Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

TÉCNICAS DE INTELIGENCIA ARTIFICIAL
4º Curso, Grupo 46
1er cuatrimestre

# Practica 3 - Clasificación supervisada

Nagore Gómez y Sergio Martín

Bilbao, 30 de noviembre de 2023

# Índice

# Parte I

# 1. Pregunta 1: Puertas lógicas con *Perceptrón*

## 1.1. Descripción

Una neurona *Perceptrón* es la estructura básica de procesamiento de una red neuronal. La neurona *Perceptrón* recibe unos valores de entrada (*inputs*) de los cuales genera una combinación lineal, calculando para ello el producto escalar de los valores de entrada y unos factores de ponderación llamados pesos.

En esencia, lo que la neurona produce es una salida (*output*) calculada de la siguiente manera:

$$y_j = \sum_{j}^{d} w_j * x_j$$

Si el uso del *Perceptrón* está dirigido a clasificación binaria, la salida de la neurona deberá ser sometida a una función de activación. Existen diversas funciones de activación (*step function*, sigmoide, función ReLU. función Tanh, función *softmax*, ...), pero todas ellas tienen los mismos objetivos: la introducción de no-linealidad en el modelo y la regularización y el control del rango de valores devueltos por la neurona.

De modo que sobre el valor de $y$ que se obtiene como salida de la neurona se podría aplicar por ejemplo una *step function* de esta forma:

$$s(y_j) = \begin{cases} 1 & : \ y > 0 \\ 0 & : \ y \leq 0 \end{cases}$$

Como se ha mencionado, la función de activación limitará la salida de la neurona a 2 únicos valores: 0 o 1.

Cabe destacar que el *Perceptrón* es un modelo paramétrico, cuyos parámetros son los pesos a asignar a cada uno de los valores de entrada. Para dar con los valores que mejor clasificación ofrece el modelo se ha de entrenar el mismo. Esa fase del entrenamiento consiste en alterar los valores de los pesos hasta conseguir la convergencia o un error aceptable según los criterios establecidos.

Adicionalmente, teniendo en cuenta que lo que alteración de los pesos genera son hiperplanos cuya pendiente es diferente, sería recomendable poder modificar también la coordenada en el origen para ese hiperplano. Para ello se añade en la ecuación el valor *BIAS*, que será un *input* adicional en la neurona que permitirá desplazar el hiperplano creado también a lo largo de la ordenada. La fórmula pasaría a verse de esta forma:

$$y_j = w_0 + \sum_{j}^{d} w_j * x_j$$

Por último, el modo de actualizar los pesos en cada iteración consiste en comprobar si la clase predicha coincide con la clase real. De ser así, los pesos no se verán actualizados (en la versión más simple del *Perceptrón*), sin embargo, si la clase predicha y la real no coinciden, se lleva a cabo la siguiente modificación sobre los pesos:

$$w_j = \begin{cases} s(y_j) = 1 & : \ w_j - x_j \quad \textit{Las instancias están mas cerca de lo que deberían} \Rightarrow \textit{Alejar} \\ s(y_j) = 0 & : \ w_j + x_j \quad \textit{Las instancias están mas lejos de lo que deberían} \Rightarrow \textit{Acercar} \end{cases}$$

Para el entrenamiento solamente se requiere de los siguientes elementos:

- Un conjunto de datos supervisado (conjunto de instancias con su valor de clase)
- Un valor inicial y aleatorio para cada uno de los pesos
- Una función de activación definida

Es importante recalcar:

- La convergencia del algoritmo de entrenamiento puede ser costosa de alcanzar, por ello se puede establecer un umbral de error y detener el entrenamiento si el error no supera el umbral.

## 1.2. Pseudocódigo del *Perceptrón* para clasificación binaria

**Algorithm 1** Algoritmo *Perceptrón* para clasificación binaria

```
w = Random()
while ¬ convergencia do
    y_j = w_0 + w_j * x_j
    ĉ = s(y_j)
    w_j = w_j + (c − ĉ) * x_j //Acercar o alejar en función del error
end while
```

## 1.3. *Averaged Perceptrón*

El *Averaged Perceptrón* es una variación sutil del algoritmo por defecto. La principal diferencia radica en el tratamiento de los pesos durante el entrenamiento. Mientras que en el algoritmo ordinario los pesos se actualizan acercando y alejando vectores mediante operaciones vectoriales, en el caso del *Averaged Perceptrón* adicional a esa actualización se calcula la media entre todos los pesos.

Realmente, almacenar los pesos por cada iteración para poder calcular la media supone un gran coste en memoria, por ello, se suele hacer la media únicamente entre los pesos al principio de la iteración y al final de la misma, tras haberlos actualizado si es que procedía.

El promedio de los pesos ayuda a suavizar las oscilaciones y variaciones que ocurren durante el entrenamiento del modelo. Esto lo hace menos sensible a las variaciones específicas de los datos de entrenamiento, lo que puede mejorar su estabilidad y generalización a datos nuevos.

Además, se reduce la tendencia del modelo a sobre ajustarse a los datos de entrenamiento específicos, conocido como *overfitting*. Y finalmente, el *Averaged Perceptrón* tiende a ser menos sensible a la inicialización inicial de los pesos, lo que puede ser beneficioso al entrenar redes neuronales.

