

Lectura 1

Ingeniería en Tecnologías de la Telecomunicación

Escuela de Ingeniería de Fuenlabrada

Universidad Rey Juan Carlos

Parte I: Implementación de una red neuronal multicapa desde cero

Basado en el capítulo 11 del libro *Machine Learning with PyTorch and Scikit-Learn*
Sebastian Raschka, Yuxi (Hayden) Liu y Vahid Mirjalili.

Introducción

- El aprendizaje profundo o ***deep learning*** es un **subcampo** del **aprendizaje automático** que se basa en las **redes neuronales** (NNs) artificiales con múltiples capas.
- Su **éxito actual** se debe en gran parte a la **disponibilidad de grandes volúmenes de datos** y **mayor capacidad de cómputo** (GPUs).
- Ha logrado **avances significativos** en tareas como reconocimiento de voz e imágenes, procesamiento de lenguaje natural, traducción automática, etc.

Introducción

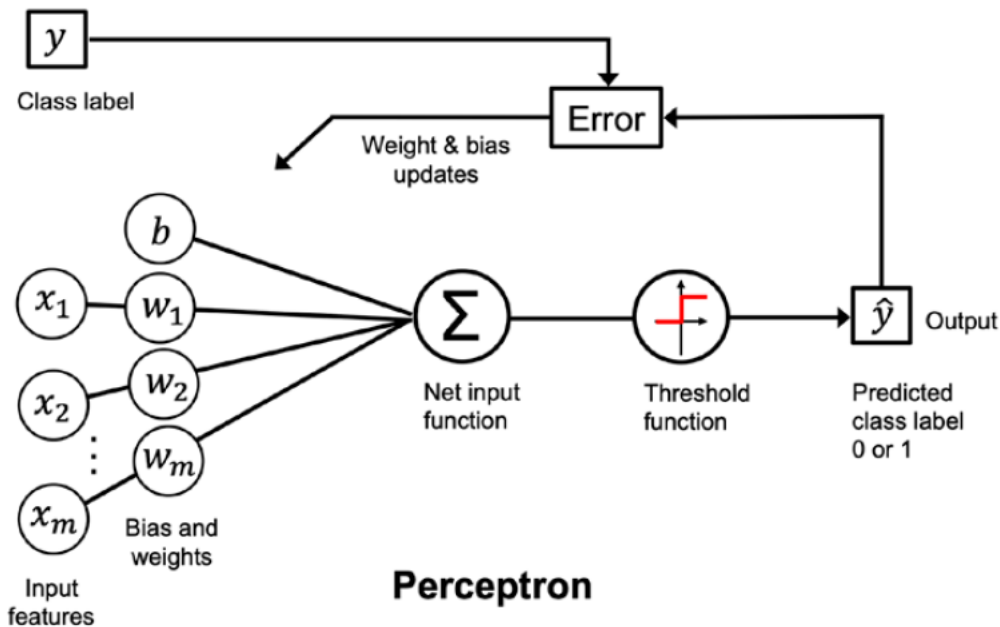
- El **concepto básico** detrás de las NNs se desarrolló a partir de hipótesis y modelos sobre **cómo funciona el cerebro humano** para resolver tareas complejas.
- Los **primeros estudios** sobre NNs se remontan a la década de **1940**, cuando Warren McCulloch y Walter Pitts.
- Tras estos primeros estudios y el **desarrollo del perceptrón** de Rosenblatt en los **años 50**, el **interés en las NNs disminuyó** por la falta de soluciones efectivas para entrenar redes con múltiples capas.
- Es en **1986**, cuando Rumelhart, **Hinton** y Williams popularizaron el algoritmo de ***backpropagation***, permitiendo entrenar NN de manera eficiente.
- Esto **reavivó el interés en las NNs**, marcando uno de los puntos de inflexión en el desarrollo del *deep learning*.

Introducción

- Sin embargo, entre 1986 y 2012 hubo períodos de mayor y menor interés en las NNs, con un enfoque en otros algoritmos de aprendizaje automático.
- En 2012, **Hinton** y su equipo marcaron **otro punto de inflexión** con la creación de AlexNet, una NN profunda diseñada para clasificación de imágenes, que ganó la competición ImageNet, superando por un amplio margen a otros métodos tradicionales.
- Este éxito consolidó a las NN profundas como la tecnología predominante en el aprendizaje automático.

Resumen de NNs de una sola capa

- Vamos a recordar conceptos de NNs de una sola capa.



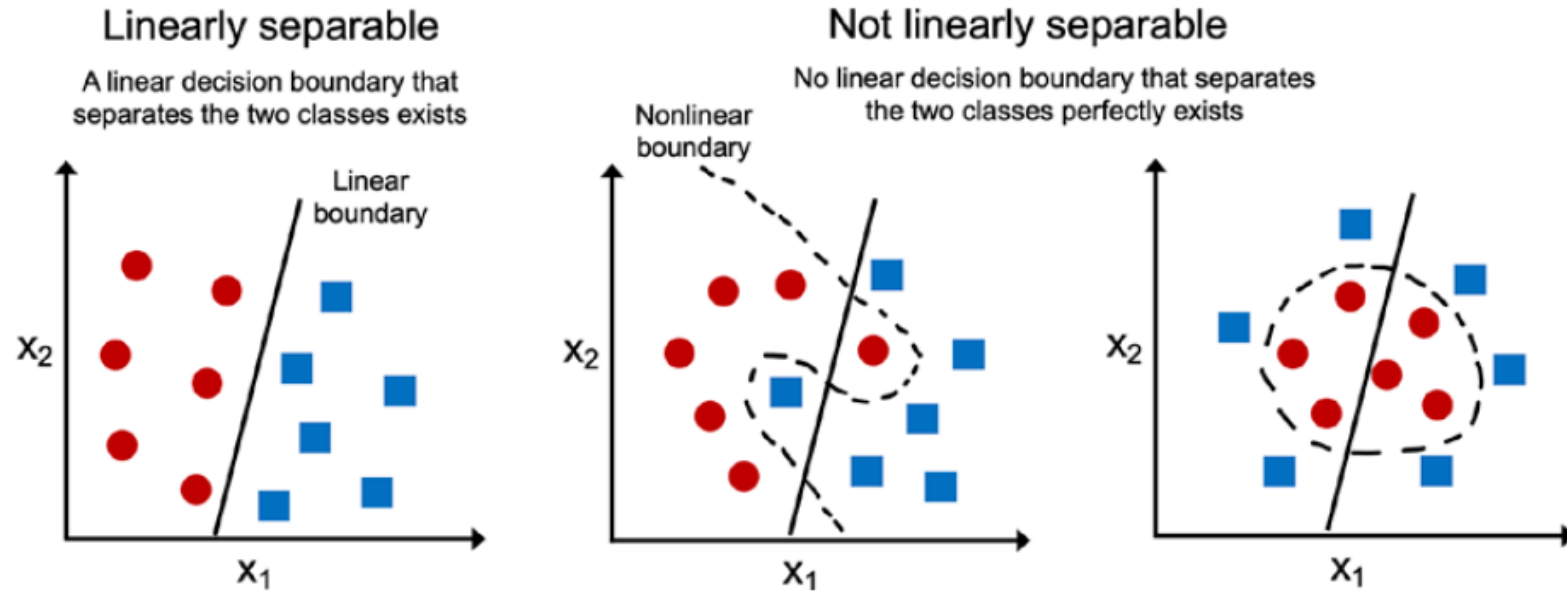
$$w_j := w_j + \Delta w_j$$
$$b := b + \Delta b$$

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$
$$\Delta b = \eta(y^{(i)} - \hat{y}^{(i)})$$

Resumen de NNs de una sola capa

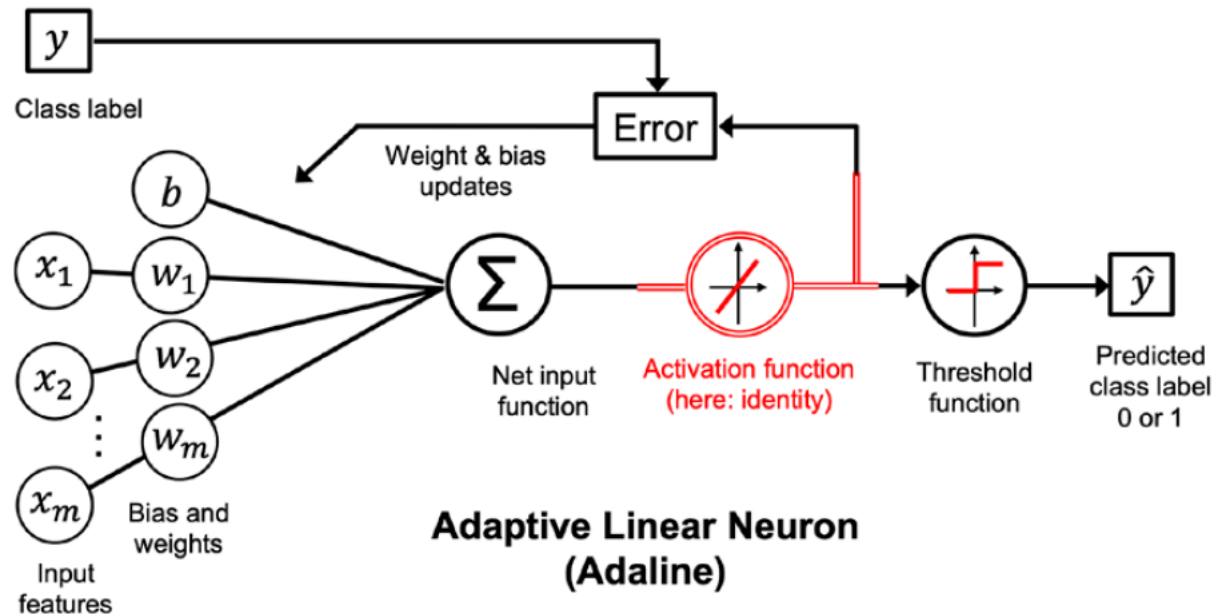
- La convergencia del perceptrón **solo está garantizada si las dos clases son linealmente separables.**
- Si las **clases son perfectamente separables** con una frontera lineal, el algoritmo ajustará los pesos y finalmente **encontrará una solución que clasifique correctamente** todos los puntos de entrenamiento.
- Sin embargo, **si las clases no son linealmente separables**, el perceptrón **no converge y continuará ajustando los pesos indefinidamente.** Por eso, en este caso, es recomendable definir un número máximo de iteraciones (*epochs*) o un umbral de error para evitar un proceso de entrenamiento infinito.

