BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

Técnicas de Inteligencia Artificial

Práctica 3 Pac-Man

2023-2024

...

Titulación:

Grado en Informática de Gestión y Sistemas de Información

...

4º Curso (1º Cuatrimestre)

...

Página de GitHub

Alan García, Álvaro Díez-Andino

2 de diciembre de 2023

Índice

1. Enunciado A

En este laboratorio se explora como realizar implementaciones de perceptrones y algoritmos de Machine Learning con el objetivo de aplicar los conocimientos vistos en clase en el juego de PAC-MAN. Sin embargo, para realizar una primera aproximación a la clasificación con perceptrones, se ha propuesto realizar la implementación de las puertas lógicas AND, OR y XOR en un python notebook.

1.1. AND, OR, XOR notebook

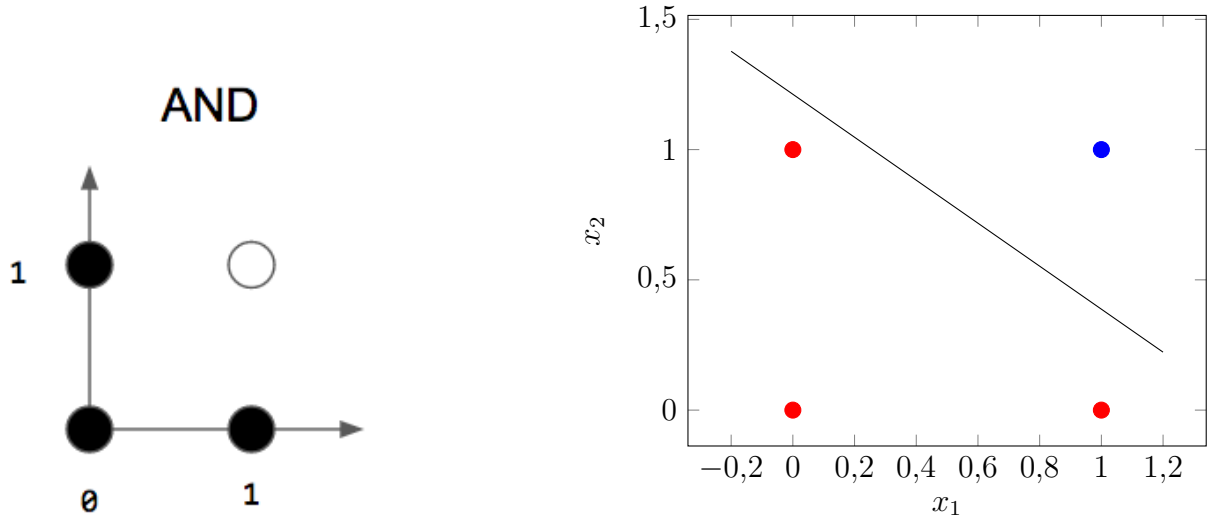
Como se ha mencionado, en esta práctica se pretende implementar un perceptrón como clasificador binario con la siguiente función de activación para las puertas lógicas mencionadas.

$$f(x) = \{1, si\ dist(x, w, bias) \geq 0; -1, si\ dist(x, w, bias) < 0\}$$

1.1.1. AND

La función de la puerta logica AND se puede visualizar en la figura 7 y, como se puede observar, la función es linealmente separable por lo que un perceptrón simple puede definir la recta que realiza la clasificación binaria para esta función.

De hecho, la práctica nos proporciona los pesos de la recta clasificadora, luego nuestra misión es realizar la implementación del perceptrón sin la necesidad de establecer un entrenamiento de los pesos. Finalmente, esta función se puede graficar como se muestra en la figura 7.