**Algorithm 2** Algoritmo *averaged Perceptrón* para clasificación binaria

w = *Random()*
**while** ¬ *convergencia* **do**
  $y_j = w_0 + w_j * x_j$
  $\hat{c} = s(y_j)$
  $w_{jact} = w_j + (c - \hat{c}) * x_j$ //Acercar o alejar en función del error
  $w_j = average(w_j, w_{jact})$
**end while**

## 1.4. Puerta *XOR*

Una de las funciones más básicas que puede tener una neurona *Perceptrón* puede ser la de imitar el comportamiento de una puerta lógica. En esta tarea se han implementado las puertas *AND*, *OR*, *NOT* y *XOR*. Sin embargo, la puerta *XOR* no es imitable utilizando una sola neurona. (Ver figura 1)

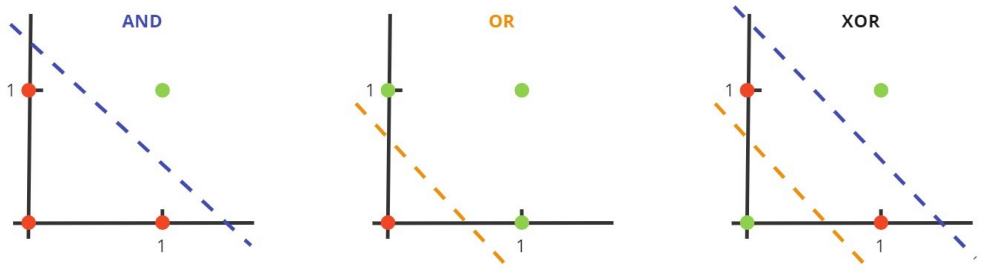


Figura 1: Representación gráfica de las funciones que emulan las puertas lógicas

No obstante, el comportamiento de la puerta *XOR* se puede conseguir combinando los resultados de las puertas *AND* y *OR*. (Ver figura 2)

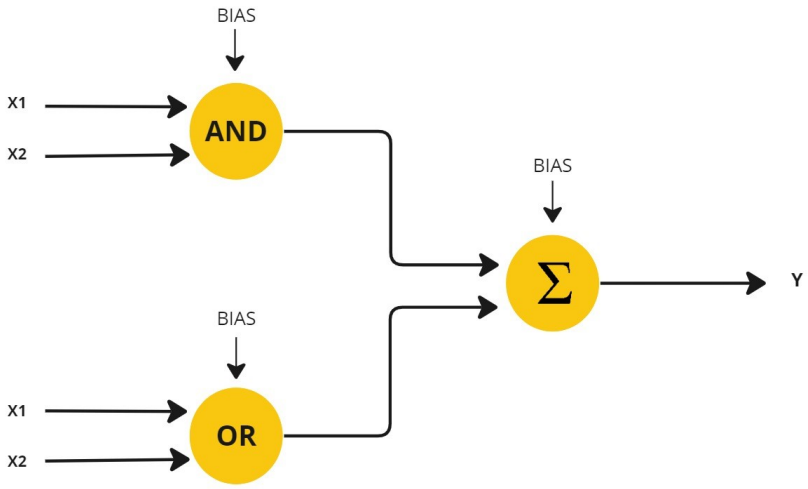


Figura 2: Red neuronal para emular la puerta lógica *XOR*

Existe la posibilidad de implementar una puerta lógica *XOR* con una sola neurona. Para ello se debe "*input*ar" un valor de entrada artificial, formado por el producto de los dos valores de entrada originales. De esta forma, la neurona tendría 3 *inputs* y sería posible distinguir las fronteras en la representación gráfica de la puerta *XOR*. (Ver figura 3)

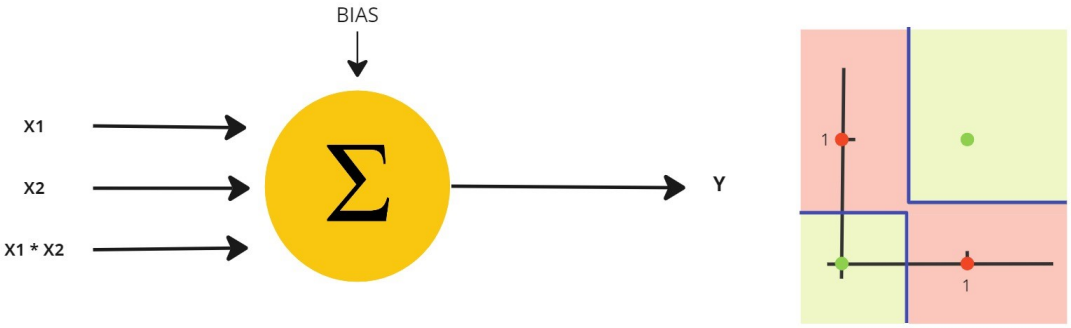


Figura 3: Puerta lógica *XOR* con una sola neurona

## 1.5. Problemas y dificultades encarados en el ejercicio

Realmente no es una tarea difícil implementar el algoritmo de entrenamiento para que una neurona imite el comportamiento de una puerta lógica. La complicación en esta pregunta reside en familiarizarse con las estructuras de datos y variables empleadas.

Algo que ha causado un pequeño inconveniente en el desarrollo ha sido que en los valores de entrada para la función *neuron()* no haya que incluir el valor de *BIAS*, pero que, sin embargo, para hacer el ajuste de pesos sí haya que tenerlo en cuenta.

# 2. Pregunta 2 Perceptrón *MNIST*

## 2.1. Descripción

Hasta ahora se ha hecho referencia al *Perceptrón* como un clasificador binario, no obstante, es posible aplicarlo sobre problemas de clasificación donde hay múltiples valores de clase. Para ello, se emplearán tantas neuronas como valores de clase posibles haya. (Ver figura 4)

Se introducirán todos los valores de entrada en todas las neuronas y cada una de ellas devolverá un valor $y_j$. Ahora no habrá un vector de pesos, sino que se representarán los pesos de manera matricial. Entonces para calcular el valor de $y_j$ por cada neurona se hará lo siguiente:

$$y_j = W_j^T * x_j$$

Y la función de activación más común en estos casos suele ser la función *softmax*, que consiste en calcular el siguiente valor:

$$s(y_j) = \frac{\exp(y_j)}{\sum_k^d \exp(y_k)}$$

La salida de la función *softmax* indica como de probable es que la instancia a clasificar pertenezca a la clase que representa esa neurona. Por ello, tras calcular $s(y)$ se aplica una función arg máx para decidir cuál es la clase más probable para la instancia a clasificar.