Resumen de NNs de una sola capa



Resumen de NNs de una sola capa

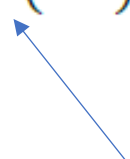
- Vamos a recordar conceptos de NNs de una sola capa.



- Se puede considerar una **mejora del Perceptrón**.
- Se minimiza una **función de pérdida** (*loss function*) continua.
- Los **pesos** se **actualizan** en base a una **función de activación lineal** en lugar de una función escalón como en el Perceptrón.
- Esta función de activación lineal, $\sigma(z)$, es simplemente la función identidad del valor de entrada, de manera que $\sigma(z) = z$.
- Se sigue utilizando una **función de umbralización** para realizar la **predicción final**.
- Adaline **no garantiza una clasificación perfecta** si las **clases no son linealmente separables**, pero tiende a encontrar una **solución que minimiza la función de pérdida**.

Resumen de NNs de una sola capa

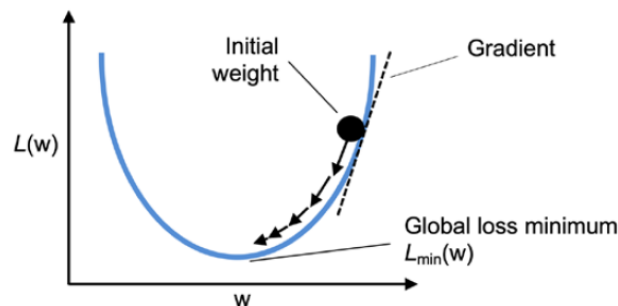
- **Minimización de la función de pérdida con descenso por gradiente.**
- Se fija una **función objetivo** que se **debe optimizar** durante el **proceso de aprendizaje**. Esta es la función de pérdida o de coste que deseamos minimizar.
- En el caso de Adaline, se define la función de pérdida, L , como el **error cuadrático medio (MSE)** entre el resultado calculado y la etiqueta de clase verdadera.

$$L(\mathbf{w}, b) = \frac{1}{2n} \sum_{i=1}^n \left(y^{(i)} - \sigma(z^{(i)}) \right)^2$$


Función de activación continua (para Adaline es la función identidad)

Resumen de NNs de una sola capa

- La principal ventaja de esta **función de activación lineal continua**, en contraste con la función escalón, es que **la función de pérdida se vuelve diferenciable**.
- Además, esta **función de pérdida es convexa**.
- Con esto, podemos utilizar un algoritmo de optimización muy simple llamado **descenso por gradiente** para encontrar los pesos que minimicen nuestra función de pérdida.



$$w := w + \Delta w, \quad b := b + \Delta b$$

$$\Delta w_j = -\eta \frac{\partial L}{\partial w_j} \quad \text{and} \quad \Delta b = -\eta \frac{\partial L}{\partial b}$$

Learning rate (tasa de aprendizaje)

Resumen de NNs de una sola capa

- **Para acelerar el aprendizaje** del modelo, se utiliza una variante denominada **descenso por gradiente estocástico (SGD)**.
- El SGD un pequeño subconjunto de muestras de entrenamiento (*batch*).

Introducción a la arquitectura de NN multicapa

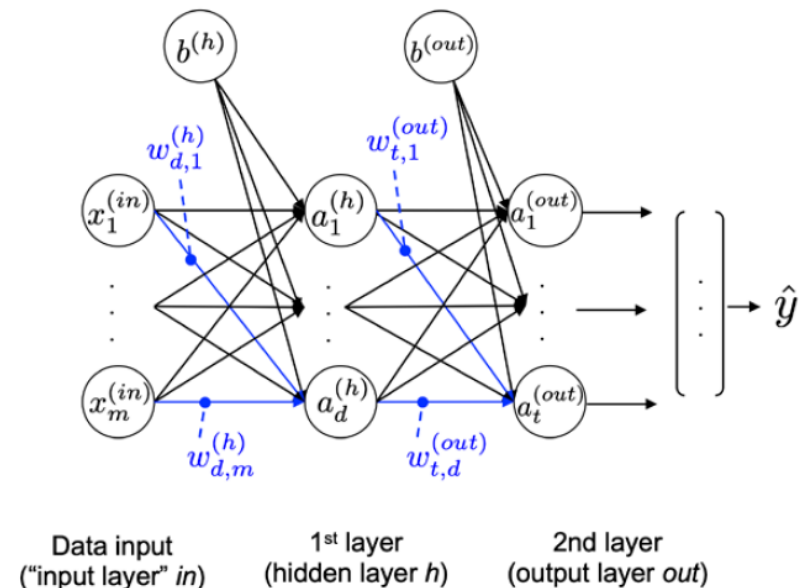
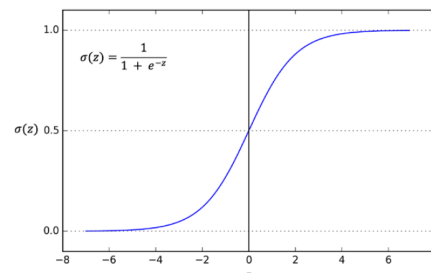
- Para crear **una arquitectura** de NN multicapa hay que **conectar** múltiples **neuronas individuales**.
- La arquitectura básica es la denominada NN multicapa *feedforward*.
- Es un tipo de **red totalmente conectada** que también se denomina Perceptrón Multicapa (MLP).

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \dots + x_m^{(in)} w_{1,m}^{(h)}$$

$$a_1^{(h)} = \sigma(z_1^{(h)})$$

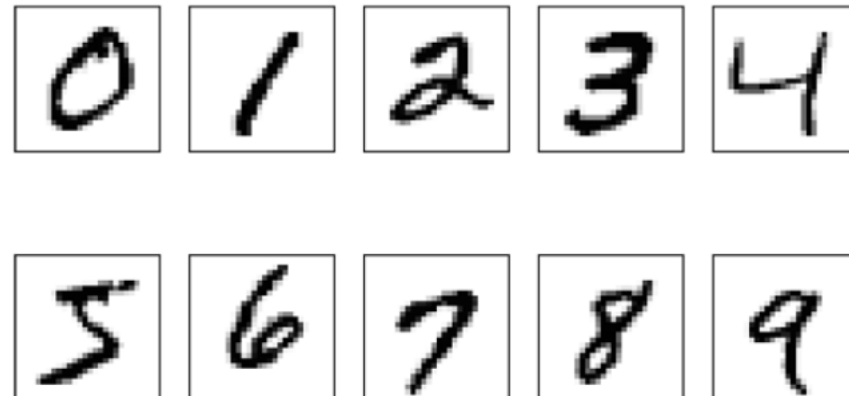
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Función de activación **sigmoide**

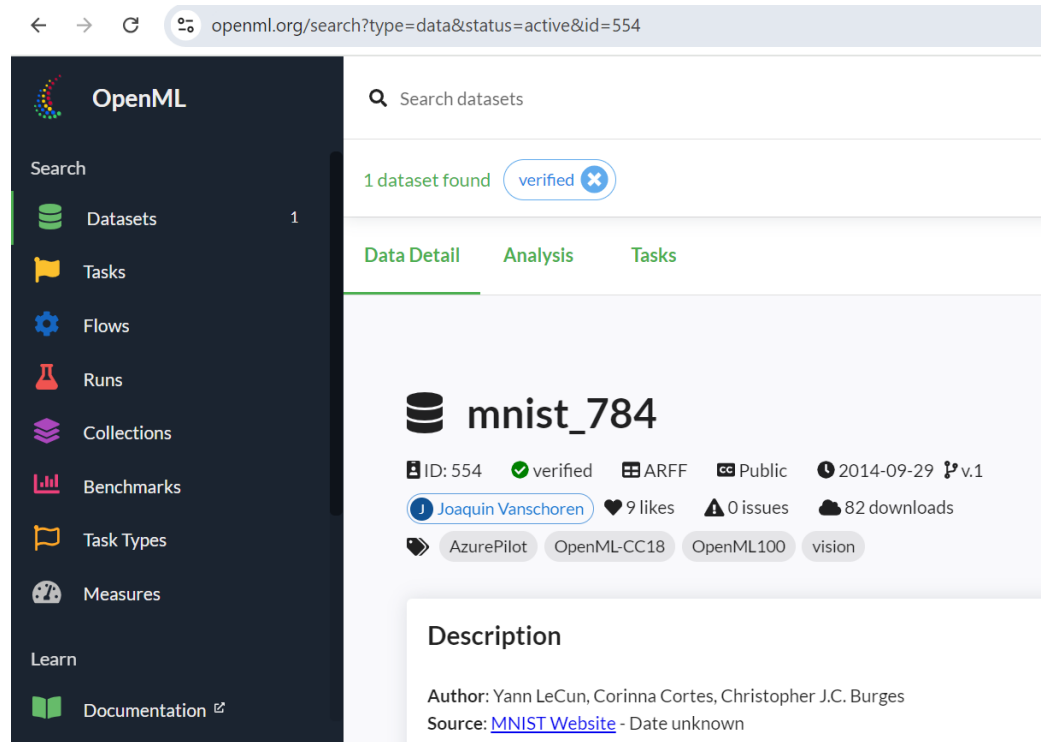


Ejemplos de clasificación de dígitos manuscritos

- Vamos a **implementar y entrenar una NN** multicapa para **clasificar dígitos manuscritos** del popular conjunto de datos **Mixed National Institute of Standards and Technology (MNIST)**, creado por Yann LeCun et al.
- Este conjunto de datos es ampliamente utilizado como referencia para evaluar el rendimiento de algoritmos de aprendizaje automático.