(a) Función AND

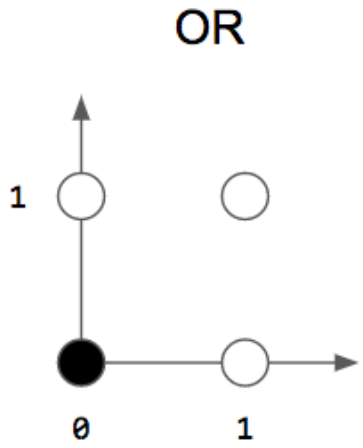
(b) Clasificación AND

Figura 1: AND

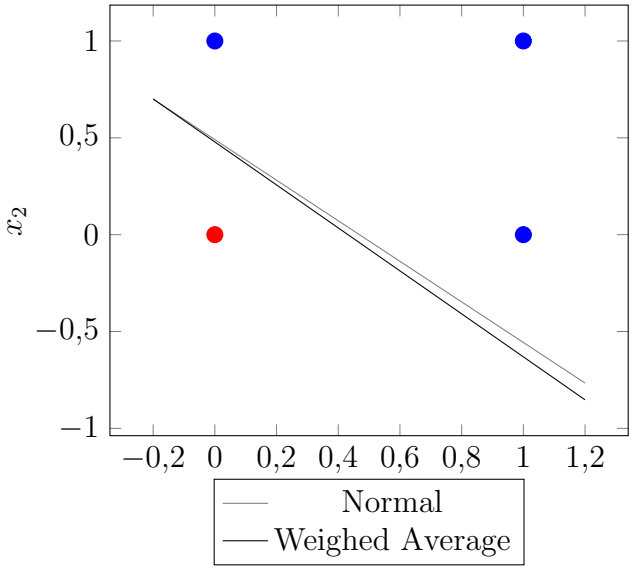
En conclusión, el uso de un perceptrón para la clasificación binaria resulta en definir una recta de tal forma que separe los conjuntos de datos.

1.1.2. OR

Para esta práctica, se pretende definir la función clasificadora para la puerta logica OR representada en la figura 2. Sin embargo, a diferencia del caso anterior, no disponemos de los pesos si no de un conjunto de datos de entrenamiento. Es por ello que tenemos que implementar la función de entrenamiento para ajustar la función clasificadora a nuestros datos de entrenamiento y luego comprobar la función con una serie de datos de test.



(a) Función OR



(b) Clasificación OR

Figura 2: OR

Hemos realizado dos implementaciones para el entrenamiento de los pesos. Ambas solo actualizan los pesos en el momento que se produce un error de clasificación (no ajusta las clasificaciones correctas), pero se diferencian en la forma de ajustarlos.

El primer método suma o resta a los pesos incorrectos los valores de cada una de las instancias mal clasificadas. Si los datos son linealmente separables, este método asegura la convergencia. Sin embargo, al producir cambios tan drásticos, puede producirse un alejamiento de la solución del problema en algún determinado momento, lo que resulta en un número elevado de epochs o iteraciones.

Para corregir esto, se ha propuesto un segundo método en el que al final de cada epoch se hace la media del nuevo vector de pesos con el anterior a la epoch. De esta manera podemos conseguir un entrenamiento con un ajuste más suave de los pesos y, en el caso general, conseguimos reducir el número de epochs. Sin embargo, en este caso el número de iteraciones es el mismo (figura 1), hecho que se contrasta con el caso de la puerta XOR (figura **??**).

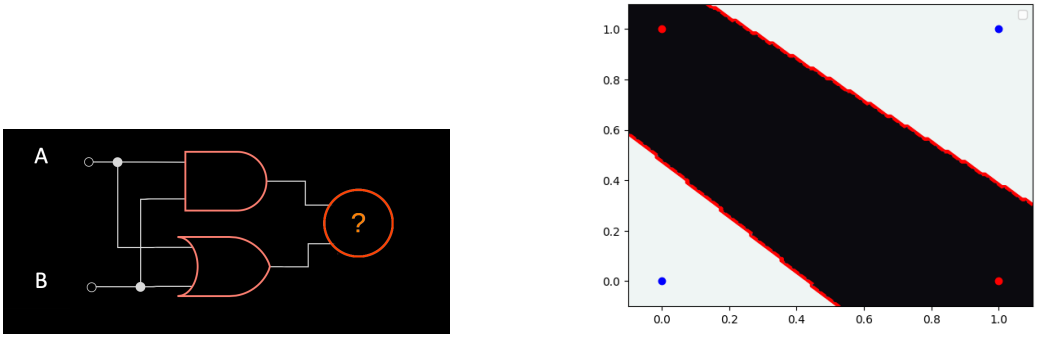
Normal	**Weighted Average**
2	2

Cuadro 1: Num Epoch OR

1.1.3. XOR

Sin nos fijamos en la función de la puerta XOR que se muestra en la figura 3 vemos que esta funcion no es linealmente separable. Para resolver este problema de clasificación necesitaríamos, como mínimo, un polinomio de grado 2 para poder graficar una función de clasificación que separe los conjuntos de datos. Sin embargo, en esta práctica hemos optado por construir la puerta XOR a partir de una puerta lógica OR y una puerta lógica AND.

