ADALINE

¿Qué aprenderemos hoy?

- **Modelo Adaline**
- **Descenso de Gradiente y Descenso de Gradiente Estocástico**
- **Implementación en Python**

Neuronas Lineales Adaptativas

Las dos diferencias claves entre la regla de **Adaline** (también conocida como la regla de Widrow-Hoff, 1960) y el Perceptron de Rosenblatt son las siguientes:

- Los pesos se actualizan considerando la salida de una **función de activación de valores continuos (o reales) en lugar de una función de paso unitario** como en el Perceptron.
- En Adaline, esta la función de activación lineal, $\phi(.)$, es simplemente la función identidad de la entrada neta, o sea $\phi(z) = z$, en consecuencia:

$$\phi(z) = \phi(w^T x) = w^T x$$

- El error se cuantifica utilizando la diferencia cuadrada entre la salida de la función de activación y el valor esperado, o sea:

$$(y^{(i)} - \phi(z^{(i)}))^2$$

In [ ]:
```python
from IPython.display import Image

Image(filename=r'Imagenes_Clase_03/3_1.png', width=700)
```

Descenso de Gradiente (GD)

In [ ]:
```python
Image(filename=r'Imagenes_Clase_03/3_2.png', width=500)
```

- Función de costo: Suma de Errores al Cuadrado (SSE)

$$J(w) = \frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2 = \frac{1}{2}\sum_i (y^{(i)} - w^T x^{(i)}))^2$$

- Recordemos: Paso de actualización (aprendizaje)

$$w := w + \Delta w$$

- Usando el descenso de gradiente, podremos actualizar los pesos dando un paso en la dirección opuesta del gradiente $\nabla J(w)$ de nuestra función de costo J.

- De este modo, Δw se define como el gradiente negativo de J multiplicado por la tasa de aprendizaje η:

$$\Delta w = -\eta \nabla J(w)$$

- Para calcular el gradiente de la función de costo, necesitamos calcular la derivada parcial de la función de costo con respecto a cada peso w_j:

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial w_j}\left(\frac{1}{2}\sum_i (y^{(i)} - \sum_j w_j x_j^{(i)})^2\right) = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

- Así, podemos escribir la actualización del peso w_j como:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

- Dado que actualizamos todos los pesos simultáneamente, nuestra regla de aprendizaje Adaline es:

$$w_j := w_j + \Delta w_j$$

Implementación Adaline con Python

In [ ]:
```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parametros
    ------------
    eta : float
      Learning rate (entre 0.0 y 1.0)
    n_iter : int
      Cantidad de épocas de entrenamiento.
    random_state : int
      Semilla para generar pesos aleatorios.

    Atributos
    -----------
    w_ : 1d-array
      Vector de pesos al término del entrenamiento.
    cost_ : list
      Valor de la función de costo en cada época.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Entrenamiento.

        Parametros
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          Vector de entrenamiento, donde n_samples es el número de muestras y
          n_features es el número de características.
        y : array-like, shape = [n_samples]
          Valor de salida (etiquetas).

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calcular entrada neta, z"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Calcular activación lineal"""
        return X

    def predict(self, X):
        """Etiqueta de clase después del paso unitario"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('https://archive.ics.uci.edu/ml/''machine-learning-databases/iris/iris.data',header=None,encodi
```

In [ ]:
```
df.columns = ['sepal_length','sepal_width','petal_length','petal_width','class']
df.head(5)
```

In [ ]:
```
df.tail(5)
```

In [ ]:
```
df.iloc[45:55, :]
```

In [ ]:
```
df.loc[95:106]
```

In [ ]:
```
df.describe(include='all')
```

In [ ]:
```
y = df.iloc[50:, 4].values
name_clases=list(np.unique(y))
y_numeric = np.where(y == name_clases[0], -1, 1)

X = df.iloc[50:, [0, 2]].values
variable_names=list(df.columns[[0,2]])
plt.scatter(X[:50, 0], X[:50, 1],color='red', marker='o', label=name_clases[0])
plt.scatter(X[50:100, 0], X[50:100, 1],color='blue', marker='x', label=name_clases[1])


plt.xlabel(f'{variable_names[0]} [cm]')
plt.ylabel(f'{variable_names[1]} [cm]')
plt.legend(loc='upper left')

#plt.savefig('02_06.png', dpi=300)
plt.show()
```

Pregunta: Qué flor es la que tiene claasificación -1??

