Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 2

Redes Neuronales Artificiales

Integrantes: Lukas Pavez
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Ayudantes: Francisco Leiva C.
Gabriel Andrés Azócar Cárcamo
Giovanni Pais L.
Nicolas Cruz Brunet

Fecha de entrega: 14 de Abril de 2019
Santiago, Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

Se tiene como objetivo implementar un clasificador de fallas de motores basado en análisis de señales de corriente. Se dispone de un set de datos entregados en la tarea que tiene datos de las 48 características de la señal de corriente, donde las 48 características pueden representar una entre 11 clases.

Para esto se quiere entrenar una red neuronal artificial utilizando tensorflow. En la red se utilizará solo una capa oculta, y se estudiará el comportamiento del entrenamiento cuando se cambia el número de neuronas de la capa oculta y su función de no-linearidad (sigmoide y ReLU).
Para el entrenamiento de la red se utilizara conjunto de entrenamiento y validación, y luego se evaluará la red cambiando la no-linearidad de la capa oculta con el conjunto de prueba.

2. Marco Teórico

Dicho de forma general, una Red Neuronal Artificial es una maquina que modela la forma en que el cerebro humano realiza una tarea particular, esta máquina es una interconexión de muchas unidades de procesamiento llamadas neuronas, que cada una realiza una operación simple, además de tener un número de entradas y una salida, cada neurona esta conectada a otra mediante un link sináptico, donde cada link está caracterizado por un peso, donde se toma el valor de la salida de la neurona, se multiplica por el peso y se lleva a la entrada de otra neurona, el peso representa la fuerza de la conexión entre dos neuronas. Las redes neuronales se dividen en capas, donde cada capa tiene un número de neuronas, y las salidas de las neuronas están conectadas a la siguiente capa. Primero se tiene la capa de entrada, donde cada neurona representa un valor de la señal de entrada, luego de esta capa se puede tener 0 o mas capas ocultas, el término *oculta* viene de que estas capas no son vistas de forma directa por el input ni el output de la red, y tienen la función de intervenir el input y output de una manera útil. También, para cada neurona se realiza una combinación lineal de las entradas y los pesos, y luego se suman, a esta suma también se le agrega un bias (también llamado offset), que tiene la función de aumentar o disminuir el total de la suma, luego, ese resultado se pasa a una función de activación, que tiene la función de limitar la amplitud del output de una neurona, se puede ver el modelo de una neurona en la Figura 1.

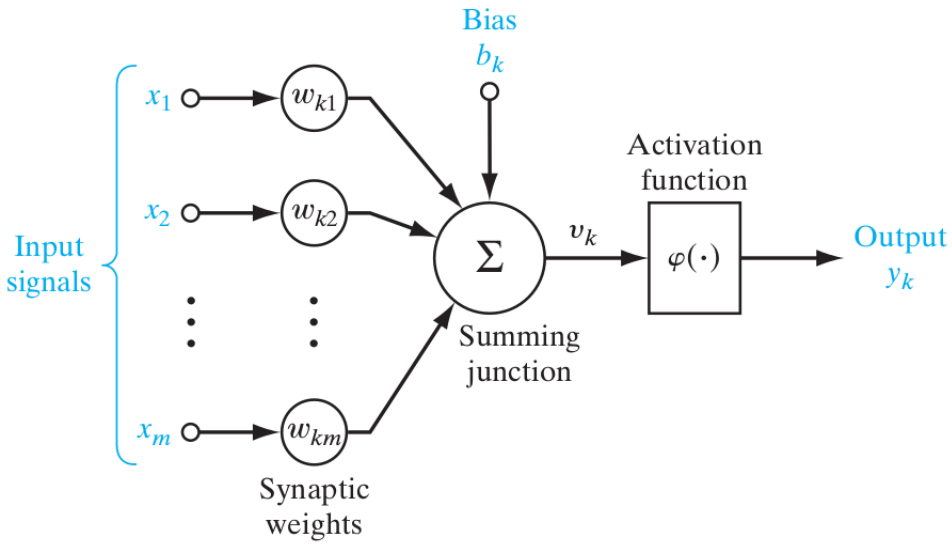

Figura 1: Haykin S, Neural Networks and Learning Machines, Third Edition, *Nonlinear model of a neuron, labeled k*

