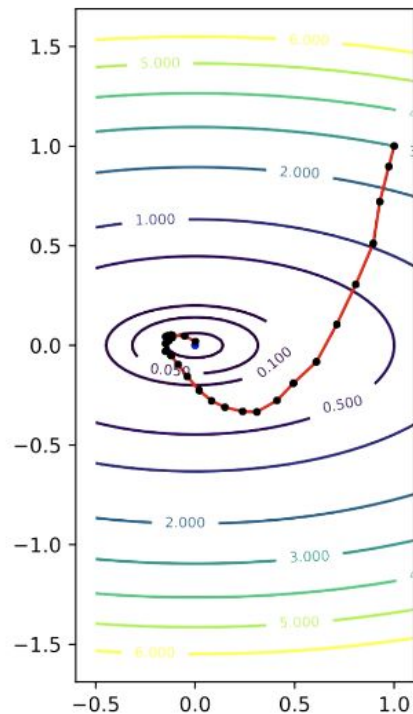
Regresión Lineal



Descenso de gradiente en la función de coste

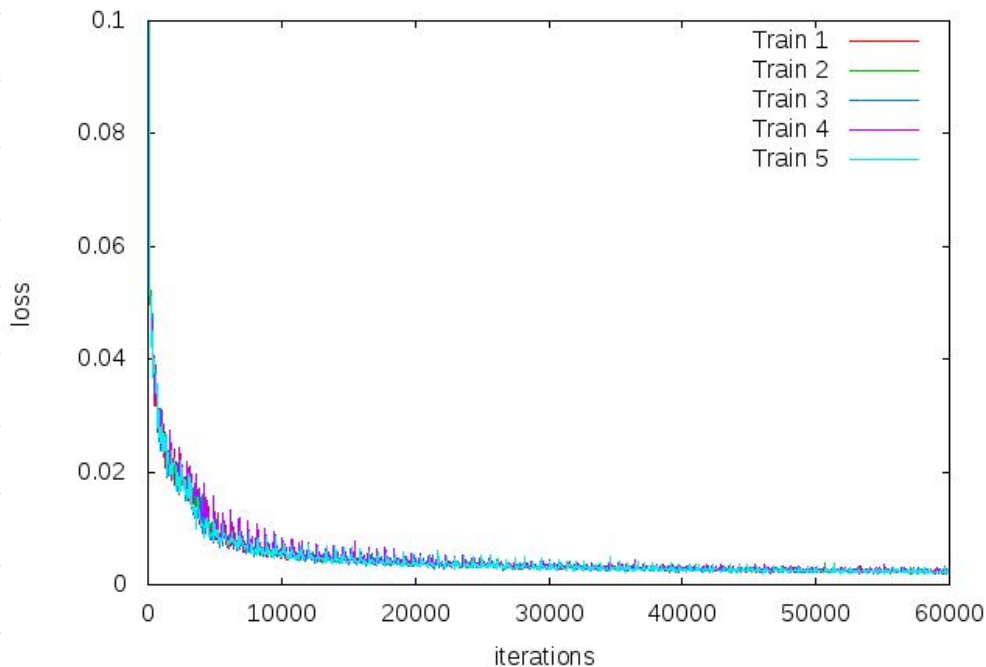


Descenso de gradiente en la función de coste



Error de entrenamiento

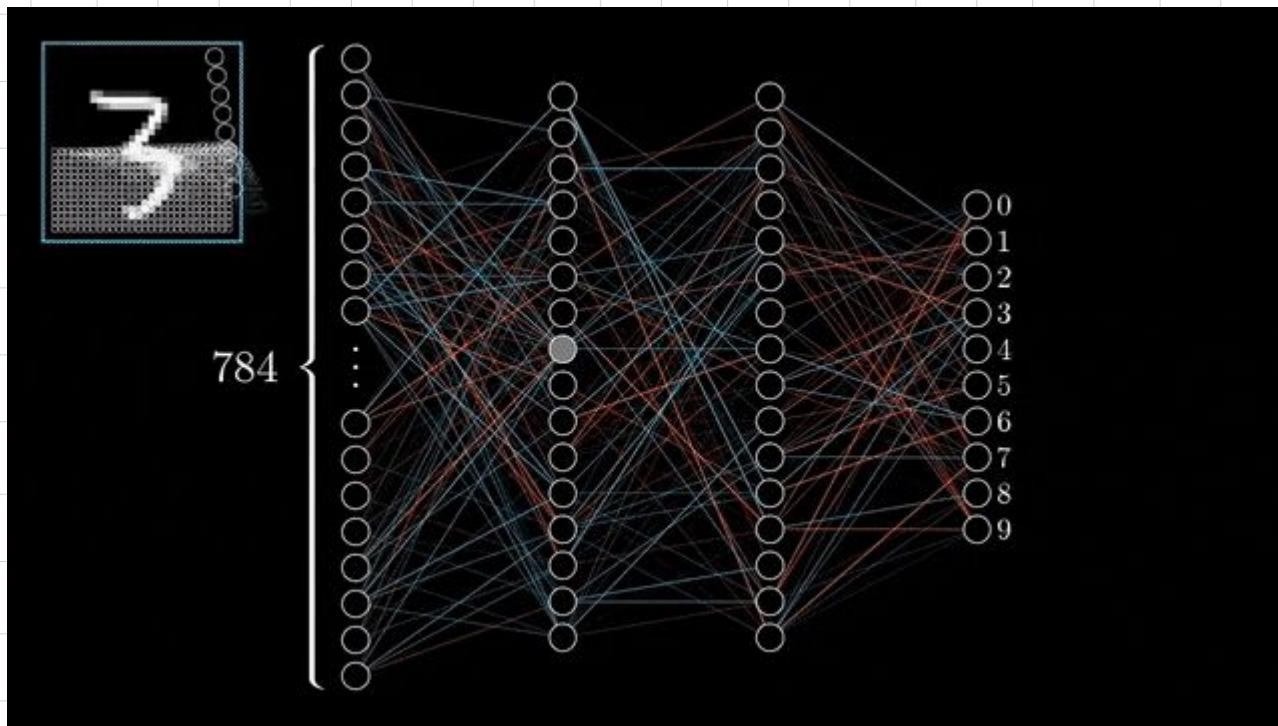
Training loss vs. iterations



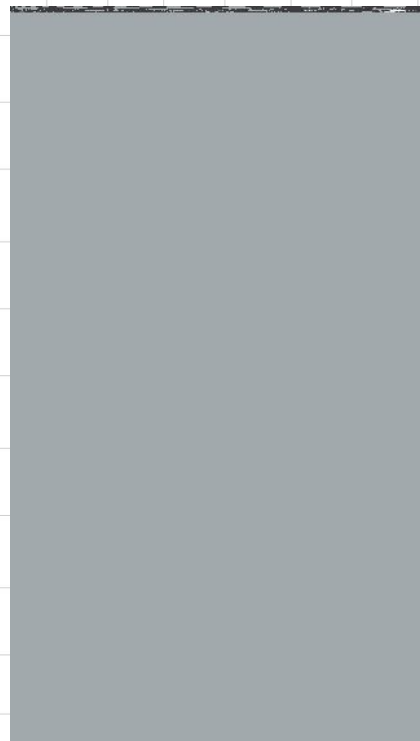
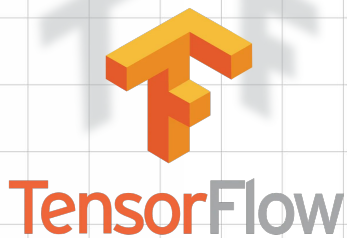
Complejidad

El costo computacional del descenso del gradiente depende del número de iteraciones que se necesitan para converger. La complejidad del descenso por gradiente es $O(kn^2)$, así que cuando N es muy grande se recomienda utilizar el descenso de gradiente en lugar de la forma cerrada de regresión lineal.