Ejemplos de clasificación de dígitos manuscritos



Vamos a descargar el conjunto de datos de OpenML directamente con scikit-learn.

Ejemplos de clasificación de dígitos manuscritos

```
>>> from sklearn.datasets import fetch_openml
>>> X, y = fetch_openml('mnist_784', version=1,
...                     return_X_y=True)
>>> X = X.values
>>> y = y.astype(int).values
```

```
>>> print(X.shape)
(70000, 784)
>>> print(y.shape)
(70000,)
```

28×28 píxeles

```
>>> X = ((X / 255.) - .5) * 2
```

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                       sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X[y == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```



Los valores de los píxeles en el conjunto de datos MNIST van de 0 a 255, donde 0 representa el nivel de intensidad negro (píxel apagado) y 255 representa el nivel de intensidad blanco (píxel encendido). La normalización se usa para escalar estos valores a un rango diferente, en este caso, de -1 a 1.

Ejemplos de clasificación de dígitos manuscritos

- Vamos a **dividir** el conjunto de datos en **subconjuntos de entrenamiento, validación y test**. El siguiente código separará el conjunto de datos de la siguiente manera:
 - **55,000 imágenes** se utilizarán para **entrenamiento**.
 - **5,000 imágenes** se utilizarán para **validación**.
 - **10,000 imágenes** se utilizarán para **test**.
- Este enfoque nos permite evaluar el rendimiento del modelo durante el entrenamiento (validación) y verificar su capacidad de generalización (test).

```
>>> from sklearn.model_selection import train_test_split
>>> X_temp, X_test, y_temp, y_test = train_test_split(
...     X, y, test_size=10000, random_state=123, stratify=y
... )
>>> X_train, X_valid, y_train, y_valid = train_test_split(
...     X_temp, y_temp, test_size=5000,
...     random_state=123, stratify=y_temp
... )
```

Ejemplos de clasificación de dígitos manuscritos

- Vamos a implementar una NN de una capa (MLP).

```
import numpy as np

def sigmoid(z):
    return 1. / (1. + np.exp(-z))

def int_to_onehot(y, num_labels):

    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1

    return ary
```

Función de activación sigmoide

Función que convierte las etiquetas de clase en formato de enteros (los números de 0 a 9) a etiquetas **one-hot encoded**.

El **one-hot encoding** convierte las etiquetas en vectores binarios donde solo una posición tiene el valor 1 (indicando la clase) y las demás son 0

[0, 1, 2, 3]

```
Clase 0: [1, 0, 0, 0]
Clase 1: [0, 1, 0, 0]
Clase 2: [0, 0, 1, 0]
Clase 3: [0, 0, 0, 1]
```

Ejemplos de clasificación de dígitos manuscritos

```
class NeuralNetMLP:

    def __init__(self, num_features, num_hidden,
                  num_classes, random_seed=123):
        super().__init__()

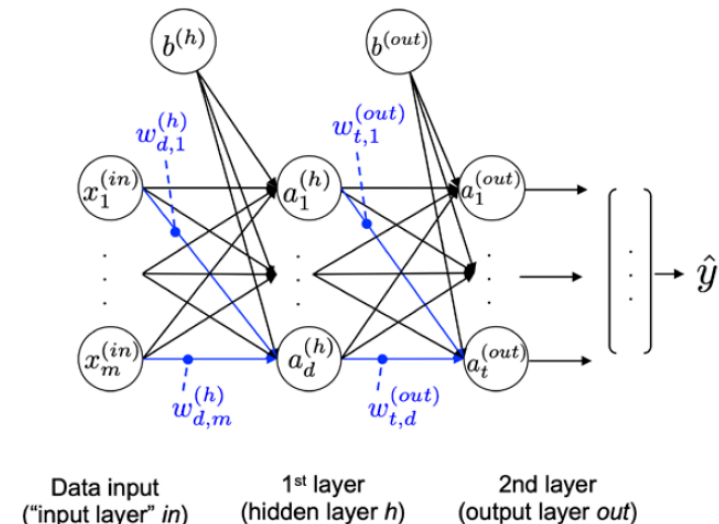
        self.num_classes = num_classes

        # hidden
        rng = np.random.RandomState(random_seed)

        self.weight_h = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # output
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)
```

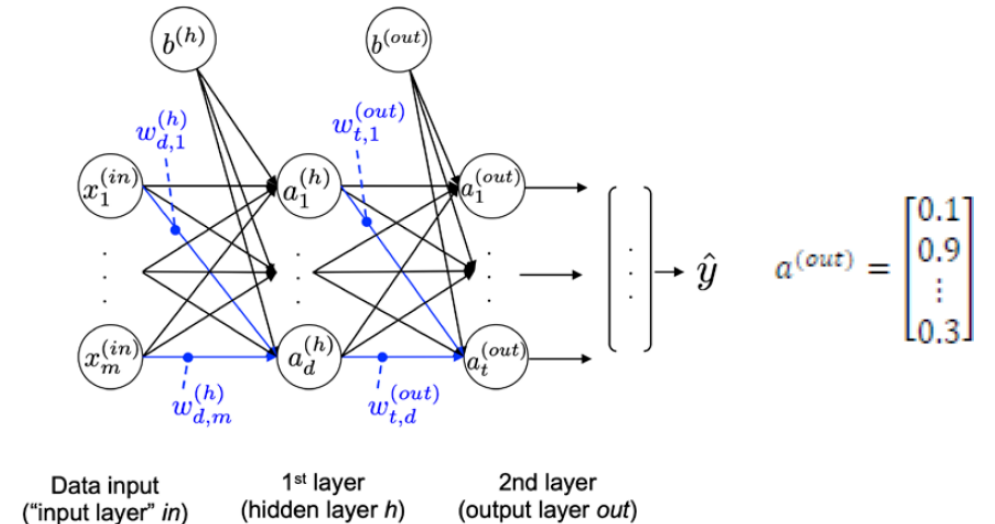
- Red de num_hidden de neuronas en la capa oculta y $num_classes$ de neuronas de salida.
- Se inicializan los pesos y el sesgo de manera aleatoria.



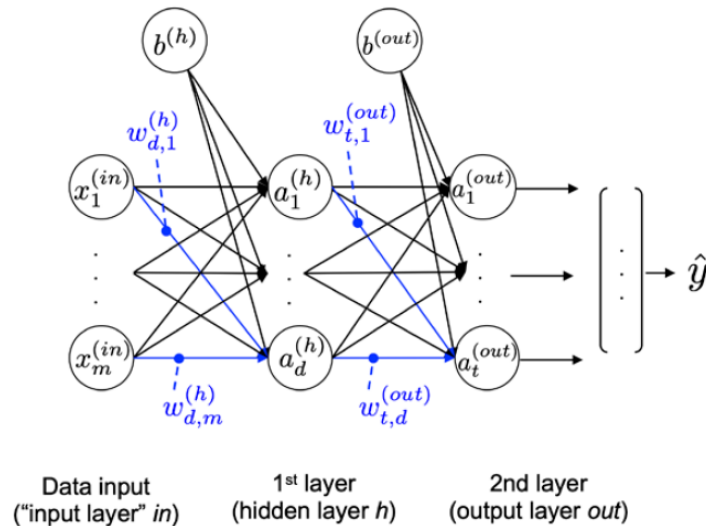
Ejemplos de clasificación de dígitos manuscritos

```
def forward(self, x):  
    # Hidden Layer  
  
    # input dim: [n_hidden, n_features]  
    #      dot [n_features, n_examples] .T  
    # output dim: [n_examples, n_hidden]  
    z_h = np.dot(x, self.weight_h.T) + self.bias_h  
    a_h = sigmoid(z_h)  
  
    # Output Layer  
    # input dim: [n_classes, n_hidden]  
    #      dot [n_hidden, n_examples] .T  
    # output dim: [n_examples, n_classes]  
    z_out = np.dot(a_h, self.weight_out.T) + self.bias_out  
    a_out = sigmoid(z_out)  
    return a_h, a_out
```

- Función que hace las predicciones.