De esta manera, el resultado de la salida XOR va a ser la aplicación de la función de activación previamente mencionada a la salida de un perceptrón que toma como entradas las puertas AND y OR de los datos x1 y x2.



(a) Función XOR (b) Clasificación XOR

Figura 3: XOR

En la figura 3.b se ha representado la función de salida de la implemetación del perceptrón XOR (zona negra) y, si prestamos atención, los límites (lineas rojas) corresponden con las dos rectas clasificadoras AND y OR.

En conclusión, en lugar de establecer un algoritmo de aprendizaje automático para modelizar una puerta XOR directamente, se ha optado por hacer un artificio en el que las entradas del perceptrón XOR son las salidas de los perceptrones AND y OR. Además, se ha hecho la comparativa entre un entrenamiento normal y un weigthed average como se puede visualizar en la siguiente figura.

Normal	**Weighted Average**
7	2

(a) Num Epoch XOR

Bias	**w0**	**w1**
0.384	-0.153	0.264

(b) Weights XOR

Figura 4: Resultados XOR

1.2. Algoritmo del Perceptrón para Clasificación Multiclase

El Perceptrón es un algoritmo de aprendizaje supervisado utilizado para la clasificación binaria. Aquí se presenta y explica el algoritmo del Perceptrón para clasificación multiclase:

Entradas:

- **Datos de entrenamiento:** $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$, donde $x^{(i)}$ es el vector de características y $y^{(i)}$ es la etiqueta de clase asociada.

Parámetros:

- **Pesos para cada clase:** $w^{(1)}, w^{(2)}, \ldots, w^{(k)}$, donde k es el número de clases.

Procedimiento:

1. Inicializar los pesos $w^{(1)}, w^{(2)}, \ldots, w^{(k)}$.
2. Para cada iteración:
   - Para cada ejemplo de entrenamiento $(x^{(i)}, y^{(i)})$:
     - Calcular la puntuación para cada clase utilizando $puntuación_j = w^{(j)} \cdot x^{(i)}$ para $j = 1, 2, \ldots, k$.
     - Predecir la clase $\hat{y} = argmax_j(puntuación_j)$.
     - Si $\hat{y}$ no es igual a $y^{(i)}$, actualizar los pesos de la clase correcta y la clase predicha:

$$w^{(y^{(i)})} \leftarrow w^{(y^{(i)})} + x^{(i)}$$

$$w^{(y^{(i)})} \leftarrow w^{(y^{(i)})} - x^{(i)}$$

3. Repetir el paso 2 hasta que se alcance un número máximo de iteraciones o hasta convergencia.

Clasificación:

- Dada una nueva instancia $x^{(new)}$, la clase se predice como $\hat{y} = argmax_j(w^{(j)} \cdot x^{(new)})$.

1.2.1. Cambios para Clasificación Multiclase

En el caso de la clasificación multiclase, durante la predicción, la clase asignada será aquella para la cual la puntuación correspondiente sea la más alta.

1.2.2. Problemas Experimentados

Durante la implementación del código, se encontraron dificultades menores.

Dificultad Principal: Familiarización con Datos y Uso dela clase `Counter`
La principal dificultad radicó en familiarizarse con la estructura y características de los datos de entrenamiento. Además, la utilización de la clase `Counter` para gestionar los cálculos de puntuación y la actualización de pesos agregó complejidad al entender cómo se realizaban las operaciones en el código.