Redes Neuronales



Frameworks



Frameworks

 PyTorch



Deep Learning - DEMO



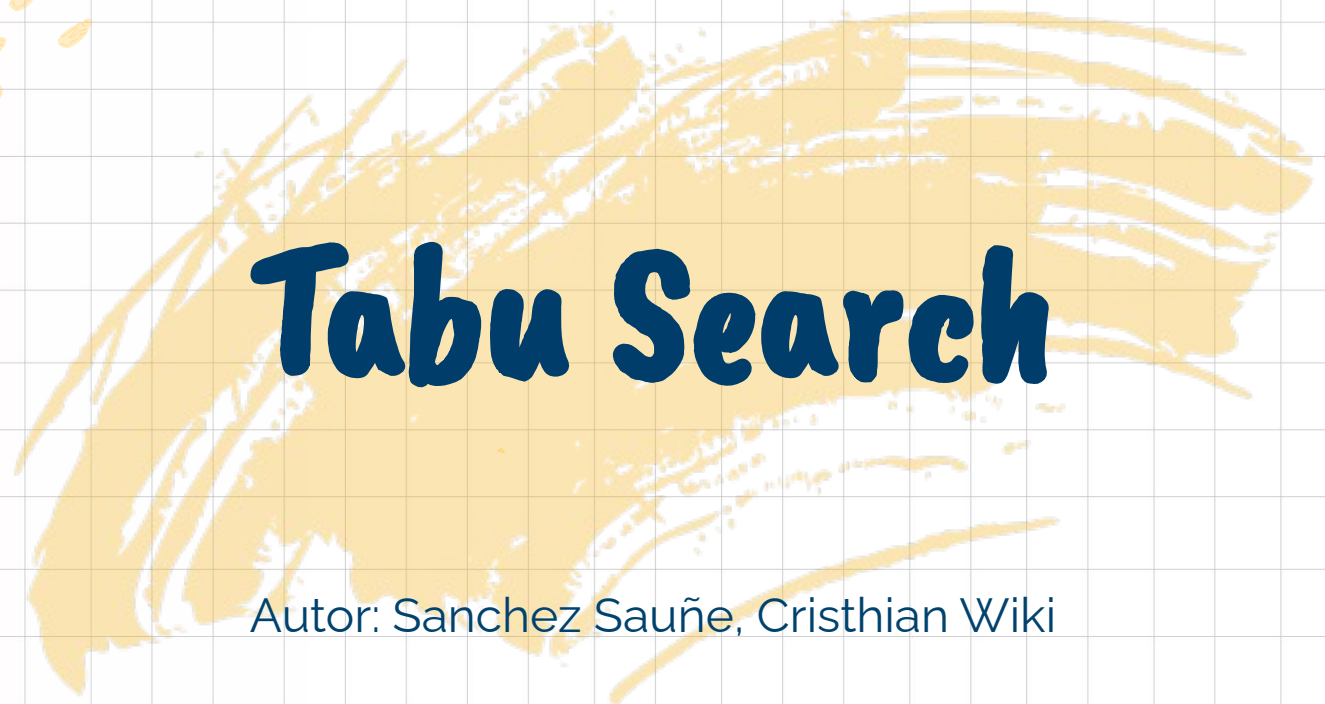
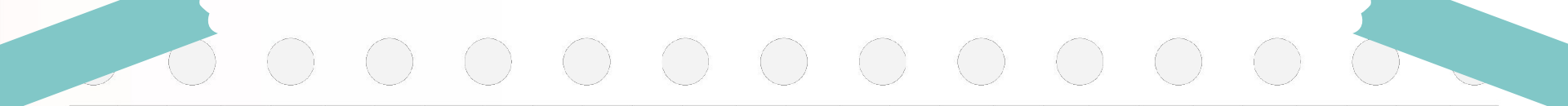
Demo UNI



Original



Detectron2



Tabu Search

Autor: Sanchez Sauñe, Cristhian Wiki



¿Qué es Tabu Search?