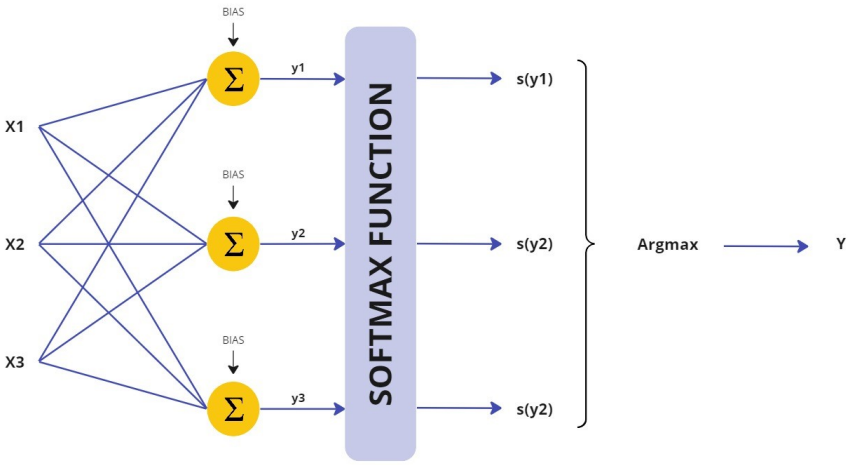


Figura 4: *Perceptrón* multiclase

En el caso de esta pregunta se propone trabajar con el *dataset MNIST*. Con estos datos la clasificación que se plantea es multiclase, existiendo 10 valores de clase diferentes

(los números entre el 0 y el 9). Por lo tanto, se precisa de 10 neuronas para solucionar este problema de clasificación. A su vez, las instancias están caracterizadas por 784 atributos, y cada uno de ellos será un *input* en cada una de las 10 neuronas del sistema. (Ver figura 5)

Es importante recalcar que en la implementación se obvia el uso de la función *softmax* por petición expresa del enunciado. Se emplea el producto escalar como *score*.

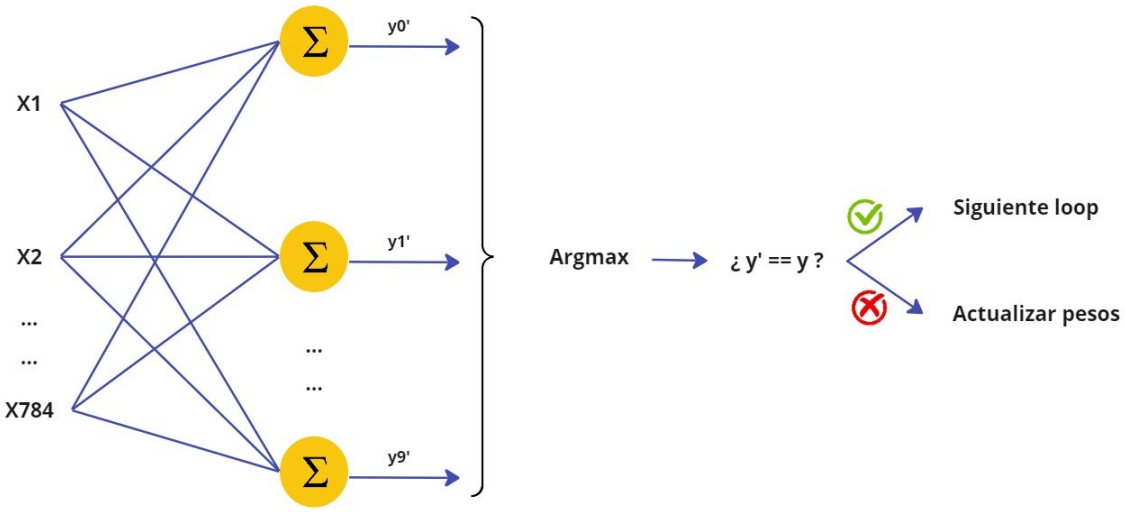


Figura 5: *Perceptrón* para clasificación *MNIST*

Para el entrenamiento solamente se requiere de los siguientes elementos:

- Un conjunto de datos supervisado (conjunto de instancias con su valor de clase)
- Un valor inicial y aleatorio para cada uno de los pesos en la matriz
- Una función de activación definida

Es importante recalcar:

- La convergencia del algoritmo de entrenamiento puede ser costosa de alcanzar, por ello se puede establecer un umbral de error y detener el entrenamiento si el error no supera el umbral.

## 2.2. Pseudocódigo del *Perceptrón* para clasificación multiclase

**Algorithm 3** Algoritmo *Perceptrón* para clasificación multiclase

w = *Random()*
**while** $\neg$ *convergencia* **do**
  $y_j = w_0 + w_j * x_j$

  $softmax_j = \frac{\exp(y_j)}{\sum_k^d \exp(y_k)}$

  $c^* = \arg \text{máx}_i(softmax_i)$

  **if** $c^* \neq c^{ok}$ **then**
    $w^{ok} = w^{ok} + x_j$ //Acercar la clase correcta (real)
    $w^* = w^* + x_j$ //Alejar la clase incorrecta (predicha)
  **end if**
**end while**

## 2.3. Código V1

```
def train(self, trainingData, trainingLabels, validationData,
    validationLabels):

        self.features = trainingData[0].keys()

        for iteration in range(self.max_iterations):
            print("Starting iteration ", iteration, "...")

            # Iterar las instancias del conjunto train
            for i in range(len(trainingData)):

                best_label = None
                instance = trainingData[i]

                # Iterar los valores de clase poosibles
                for label in self.legalLabels:

                    # Calcular el producto  escalar
                    output = self.weights[label].__mul__(
                        instance)

                    best_label = label

                # Obtener el valor de clase real
                real_label = trainingLabels[i]

                # Si la clase predicha no es correcta -->
                    Actualizar pesos
                if best_label != real_label:
```

```
                    # Acercar a la clase real
                    self.weights[real_label] = self.weights[
                        real_label] + instance

                    # Alejar de la clase mal predicha
                    self.weights[best_label] = self.weights[
                        best_label] + instance
```

```
Question q2
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
11 correct out of 100 (11.0%).
*** FAIL: test_cases\q2\grade.test (0 of 4 points)
***      11.0 correct (0 of 4 points)
***          Grading scheme:
***           < 70:  0 points
***          >= 70:  4 points

### Question q2: 0/4 ###
```