1.3. Ejercicios del Clonado

1.3.1. Uso del Perceptrón para Hacer Ranking

El Perceptrón puede ser empleado para hacer ranking al considerar la puntuación asociada a cada posible acción y luego seleccionar la acción con la puntuación más alta. En el contexto de clonado, esto implica evaluar la puntuación de cada acción posible dadas las características del estado actual. La acción con la puntuación más alta se selecciona como la predicción del modelo.

1.3.2. Rasgos Considerados

En el código proporcionado, los rasgos considerados para el entrenamiento del Perceptrón se obtienen de los datos de entrenamiento:

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

Aquí, `self.features` contiene las claves de los rasgos asociados a la acción en el conjunto de datos de entrenamiento.

- En training data nos encontramos una lista de tuplas, por un lado las acciones posibles y por el otro un diccionario con el resultado, en este caso la comida, asociado a cada acción

1.3.3. Razonamiento del Perceptrón en el Clonado

El código implementa un Perceptrón para el problema de clonado, donde el objetivo es aprender a imitar el comportamiento de un experto. Aquí está el razonamiento detrás del Perceptrón en este contexto:

1.3.4. Inicialización de Pesos

```
self.weights = util.Counter()
```

Se inicializan los pesos del Perceptrón. Cada peso está asociado a un rasgo específico de una acción.

1.3.5. Selección de Rasgos

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

Los rasgos considerados para el entrenamiento se obtienen del primer ejemplo de entrenamiento. Estos rasgos representan las características relevantes de una acción, como la posición de pacman, o como en este caso, la comida faltante.

1.3.6. Entrenamiento del Perceptrón

```
for iteration in range(self.max_iterations):
    for i in range(len(trainingData)):
        train_act_dic, acc = trainingData[i]
        puntos = util.Counter()
        for x in acc:
            dato = train_act_dic[x]
            puntos[x] = dato.__mul__(self.weights)
        puntuacion = puntos.argMax()
        if puntuacion != trainingLabels[i]:
            self.weights += train_act_dic[trainingLabels[i]]
            self.weights -= train_act_dic[puntuacion]
```

- Se itera sobre el conjunto de entrenamiento para actualizar los pesos del Perceptrón.
- Para cada ejemplo de entrenamiento, se obtiene un diccionario (`train_act_dic`) que contiene las características de la acción y una lista de acciones disponibles (`acc`).

- Se calcula la puntuación de cada acción multiplicando las características por los pesos. La acción con la puntuación más alta se selecciona (`puntuacion`).
- Si la acción seleccionada no coincide con la etiqueta real (`trainingLabels`), se actualizan los pesos para penalizar la acción predicha incorrecta y recompensar la acción correcta.

1.3.7. Predicción del Perceptrón

- Durante la predicción, se calcula la puntuación de cada acción disponible dada una situación.
- La acción con la puntuación más alta se elige como la predicción del Perceptrón.

1.3.8. Problemas Enfrentados

Durante la implementación del código, se encontraron pocas dificultades:

- **Entendimiento de los datos**, la única dificultad implementando esta sección ha sido adaptarse a los datos ya que el resto era muy similar al anterior ejercicio. Una vez entendido el formato de los datos no ha supuesto mayor dificultad.

1.4. Diseño de características de Pacman

1.4.1. Ponderación de la distancia al alimento por la cantidad de comida restante:

Intenta ajustar la característica 'closest food' ponderando la distancia al alimento más cercano por la cantidad total de comida restante en el tablero. Por ejemplo:

features['closest food'] = $1.0 \frac{}{minD \times len(foods)}$

Los resultados son ligeramente inferiores a la versión ya implementada

1.4.2. Consideración del estado de miedo de los fantasmas:

Modifica la característica 'closest ghost' para tener en cuenta si el fantasma más cercano está asustado. Por ejemplo:

$if scared\_ghosts : features["closestghost"] = -minD \quad \#Moversehaciafantasmasasustados$

$$else: features["closestghost"] = minD \quad \#Evitar fantasmas no asustados$$

No se ha conseguido una mejoría frente al modelo ya implementado.

1.4.3. Inclusión de la cantidad de cápsulas de energía restantes:

Agrega una nueva característica que refleje la cantidad de cápsulas de energía restantes en el tablero. Por ejemplo:

$$features["numpowercapsules"] = len(state.getCapsules())$$

De nuevo, esta prueba tampoco ha resultado en una mejora.