Tabu Search es un meta-heurística de uso común utilizado para optimizar los parámetros de un modelo.

Una metaheurística es una estrategia general que se utiliza para guiar y controlar la heurística real.

Tabu Search se considera a menudo como la integración de estructuras de memoria en estrategias de búsqueda local. Como la búsqueda local tiene muchas limitaciones, Tabu Search está diseñado para combatir muchos de esos problemas.

Búsqueda Tabu (TS)

La idea básica de Tabu Search es penalizar los movimientos que llevan la solución a espacios de búsqueda visitados anteriormente (también conocido como tabu). Tabu Search, sin embargo, acepta determinísticamente soluciones que no mejoran con el fin de evitar quedarse atascado en los mínimos locales.

Memoria a corto plazo frente a memoria a largo plazo

La **memoria a corto plazo** se basa en la actualidad de la ocurrencia y se usa para evitar que el algoritmo de búsqueda vuelva a visitar las soluciones visitadas anteriormente y también se puede usar para regresar a los componentes buenos para localizar e intensificar una búsqueda. Esto se logra mediante la Lista tabú y también se conoce como **intensificación**.

La **memoria a largo plazo** se basa en la frecuencia de aparición y se utiliza para diversidad de la búsqueda y explorar áreas no vistas del espacio de búsqueda evitando áreas exploradas. Esto se logra mediante la memoria de frecuencia y también se conoce como **diversificación**.

Lista Tabu

La lista Tabu es la ***piedra angular*** de la utilización de la memoria a corto plazo. Esta lista almacena una cantidad fija de movimientos realizados recientemente. En algunas implementaciones, se utilizan soluciones completas en lugar de los movimientos utilizados, pero esto no es ideal si las soluciones completas son muy grandes debido a limitaciones de espacio. Algunos ejemplos de estos movimientos son:

- Intercambiar nodos en un grafo/tour
- Alternando un poco entre 0 y 1
- Insertar o eliminar aristas en un grafo

Tenencia Tabu

La tenencia de Tabu es el número de iteraciones que un movimiento permanece en la lista Tabu. Los movimientos que están en la lista Tabu son movimientos que no se pueden hacer de nuevo. Hay dos maneras de implementar la tenencia Tabu (T):

- **Estático:** elija T para que sea una constante (a menudo \sqrt{T})
- **Dinámico:** elija T aleatoriamente entre algunos T_{\min} y T_{\max}

Criterios de aspiración

Esta es una parte opcional de Tabu Search. Los movimientos o soluciones que forman parte de los Criterios de Aspiración cancelan el Tabu y el movimiento se puede hacer incluso si está en la Lista Tabu. Esto también se puede utilizar para evitar el estancamiento en los casos en que todos los movimientos posibles están prohibidos por la Lista Tabu.

Algunos ejemplos de criterios de aspiración son:

- si la nueva solución es mejor que la mejor solución actual, entonces la nueva solución se utiliza incluso si está en la lista Tabu.
- estableciendo la tenencia tabu para que sea un valor más pequeño

Memoria de frecuencia

Esta memoria contiene el número total de iteraciones que cada solución se eligió desde el principio de la búsqueda. Las soluciones que se visitaron más son menos propensas a ser recogidas de nuevo y promoverán soluciones más diversas. Existen dos enfoques principales para diversificar:

Reiniciar diversificación: permite que los componentes que rara vez parecen, estar en la solución actual reiniciando la búsqueda desde estos puntos

Diversificación continua: sesga la evaluación de posibles movimientos con la frecuencia de estos movimientos. Los movimientos que no aparecen con la frecuencia tendrán una probabilidad más alta de realizarse.

Algoritmo

Paso 1: Primero comenzamos con una solución inicial s_0 . Esto puede ser cualquier solución que se ajuste a los criterios para una solución aceptable.

