GLOBAL CAMPUS
NEBRIJA

PRACTICA 4

INTELIGENCIA ARTIFICIAL

José María Fernández Gómez

Question 1 (6 points): Perceptron

----------CODIGO----------

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


def get_prediction(self, x):

        floatNum = nn.as_scalar(self.run(x))
        return (floatNum >= 0) - (floatNum < 0)


def train(self, dataset):
        iterations = 1
        checker = True
        while checker:
                checker = False
                for x, y in dataset.iterate_once(iterations):
                        result = self.get_prediction(x)
                        if result != nn.as_scalar(y):
                                self.w.update(nn.Constant(nn.as_scalar(y)*x.data), 1)
                                checker = True
```

----------END CODIGO----------

En este apartado se comienza actualizando el metodo run que calcula el producta vectorial del peso del propio objeto nn con el vector de entrada, despues se completa el metodo para sacar predicciones, este metodo solo devuelve 1 o –1 si el vector que se le pasa tiene un producto vectorial positivo con el vector de sus pesos. Y por último entrenamos el perceptron, el cual recorre todos los datos y actualiza los vectores de pesos de los datos que considere que estan mal clasificados, hace esto hasta que no detecta mas errores y la flag de checkeo se maniene verdadera.

Pregunta 2 (6 puntos): Regresión no lineal

----------CODIGO----------

```
def __init__(self):
self.lr = .01
self.w1 = nn.Parameter(1, 128)
self.b1 = nn.Parameter(1, 128)
self.w2 = nn.Parameter(128, 64)
self.b2 = nn.Parameter(1, 64)
self.w3 = nn.Parameter(64, 1)
self.b3 = nn.Parameter(1, 1)
self.params = [self.w1, self.b1, self.w2, self.b2, self.w3, self.b3]

def run(self, x):
c_1 = nn.ReLU(nn.AddBias(nn.Linear(x, self.w1),self.b1))
c_2 = nn.ReLU(nn.AddBias(nn.Linear(c_1, self.w2),self.b2))
c_fin = nn.AddBias(nn.Linear(c_2, self.w3),self.b3)
return c_fin


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


def train(self, dataset):

        batch_size = 10
        loss = 99999999999999999999999
        while loss >= .015:
                for x, y in dataset.iterate_once(batch_size):
                        loss = self.get_loss(x, y)
                        grads = nn.gradients(loss, self.params)
                        loss = nn.as_scalar(loss)
                        for i in range(len(self.params)):
                                self.params[i].update(grads[i], -self.lr)
```

----------END CODIGO----------

En este apartado se ha programado la red para que la red aproxime el valor de una función trigonométrica dada, el funcionamiento del entrenamiento es similar al apartado anterior pero toma en cuenta la posible pérdida que viene dada por el método get_loss, también mostrado, este método aproxima la pérdida con una función de error cuadrática que viene ya dada. Y por último el modelo viene dado por el método run, que crea ese modelo y devuelve las predicciones en un nodo de tamaño batch_sizex1, en este caso hemos cogido 10 iteraciones para que el modelo trabaje por lo que ese nodo es de 10x1. Después los pesos se han inicializado con una forma de 1 x128 para la primera capa, de 64x128 y de 64x1 para el nodo final de predicciones.A cada capa se le pondera por el peso de la siguiente y se le calcula el sesgo con los definidos en init. La diferencia más relevante con respecto al perceptrón es la menara con la que actualizamos los parámetros usando la función gradiente de estos mismos con la función de pérdidas.

Pregunta 3 (6 puntos): Clasificación de dígitos

----------CODIGO----------

```
def run(self, x):
        c_1 = nn.ReLU(nn.AddBias(nn.Linear(x, self.w1),self.b1))
        c_2 = nn.ReLU(nn.AddBias(nn.Linear(c_1, self.w2),self.b2))
        c_3 = nn.ReLU(nn.AddBias(nn.Linear(c_2, self.w3),self.b3))
        c_fin = nn.AddBias(nn.Linear(c_3, self.w4),self.b4)
        return c_fin