2. Enunciado B

2.1. Pregunta Q1

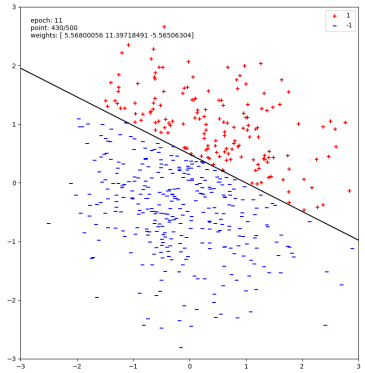
En esta practica se propone realizar la implementación de un clasificador binario con un perceptrón igual que en las implementaciones de las puertas lógicas (section1). A diferencia de la implementación previa, en esta ocasión disponemos de librerías para agilizar la implementación, por lo que el código resultante es más "simple"por nuestra parte.

2.1.1. Implementación

En este caso, al tratarse de una implementación muy similar a la de los perceptrones de las puertas lógicas, se ha conseguido la puntuación máxima con el autograder en la primera implementación.

```python
class PerceptronModel(object):
    def __init__(self, dimensions):
        self.w = nn.Parameter(1, dimensions)

    def get_weights(self):
        return self.w

    def run(self, x):
        return nn.DotProduct(x, self.w)

    def get_prediction(self, x):
        return 1 if ( nn.as_scalar(self.run(x)) >= 0 ) else -1

    def train(self, dataset):
        converge = False
        while not converge:
            # Inicio de epoch
            cambio = False
            for x, y in dataset.iterate_once(1):
                # x es el vector de entrada e y es el gold
                pred = self.get_prediction(x)
                y = nn.as_scalar(y)
                if pred != y:
                    cambio = True
                    self.w.update( x, y )

            # Fin de epoch
            if not cambio:
                converge = True
        # Fin de entrenamiento
        return
```

(a) PerceptronModel



(b) Autograder.py (6/6)

Figura 5: Primera iteración Q1

2.1.2. Problemas y Conclusiones

El mayor problema encontrado durante esta implementación ha sido adaptar las librerías para *numpy* > 1,16, con lo que hemos tenido que cambiar la expresión *"np.asscalar()"* por *"np.ndarray.item()"*.

2.2. Pregunta Q2

En esta practica se propone diseñar una red neuronal para representar una función coseno. Por ello, a parte del diseño de la arquitectura de la red, es necesario ajustar los parámetros de *learningRate* y *batch_size* para conseguir un entrenamiento adecuado. Además, es necesario incluir capas de *ReLus* para introducir no linearidades y poder aprender relaciones y funciones complejas.

2.2.1. Implementación

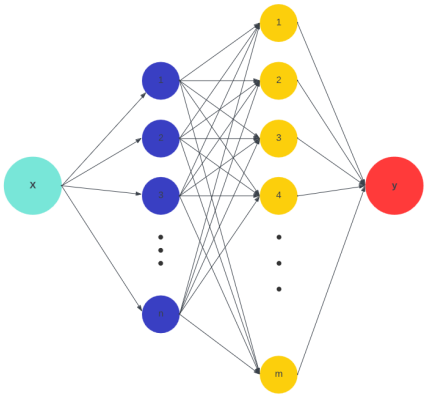
En la primera iteración del código, se definió una red neuronal de 1 capa oculta con un numero bajo de neuronas y un $Lr = -0{,}01$ y $batch = 1$. Sin embargo, después de varias pruebas con una sola capa oculta, no conseguíamos resultados aceptables e incluso el error comenzaba a aumentar. Tras varias revisiones, vimos un erro en la función *run(x)* en el que estábamos añadiendo una *ReLu* de más.

```
1  def run(self, x):
2      return nn.ReLU( nn.AddBias(nn.Linear(nn.ReLU( nn.AddBias(nn.Linear(nn.ReLU( nn.AddBias(nn.Linear(x, self.
       ↪ w0), self.b0) ), self.w1), self.b1) ), self.w2), self.b2) )
```