## 2.4. Código V-Final

```
def train(self, trainingData, trainingLabels, validationData,
    validationLabels):

        self.features = trainingData[0].keys()

        for iteration in range(self.max_iterations):
            print("Starting iteration ", iteration, "...")

            # Iterar las instancias del conjunto train
            for i in range(len(trainingData)):

                topScore = -1
                best_label = None
                instance = trainingData[i]

                # Iterar los valores de clase poosibles
                for label in self.legalLabels:
```

```
                    # Calcular el producto  escalar
                    output = self.weights[label].__mul__(
                        instance)

                    # El valor de clase que mejor putput consiga
                         sera el predicho
                    if output > topScore:
                        topScore = output
                        best_label = label

                # Obtener el valor de clase real
                real_label = trainingLabels[i]

                # Si la clase predicha no es correcta -->
                    Actualizar pesos
                if best_label != real_label:

                    # Acercar a la clase real
                    self.weights[real_label] = self.weights[
                        real_label] + instance

                    # Alejar de la clase mal predicha
                    self.weights[best_label] = self.weights[
                        best_label] - instance
```

```
Question q2
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
79 correct out of 100 (79.0%).
*** PASS: test_cases\q2\grade.test (4 of 4 points)
***      79.0 correct (4 of 4 points)
***          Grading scheme:
***           < 70:  0 points
***          >= 70:  4 points

### Question q2: 4/4 ###
```

## 2.5. Problemas y dificultades encarados en el ejercicio

Por una parte, el hecho de tener que elegir en cada iteración la salida de cuál de las neuronas es la que mejor representa la clase de la instancia ha supuesto un problema. Sin

embargo, ha sido un fallo de no haberlo tenido en cuenta, una vez detectado el problema ha sido fácil de corregirlo.

Finalmente, en esta pregunta las estructuras de datos no son tan explícitas y, por lo tanto, comprenderlas y trabajar con ellas ha resultado más costoso. Así y todo el algoritmo a implementar es muy similar al implementado para las puertas lógicas.

# 3. Pregunta 3: Clonación del comportamiento

## 3.1. Descripción

A continuación se propone emplear el clasificador *Perceptrón* multiclase sobre el conocido juego de *Pacman*. En este caso, las instancias o los datos serán estados y los valores de clase para cada instancia serán el conjunto de acciones legales desde el estado que la instancia representa.

El problema es el mismo que en la pregunta 2, solo que esta vez las instancias tiene otra estructura, y además, el vector de pesos será único para las etiquetas reales y las predichas. Esto ocurre porque estamos ante un problema de *ranking*. Se trata de un caso especial donde se aplica el mismo vector de pesos a todos los ejemplos, puesto que no hay una clase correcta o una incorrecta, simplemente se está tratando de cuantificar cuanto se parece cada ejemplo a la solución deseada.

Adicionalmente, en esta ocasión se ha hecho uso de la función *classify()* que se ofrece en el fichero *'perceptron_pacman.py'*. Sirve para abstraer el bucle que compara los resultados de todas las neuronas y obtiene la clase que más se ajusta a la instancia. Internamente, itera todos los valores de clase posibles y calcula el producto escalar con ellos, devolviendo la clase predicha para la instancia.

De esta forma, se obtiene la salida de cada una de las neuronas (habrá tantas valores de clase posibles) y se busca el mayor producto escalar. La neurona que mayor producto escalar ofrezca será la que le atribuya la clase predicha a la instancia a clasificar. Finalmente, si la predicción es correcta, se pasará a la siguiente iteración y si no se ajustarán los pesos. (Ver figura 6)

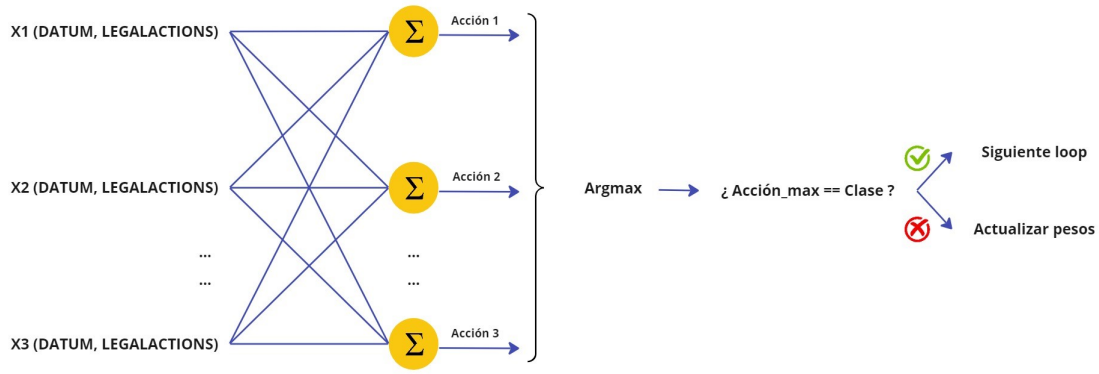


Figura 6: *Perceptrón* multiclase para juego *Pacman*

Para el entrenamiento solamente se requiere de los siguientes elementos:

- Un conjunto de datos supervisado (conjunto de instancias con su valor de clase)

- Un valor inicial y aleatorio para cada uno de los pesos

Es importante recalcar:

- La convergencia del algoritmo de entrenamiento puede ser costosa de alcanzar, por ello se puede establecer un umbral de error y detener el entrenamiento si el error no supera el umbral.