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

def train(self, dataset):
        batch_size = 100
        loss = 999999999999999999999999
        valid_acc = 0
        while valid_acc < .98:
                for x, y in dataset.iterate_once(batch_size):
                        loss = self.get_loss(x, y)
                        grads = nn.gradients(loss, self.params)
                        loss = nn.as_scalar(loss)
                        for i in range(len(self.params)):
                                self.params[i].update(grads[i], -self.lr)
                                valid_acc = dataset.get_validation_accuracy()
```

----------END CODIGO----------

Este ejercicio es muy similar al anterior en cuanto al init y al run se refiere salvo porque en el vector de pesos de la primera capa se le da un tamaño de 748 para poder almacenar cada uno de los caracteres y la salida del del método run tiene un tamaño de 1 x 10 ya que con este vector expresamos la predicción de que numero es el que estamos reconociendo. También cambiamos la función para calcular la pérdida, en vez de ser de error cuadrático en esta ocasión es SoftMax, que es una función proporcionada dedicada al proceso de clasificación, que es lo que estamos haciendo. Por último la función de entrenamiento vuelve a ser bastante similar a las últimas vistas pero en esta ocasión también se encarga comprobar la precisión con la que estamos comprobando.

Pregunta 4 (7 puntos): Identificación lingüística

----------CODIGO----------

```
def __init__(self):
        self.lr = .1
        self.initial_w = nn.Parameter(self.num_chars, 256)
        self.initial_b = nn.Parameter(1, 256)
        self.x_w = nn.Parameter(self.num_chars, 256)
        self.h_w = nn.Parameter(256, 256)
        self.b = nn.Parameter(1, 256)
        self.output_w = nn.Parameter(256, len(self.languages))
        self.output_b = nn.Parameter(1, len(self.languages))
        self.params = [self.initial_w, self.initial_b, self.x_w, self.h_w,self.b, self.output_w,
        self.output_b]

def run(self, xs):
        h_i = nn.ReLU(nn.AddBias(nn.Linear(xs[0], self.initial_w), self.initial_b))
        for char in xs[1:]:
                h_i = nn.ReLU(nn.AddBias(nn.Add(nn.Linear(char, self.x_w),nn.Linear(h_i, self.h_w)),
                self.b))
        c_fin = nn.AddBias(nn.Linear(h_i, self.output_w), self.output_b)
        return c_fin

def get_loss(self, xs, y):
        y_predictions = self.run(xs)
        return nn.SoftmaxLoss(y_predictions, y)

def train(self, dataset):
        batch_size = 100
        loss = 9999999999999999999999999999999
        valid_acc = 0
        while valid_acc < .85:
                for x, y in dataset.iterate_once(batch_size):
                        loss = self.get_loss(x, y)
                        grads = nn.gradients(loss, self.params)
                        loss = nn.as_scalar(loss)
                        for i in range(len(self.params)):
                                self.params[i].update(grads[i], -self.lr)
                                valid_acc = dataset.get_validation_accuracy()
```

----------END CODIGO----------

Por último tenemos un caso que difiere en el anterior en el numero de elementos a reconocer y el tamaño de estos, el cual puede ser variable, esto lo vemos en el init cuando la matriz de pesos inicial depende del tamaño de caracteres que tenga la palabra a analizar y despues se declaran las variables x y h iniciales para que pueda comenzar con algun valor la funcion iterativa para el calculo de las h, que es lo que se muestra en la función run (Se ha tomado referencia en este apartado del trabajo realizado por zhiming-xu refrencia: https://github.com/zhiming-xu/CS188/blob/master/p5-ml/models.py) La parte de la función run no la he terminado de entender, ya que tengo dudas sobre el funcionamiento iterativo de esta y que va calculando en cada iteración. El modelo de entrenamiento es el mismo que en los apartados anteriores.