Una vez solucionado este error, volvimos a probar con una arquitectura de una sola capa y varias configuraciones de los hiperparámetros. Finalmente, añadimos una segunda capa oculta y los resultados comenzaron a mejorar significativamente y descubrimos que las arquitecturas con más neuronas en la segunda capa oculta que en la primera producen mejores resultados. De esta forma, logramos la implementación mostrada en la figura 6.

```python
class RegressionModel(object):
    def __init__(self):
        self.batch_size = 20
        self.lr = -0.01
        self.w0 = nn.Parameter(1, 12)
        self.b0 = nn.Parameter(1, 12)

        self.w1 = nn.Parameter(12, 50)
        self.b1 = nn.Parameter(1, 50)

        self.w2 = nn.Parameter(50, 1)
        self.b2 = nn.Parameter(1, 1)

    def run(self, x):
        o1 = nn.Linear(x, self.w0)
        o1 = nn.AddBias(o1, self.b0)
        o1 = nn.ReLU( o1 )

        o2 = nn.Linear(o1, self.w1)
        o2 = nn.AddBias(o2, self.b1)
        o2 = nn.ReLU( o2 )

        o3 = nn.Linear(o2, self.w2)
        o3 = nn.AddBias(o3, self.b2)
        return o3

    def get_loss(self, x, y):
        return nn.SquareLoss(self.run(x), y)

    def train(self, dataset):
        total_loss = 100000
        while total_loss > 0.02:
            t_loss = 0
            w = 0
            for x, y in dataset.iterate_once(self.batch_size):
                loss = self.get_loss(x, y)
                gradientes = nn.gradients(loss, [self.w0, self.w1, self.w2])

                # Actualizacion
                for i, wi in enumerate([self.w0, self.w1, self.w2]):
                    wi.update(gradientes[i], self.lr)

                w += 1
                t_loss += loss.data

            total_loss = t_loss / w
            print(total_loss)

        # Fin de entrenamiento
```

(a) RegressionModel

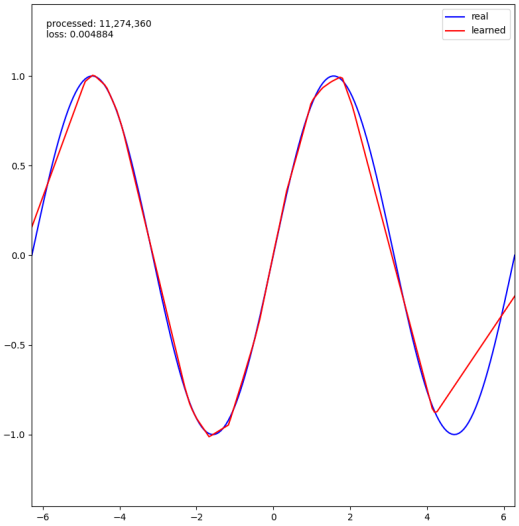


(b) Arquitectura de la red ($n = 12$ y $m = 50$)

Figura 6: Iteración final

2.2.2. Resultados

El resultado final se puede visualizar en las siguientes gráficas, habiendo conseguido un 6/6 en *autograder.py*.



(a) Regresión a sin(x)

```
Your final loss is: 0.003230
*** PASS: check_regression

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

Finished at 0:29:22

Provisional grades
==================
Question q2: 6/6
------------------
Total: 6/6
ag6154lk@msidealan:~/IngInfor/TIA/Labo_3/machinelearning$
```

(b) Autograder.py

Figura 7: Primera iteración Q1

2.3. Pregunta Q3

En esta práctica se propone diseñar y entrenar una red neuronal para clasificar imágenes de dígitos numéricos entrenando la red con el dataset *MNIST*.

Para el desarrollo de esta cuestión, se ha seguido la misma metodología que en la pregunta anterior: se ha empezado por una red simple de una capa oculta y se ha evaluado su desempeño con distintos hiperparámetros y tamaños de red. Finalmente, tras observar que con una sola capa oculta la red no era capaz de resolver el problema, se optó por añadir una segunda capa. Los resultados mejoraron significativamente y la resolución final consiste en una red en la que la primera capa oculta tiene más neuronas que la segunda (una especie de codificador).