Para entrenar una red neuronal utilizando aprendizaje supervisado, se tiene un conjunto de entrenamiento que contiene diferentes sets de características y la clase a la que pertenecen, entonces para el entrenamiento se entregan las características a la capa de entrada, y luego se compara el output de la red con la clase de las características entregadas, luego se modifican los pesos utilizando back-propagation, luego se toma otro set de características del conjunto y se vuelven a modificar los pesos, y así hasta utilizar todos los sets de características, al momento utilizar todo el conjunto de entrenamiento se llama una época, luego de cada época se calcula el error de la red, y se agregan épocas para disminuirlo. Si se tienen muchas épocas se puede tener un sobreajuste de los datos, lo que significa que el error es muy pequeño, pero solo con el conjunto de entrenamiento, por lo que podría clasificar de forma correcta cualquier set de características del conjunto de entrenamiento, pero si se entrega otro set que no sea del conjunto, puede tener un error mucho mayor, ya que la red solo clasificaría de forma correcta los sets del conjunto de entrenamiento. Para evitar esto, se utiliza el conjunto de validación, donde por cada época el error va disminuyendo, pero se tiene un sobreajuste del conjunto de entrenamiento cuando el error del conjunto de validación comienza a aumentar.
Otro método para entrenar es en vez de ir modificando los pesos por cada set de características, se se guarda el valor de modificación y se sigue con el siguiente set, hasta utilizar un número determinado de sets, luego se aplican todas las modificaciones, el conjunto de sets de características utilizado se denomina batch, el total de sets del conjunto de entrenamiento se divide en un número de batchs, y las modificaciones se aplican después de utilizar todo un batch, y después de utilizar todos los batchs se tiene una época.

Al usar sólo una capa oculta, se debe elegir el número de neuronas que va a contener, para esto existen distintas “Rules of Thumb”, a continuación se describirán 3 reglas encontradas:

- regla basada en mejorar la convergencia: según esta regla, deben haber más neuronas en la

capa oculta que en la capa de entrada.

- regla basada en el tamaño de la capa de entrada y salida: el numero de neuronas en la capa oculta es (neuronas en capa de entrada + neuronas en capa de salida)*2/3
- regla basada en componentes principales: en la capa oculta deben haber el numero de neuronas necesarias para capturar el 70 %-90 % de la varianza del conjunto de entrenamiento

Para esta tarea se utilizó la regla basada en el tamaño de la capa de entrada y salida, ya que es más fácil obtener el número de neuronas mediante una fórmula que utilizar alguna de las otras opciones.

Para conocer la calidad de un clasificador se usa el *accuracy*, que mide que tan cerca de lo correcto está el valor entregado por el clasificador. El valor del accuracy se puede obtener mediante la matriz de confusión, que es una tabla que muestra el desempeño de la red, con respecto a cuantas clases clasificó correctamente, y cuantas clases clasificó como otra clase. La matriz se construye teniendo en cuenta que cada fila representa las instancias de las clases de la realidad, y cada columna son las instancias de la clase que entrega el clasificador, por lo que cada elemento (i, j) de la matriz representa que el clasificador dijo que el set de input corresponde a la clase j, y en verdad es de la clase i. En el caso de la diagonal de la matriz, se tiene que todos los elementos corresponden a una casilla (i, i) de la matriz, lo que significa que el clasificador dijo que era de clase i y en realidad es de clase i. Para calcular el accuracy se debe sumar la diagonal de la matriz y dividirla por el total de datos, osea todos las clasificaciones correctas sobre el total de las pruebas realizadas, esto es otra forma de decir que el accuracy es el rate de respuestas correctas entregadas por el clasificador. Para normalizar la matriz, se debe hacer por fila, ya que así se puede obtener el rate de cuantas predicciones correctas hizo el clasificador.

