Resumen del capítulo: Entrenamiento del descenso de gradiente

Descenso de gradiente para regresión lineal

Recuerda la tarea de entrenamiento de regresión lineal:

$$w = \arg\min_{w} ECM(Xw, y)$$

Vamos a entrenar un modelo usando descenso de gradiente. Pero primero, escribe la función de pérdida en forma de vector para encontrar su gradiente. Expresa el *ECM* como un producto escalar de la diferencia de vectores:

$$ECM(y, a) = \frac{1}{n}\sum_{i=1}^{n}(a_i - y_i)^2 = \frac{1}{n}(a - y,\ a - y)$$

donde *y* es el vector de respuesta correcta y *a* es el vector de predicción.

Convierte el producto escalar en un producto matricial:

$$(a - y,\ a - y) = (a - y)^T(a - y)$$

Después de transponer el vector, es decir, convertirlo de una columna a una fila, podemos multiplicarlo por otro vector.

Combina las fórmulas de *ECM* y regresión lineal:

$$ECM(X w, y) = \frac{1}{n} (X w - y)^T (X w - y)$$

Encuentra la función gradiente para el vector de parámetros *w*. Los gradientes de las funciones con valores vectoriales se calculan de manera similar a las derivadas. Por ejemplo, cuando se trabaja con números, la derivada parcial de $(xw - y)^2$ para el vector de parámetros *w* es igual a *2x(xw - y).* Cuando se trabaja con vectores, solo permanece el factor de *w* del primer paréntesis (X^T):

$$\nabla ECM(X w, y) = \frac{2}{n} X^T (X w - y)$$

Descenso de gradiente estocástico

Podemos calcular el gradiente usando pequeñas partes del conjunto de entrenamiento. Estas piezas se conocen como **minilotes** o **lotes**. Para que el algoritmo "vea" todo el conjunto de entrenamiento, deben cambiarse sus lotes en cada iteración de forma aleatoria. Aquí necesitamos el **descenso de gradiente estocástico en minilotes** o **descenso de gradiente estocástico, DGE.**

Para obtener lotes, necesitamos barajar todos los datos del conjunto de entrenamiento y dividirlo en partes. Un lote debe contener un promedio de 100-200 observaciones (el **tamaño del lote)**. Cuando el algoritmo *DGE* ha pasado por todos los lotes una vez, significa que una **época** ha terminado. El número de épocas depende del tamaño del

conjunto de entrenamiento. Pueden ser una o dos si el conjunto es grande o varias docenas si el conjunto es pequeño. El número de lotes es igual al número de iteraciones para completar una época.

Así es como funciona el algoritmo *DGE*:

1. Hiperparámetros de entrada: tamaño del lote, número de épocas y tamaño del paso.
2. Define los valores iniciales de los pesos del modelo.
3. Divide el conjunto de entrenamiento en lotes para cada época.
4. Para cada lote:

   4.1. Calcula el gradiente de la función de pérdida;

   4.2. Actualiza los pesos del modelo (agrega el gradiente negativo multiplicado por el tamaño del paso a los pesos actuales).
5. El algoritmo devuelve los pesos finales del modelo.

Encontremos la complejidad computacional del *DGE* con las siguientes definiciones:

- *n* — el número de observaciones en todo el conjunto de entrenamiento;
- *b* — el tamaño del lote;
- *p* — el número de funciones

DGE en Python

Vamos a aprender a pasar hiperparámetros a un modelo. Necesitamos declarar la clase del modelo y crear el método "**inicializador de clase**" (`__init__`):

```
class SGDLinearRegression:
    def __init__(self):
        ...
```

Agrega un hiperparámetro `step_size` al inicializador de clase:

```
class SGDLinearRegression:
    def __init__(self, step_size):
        ...
```

Ahora podemos pasar el tamaño del paso al modelo al crear una clase:

```
# puedes elegir el tamaño del paso de forma arbitraria
model = SGDLinearRegression(0.01)
```

Guarda el tamaño del paso como un atributo:

```
class SGDLinearRegression:
    def __init__(self, step_size):
        self.step_size = step_size
```