In [ ]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ada1 = AdalineGD(n_iter=50, eta=0.01).fit(X, y_numeric)
#ax[0].plot(range(1, len(ada1.cost_) + 1), ada1.cost_, marker='o')
ax[0].plot(range(1, len(ada1.cost_) + 1), ada1.cost_, marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Sum-squared-error')
ax[0].set_title('Adaline - Learning rate 0.01')

#np.log10(

ada2 = AdalineGD(n_iter=50, eta=0.0001).fit(X, y_numeric)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Sum-squared-error')
ax[1].set_title('Adaline - Learning rate 0.0001')

#plt.savefig('02_11.png', dpi=300)
plt.show()
```

In [ ]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ada1 = AdalineGD(n_iter=50, eta=0.01).fit(X, y_numeric)
#ax[0].plot(range(1, len(ada1.cost_) + 1), ada1.cost_, marker='o')
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Sum-squared-error)')
ax[0].set_title('Adaline - Learning rate 0.01')

#np.log10(

ada2 = AdalineGD(n_iter=50, eta=0.0001).fit(X, y_numeric)
ax[1].plot(range(1, len(ada2.cost_) + 1), np.log10(ada2.cost_), marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('log(Sum-squared-error)')
ax[1].set_title('Adaline - Learning rate 0.0001')

#plt.savefig('02_11.png', dpi=300)
plt.show()
```

In [ ]:
```
Image(filename=r'Imagenes_Clase_03/3_3.png', width=700)
```

- **Estandarización para el Descenso de Gradiente**

  La estandarización cambia la **media** de cada característica para que esté **centrada en cero** y cada característica tenga una **desviación estándar igual a 1**. Por ejemplo, para estandarizar la j-ésima característica, simplemente podemos restar la media muestral μ_j de cada muestra de entrenamiento y dividirla por su desviación estándar σ_j:

$$x_j^{'} = \frac{x_j - \mu_j}{\sigma_j}$$

  Una de las razones por las que la estandarización ayuda con el aprendizaje del descenso de gradiente es que el optimizador tiene que pasar por menos pasos para encontrar una solución óptima.

In [ ]:
```
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
```

```
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```

In [ ]:
```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier,  clases_names=['clase 0','clase 1'], resolution=0.02):
    markers = ('s', 'o', '^', 'v', 'x')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        if cl == -1:
            label = clases_names[0]
        else:
            label = clases_names[1]
        plt.scatter(x=X[y == cl, 0],y=X[y == cl, 1],alpha=0.8,c=colors[idx],
                    marker=markers[idx], label=label,edgecolor='black')
```

In [ ]:
```
ada = AdalineGD(n_iter=50, eta=0.01)
ada.fit(X_std, y_numeric)

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')

plt.tight_layout()
#plt.savefig('images/02_14_2.png', dpi=300)
plt.show()
```

In [ ]:
```
plot_decision_regions(X_std, y, classifier=ada,clases_names=name_clases)
plt.title('Adaline - Gradient Descent (scaled data)')

plt.xlabel(f'{variable_names[0]} [cm]')
plt.ylabel(f'{variable_names[1]} [cm]')


plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/02_14_1.png', dpi=300)
plt.show()
```

Descenso de Gradiente Estocástico (SGD)

- Una alternativa popular al algoritmo de descenso de gradiente por lotes es el **Descenso de gradiente Estocástico**, a veces también llamado descenso de gradiente iterativo o en línea. **En lugar de actualizar los pesos basados en la suma de los errores acumulados en todas las muestras $x^{(i)}$:**

$$\Delta w_j = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

- Se estima el gradiente solo a partir de una observación del conjunto de entrenamiento, elegida de manera aleatoria (por ejemplo, la observación $x^{(k)}$). De esta manera:

$$\Delta w_j = \eta (y^{(k)} - \phi(z^{(k)}))x_j^{(k)}$$

- En algunas implementaciones de descenso de gradiente estocástico, la tasa de aprendizaje fija η a menudo se reemplaza por una **tasa de aprendizaje adaptativa que disminuye con el tiempo**, por ejemplo, multiplicándola por:

$$\frac{c_1}{[\text{num iteraciones}] + c_2}$$

  Donde c_1 y c_2 son constantes positivas.

- Otra ventaja del descenso de gradiente estocástico es que podemos usarlo para el **aprendizaje en línea**.

In [ ]:
```
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.
```