3. Desarrollo

3.1. División de la base de datos

Para dividir la base de datos de forma proporcional se creó la función split_data, que recibe los datos a separar y la proporción del set de training (por default es 0.8) y retorna el set de training y el de validación. El código es el siguiente:

```
def split_data(data, train_size=0.8):
    size = len(data[0])  # number of characteristics + class number
    n_classes = int(max(data[:, size-1]))  # number of classes
    classes = [np.zeros(size)]*n_classes  # each class start with a array of 0s (for vstack function)
    for x in data:
        classes[int(x[size-1])-1] = np.vstack((classes[int(x[size-1])-1], x))  # example: if x belong to class
     ↪  1, x is stored in classes[0]

    train = [np.zeros(size)]  # the set start with a array of 0s (for vstack function)
    validation = [np.zeros(size)]  # the set start with a array of 0s (for vstack function)
    for i in range(n_classes):
        classes[i] = classes[i][1:]  # delete the first row of each class matrix (the 0s row for vstack)
        np.random.shuffle(classes[i])  # shuffle data
        train_len = int(len(classes[i]) * train_size)  # calculate size of train set
        train = np.vstack((train, np.array(classes[i][:train_len])))  # append a new class matrix to the
     ↪ train matrix
        validation = np.vstack((validation, np.array(classes[i][train_len:])))  # append a new class
     ↪ matrix to the validation matrix

    train = train[1:]  # delete first row
    validation = validation[1:]  # delete first row
    np.random.shuffle(train)  # the data is ordered by class
    np.random.shuffle(validation)
    return train, validation
```

La función parte calculando el tamaño del conjunto (numero de características + la clase), luego calcula el número de clases calculando el máximo de la columna que representa la clase (se asume que hay al menos un ejemplo de cada clase), luego se crea una lista classes donde cada elemento va a ser las filas de una sola clase, por ejemplo classes[1] va a ser la matriz con todas las características de la clase 1, la lista empieza con una lista de ceros para cada clases para usar la función vstack de numpy, y esta función necesita que todas las filas sean del mismo largo, por lo que no se puede empezar de una fila vacía, luego se recorre todo el conjunto recibido y se va separando en la lista classes, luego se inician train y validation en 0, y se le van agregando una parte de cada matriz de la lista classes (a train se le agrega el 80 % y a validation el 20 %), teniendo en cuenta de que antes se les debe borrar la primera fila que estaba iniciada en 0, y después se les hace un shuffle. Después de hacer esto con todas las matrices dentro de la lista classes, se les borra la primera fila a train y validation (que estaba iniciada en 0 para usar la función vstack) y se les hace un shuffle, finalmente se entrega train y validation.
Este método los separa de forma proporcional, ya que separa todas las clases, a cada clase le saca un porcentaje para testing y el resto para validación, y después las junta en las 2 listas, donde se mantiene la proporción de cada clase que había al principio.

Después de separar los conjuntos, se separa la columna que representa la clase de la matriz de características, esto se hace en la función separate_characteristics_and_class, que recibe los datos, y un objeto label binarizer, esto es para hacer la representación one-hot encoding de las clases. La función deja la matriz de características en la variable data_x y las clases en data_y. El código es el siguiente:

```
def separate_characteristics_and_class(data, label_binarizer):
    nc = data.shape[1] - 1
    data_x = data[:, :nc]  # delete last column (class number column)
    data_y = data[:, nc] - 1  # class column
    data_y = data_y.astype(int)  # class column, each value as int

    data_y = label_binarizer.transform(data_y).astype(float)  # one-hot encoding representation

    return data_x, data_y
```

Se creó una función para cargar y separar los datos llamada load_data, que recibe un seed, esta función parte por setear los randoms con el seed, luego carga los archivos para training-validación y testing, separa el conjunto de training-validación con la función split_data, después crea el label_binarizer y llama a la función separate_characteristics_and_class para separar los conjuntos de training, validación y testing, finalmente entrega un diccionario con todos los conjuntos obtenidos. El código es el siguiente:

```
def load_data(seed):
    random_state = seed
    np.random.seed(random_state)
    tf.set_random_seed(random_state)

    D = np.loadtxt('sensorless_tarea2_train.txt', delimiter=',')  # load train and validation data
    T = np.loadtxt('sensorless_tarea2_test.txt', delimiter=',')  # load test data

    train, validation = split_data(D)  # split train and validation sets
    nc = D.shape[1] - 1

    label_binarizer = LabelBinarizer()
    label_binarizer.fit(range(int(max(D[:, nc]))))
    x_train, y_train = separate_characteristics_and_class(train, label_binarizer)
    x_valid, y_valid = separate_characteristics_and_class(validation, label_binarizer)
    x_test, y_test = separate_characteristics_and_class(T, label_binarizer)
    return {"x_train": x_train, "y_train": y_train, "x_valid": x_valid, "y_valid": y_valid, "x_test":
     ↪ x_test, "y_test": y_test}
```