Ejemplos de clasificación de dígitos manuscritos



$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

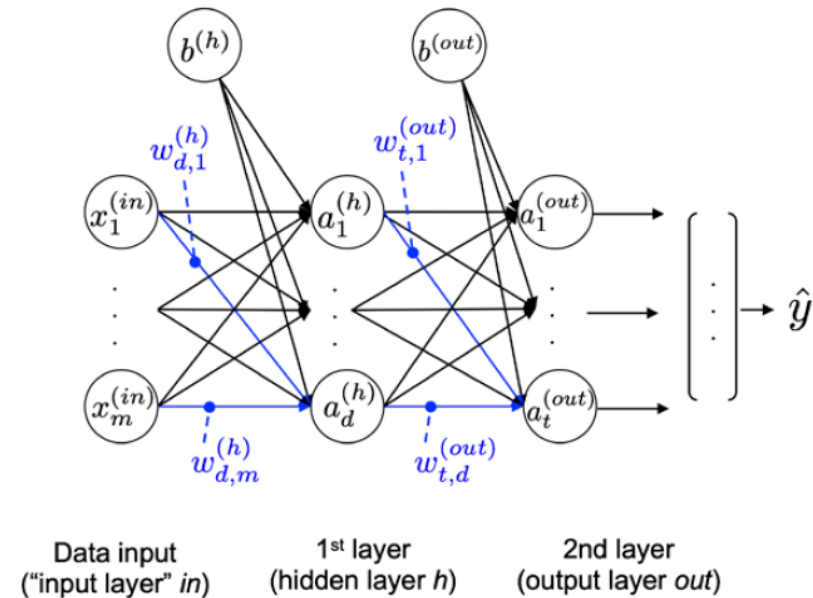
Comparar la salida de la red con las etiquetas.

$$L(W, b) = \frac{1}{n} \sum_1^n \frac{1}{t} \sum_{j=1}^t (y_j^{[i]} - a_j^{(out)[i]})^2$$

Función de pérdida (MSE).

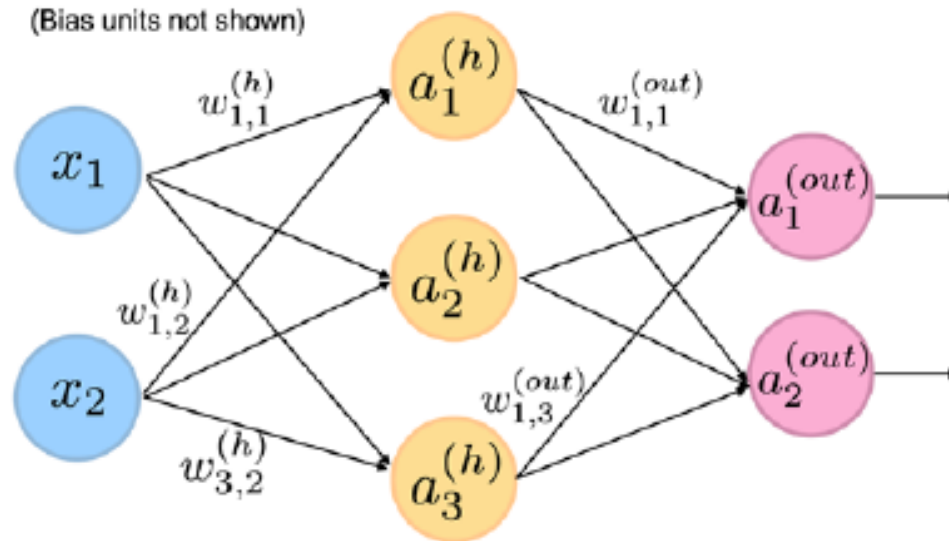
Ejemplos de clasificación de dígitos manuscritos

- El **objetivo** es **minimizar la función de pérdida**.
- Para ello usamos el **descenso por gradiente**.
- Necesitamos calcular la **derivada de la función de pérdida con respecto a los parámetros de la red neuronal** -> algoritmo de **backpropagation**.



Ejemplos de clasificación de dígitos manuscritos

Backpropagation



Ejemplo para **una muestra** de entrada:

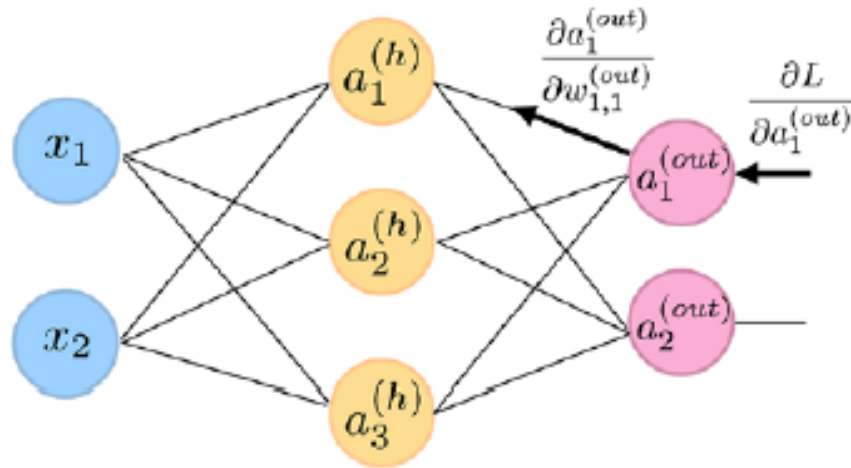
$$L(W, b) = \frac{1}{n} \sum_1^n \frac{1}{t} \sum_{j=1}^t (y_j^{[i]} - a_j^{(out)[i]})^2$$

↓

$$L(W, b) = \frac{1}{t} \sum_{j=1}^t (y_j^{[i]} - a_j^{(out)[i]})^2$$

Ejemplos de clasificación de dígitos manuscritos

Backpropagation



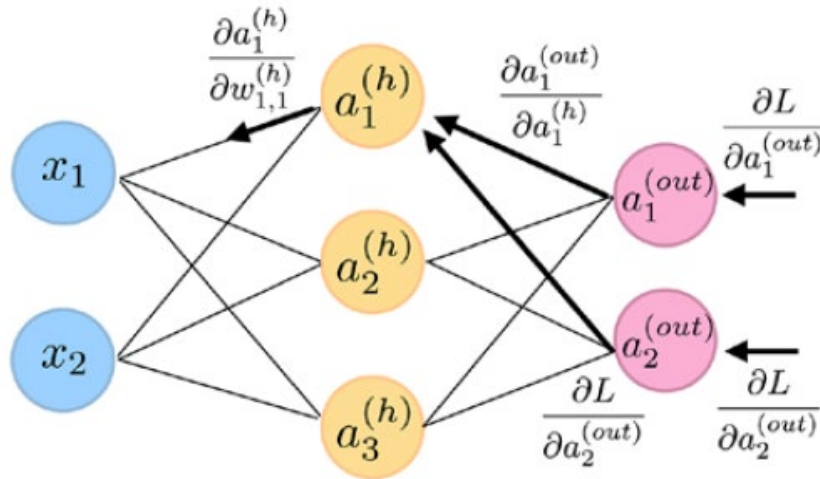
$$\delta_1^{(out)} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}$$

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = 2 \cdot \frac{1}{2} \cdot (a_1^{(out)} - y) \cdot a_1^{(out)} (1 - a_1^{(out)}) \cdot a_1^{(h)}$$

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}$$

Ejemplos de clasificación de dígitos manuscritos

Backpropagation



$$\begin{aligned} \frac{\partial L}{\partial w_{1,1}^{(out)}} &= \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \\ &+ \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial w_{1,1}^{(out)}} &= \delta_1^{(out)} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \\ &+ \delta_2^{(out)} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \end{aligned}$$

Ejemplos de clasificación de dígitos manuscritos

Backpropagation

```
def backward(self, x, a_h, a_out, y):
```

```
#####  
### Output Layer weights  
#####
```

```
# one-hot encoding
```

```
y_onehot = int_to_onehot(y, self.num_classes)
```

```
# Part 1: dLoss/dOutWeights
```

```
## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
```

```
## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet
```

```
## for convenient re-use
```

```
# input/output dim: [n_examples, n_classes]
```

```
d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]
```

```
# input/output dim: [n_examples, n_classes]
```

```
d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative
```

```
# output dim: [n_examples, n_classes]
```

```
delta_out = d_loss__d_a_out * d_a_out__d_z_out
```

```
# gradient for output weights
```

```
# [n_examples, n_hidden]
```

```
d_z_out__dw_out = a_h
```

```
# input dim: [n_classes, n_examples]
```

```
# dot [n_examples, n_hidden]
```

```
# output dim: [n_classes, n_hidden]
```

```
d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
```

```
d_loss__db_out = np.sum(delta_out, axis=0)
```

```
#####
```

```
# Part 2: dLoss/dHiddenWeights
```

```
## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet
```

```
# * dHiddenNet/dWeight
```

```
# [n_classes, n_hidden]
```

```
d_z_out__a_h = self.weight_out
```

```
# output dim: [n_examples, n_hidden]
```

```
d_loss__a_h = np.dot(delta_out, d_z_out__a_h)
```

```
# [n_examples, n_hidden]
```

```
d_a_h__d_z_h = a_h * (1. - a_h) # sigmoid derivative
```

```
# [n_examples, n_features]
```

```
d_z_h__d_w_h = x
```

```
# output dim: [n_hidden, n_features]
```

```
d_loss__d_w_h = np.dot((d_loss__a_h * d_a_h__d_z_h).T,  
                        d_z_h__d_w_h)
```

```
d_loss__d_b_h = np.sum((d_loss__a_h * d_a_h__d_z_h), axis=0)
```

```
return (d_loss__dw_out, d_loss__db_out,
```