### 3.1.1. Código V1

```
def train(self, trainingData, trainingLabels, validationData,
    validationLabels):

    self.features = trainingData[0][0]['Stop'].keys()

    for iteration in range(self.max_iterations):
        print("Starting iteration ", iteration, "...")
        for i in range(len(trainingData)):

            # Calcular la clase predicha
            best_label = self.classify([trainingData[i]])[0]

            # Obtener el valor de clase real
            real_label = trainingLabels[i]

            # Si la clase predicha no es correcta --> Actualizar
                pesos
            if best_label != real_label:

                # Acercar a la clase real
                self.weights = self.weights + trainingData[i
                    ][0][real_label]

                # Alejar de la clase mal predicha
                self.weights = self.weights - trainingData[i
                    ][0][best_label]
```

```
Question q3
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
80 correct out of 100 (80.0%).
*** PASS: test_cases\q3\contest.test (2 of 2 points)
***      80.0 correct (2 of 2 points)
***          Grading scheme:
***           < 70:  0 points
***          >= 70:  2 points
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
```

```
Starting iteration  4 ...
72 correct out of 100 (72.0%).
*** PASS: test_cases\q3\suicide.test (2 of 2 points)
***      72.0 correct (2 of 2 points)
***          Grading scheme:
***           < 70:  0 points
***          >= 70:  2 points

### Question q3: 4/4 ###
```

### 3.1.2. Código V-Final

```
def train(self, trainingData, trainingLabels, validationData,
    validationLabels):

    self.features = trainingData[0][0]['Stop'].keys()

    for iteration in range(self.max_iterations):
        print("Starting iteration ", iteration, "...")
        for i in range(len(trainingData)):

            # Calcular la clase predicha
            best_label = self.classify([trainingData[i]])[0]

            # Obtener el valor de clase real
            real_label = trainingLabels[i]

            # Si la clase predicha no es correcta --> Actualizar
                pesos
            if best_label != real_label:

                # Acercar a la clase real
                self.weights = self.weights + trainingData[i
                    ][0][real_label]

                # Alejar de la clase mal predicha
                self.weights = self.weights - trainingData[i
                    ][0][best_label]
```

```
Question q3
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
```

```
Starting iteration  3 ...
Starting iteration  4 ...
80 correct out of 100 (80.0%).
*** PASS: test_cases\q3\contest.test (2 of 2 points)
***      80.0 correct (2 of 2 points)
***          Grading scheme:
***           < 70:  0 points
***          >= 70:  2 points
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
72 correct out of 100 (72.0%).
*** PASS: test_cases\q3\suicide.test (2 of 2 points)
***      72.0 correct (2 of 2 points)
***          Grading scheme:
***           < 70:  0 points
***          >= 70:  2 points

### Question q3: 4/4 ###
```

#### 3.1.3. Problemas y dificultades encarados en el ejercicio

Más allá de familiarizarse con las estructuras de datos, la solución de este problema ha sido instantánea. Realmente, es aplicar el mismo algoritmo presentado en la pregunta 2 siendo cuidadoso con qué información se le pasa a cada parte del código.

Además, el uso de la función *classify()* facilita las cosas evitando problemas en el código y dejando una estructura bastante más limpia.

# 4. Pregunta 4: Diseño de características de *Pacman*

## 4.1. Descripción

Para que un algoritmo de clasificación pueda lidiar con los datos que se van a usar en el entrenamiento deben tener un formato específico. Las instancias deberán estar caracterizadas con diferentes atributos o *features*. (Ver figura 7)

El objetivo del *Perceptrón* en esta ocasión será establecer como de importante es una *feature* para inferir el valor de clase, esto es, debe asignar pesos a los diferentes atributos de entrada.

En esta pregunta se pide probar con diferentes caracterizaciones de las instancias para ver como influye la selección de atributos en el proceso de aprendizaje del *Perceptrón*. Sin embargo, la solución del problema viene dada, siendo la mejor alternativa emplear la distancia a la comida más cercana y la distancia al fantasma más cercano como atributos para los estados representados en las instancias.

Obviando que la solución se ha proporcionado, se ha procedido a probar con algunas alternativas para testar si existe alguna otra posibilidad de obtener la máxima puntuación. Entre otras cosas, se ha probado a contemplar también la distancia a la cápsula más cercana o a emplear la media de las distancias a los diferentes elementos. No obstante, no se han conseguido superar los resultados de la versión inicial del código.

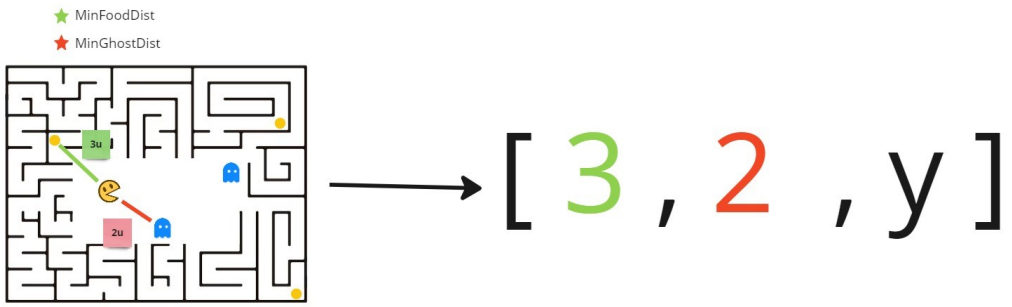


Figura 7: Vectorización de los estados

## 4.2. Código V1

```
def enhancedPacmanFeatures(state, action):
    """
    For each state, this function is called with each legal
        action.
    It should return a counter with { <feature name> : <feature
        value>, ... }
    """
    features = util.Counter()
    state = state.generateSuccessor(0, action)
    foods = state.getFood().asList()
    capsules = state.getCapsules()
    pac = state.getPacmanPosition()


    foodDistances = []  # Lista para almacenar las distancias a
        la comida
    ghostDistances = []  # Lista para almacenar las distancias a
         los fantasmas
    capsuleDistances = []  # Lista para almacenar las distancias
         a las c psulas

    for food in foods:
        d = util.manhattanDistance(food, pac)
        foodDistances.append(d)

    if foodDistances:  # Comprueba si hay distancias en la lista
        avgFoodD = sum(foodDistances) / len(foodDistances)  #
            Calcula la distancia media
        features["average-food-distance"] = 1.0 / avgFoodD
    else:
        features["average-food-distance"] = float('inf')

    for ghost in state.getGhostPositions():
        d = util.manhattanDistance(pac, ghost)
        ghostDistances.append(d)

    if ghostDistances:  # Comprueba si hay distancias en la
        lista
        avgGhostD = sum(ghostDistances) / len(ghostDistances)  #
             Calcula la distancia media
        features["average-ghost-distance"] = avgGhostD

    for capsule in capsules:
        d = util.manhattanDistance(pac, capsule)
        capsuleDistances.append(d)
```

```
    if capsuleDistances:  # Comprueba si hay distancias en la
        lista
        avgCapsuleD = sum(capsuleDistances) / len(
            capsuleDistances)  # Calcula la distancia media
        features["average-capsule-distance"] = avgCapsuleD

    return features
```

```
Question q4
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
79 correct out of 100 (79.0%).
*** FAIL: test_cases\q4\contest.test (0 of 2 points)
***      79.0 correct (0 of 2 points)
***          Grading scheme:
***           < 90:  0 points
***          >= 90:  2 points
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
88 correct out of 100 (88.0%).
*** PASS: test_cases\q4\suicide.test (2 of 2 points)
***      88.0 correct (2 of 2 points)
***          Grading scheme:
***           < 80:  0 points
***          >= 80:  2 points

### Question q4: 2/4 ###
```