3.2. Entrenamiento

Para entrenar la red, se comienza por definir el número de input, que va a ser el número de características, el de outputs, que es el número de clases, y el número de neuronas en la capa oculta, que, por lo dicho en el marco teórico, se utiliza 2/3 de la suma de input y output, luego se llama a la función create_and_train_perceptron, que parte por definir los pesos y los bias con números aleatorios, luego define variables de tensorlfow de tipo placeholder (x, y, keep_prob), y con todo eso crea un perceptrón (las predicciones) con la función multilayer_perceptron, que recibe x, los pesos, los bias, keep_prob y un bool que indica si se usa sigmoid o relu, esta función retorna la capa de salida de la red. Después de crear el perceptrón crea la función de costos y el optimizador que usa el algoritmo Adam, con todo esto ahora comienza el entrenamiento de la red, para esto llama a la función train_perceptron, que recibe x, y, las predicciones, la función de costos, el optimizador, el set de entrenamiento, el set a evaluar (que puede ser el de validación o el de testing), y keep_prob. Esta función parte definiendo el número de épocas y el tamaño de cada batch, luego crea el grafo de la red para calcular las predicciones, esto en la variable correct_prediction, donde compara el máximo de la salida obtenida y lo compara con la clase, para luego comenzar a entrenar, parte iniciando una lista vacía para ir guardando los valores del accuracy en cada momento del entrenamiento, también obtiene el tiempo de inicio con la librería time, luego, para cada época, inicia un costo en 0, y después comienza a sumarle el costo de cada batch, después de de utilizar todos los batch, termina una época y se guarda el valor de accuracy, y así hasta que se acaban las épocas, después la función calcula el tiempo de entrenamiento con la diferencia de el tiempo actual y el tiempo obtenido al principio, y retorna un diccionario con el tiempo, la matriz de confusión y la lista de accuracy. El código de las funciones es el siguiente:

```
1  def multilayer_perceptron(x, weights, biases, keep_prob, sigmoid=1):
2      layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
3      if sigmoid:
4          layer_1 = tf.nn.sigmoid(layer_1)
5      else:
6          layer_1 = tf.nn.relu(layer_1)
7      layer_1 = tf.nn.dropout(layer_1, keep_prob)
8      out_layer = tf.matmul(layer_1, weights['out']) + biases['out']
9      return out_layer
10
11
12 def train_perceptron(x, y, predictions, cost, optimizer, x_train, y_train, x_data, y_data, keep_prob
       ↪ ):
13     training_epochs = 1500
14     display_step = 100
15     batch_size = 32
16
17     with tf.Session() as sess:
18         sess.run(tf.global_variables_initializer())
19         t1 = time.time()
20         acc = []
21         correct_prediction = tf.equal(tf.argmax(predictions, 1), tf.argmax(y, 1))
22         accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
23         for epoch in range(training_epochs):
24             avg_cost = 0.0
```

```
25          total_batch = int(len(x_train) / batch_size)
26          x_batches = np.array_split(x_train, total_batch)
27          y_batches = np.array_split(y_train, total_batch)
28          for i in range(total_batch):
29              batch_x, batch_y = x_batches[i], y_batches[i]
30              _, c = sess.run([optimizer, cost],
31                              feed_dict={
32                                  x: batch_x,
33                                  y: batch_y,
34                                  keep_prob: 0.8
35                              })
36              avg_cost += c / total_batch
37          data_acc = accuracy.eval({x: x_data, y: y_data, keep_prob: 1.0})
38          acc.append(data_acc)
39          if epoch % display_step == 0:
40              print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
41              print("Accuracy validation:", data_acc)
42      print("Optimization Finished!")
43      t2 = time.time()
44      confm = tf.confusion_matrix(tf.argmax(y,1),tf.argmax(predictions, 1), num_classes=y_train.
    ↪ shape[1])
45      confm_eval = confm.eval({x: x_data, y: y_data, keep_prob: 1.0})
46      res = {"time": (t2-t1), "confm": confm_eval, "accuracy": acc}
47      return res
48
49
50  def create_and_train_perceptron(x_train, y_train, x_data, y_data, n_hidden, sigmoid=1):
51      n_input = x_train.shape[1]
52      n_classes = y_train.shape[1]
53
54      weights = {
55          'h1': tf.Variable(tf.random_normal([n_input, n_hidden])),
56          'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
57      }
58
59      biases = {
60          'b1': tf.Variable(tf.random_normal([n_hidden])),
61          'out': tf.Variable(tf.random_normal([n_classes]))
62      }
63
64      keep_prob = tf.placeholder("float")
65
66      x = tf.placeholder("float", [None, n_input])
67      y = tf.placeholder("float", [None, n_classes])
68
69      predictions = multilayer_perceptron(x, weights, biases, keep_prob, sigmoid)
70      cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=predictions, labels=y))
71
72      optimizer = tf.train.AdamOptimizer(learning_rate=0.001).minimize(cost)
73
74      return train_perceptron(x, y, predictions, cost, optimizer, x_train, y_train, x_data, y_data,
    ↪ keep_prob)
```