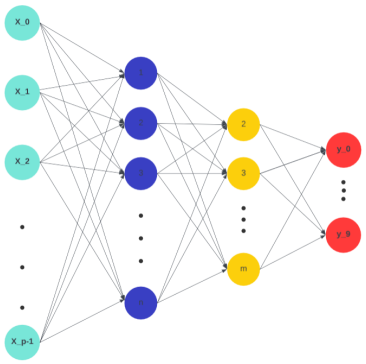
2.3.1. Implementación

Por ello, la arquitectura final consiste en una entrada de 784 parámetros (número de pixeles de las imágenes); una primera capa oculta de 256 neuronas; una segunda capa de 128 neuronas y una salida de 10 clases. Esta arquitectura se muestra en la figura 8b.

En cuanto a las versiones de código, se ha partido del código final de la pregunta anterior. Lo único que ha variado han sido las pruebas de las arquitecturas, por lo que la versión inicial solo varía de la final en el número de neuronas de la red.

```python
class DigitClassificationModel(object):
    def __init__(self):
        output_size = 10
        pixel_dim_size = 784
        pixel_vector_length = pixel_dim_size* pixel_dim_size
        # La entrada es de 1x784 ya que hay 784 pixeles

        self.batch_size = 20
        self.lr = -0.01

        n = 256
        m = 128

        self.w0 = nn.Parameter(pixel_dim_size, n)
        self.b0 = nn.Parameter(1, n)

        self.w1 = nn.Parameter(n, m)
        self.b1 = nn.Parameter(1, m)

        self.w2 = nn.Parameter(m, output_size)
        self.b2 = nn.Parameter(1, output_size)

    def run(self, x):
        o1 = nn.Linear(x, self.w0)
        o1 = nn.AddBias(o1, self.b0)
        o1 = nn.ReLU( o1 )

        o2 = nn.Linear(o1, self.w1)
        o2 = nn.AddBias(o2, self.b1)
        o2 = nn.ReLU( o2 )

        o3 = nn.Linear(o2, self.w2)
        o3 = nn.AddBias(o3, self.b2)

        return o3

    def get_loss(self, x, y):
        return nn.SoftmaxLoss(self.run(x), y)

    def train(self, dataset):
        while dataset.get_validation_accuracy() < 0.97:
            for x, y in dataset.iterate_once(self.batch_size):
                # x es el vector de entrada e y es el gold
                loss = self.get_loss(x, y)
                gradientes = nn.gradients(loss, [self.w0, self.w1, self.w2])

                # Actualizacin
                for i, wi in enumerate([self.w0, self.w1, self.w2]):
                    wi.update(gradientes[i], self.lr)

                i += 1
```

(a) DigitClassificationModel

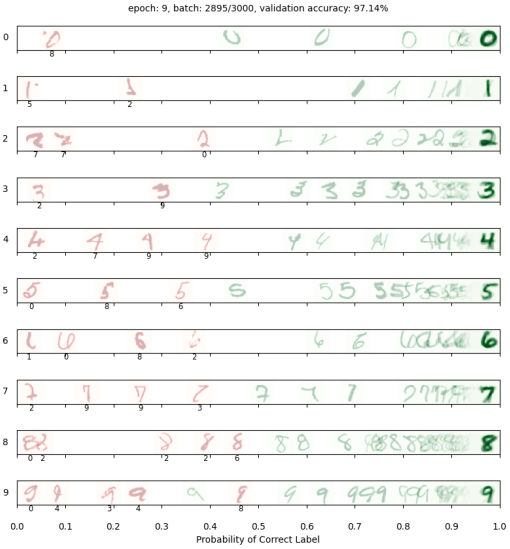


(b) Arquitectura clasificación numérica

Figura 8: Implementación final

2.3.2. Resultados

El resultado final se puede obserbar en las siguientes gráficas:



(a) Clasificación

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

Finished at 22:43:26

Provisional grades
==================
Question q3: 6/6
------------------
Total: 6/6
ag6154lk@msidealan:~/IngInfor/TIA/Labo_3/machinelearning$
```

(b) Autograder.py

Figura 9: Resultados clasificación dígitos