## 4.3. Código V-Final

```
def enhancedPacmanFeatures(state, action):
    """
    For each state, this function is called with each legal
        action.
    It should return a counter with { <feature name> : <feature
        value>, ... }
    """
    features = util.Counter()
```

```
    state = state.generateSuccessor(0, action)
    foods = state.getFood().asList()
    capsules = state.getCapsules()
    pac = state.getPacmanPosition()


    minFoodD = float('inf')

    for food in foods:
        d = util.manhattanDistance(food, pac)
        minFoodD = min(d, minFoodD)

    if minFoodD != float('inf'):
        features["closest-food"] = 1.0 / minFoodD
    else:
        features["closest-food"] = float('inf')

    minGhostD = float('inf')

    for ghost in state.getGhostPositions():
        d = util.manhattanDistance(pac, ghost)
        minGhostD = min(d, minGhostD)

    features["closest-ghost"] = minGhostD

    return features
```

```
Question q4
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
95 correct out of 100 (95.0%).
*** PASS: test_cases\q4\contest.test (2 of 2 points)
***      95.0 correct (2 of 2 points)
***          Grading scheme:
***           < 90:  0 points
***          >= 90:  2 points
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
85 correct out of 100 (85.0%).
*** PASS: test_cases\q4\suicide.test (2 of 2 points)
***      85.0 correct (2 of 2 points)
```

```
***          Grading scheme:
***           < 80:  0 points
***          >= 80:  2 points

### Question q4: 4/4 ###
```

## 4.4. Problemas y dificultades encarados en el ejercicio

Esta tarea no ha supuesto mayor problema porque la solución venía dada. En vista de ello, se ha procedido a experimentar con diferentes alternativas. A pesar de no haber conseguido mejorar la solución dada, se ha experimentado y se ha comprendido la relevancia de hacer una buena elección de las *features* que describen las instancias.

La selección de unos buenos atributos permite describir fielmente las instancias, facilitando el entrenamiento del algoritmo de clasificación.

# Parte II

# 5. Pregunta 1: *Perceptrón*

## 5.1. Descripción

Este problema contiene algunas modificaciones respecto al *Perceptrón* básico. En este caso, la red neuronal está diseñada para admitir arquitecturas de varias capas, donde cada una de ellas realiza una operación lineal.

El *Perceptrón* se entrena hasta que se alcanza un mínimo o se estabiliza la función de pérdida, esta función mide la diferencia entre las predicciones y los datos reales.

El descenso del gradiente es un algoritmo cuyo objetivo es ajustar los parámetros del modelo para minimizar la función de pérdida, calculando el gradiente de la misma con respecto a los parámetros del modelo. Este gradiente indica la dirección en la que la pérdida aumenta, por lo que los parámetros se deben ajustar en dirección opuesta, para reducir la pérdida.

En este proyecto, el ajuste o la actualización se debe hacer por lotes de ejemplos o *mini-batches* de tamaño menor que el conjunto de entrenamiento, y el *Perceptrón* se actualiza después de procesar cada *mini-batch*.

El *learning rate* o tasa de aprendizaje es un hiperparámetro, es decir, un parámetro externo al modelo (no se aprende del conjunto de datos durante el entrenamiento), por lo que permite generar distintos modelos. El learning rate determina el tamaño de los pasos que se dan al ajustar los parámetros de un modelo durante el proceso de entrenamiento.

### 5.1.1. Código V1

```
    def run(self, x):

        return nn.DotProduct(self.w,x)


    def get_prediction(self, x):

        score=nn.as_scalar(self.run(x))
        return 1 if score>=0 else -1


    def train(self, dataset):

        notConverge=True
        while notConverge:
            notConverge=False
            for x,y in dataset.iterate_once(1):
```

```
                output=self.get_prediction(x)

                if output!=y:
                    notConverge=True
                    self.w.update(x,y)
```

```
Question q1
===========
*** q1) check_perceptron
Sanity checking perceptron...
Sanity checking perceptron weight updates...
Traceback (most recent call last):
  File "/home/nagore/Mahaigaina/TIA/LABO/LAB3-2/machinelearning/
     autograder.py", line 204, in main
    fn(tracker)
  File "/home/nagore/Mahaigaina/TIA/LABO/LAB3-2/machinelearning/
     autograder.py", line 363, in check_perceptron
    p.train(sanity_dataset)
  File "/home/nagore/Mahaigaina/TIA/LABO/LAB3-2/machinelearning/
     models.py", line 80, in train
    self.w.update(x,y)
  File "/home/nagore/Mahaigaina/TIA/LABO/LAB3-2/machinelearning/
     nn.py", line 53, in update
    assert isinstance(multiplier, (int, float)), (
AssertionError: Multiplier must be a Python scalar, instead has
   type 'Constant'


### Question q1: 2/6 ###

Finished at 11:34:21

Provisional grades
==================
Question q1: 2/6
----------------
Total: 2/6
```

#### 5.1.2. Código V-Final

```
    def run(self, x):

        return nn.DotProduct(self.w,x)
```

```
    def get_prediction(self, x):

        score=nn.as_scalar(self.run(x))
        return 1 if score>=0 else -1


    def train(self, dataset):

        notConverge=True
        while notConverge:
            notConverge=False
            for x,y in dataset.iterate_once(1):
                output=self.get_prediction(x)
                label=nn.as_scalar(y)
                if output!=label:
                    notConverge=True
                    self.w.update(x,label)
```

```
Question q1
===========
*** q1) check_perceptron
Sanity checking perceptron...
Sanity checking perceptron weight updates...
Sanity checking complete. Now training perceptron
*** PASS: check_perceptron

### Question q1: 6/6 ###

Finished at 11:37:52

Provisional grades
==================
Question q1: 6/6
------------------
Total: 6/6
```

### 5.1.3. Problemas y dificultades encarados en el ejercicio

La mayor dificultad encarada en este ejercicio ha sido interiorizar la manera en la que trabajar con un *Perceptrón*. Los demás problemas han sido meramente técnicos, como se puede ver entre las dos versiones de código del ejercicio propuesto.