Luego, para mostrar el gráfico con el conjunto de validación y el número óptimo de neuronas en la capa oculta, se tiene la función p4a, que no recibe nada, parte por cargar los datos con load_data, calcula el número de neuronas en la capa oculta, llama a la función create_and_train_perceptron, y luego con el resultado obtenido imprime la matriz de confusión, el tiempo de entrenamiento, el accuracy (calculado como la suma de la diagonal partido por la suma de toda la matriz) y el promedio y desviación estándar de la diagonal, luego grafica el accuracy y guarda la figura en la misma carpeta de donde esta el programa, con el nombre *p4a*, el gráfico obtenido se puede ver en la Figura 2, y la matriz de confusión con el resto de los prints en la Figura 3.

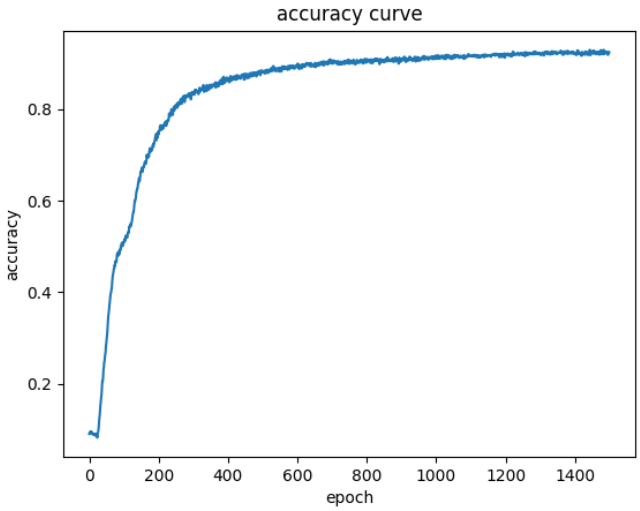

Figura 2: Accuracy, numero óptimo de neuronas utilizando función sigmoidal

```
-------------------------------------------
[[149   0   0   0   0  12   0   0   0   0   0]
 [  0 126   0   0   0   0   0   0   1  25   0]
 [  0   0 151   0   4   0   0   0   0   0   0]
 [  0   0   1 142   0   0   0   0   0   0   0]
 [  0   0   5   4 131   0   0  13   0   0   0]
 [  5   0   1   1   0 130   0   0  11   0   0]
 [  0   0   0   0   0   0 154   0   0   0   0]
 [  0   0   0   3  11   1   0 135   0   0   0]
 [  1   3   0   0   0   9   0   0 158   0   0]
 [  0  17   0   0   0   0   0   0   0 122   0]
 [  1   1   0   0   0   0   0   0   0   0 169]]
training time: 287.53
Accuracy: 0.9233942
mean of confusion matrix diagonal: 142.45
standard deviation of confusion matrix diagonal: 14.23
-----------------------------------------------
```

Figura 3: Matriz de confusión y otros datos del entrenamiento

3.3. Entrenamiento con distinto número de neuronas

Para esta parte se creó la función p5a, que no recibe nada, la función parte cargando los datos con load_data, con seed 42, no importa el valor del seed ya que se quiere comparar lo que pasa cuando se cambia el número de neuronas, luego se calcula el número de neuronas en la capa oculta de tres formas diferentes: 1/3 de la suma de las neuronas de la capa de entrada + la capa oculta, 2/3 de la suma (el óptimo), y el total de la suma, luego se guardan esos valores en una lista y se recorre con un for, dentro del for se llama a la función create_and_train_perceptron, dándole el conjunto de validación para graficar, y con los resultados se iba imprimiendo la matriz de confusión junto con agregar una curva a un gráfico. Después de hacer el proceso para los tres valores, se muestra el gráfico, y se guarda en la carpeta del programa, con el nombre *validation_accuracy*. El gráfico se puede ver en la Figura 4, y las matrices de confusión con el resto de los datos en las Figuras 5, 6 y 7.