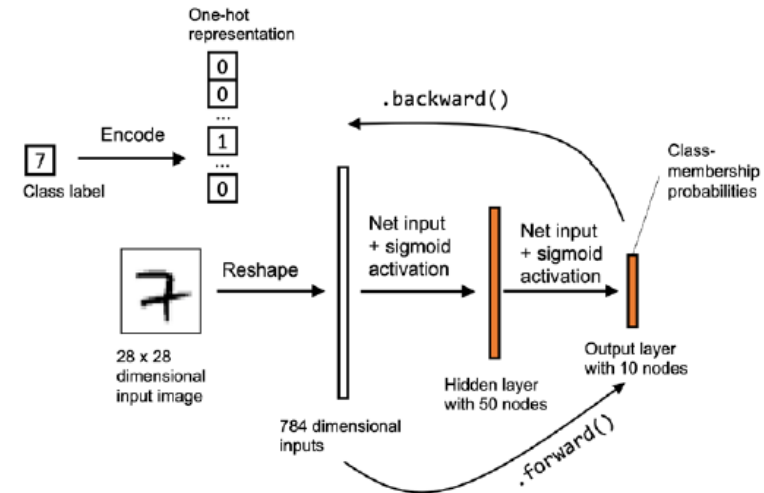
```
        d_loss__d_w_h, d_loss__d_b_h)
```

Ejemplos de clasificación de dígitos manuscritos

```
>>> model = NeuralNetMLP(num_features=28*28,  
...                        num_hidden=50,  
...                        num_classes=10)
```

```
num_epochs = 50  
minibatch_size = 100  
  
def minibatch_generator(X, y, minibatch_size):  
    indices = np.arange(X.shape[0])  
    np.random.shuffle(indices)  
    for start_idx in range(0, indices.shape[0] - minibatch_size  
        + 1, minibatch_size):  
        batch_idx = indices[start_idx:start_idx + minibatch_size]  
        yield X[batch_idx], y[batch_idx]
```

```
def mse_loss(targets, probas, num_labels=10):  
    onehot_targets = int_to_onehot(  
        targets, num_labels=num_labels  
    )  
    return np.mean((onehot_targets - probas)**2)  
  
def accuracy(targets, predicted_labels):  
    return np.mean(predicted_labels == targets)
```



```
def compute_mse_and_acc(nnet, X, y, num_labels=10,  
                        minibatch_size=100):  
    mse, correct_pred, num_examples = 0., 0, 0  
    minibatch_gen = minibatch_generator(X, y, minibatch_size)  
    for i, (features, targets) in enumerate(minibatch_gen):  
        _, probas = nnet.forward(features)  
        predicted_labels = np.argmax(probas, axis=1)  
        onehot_targets = int_to_onehot(  
            targets, num_labels=num_labels  
        )  
        loss = np.mean((onehot_targets - probas)**2)  
        correct_pred += (predicted_labels == targets).sum()  
        num_examples += targets.shape[0]  
        mse += loss  
    mse = mse/i  
    acc = correct_pred/num_examples  
    return mse, acc
```

Ejemplos de clasificación de dígitos manuscritos

```
def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
          learning_rate=0.1):
    epoch_loss = []
    epoch_train_acc = []
    epoch_valid_acc = []

    for e in range(num_epochs):
        # iterate over minibatches
        minibatch_gen = minibatch_generator(
            X_train, y_train, minibatch_size)
        for X_train_mini, y_train_mini in minibatch_gen:
            ### Compute outputs ###
            a_h, a_out = model.forward(X_train_mini)

            ### Compute gradients ###
            d_loss_d_w_out, d_loss_d_b_out, \
            d_loss_d_w_h, d_loss_d_b_h = \
                model.backward(X_train_mini, a_h, a_out,
                               y_train_mini)

            ### Update weights ###
            model.weight_h -= learning_rate * d_loss_d_w_h
            model.bias_h -= learning_rate * d_loss_d_b_h
            model.weight_out -= learning_rate * d_loss_d_w_out
            model.bias_out -= learning_rate * d_loss_d_b_out

        ### Epoch Logging ###
        train_mse, train_acc = compute_mse_and_acc(
            model, X_train, y_train
        )
        valid_mse, valid_acc = compute_mse_and_acc(
            model, X_valid, y_valid
        )
        train_acc, valid_acc = train_acc*100, valid_acc*100
        epoch_train_acc.append(train_acc)
        epoch_valid_acc.append(valid_acc)
        epoch_loss.append(train_mse)
        print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
              f'| Train MSE: {train_mse:.2f} '
              f'| Train Acc: {train_acc:.2f}% '
              f'| Valid Acc: {valid_acc:.2f}%')

    return epoch_loss, epoch_train_acc, epoch_valid_acc
```

Ejemplos de clasificación de dígitos manuscritos

Entrenamiento:

```
np.random.seed(123) # for the training set shuffling
epoch_loss, epoch_train_acc, epoch_valid_acc = train(
    model, X_train, y_train, X_valid, y_valid,
    num_epochs=50, learning_rate=0.1)
```

```
Epoch: 001/050 | Train MSE: 0.05 | Train Acc: 76.17% | Valid Acc: 76.02%
Epoch: 002/050 | Train MSE: 0.03 | Train Acc: 85.46% | Valid Acc: 84.94%
Epoch: 003/050 | Train MSE: 0.02 | Train Acc: 87.89% | Valid Acc: 87.64%
Epoch: 004/050 | Train MSE: 0.02 | Train Acc: 89.36% | Valid Acc: 89.38%
Epoch: 005/050 | Train MSE: 0.02 | Train Acc: 90.21% | Valid Acc: 90.16%
...
Epoch: 048/050 | Train MSE: 0.01 | Train Acc: 95.57% | Valid Acc: 94.58%
Epoch: 049/050 | Train MSE: 0.01 | Train Acc: 95.55% | Valid Acc: 94.54%
Epoch: 050/050 | Train MSE: 0.01 | Train Acc: 95.59% | Valid Acc: 94.74%
```

Parte II: Funciones de pérdida y evaluación de prestaciones en aprendizaje profundo

Basado en las secciones 1, 2 y 3 del artículo *Loss Functions and Metrics in Deep Learning*
Juan Terven et al.

<https://arxiv.org/pdf/2307.02694>

Introducción

- La **función de pérdida** y las **medidas de evaluación de prestaciones (o medidas de rendimiento)** son dos **componentes críticos** en un problema de *deep learning*, esenciales para el éxito del entrenamiento y el funcionamiento del modelo.
- Las **funciones de pérdida** cuantifican la diferencia entre las predicciones del modelo y los resultados esperados.
- Las **medidas de rendimiento** son criterios utilizados para evaluar la eficacia del modelo, proporcionando una medida objetiva de su funcionamiento en tareas específicas.
- Hay **distintas funciones de pérdida y distintas medidas de rendimiento** dependiendo de la tarea a realizar: **regresión o clasificación**.

Función de pérdida vs. Medidas de rendimiento

- **Funciones de pérdida:** Se **utilizan**, en general, durante el **entrenamiento** de un modelo para **optimizar sus parámetros**. Miden la **diferencia entre las salidas del modelo y las esperadas**, y el **objetivo** del entrenamiento es **minimizar esta diferencia**.
 - Se usa con el **conjunto de entrenamiento**: ayuda a entender cómo está aprendiendo el modelo.
 - También se usa con el **conjunto de validación**: ayuda a entender si el modelo está aprendiendo con *overfitting* o no.
- Las funciones de pérdida pueden ser **difíciles de interpretar** porque **sus valores** suelen ser **arbitrarios** y **dependen de la tarea** y el **conjunto de datos** específicos.

Función de pérdida vs. Medidas de rendimiento

- **Medidas de rendimiento:** Se **utilizan** para evaluar el modelo **después del entrenamiento**. Ayudan a determinar **cómo de bien el modelo generaliza** en nuevas muestras y hacer predicciones precisas.
 - Se usa con el **conjunto de test**: ayuda a determinar cómo el modelo generaliza.
- Las **medidas de rendimiento** suelen ser **interpretables** y se pueden aplicar a diferentes tareas, facilitando la **comparación entre modelos**.

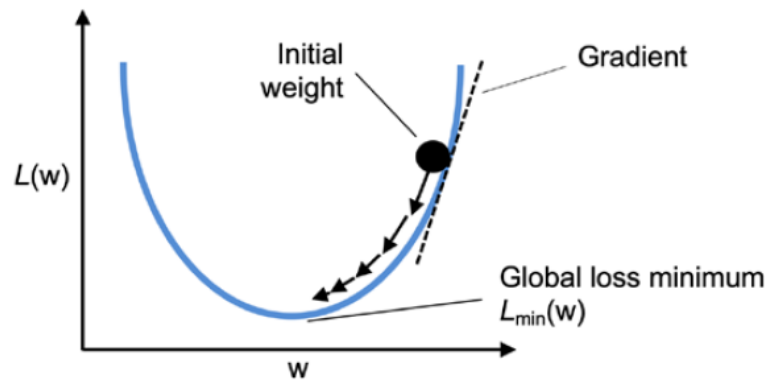
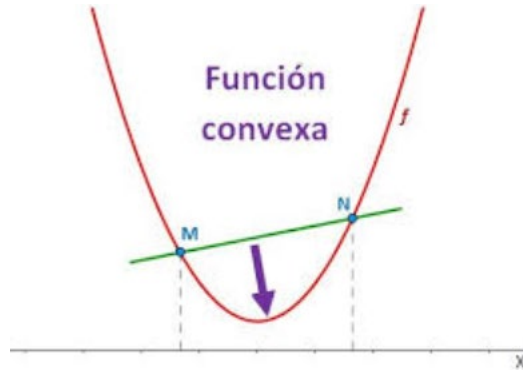
Función de pérdida vs. Medidas de rendimiento

- En el **entrenamiento** también se suele **usar medidas de rendimiento** en el conjunto de **validación** como **complemento** a la función de pérdida.
- Al monitorear estas medidas, se puede obtener una **mejor comprensión** del **entrenamiento** del modelo, **ayudando a detectar problemas**. Por ejemplo: un modelo puede tener una pérdida baja, pero no necesariamente una alta precisión.