Aquí está la implementación completa del algoritmo DGE:

```
class SGDLinearRegression:
    def __init__(self, step_size, epochs, batch_size):
        self.step_size = step_size
        self.epochs = epochs
        self.batch_size = batch_size

    def fit(self, train_features, train_target):
        X = np.concatenate((np.ones((train_features.shape[0], 1)), train_features), axis=
1)
        y = train_target
        w = np.zeros(X.shape[1])

        for _ in range(self.epochs):
            batches_count = X.shape[0] // self.batch_size
            for i in range(batches_count):
                begin = i * self.batch_size
                end = (i + 1) * self.batch_size
                X_batch = X[begin:end, :]
                y_batch = y[begin:end]

                gradient = 2 * X_batch.T.dot(X_batch.dot(w) - y_batch) / X_batch.shape[0]

                w -= self.step_size * gradient

        self.w = w[1:]
```

```
        self.w0 = w[0]

    def predict(self, test_features):
        return test_features.dot(self.w) + self.w0
```

Regularización de regresión lineal

Vamos a modificar la función de pérdida para eliminar el sobreajuste. Para reducir el sobreajuste, podemos usar la **regularización**, que "refina" el modelo si los valores de los parámetros complican la operación del algoritmo. Para un modelo de regresión lineal, la regularización implica la limitación de pesos. Cuanto más bajos sean los valores de peso, más fácil será entrenar el algoritmo. Para averiguar qué tan grandes son los pesos, calculamos la distancia entre el vector de peso y el vector que consta de ceros. Por ejemplo, la distancia euclidiana $d_2(w, 0)$ es igual al producto escalar de los pesos solos: (w, w).

Para limitar los valores de peso, incluye el producto escalar de los pesos en la fórmula de la función de pérdida:

$$L(w) = ECM(Xw, y) + \lambda(w, w)$$

La derivada (w, w) es igual a $2w$. Calcula el gradiente de la función de pérdida:

$$\nabla L(w) = \frac{2}{n} X^T (Xw - y) + 2w$$

Para controlar la magnitud de la regularización, agrega el **peso de regularización** a la fórmula de la función de pérdida. Se denota como λ.

$$L(w) = ECM(Xw, y) + \lambda(w, w)$$

El peso de regularización también se agrega a la fórmula de cálculo de gradiente:

$$\nabla L(w) = \frac{2}{n} X^T (Xw - y) + 2\lambda w$$

Cuando usamos la distancia euclidiana para la regularización del peso, dicha regresión lineal se denomina **regresión de cresta**.

Así es como se verá el DGE si consideramos la regularización:

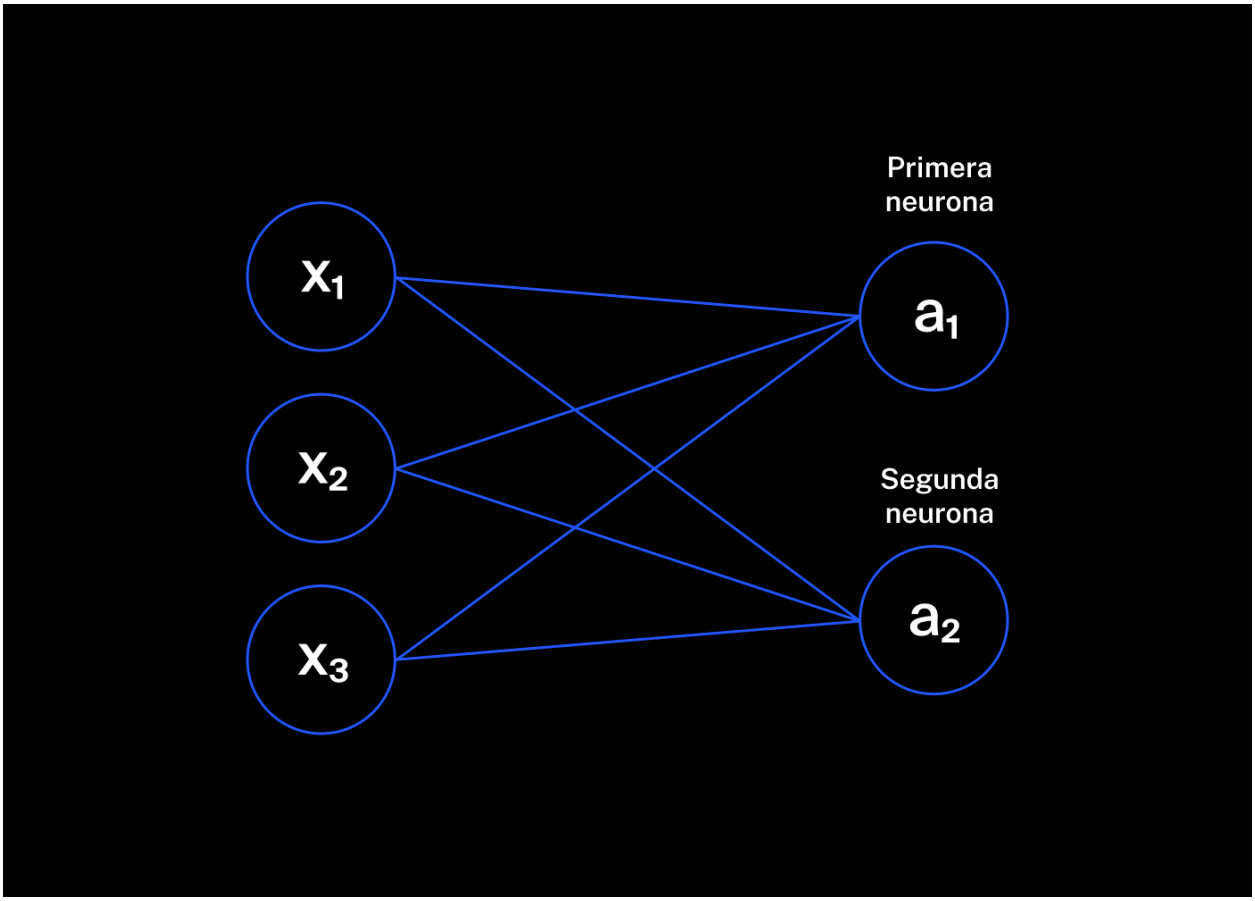
```
class SGDLinearRegression:
    def __init__(self, step_size, epochs, batch_size, reg_weight):
        self.step_size = step_size
        self.epochs = epochs
        self.batch_size = batch_size
        self.reg_weight = reg_weight

    def fit(self, train_features, train_target):
        X = np.concatenate((np.ones((train_features.shape[0], 1)), train_features), axis=
1)
        y = train_target
        w = np.zeros(X.shape[1])

        for _ in range(self.epochs):
            batches_count = X.shape[0] // self.batch_size
            for i in range(batches_count):
                begin = i * self.batch_size
                end = (i + 1) * self.batch_size
                X_batch = X[begin:end, :]
                y_batch = y[begin:end]

                gradient = 2 * X_batch.T.dot(X_batch.dot(w) - y_batch) / X_batch.shape[0]
                reg = 2 * w.copy()
                reg[0] = 0
                gradient += self.reg_weight * reg
```

```
                w -= self.step_size * gradient

        self.w = w[1:]
        self.w0 = w[0]

    def predict(self, test_features):
        return test_features.dot(self.w) + self.w0
```

Fundamentos de redes neuronales

Una **red neuronal** es un modelo que consta de muchos modelos simples (por ejemplo, modelos de regresión lineal). El nombre proviene de la biología: una red neuronal artificial utiliza el principio de una operación de red de células neuronales donde las neuronas construyen relaciones complejas entre los datos de entrada y salida.

Este es un ejemplo de una red neuronal con tres entradas x_1, x_2, x_3 y dos salidas a_1, a_2:



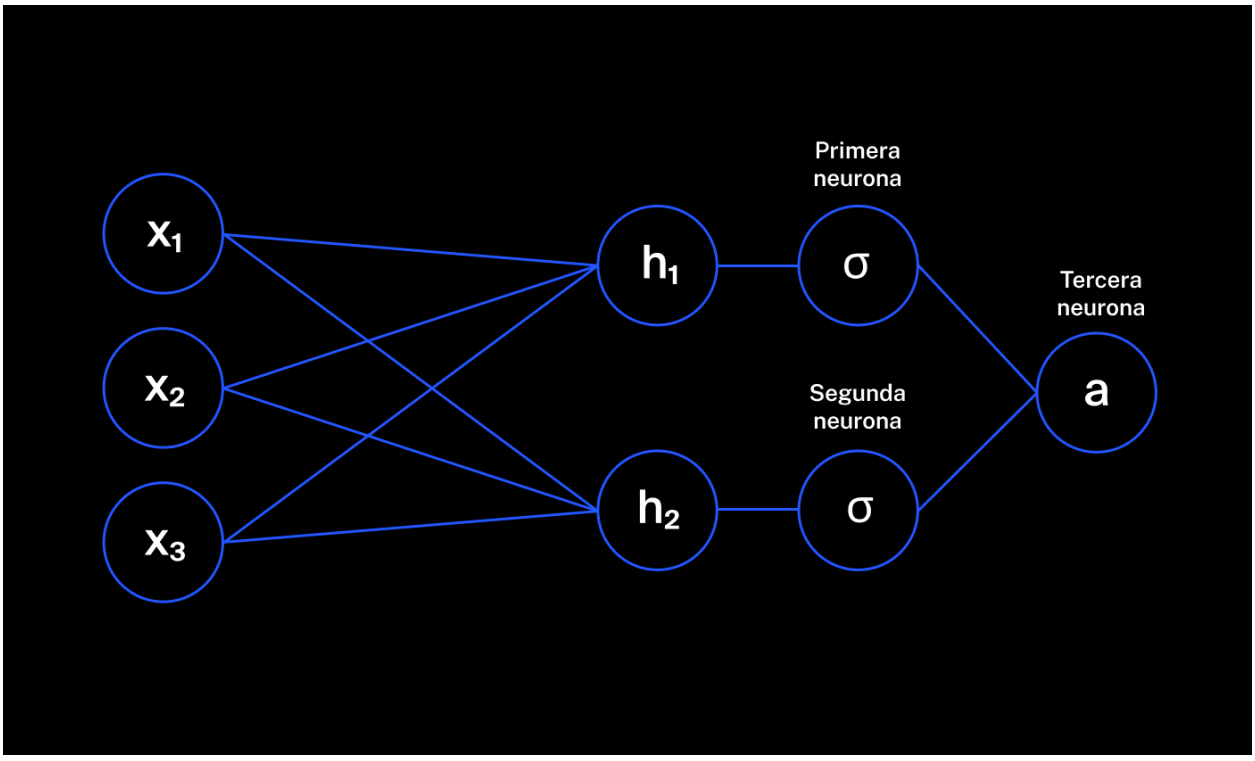
El valor de cada salida, o neurona, se calcula de la misma manera que una predicción de regresión lineal.

$$a_1 = xw_1$$

$$a_2 = xw_2$$

Cada valor de salida tiene sus propios pesos (w_1 y w_2).

Aquí hay otro ejemplo. La red tiene tres entradas x_1, x_2, x_3, dos variables ocultas h_1 y h_2, y una salida a.



Los valores h_1 y h_2 se pasan a la función logística $\sigma(x)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

donde e es el número de Euler (aproximadamente 2.718281828).

La función logística en una red neuronal se denomina **función de activación**. Se incluye en la neurona después de multiplicar los valores de entrada por los pesos, cuando las salidas de la neurona se convierten en entradas para otras neuronas. De esta manera, podemos describir dependencias más complejas.

Cada variable oculta *(h_1, h_2)* es igual al valor de entrada multiplicado por un peso:

$$h_1 = xw_1$$

$$h_2 = xw_2$$

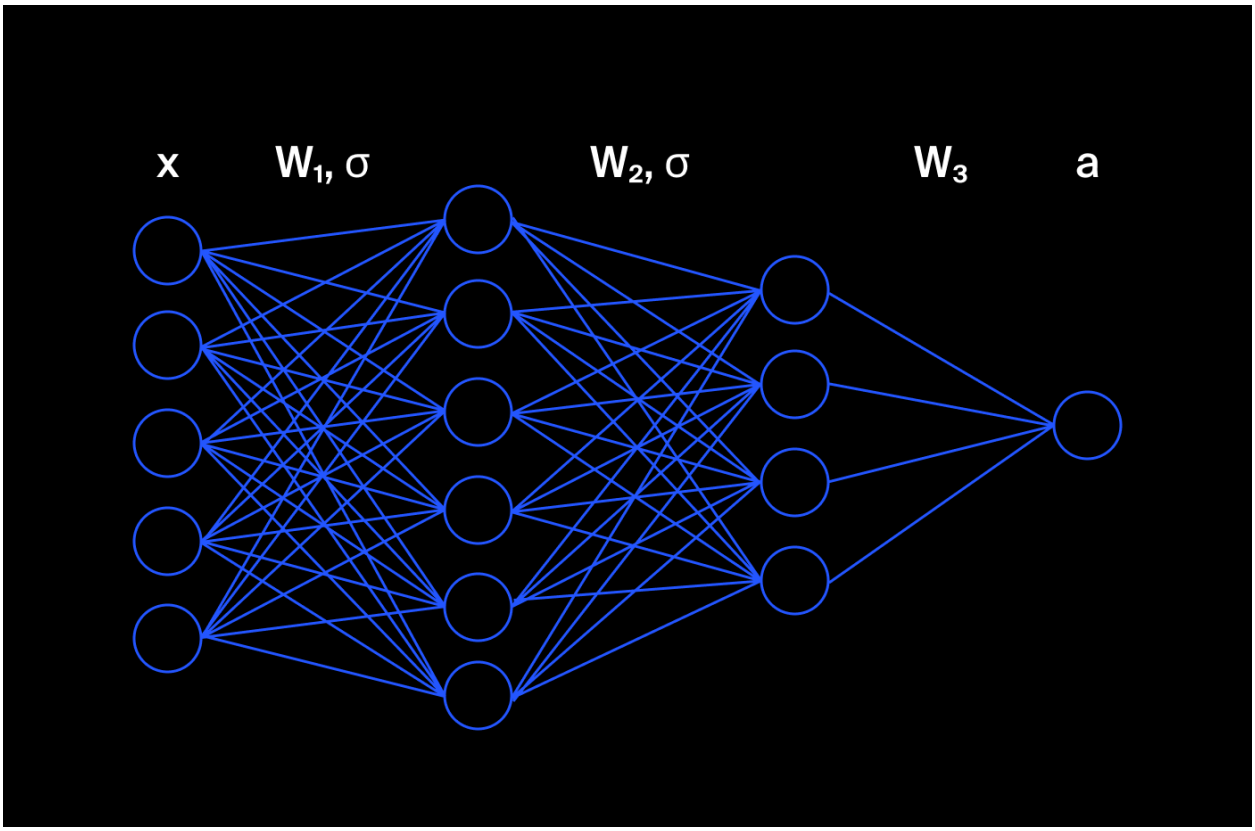
Por comodidad, expresamos las variables ocultas *h_1* y *h_2* como vector *h*. Aquí está la fórmula para calcular la predicción de la red neuronal:

$$a = \sigma(h)w_3$$

Por comodidad, expresamos las variables ocultas *h_1* y *h_2* como vector *h*. Aquí está la fórmula para calcular la predicción de la red neuronal:

$$a = \sigma(xW)w_3$$

Si ponemos los pesos de varias neuronas en matrices, podemos obtener una red aún más compleja, por ejemplo:



donde:

- *x*: vector de entrada con dimensión *p* (número de características)
- *W₁* — matriz con dimensión *p x m*
- *W₂* — matriz con dimensión *m x k*
- *W₃* — matriz con dimensión *k x 1*
- a — predicción del modelo (número único)

Cuando una red neuronal de este tipo calcula una predicción, realiza todas las operaciones de forma secuencial:

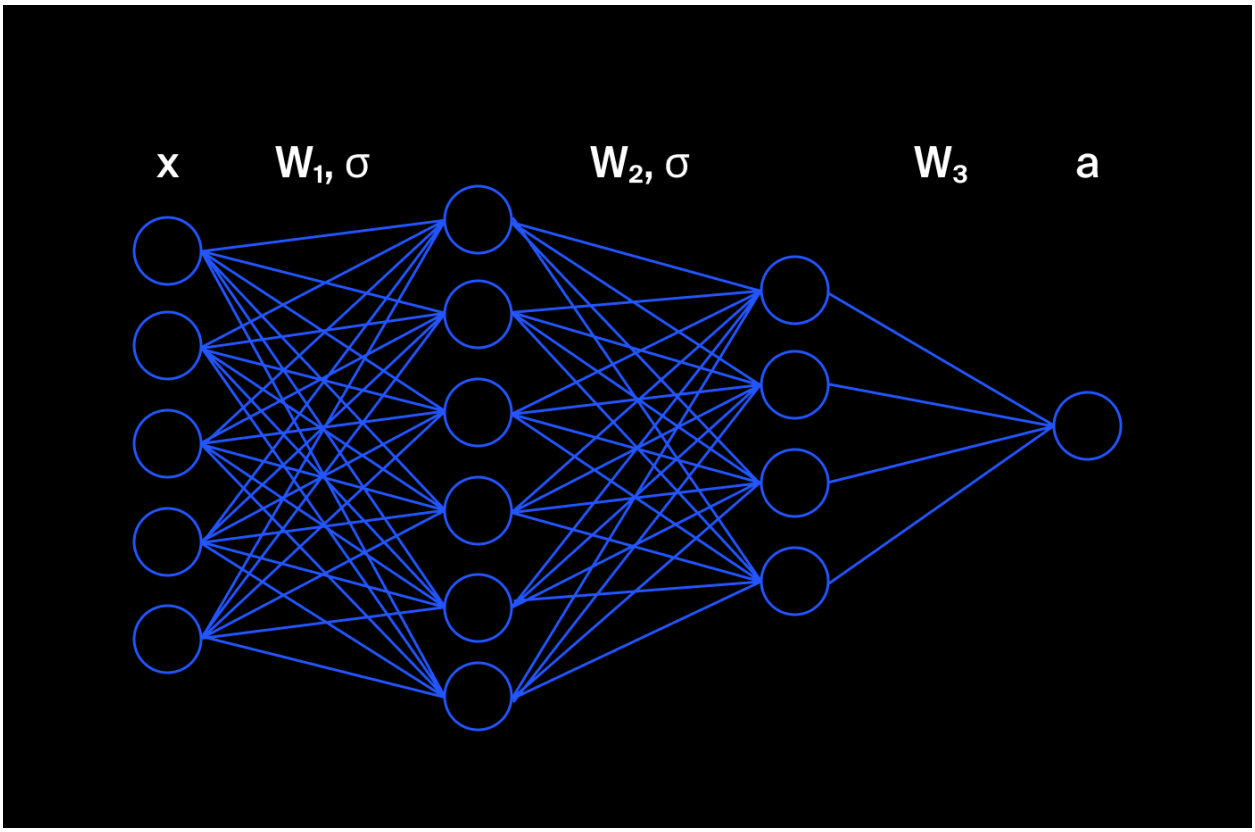
$$a = \sigma(\sigma(xW_1)W_2)W_3$$

Entrenamiento de redes neuronales

Para entrenar una red neuronal, necesitamos establecer el objetivo de entrenamiento. Cualquier red neuronal se puede escribir como una función a partir de su vector de entrada y sus parámetros. Definamos lo siguiente:

- *X* — características del conjunto de entrenamiento
- *P* — conjunto de todos los parámetros de la red neuronal
- *N(X, P)* — función de red neuronal

Tomemos esta red neuronal:



Los parámetros de la red neuronal son pesos en las neuronas:

$$P = W_1, W_2, W_3$$

Aquí está la función de red neuronal:

$$N(X, P) = \sigma(\sigma(XW_1)W_2)W_3$$

También vamos a definir:

- *y* — respuestas del conjunto de entrenamiento
- *L(a, y)* — función de pérdida (por ejemplo, *ECM)*

Entonces podemos declarar el objetivo del entrenamiento de redes neuronales de la siguiente manera:

$$\min_{P} L(N(X, P), y)$$

El mínimo de esta función también se puede encontrar usando el algoritmo *DGE*.

El algoritmo de aprendizaje de la red neuronal es el mismo que el algoritmo DGE para la regresión lineal. Solo calculamos el gradiente de la red neuronal, en lugar del gradiente para la regresión lineal.

$$\nabla L(N(X, P), y)$$