Paso 2: Generar un conjunto de soluciones vecinas a la solución actual etiquetada $N(s)$. De este conjunto de soluciones, las soluciones que se encuentran en la lista Tabu se eliminan con la excepción de las soluciones que se ajustan a los criterios de aspiración. Este nuevo conjunto de resultados es el nuevo(s) $N(s)$.

$$s' \in N(s) = \{N(s) - T(s)\} + A(s)$$

Paso 3: Elija la mejor solución de $N(s)$ y etiquete esta nueva solución s' . Si la solución s' es mejor que la mejor solución actual, actualice la mejor solución actual. Después, independientemente de si s' es mejor que s , actualizamos s para ser s' .

Algoritmo

$$s' \in N(s) = \{N(s) - T(s)\} + A(s)$$

Paso 4: Actualiza las T de la lista Tabu eliminando todos los movimientos que hayan expirado más allá de la tenencia de Tabu y agregue el nuevo movimiento **s'** a la lista tabu. Además, actualice el conjunto de soluciones que se ajustan a los criterios de aspiración **A**. Si se utiliza memoria de frecuencia, incrementa también el contador de memoria de frecuencia con la nueva solución.

Paso 5: Si se cumplen los criterios de terminación, la búsqueda se detiene o de lo contrario se moverá a la siguiente iteración. Los criterios de terminación dependen del problema en cuestión, pero algunos ejemplos posibles son:

- un número máximo de iteraciones
- si la mejor solución encontrada es mejor que algún umbral

Código

```
class TabuSearch:
    def __init__(self, initialSolution, solutionEvaluator, neighborOperator,
                  aspirationCriteria, acceptableScoreThreshold, tabuTenure):
        self.currSolution = initialSolution
        self.bestSolution = initialSolution
        self.evaluate = solutionEvaluator
        self.aspirationCriteria = aspirationCriteria
        self.neighborOperator = neighborOperator
        self.acceptableScoreThreshold = acceptableScoreThreshold
        self.tabuTenure = tabuTenure

    def isTerminationCriteriaMet(self):
        # can add more termination criteria
        return self.evaluate(self.bestSolution) < self.acceptableScoreThreshold \
            or self.neighborOperator(self.currSolution) == 0
```

```
def run(self):
    tabuList = {}

    while not self.isTerminationCriteriaMet():
        # get all of the neighbors
        neighbors = self.neighborOperator(self.currSolution)
        # find all tabuSolutions other than those
        # that fit the aspiration criteria
        tabuSolutions = tabuList.keys()
        # find all neighbors that are not part of the Tabu list
        neighbors = filter(lambda n: self.aspirationCriteria(n), neighbors)
        # pick the best neighbor solution
        newSolution = sorted(neighbors, key=lambda n: self.evaluate(n))[0]
        # get the cost between the two solutions
        cost = self.evaluate(self.solution) - self.evaluate(newSolution)
        # if the new solution is better,
        # update the best solution with the new solution
        if cost >= 0:
            self.bestSolution = newSolution
        # update the current solution with the new solution
        self.currSolution = newSolution

        # decrement the Tabu Tenure of all tabu list solutions
        for sol in tabuList:
            tabuList[sol] -= 1
            if tabuList[sol] == 0:
                del tabuList[sol]
        # add new solution to the Tabu list
        tabuList[newSolution] = self.tabuTenure

    # return best solution found
    return self.bestSolution
```

Ventajas y desventajas de TS

Ventajas

- Puede escapar de los óptimos locales seleccionando soluciones que no mejoran
- La lista Tabu se puede utilizar para evitar ciclos y volver a soluciones antiguas
- Se puede aplicar tanto a soluciones discretas como continuas

Desventajas

- El número de iteraciones puede ser muy alto
- Hay una gran cantidad de parámetros sintonizables en este algoritmo

Ejemplos de problemas para resolver con TS

- Problema de N-Queens
- Problema de vendedor viajero (TSP)
- Árbol de expansión mínimo (MST)
- Problemas de asignación
- Enrutamiento de vehículos
- Secuenciación de ADN