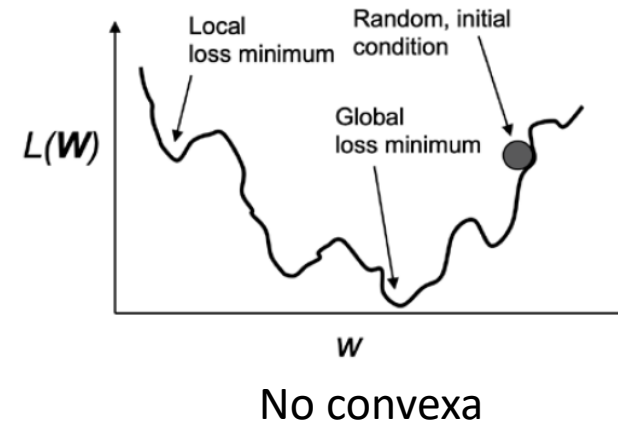
Propiedades de las funciones de pérdida

- **Convexidad:** Una función de pérdida es convexa si, para **cualquier par de puntos en su dominio, la línea recta que une la función en esos puntos está por encima de la función**. Esto implica que cualquier mínimo local es también un mínimo global.
 - Por ejemplo, el error cuadrático medio (**MSE**) es una función de pérdida **convexa** cuando se aplica a **modelos totalmente lineales**.
 - En **redes neuronales con funciones de activación no lineales**, ni el MSE ni **ninguna** otra función de pérdida es **convexa** con respecto a los parámetros de la red.
 - Las funciones de pérdida convexas son **deseables** porque pueden **optimizarse** fácilmente utilizando métodos **basados en gradientes**.

Propiedades de las funciones de pérdida



Convexa



No convexa

Propiedades de las funciones de pérdida

- **Diferenciabilidad:** Una función de pérdida es diferenciable si su **derivada con respecto a los parámetros** del modelo **existe** y es **continua**.
 - La diferenciabilidad es **esencial** para permitir el uso de métodos de **optimización basados en gradientes**.
- **Robustez:** Las funciones de pérdida deben ser capaces de **manejar valores atípicos** y **no verse afectadas** por un pequeño número de **valores extremos**.
- **Suavidad:** Una función de pérdida debe tener un **gradiente continuo** y **no presentar transiciones bruscas** o picos.

Propiedades de las funciones de pérdida

- **Función monótona:** Una función de pérdida es monótona si su **valor disminuye** a medida que la **salida dada por el modelo se acerca a la salida verdadera**.
 - La función monótona **asegura** que el proceso de **optimización se dirige hacia la solución correcta**.

Regresión

- La **regresión** es un problema de **aprendizaje automático supervisado** que tiene como objetivo **predecir un valor de salida continuo** basado en una o más características de entrada.
- Se utiliza en diversos dominios, como finanzas, salud, ciencias sociales, deportes e ingeniería.

Funciones de pérdida en regresión

- El error cuadrático medio (*Mean Squared Error*, **MSE**) mide el promedio de las diferencias al cuadrado entre los valores de salida que predice el modelo y los valores de salida verdaderos.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Número de muestras

Valor de salida verdadero para la muestra i-ésima

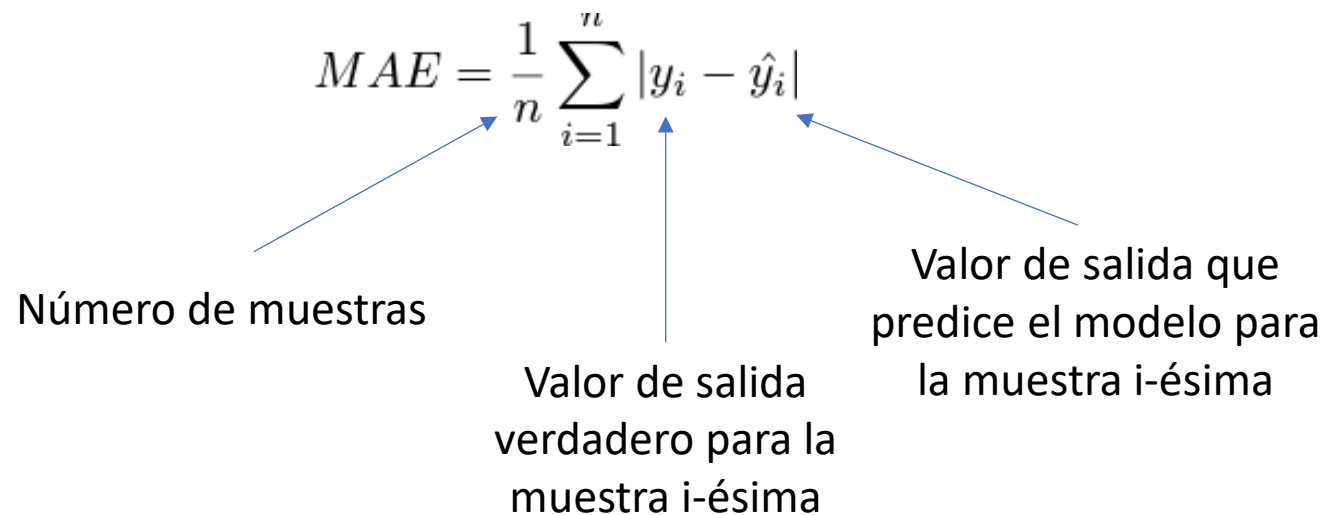
Valor de salida que predice el modelo para la muestra i-ésima

Funciones de pérdida en regresión

- El MSE también se denomina función de pérdida **L2**.
- Es **no negativa**, lo que significa que siempre es igual o mayor a cero.
- Es **sensible a los valores atípicos** debido a su naturaleza cuadrática: da más peso a los errores grandes.
- Es **diferenciable**, permitiendo el cálculo de gradientes necesarios para algoritmos de optimización.
- Es **convexa en modelos lineales**.

Funciones de pérdida en regresión

- El error absoluto medio (*Mean Absolute Error*, **MAE**) mide el promedio de las diferencias absolutas entre los valores de salida que predice el modelo y los valores de salida verdaderos.

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$


Número de muestras

Valor de salida verdadero para la muestra i-ésima

Valor de salida que predice el modelo para la muestra i-ésima

Funciones de pérdida en regresión

- El MAE también se denomina función de pérdida **L1**.
- Es **no negativa**, lo que significa que siempre es igual o mayor a cero.
- Es **robusta a los valores atípicos**, ya que trata todos los errores de manera igual, sin dar un peso desproporcionado a los errores grandes.
- Aunque es continua, **no es diferenciable** cuando el error de predicción es cero debido a la función de valor absoluto.
- Es **convexa en modelos lineales**.
- A menudo se usa como **medida de rendimiento** por su simplicidad computacional y facilidad de comprensión.

Funciones de pérdida en regresión

- La **pérdida de Huber** combina las propiedades del MSE y MAE ofreciendo una **solución más robusta a los valores atípicos** al tiempo que **manteniendo la diferenciabilidad**.

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise,} \end{cases}$$

Valor umbral definido por el usuario (hiperparámetro a elegir)

Valor de salida verdadero para la muestra i-ésima

Valor de salida que predice el modelo para la muestra i-ésima

Funciones de pérdida en regresión

- **Parte cuadrática** se utiliza cuando el **error es pequeño** (la pérdida de Huber se comporta como la pérdida de MSE).
- **Parte lineal** (no cuadrática) se utiliza cuando el **error es más grande** (la pérdida de Huber se comporta como la pérdida de MAE).
 - Esta parte hace que la función de pérdida sea **menos sensible a los valores atípicos**. En lugar de aumentar cuadráticamente con el error, la penalización aumenta linealmente.

Funciones de pérdida en regresión

- La **función de pérdida Log-Cosh** es suave y diferenciable.

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n \log(\cosh(y_i - \hat{y}_i))$$

Número de muestras

Función coseno hiperbólico

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

Valor de salida verdadero para la muestra i-ésima

Valor de salida que predice el modelo para la muestra i-ésima

Funciones de pérdida en regresión

- Una de las ventajas de la función de pérdida Log-Cosh es que es **menos sensible a los valores atípicos** que el MSE, ya que no se ve afectada por valores extremos.
- Sin embargo, es **más sensible a errores pequeños** que la pérdida de Huber.

Funciones de pérdida en regresión

- La **pérdida de cuantiles**, también conocida como pérdida de regresión cuantílica, se utiliza a menudo para **predecir un intervalo** en lugar de un solo valor.

$$L(y, \hat{y}) = q \cdot \max(y - \hat{y}, 0) + (1 - q) \cdot \max(\hat{y} - y, 0)$$

Cuantil q , con $0 < q < 1$

Valor de salida
verdadero para la
muestra i -ésima

Valor de salida que
predice el modelo para
la muestra i -ésima

El cuantil q representa el porcentaje de datos que se espera que esté por debajo de un cierto valor. Por ejemplo, el cuantil 0.5 (mediana) divide el conjunto de observaciones en dos mitades iguales.