# 6. Pregunta 2: Regresión no lineal

## 6.1. Descripción

Pese a que el *Perceptrón* suele ser utilizado para problemas de clasificación, también puede utilizarse para problemas de regresión. En estos, el *Perceptrón* se entrena para predecir un número, es decir, dado un valor $X$ el *Perceptrón* se entrena para aprender a obtener la misma $Y$ (predicción) que en la función original.

Para conseguir que los valores predichos y los reales coincidan, se debe minimizar la función de pérdida al cuadrado:

$$Error = \frac{1}{2N} \sum_{(x,y)} (y - f(x))^2$$

Además, la capa de entrada debe tener tantas neuronas como características tenga el conjunto de datos y la de salida debe tener una sola neurona, puesto que se predice un valor continuo.

### 6.1.1. Código V1

```
class RegressionModel(object):

    def __init__(self):
        salida_tama o=1
        entrada_tama o=1
        oculta_tama o=30
        self.batch_size = 100
        self.lr = -0.001
        self.numero_ocultas=3

        self.layers=[]

        # capa de entrada
        self.layers.append(nn.Parameter(entrada_tama o,
            oculta_tama o))
        self.layers.append(nn.Parameter(1, oculta_tama o))

        # capas ocultas
        for i in range(self.numero_ocultas):
            self.layers.append(nn.Parameter(oculta_tama o,
                oculta_tama o))
            self.layers.append(nn.Parameter(1, oculta_tama o))

        # capa de salida
        self.layers.append(nn.Parameter(oculta_tama o,
            salida_tama o))
```

```
        self.layers.append(nn.Parameter(1, salida_tama o))


    def run(self, x):

        # capa de entrada
        entrada = nn.Linear(x, self.layers[0])
        entrada = nn.AddBias(entrada, self.layers[1])

        # Capas ocultas
        for i in range(2, len(self.layers)-2, 2):
            oculta = nn.Linear(nn.ReLU(entrada), self.layers[i])
            oculta = nn.AddBias(oculta, self.layers[i + 1])
            entrada = oculta

        # Capa de salida
        salida = nn.Linear(nn.ReLU(entrada), self.layers[-2])
        salida = nn.AddBias(salida, self.layers[-1])

        return salida


    def get_loss(self, x, y):

        return nn.SquareLoss(self.run(x),y)


    def train(self, dataset):

        batch_size = self.batch_size
        total_loss = 100000

        while total_loss > 0.02:
            for x, y in dataset.iterate_once(batch_size):
                prediction=self.run(x)
                if prediction!=y:
                    total_loss=self.get_loss(x,y)
                    grad=nn.gradients(total_loss ,self.layers)
                    total_loss=nn.as_scalar(total_loss)
                    for i in range(0, len(self.layers), 2):
                        self.layers[i].update(grad[i], self.lr)
                        self.layers[i + 1].update(grad[i + 1],
                            self.lr)
```

```
Question q2
===========
*** q2) check_regression
```

```
Your final loss is: 0.018139
*** PASS: check_regression

### Question q2: 6/6 ###

Finished at 12:32:09

Provisional grades
==================
Question q2: 6/6
------------------
Total: 6/6
```

### 6.1.2. Código V-Final

```
class RegressionModel(object):

    def __init__(self):
        salida_tama o=1
        entrada_tama o=1
        oculta_tama o=30
        self.batch_size = 100
        self.lr = -0.001
        self.numero_ocultas=3

        self.layers=[]

        # capa de entrada
        self.layers.append(nn.Parameter(entrada_tama o,
            oculta_tama o))
        self.layers.append(nn.Parameter(1, oculta_tama o))

        # capas ocultas
        for i in range(self.numero_ocultas):
            self.layers.append(nn.Parameter(oculta_tama o,
                oculta_tama o))
            self.layers.append(nn.Parameter(1, oculta_tama o))

        # capa de salida
        self.layers.append(nn.Parameter(oculta_tama o,
            salida_tama o))
        self.layers.append(nn.Parameter(1, salida_tama o))


    def run(self, x):
```

```
        # capa de entrada
        entrada = nn.Linear(x, self.layers[0])
        entrada = nn.AddBias(entrada, self.layers[1])

        # Capas ocultas
        for i in range(2, len(self.layers)-2, 2):
            oculta = nn.Linear(nn.ReLU(entrada), self.layers[i])
            oculta = nn.AddBias(oculta, self.layers[i + 1])
            entrada = oculta

        # Capa de salida
        salida = nn.Linear(nn.ReLU(entrada), self.layers[-2])
        salida = nn.AddBias(salida, self.layers[-1])

        return salida


    def get_loss(self, x, y):

        return nn.SquareLoss(self.run(x),y)


    def train(self, dataset):

        batch_size = self.batch_size
        total_loss = 100000

        while total_loss > 0.02:
            for x, y in dataset.iterate_once(batch_size):
                prediction=self.run(x)
                if prediction!=y:
                    total_loss=self.get_loss(x,y)
                    grad=nn.gradients(total_loss,self.layers)
                    total_loss=nn.as_scalar(total_loss)
                    for i in range(0, len(self.layers), 2):
                        self.layers[i].update(grad[i], self.lr)
                        self.layers[i + 1].update(grad[i + 1],
                            self.lr)
```

```
Question q2
===========
*** q2) check_regression
Your final loss is: 0.018409
*** PASS: check_regression

### Question q2: 6/6 ###
```

```
Finished at 12:36:15

Provisional grades
==================
Question q2: 6/6
---------------
Total: 6/6
```

### 6.1.3. Problemas y dificultades encarados en el ejercicio

El mayor problema encarado en este ejercicio ha sido entender la manera en la que trabaja el *Perceptrón* en regresión. Además de esto, la elección de la arquitectura e hiperparámetros también ha supuesto un reto.