```
    Parametros
    ------------
    eta : float
      Learning rate (entre 0.0 y 1.0)
    n_iter : int
      Cantidad de épocas de entrenamiento.
    shuffle : bool (default: True)
      Si es True, mezcla los datos de entrenamiento cada época, para evitar ciclos..
    random_state : int
      Semilla para generar pesos aleatorios.

    Atributos
    -----------
    w_ : 1d-array
      Vector de pesos al término del entrenamiento.
    cost_ : list
      Valor de la función de costo en cada época.

    """
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        """ Entrenamiento.

        Parametros
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          Vector de entrenamiento, donde n_samples es el número de muestras y
          n_features es el número de características.
        y : array-like, shape = [n_samples]
          Valor de salida (etiquetas).

        Returns
        -------
        self : object

        """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Ajustar los datos de entrenamiento sin reiniciar los pesos"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Barajar datos de entrenamiento"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Inicializar pesos con pequeños números aleatorios"""
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Aplicar la regla de aprendizaje de Adaline para actualizar los pesos"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
```

```python
        return cost

    def net_input(self, X):
        """Calcular entrana neta, z"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Calcular activación lineal"""
        return X

    def predict(self, X):
        """Etiqueta de clase después del paso unitario"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

In [ ]:
```python
ada = AdalineSGD(n_iter=3, eta=0.01, random_state=1)
ada.fit(X_std, y_numeric)

plot_decision_regions(X_std, y, classifier=ada,clases_names=name_clases)
plt.title('Adaline - Gradient Descent (scaled data)')
plt.xlabel(f'{variable_names[0]} [cm]')
plt.ylabel(f'{variable_names[1]} [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/02_15_1.png', dpi=300)
plt.show()
```

In [ ]:
```python
for i in range(len(ada.w_)):
    print('w[{}] = {}'.format(i,ada.w_[i]))
```

In [ ]:
```python
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')

plt.tight_layout()
#plt.savefig('images/02_15_2.png', dpi=300)
plt.show()
```

In [ ]:
```python
ada.partial_fit(X_std, y_numeric)

plot_decision_regions(X_std, y, classifier=ada,clases_names=name_clases)
plt.title('Adaline - Gradient Descent (scaled data)')
plt.xlabel(f'{variable_names[0]} [cm]')
plt.ylabel(f'{variable_names[1]} [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/02_15_1.png', dpi=300)
plt.show()
```

In [ ]:
```python
for i in range(len(ada.w_)):
    print('w[{}] = {}'.format(i,ada.w_[i]))
```

In [ ]:
```python
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')

plt.tight_layout()
#plt.savefig('images/02_15_2.png', dpi=300)
plt.show()
```

In [ ]:
```python
ada.n_iter=20
ada.partial_fit(X_std, y_numeric)

plot_decision_regions(X_std, y, classifier=ada,clases_names=name_clases)
plt.title('Adaline - Gradient Descent (scaled data)')
plt.xlabel(f'{variable_names[0]} [cm]')
plt.ylabel(f'{variable_names[1]} [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/02_15_1.png', dpi=300)
plt.show()
```

In [ ]:
```python
for i in range(len(ada.w_)):
    print('w[{}] = {}'.format(i,ada.w_[i]))
```

¿Cómo realizamos predicciones?

Ejercicio

1. Incorporar dos funciones a la clase AdalineSGD: una que imprima en pantalla los pesos actuales y otra que imprima el costo promedio asociado a un conjunto de datos.
2. Comprobar el cambio de los pesos y el avance de los costos utilizando partial_fit para 5, 10, 15 y 20 iteraciones sucesivas.