Funciones de pérdida en regresión

- **Subestimación** ($y - \hat{y} > 0$):

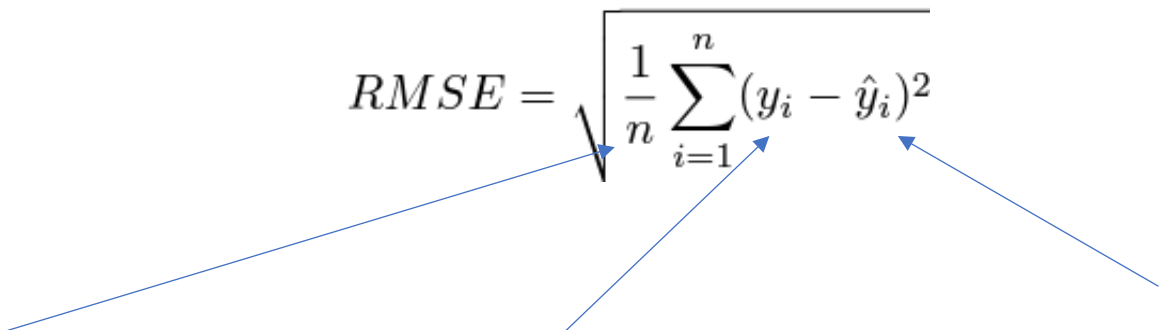
Si la **predicción subestima el valor real**, la **pérdida es proporcional a q** . Esto significa que el modelo penaliza las subestimaciones en función del cuantil q .

- **Sobreestimación** ($\hat{y} - y > 0$):

Si la **predicción sobreestima el valor**, la **pérdida es proporcional a $1-q$** . Esto significa que el modelo penaliza las sobreestimaciones en función de $1-q$.

Medidas de rendimiento en regresión

- Raíz del error cuadrático medio (*Root Mean Squared Error*, **RMSE**) mide la **desviación promedio de las predicciones con respecto a los valores verdaderos**.
- Este valor es **fácil de interpretar** porque está en las mismas unidades que los datos.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$


Número de muestras

Valor de salida verdadero para la muestra i-ésima

Valor de salida que predice el modelo para la muestra i-ésima

Medidas de rendimiento en regresión

- Error porcentual absoluto medio (*Mean Absolute Percentage Error* , **MAPE**) mide el **error porcentual promedio de las predicciones del modelo en comparación con los valores verdaderos**.

$$MAPE = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{y_i} \times 100$$

Número de muestras

Valor de salida verdadero para la muestra i-ésima

Valor de salida que predice el modelo para la muestra i-ésima

Medidas de rendimiento en regresión

- El **coeficiente de determinación** (R^2) muestra cómo de bien las **variaciones en las predicciones del modelo reflejan las variaciones en los datos reales**.

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Valor de salida
verdadero para la
muestra i-ésima

Valor medio de todas
las muestras de salida
verdaderas

Valor de salida que
predice el modelo para
la muestra i-ésima

Clasificación


- La **clasificación** es un problema de **aprendizaje automático supervisado** que entrena un modelo para **predecir la categoría o clase** de una muestra de entrada dada.
- Existen diferentes tipos de tareas de clasificación:
 - La **clasificación binaria** implica entrenar al modelo para predecir una de **dos clases**, como "spam" o "no spam" en el caso de un correo electrónico.
 - La **clasificación multiclase** requiere que el modelo prediga una de varias clases, como "perro", "gato" o "pájaro" para una imagen.
 - La **clasificación multietiqueta**, el modelo se entrena para predecir múltiples etiquetas para una sola muestra de entrada, como "perro" y "al aire libre" para una imagen de un perro en el parque.

Funciones de pérdida en clasificación

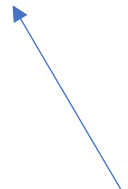
- La pérdida de entropía cruzada binaria (*Binary Cross-Entropy Loss*, **BCE**), es utilizada en problemas de **clasificación binaria** que evalúa la **disimilitud** entre la **probabilidad que se predice para la clase positiva** y la etiqueta de clase verdadera.

$$L(y, p) = -(y \log(p) + (1 - y) \log(1 - p))$$

Valor de salida
verdadero para la
muestra i-ésima



Probabilidad que se
predice para la clase
positiva



Funciones de pérdida en clasificación

- **Cuando $y=1$** (clase positiva), la pérdida es $-\log(p)$. Si p es cercano a 1, la pérdida es baja, indicando una buena predicción. Si p es cercano a 0, la pérdida es alta, indicando una mala predicción.
- **Cuando $y=0$** (clase negativa), la pérdida es $-\log(1-p)$. Si p es cercano a 0, la pérdida es baja, indicando una buena predicción. Si p es cercano a 1, la pérdida es alta, indicando una mala predicción.
- Es una función **suave y diferenciable**, lo que la hace **adecuada para optimización** mediante algoritmos de **descenso por gradiente**.

Funciones de pérdida en clasificación

- BCE puede ser sensible al **desbalanceo de clases**, donde el número de instancias en una clase supera significativamente a la otra. Para abordar esto, se puede utilizar una variante llamada **entropía cruzada binaria ponderada** (*Weighted Binary Cross-Entropy*), que **asigna un peso a cada clase**, ayudando a equilibrar la influencia de cada clase en el proceso de entrenamiento.

$$L_{weighted}(y, p) = -(w_1 \cdot y \log(p) + w_0 \cdot (1 - y) \log(1 - p))$$

Peso para la clase 1

Peso para la clase 0

Los **pesos** suelen establecerse de **manera inversamente proporcional a las frecuencias de las clases** en el conjunto de datos.

Funciones de pérdida en clasificación

- La pérdida de entropía cruzada categórica (*Categorical Cross-Entropy Loss*, **CCE**), es utilizada en problemas de **clasificación multiclase** que evalúa la **disimilitud** entre la **probabilidad que se predice para la clase 1** y la etiqueta de clase verdadera:

Número de muestras $\xrightarrow{\hspace{10em}}$

$$L = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{i,j} \log(p_{i,j})$$

Número de clases $\xleftarrow{\hspace{10em}}$

Valor de salida verdadero para la clase i , que es 1 si la clase i es la verdadera y 0 en caso contrario

Probabilidad predicha de que la muestra pertenezca a la clase i

The diagram illustrates the Categorical Cross-Entropy Loss formula. A horizontal line with arrows at both ends is labeled 'Número de muestras' on the left and 'Número de clases' on the right. The formula $L = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{i,j} \log(p_{i,j})$ is centered below this line. A blue arrow points from the text 'Valor de salida verdadero para la clase i, que es 1 si la clase i es la verdadera y 0 en caso contrario' to the term $y_{i,j}$ in the formula. Another blue arrow points from the text 'Probabilidad predicha de que la muestra pertenezca a la clase i' to the term $p_{i,j}$ in the formula.

Funciones de pérdida en clasificación

- **Predicciones correctas:** Si la probabilidad que se predice para la clase verdadera es alta, la pérdida será baja, indicando una buena predicción.
- **Predicciones incorrectas:** Si la probabilidad que se predice para la clase verdadera es baja, la pérdida será alta, indicando una mala predicción.
- Al igual que la BCE, la CCE es **suave y diferenciable**, lo que la hace adecuada para la optimización mediante algoritmos de **descenso por gradiente**.

Medidas de rendimiento en clasificación

- Verdaderos positivos (*True Positives*, **TP**): Casos en los que el modelo predice correctamente la clase positiva. En otras palabras, son **las muestras que el modelo predice como positivas y que realmente son positivas**.
- Verdaderos negativos (*True Negatives*, **TN**): Casos en los que el modelo predice correctamente la clase negativa. Es decir, son las **muestras que el modelo predice como negativas y que realmente son negativas**.
- Falsos positivos (*False Positives*, **FP**): También conocidos como errores de Tipo I, ocurren cuando el modelo predice incorrectamente la clase positiva. Esto es, son las **muestras que el modelo predice como positivas y que realmente son negativas**.
- Falsos Negativos (*False Negative*, **FN**): También conocidos como errores de Tipo II, ocurren cuando el modelo predice incorrectamente la clase negativa. Es decir, son las **muestras que el modelo predice como negativas y que realmente son positivas**.

Medidas de rendimiento en clasificación

- **Matriz de confusión:** Contiene la cantidad de TP, TN, FP y FN de un conjunto de datos.
- Para una clasificación binaria y multiclase:

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

	Predicted: A	Predicted: B	Predicted: C
Actual: A	$M[A, A]$	$M[A, B]$	$M[A, C]$
Actual: B	$M[B, A]$	$M[B, B]$	$M[B, C]$
Actual: C	$M[C, A]$	$M[C, B]$	$M[C, C]$

Medidas de rendimiento en clasificación

- **Exactitud** (*Accuracy*): Se define como la proporción de muestras clasificadas correctamente respecto al número total de muestras.

$$Accuracy = \frac{\text{Number of Correctly Classified Samples}}{\text{Total Number of Samples}}$$

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Medidas de rendimiento en clasificación

- **Precisión (*Precision*):** Indica **cuántas de las predicciones positivas son realmente correctas.**

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Sensibilidad (*Recall*)** o tasa de verdaderos positivos (*True Positive Rate*, TPR) : Indica **cuántas de las instancias positivas son correctamente identificadas entre todas las instancias que realmente son positivas.**

$$\text{Recall} = \frac{TP}{TP + FN}$$

Medidas de rendimiento en clasificación

- El ***trade-off*** entre **precisión** y **sensibilidad** se refiere que el ajustar un modelo para optimizar una métrica puede afectar negativamente a la otra.
- **Alta precisión, baja sensibilidad:** Si el modelo es muy conservador, solo predice positivo cuando está muy seguro. Esto reduce los falsos positivos, pero puede aumentar los falsos negativos.
- **Alta sensibilidad, baja precisión:** Si el modelo no es tan conservador, predice positivo más fácilmente. Esto reduce los falsos negativos, pero puede aumentar los falsos positivos.