# 7. Pregunta 3: Clasificación de dígitos

## 7.1. Descripción

Rectified Linear Unit, o ReLU, es una función de activación que se emplea habitualmente en las arquitecturas de redes neuronales modernas para aportar no-linearidad. Se define como:

$$f(x) = max(0, x)$$

En este caso, añadir ReLUs a este ejercicio permite que el modelo aprenda patrones más complejos y no lineales en los datos. Sin ReLUs la red neuronal se reduciría a una combinación lineal de las capas de la misma.

### 7.1.1. Código V1

```
class DigitClassificationModel(object):

    def __init__(self):
        output_size = 10
        pixel_dim_size = 28
        pixel_vector_length = pixel_dim_size* pixel_dim_size
        entrada_tama o=784
        oculta_tama o=200
        self.batch_size = 300
        self.lr = -0.1
        self.numero_ocultas=3

        self.layers=[]

        # capa de entrada
        self.layers.append(nn.Parameter(entrada_tama o,
            oculta_tama o))
        self.layers.append(nn.Parameter(1, oculta_tama o))

        # capas ocultas
        for i in range(self.numero_ocultas):
            self.layers.append(nn.Parameter(oculta_tama o,
                oculta_tama o))
            self.layers.append(nn.Parameter(1, oculta_tama o))

        # capa de salida
        self.layers.append(nn.Parameter(oculta_tama o,
            output_size))
        self.layers.append(nn.Parameter(1, output_size))
```

```
    def run(self, x):
        # capa de entrada
        entrada = nn.Linear(x, self.layers[0])
        entrada = nn.AddBias(entrada, self.layers[1])

        # Capas ocultas
        for i in range(2, len(self.layers)-2, 2):
            oculta = nn.Linear(nn.ReLU(entrada), self.layers[i])
            oculta = nn.AddBias(oculta, self.layers[i + 1])
            entrada = oculta

        # Capa de salida
        salida = nn.Linear(nn.ReLU(entrada), self.layers[-2])
        salida = nn.AddBias(salida, self.layers[-1])

        return salida


    def get_loss(self, x, y):

        return nn.SoftmaxLoss(self.run(x), y)

    def train(self, dataset):

        while dataset.get_validation_accuracy() < 0.975:
            for x, y in dataset.iterate_once(self.batch_size):
                prediction=self.run(x)
                if prediction!=y:
                    total_loss=self.get_loss(x,y)
                    grad=nn.gradients(total_loss ,self.layers)
                    total_loss=nn.as_scalar(total_loss)
                    for i in range(0, len(self.layers), 2):
                        self.layers[i].update(grad[i], self.lr)
                        self.layers[i + 1].update(grad[i + 1],
                            self.lr)
```

```
Question q3
===========
*** q3) check_digit_classification
Your final test set accuracy is: 97.480000%
*** PASS: check_digit_classification

### Question q3: 6/6 ###

Finished at 17:11:40

Provisional grades
```

```
==================
Question q3: 6/6
------------------
Total: 6/6
```

### 7.1.2. Código V-Final

```
class DigitClassificationModel(object):

    def __init__(self):
        output_size = 10
        pixel_dim_size = 28
        pixel_vector_length = pixel_dim_size* pixel_dim_size
        entrada_tama o=784
        oculta_tama o=200
        self.batch_size = 300
        self.lr = -0.1
        self.numero_ocultas=3

        self.layers=[]

        # capa de entrada
        self.layers.append(nn.Parameter(entrada_tama o,
            oculta_tama o))
        self.layers.append(nn.Parameter(1, oculta_tama o))

        # capas ocultas
        for i in range(self.numero_ocultas):
            self.layers.append(nn.Parameter(oculta_tama o,
                oculta_tama o))
            self.layers.append(nn.Parameter(1, oculta_tama o))

        # capa de salida
        self.layers.append(nn.Parameter(oculta_tama o,
            output_size))
        self.layers.append(nn.Parameter(1, output_size))


    def run(self, x):
        # capa de entrada
        entrada = nn.Linear(x, self.layers[0])
        entrada = nn.AddBias(entrada, self.layers[1])

        # Capas ocultas
        for i in range(2, len(self.layers)-2, 2):
            oculta = nn.Linear(nn.ReLU(entrada), self.layers[i])
```

```
            oculta = nn.AddBias(oculta, self.layers[i + 1])
            entrada = oculta

        # Capa de salida
        salida = nn.Linear(nn.ReLU(entrada), self.layers[-2])
        salida = nn.AddBias(salida, self.layers[-1])

        return salida


    def get_loss(self, x, y):

        return nn.SoftmaxLoss(self.run(x), y)

    def train(self, dataset):

        while dataset.get_validation_accuracy() < 0.975:
            for x, y in dataset.iterate_once(self.batch_size):
                prediction=self.run(x)
                if prediction!=y:
                    total_loss=self.get_loss(x,y)
                    grad=nn.gradients(total_loss,self.layers)
                    total_loss=nn.as_scalar(total_loss)
                    for i in range(0, len(self.layers), 2):
                        self.layers[i].update(grad[i], self.lr)
                        self.layers[i + 1].update(grad[i + 1],
                            self.lr)
```

```
Question q3
===========
*** q3) check_digit_classification
Your final test set accuracy is: 97.480000%
*** PASS: check_digit_classification

### Question q3: 6/6 ###

Finished at 17:11:40

Provisional grades
==================
Question q3: 6/6
----------------
Total: 6/6
```

### 7.1.3. Problemas y dificultades encarados en el ejercicio

La dificultad de este ejercicio ha sido la elección de la arquitectura e hiperparámetros, puesto que el código no cambia significativamente respecto al ejercicio anterior.