Medidas de rendimiento en clasificación

- El ***F1-Score*** proporciona una medida única que **tiene en cuenta** tanto la **precisión** (la capacidad del modelo para identificar correctamente las instancias positivas) como la **sensibilidad** (la capacidad del modelo para capturar todas las instancias positivas en el conjunto de datos).

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Medidas de rendimiento en clasificación

- La **especificidad** identifica la **proporción de negativos** que son correctamente identificados como tal por un modelo de clasificación.

$$\text{Specificity} = \frac{TN}{TN + FP}$$

Medidas de rendimiento en clasificación

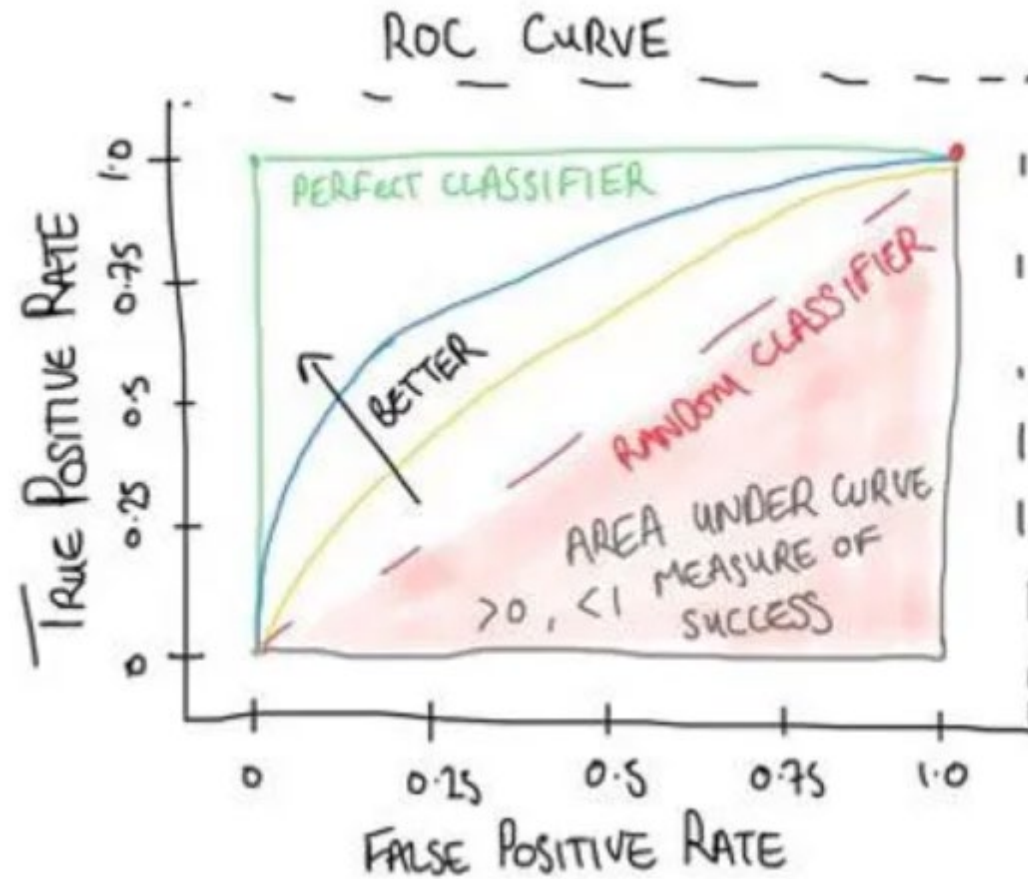
- **La tasa de falsos positivos** (*False Positive Rate*, FPR) mide la proporción de falsos positivos entre todas las instancias realmente negativas.

$$FPR = \frac{FP}{TN + FP}$$

Medidas de rendimiento en clasificación

- La curva ***Receiver Operating Characteristic (ROC)*** es un gráfico que muestra la **relación entre la tasa de verdaderos positivos** (TPR o sensibilidad) y la **tasa de falsos positivos** (FPR) a través de varios umbrales de decisión de un modelo de **clasificación binaria**.
- Área bajo la curva (*Area Under the ROC Curve, AUC-ROC*) es un valor numérico que representa la **capacidad de un modelo para distinguir entre clases**. Un AUC más alto indica un mejor rendimiento, con 1 siendo un modelo perfecto y 0.5 indicando un rendimiento aleatorio.

Medidas de rendimiento en clasificación



Ejercicios

1) Ejercicio: Cálculo de funciones de pérdida en regresión

- Dado un conjunto de predicciones de un modelo y los valores verdaderos correspondientes, se te pide calcular las siguientes funciones de pérdida:
 - MSE
 - MAE
 - Pérdida Log-Cosh
 - Pérdida de Huber
- Utiliza los siguientes vectores para realizar los cálculos:
 - Predicciones del modelo: [3.5, 2.8, 4.0, 5.1, 3.3, 4.8, 2.9, 3.7, 4.2, 5.0]
 - Valores verdaderos:[3.7, 2.5, 4.1, 5.0, 3.0, 4.9, 3.0, 3.8, 4.3, 5.2]

Ejercicios

2) Ejercicio: Medidas de rendimiento en clasificación

- Dado un conjunto de probabilidades de predicción de un modelo y los valores verdaderos correspondientes (clases binarias 0 y 1), se te pide calcular las siguientes medidas de rendimiento para los umbrales 0, 0.25, 0.5, 0.75, y 1:
 - Verdaderos Positivos (TP), Verdaderos Negativos (TN), Falsos Positivos (FP), Falsos Negativos (FN), Matriz de Confusión, Exactitud (Accuracy), Precisión (Precision), Sensibilidad (Recall), F1-Score, Especificidad, Tasa de Falsos Positivos (FPR) y Curva ROC.
 - Utiliza los siguientes vectores para realizar los cálculos:
 - Probabilidades de predicción del modelo:[0.1, 0.4, 0.35, 0.8, 0.65, 0.2, 0.9, 0.55, 0.3, 0.7]
 - Valores verdaderos:[0, 0, 1, 1, 1, 0, 1, 0, 0, 1]

Ejercicios

2) Ejercicio: Medidas de rendimiento en clasificación

- Instrucciones:
 - Para cada umbral, convierte las probabilidades de predicción en clases binarias.
 - Calcula TP, TN, FP, FN, y la matriz de confusión.
 - Calcula las medidas de rendimiento: exactitud, precisión, sensibilidad, F1-Score, especificidad, y FPR.
 - Genera la curva ROC utilizando los valores de FPR y sensibilidad para los umbrales dados.

Soluciones

1) Ejercicio: Cálculo de funciones de pérdida en regresión

- MSE: 0.0319
- MAE: 0.1599
- Pérdida Log-Cosh: 0.0158
- Pérdida de Huber: 0.0159

Soluciones

2) Ejercicio: Medidas de rendimiento en clasificación

```
{'umbral': 0,  
'TP': 5,  
'TN': 0,  
'FP': 5,  
'FN': 0,  
'Exactitud': 0.5,  
'Precisión': 0.5,  
'Sensibilidad': 1.0,  
'F1-Score': 0.6666666666666666,  
'Especificidad': 0.0,  
'FPR': 1.0}
```

```
{'umbral': 0.25,  
'TP': 5,  
'TN': 2,  
'FP': 3,  
'FN': 0,  
'Exactitud': 0.7,  
'Precisión': 0.625,  
'Sensibilidad': 1.0,  
'F1-Score': 0.7692307692307693,  
'Especificidad': 0.4,  
'FPR': 0.6}
```

```
{'umbral': 0.5,  
'TP': 4,  
'TN': 4,  
'FP': 1,  
'FN': 1,  
'Exactitud': 0.8,  
'Precisión': 0.8,  
'Sensibilidad': 0.8,  
'F1-Score': 0.8000000000000002,  
'Especificidad': 0.8,  
'FPR': 0.2}
```

```
{'umbral': 0.75,  
'TP': 2,  
'TN': 5,  
'FP': 0,  
'FN': 3,  
'Exactitud': 0.7,  
'Precisión': 1.0,  
'Sensibilidad': 0.4,  
'F1-Score': 0.5714285714285715,  
'Especificidad': 1.0,  
'FPR': 0.0}
```

```
{'umbral': 1,  
'TP': 0,  
'TN': 5,  
'FP': 0,  
'FN': 5,  
'Exactitud': 0.5,  
'Precisión': 0,  
'Sensibilidad': 0.0,  
'F1-Score': 0,  
'Especificidad': 1.0,  
'FPR': 0.0}
```

Soluciones

2) Ejercicio: Medidas de rendimiento en clasificación