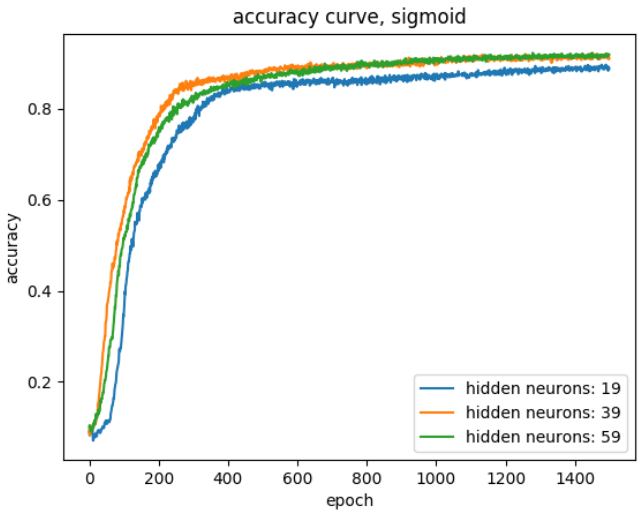

Figura 4: Accuracy del conjunto de validación, cambiando el número de neuronas en la capa oculta

```
------------------------------------------------
[[65  0  0  0  0  3  0  0  0  0  0]
 [ 0 53  0  0  1  0  0  0  0 16  0]
 [ 0  0 63  0  6  0  0  0  0  0  0]
 [ 0  0  2 68  0  0  2  0  0  0  0]
 [ 0  0  0  0 58  0  0 12  0  0  0]
 [12  0  0  0  0 54  0  0  5  0  0]
 [ 0  0  0  0  0  0 70  0  0  0  0]
 [ 0  0  0  0  6  0  0 64  0  0  0]
 [ 0  0  0  0  0 11  0  0 55  0  0]
 [ 0  9  0  0  0  0  0  0  0 64  0]
 [ 0  0  0  0  0  0  0  0  0  0 66]]
training time: 268.73
Accuracy: 0.8888889
mean of confusion matrix diagonal: 61.82
standard deviation of confusion matrix diagonal: 5.59
------------------------------------------------
```

Figura 5: Matriz de confusión y otros datos del entrenamiento, 19 neuronas

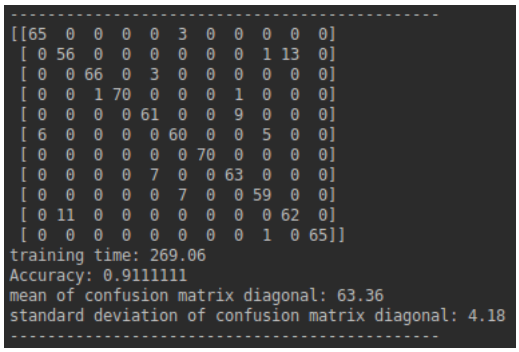

Figura 6: Matriz de confusión y otros datos del entrenamiento, 39 neuronas

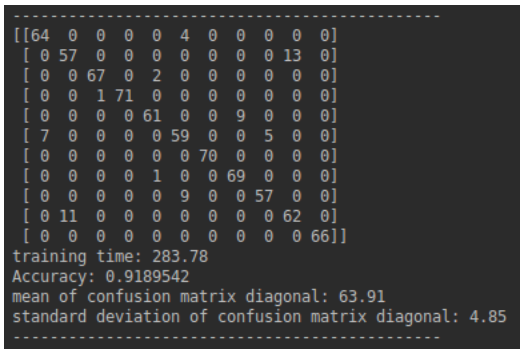

Figura 7: Matriz de confusión y otros datos del entrenamiento, 59 neuronas

El código es el siguiente:

```
def p5a():
    data = load_data(42)
    x_train, y_train = data["x_train"], data["y_train"]
    x_valid, y_valid = data["x_valid"], data["y_valid"]

    n_input = x_train.shape[1]
    n_classes = y_train.shape[1]
    s = n_input + n_classes

    hidden_neurons = [int(s*1/3), int(s*2/3), s]
    for n_hidden in hidden_neurons:
        res = create_and_train_perceptron(x_train, y_train, x_valid, y_valid, n_hidden, sigmoid=1)
        plt.plot(range(len(res["accuracy"])), res["accuracy"], label="hidden neurons: {}".format(
    ↪ n_hidden))
        print("-----------------------------------------------")
        confm = res["confm"]
        confm_diagonal = np.diag(confm)
        print(confm)
        print("training time: {:.2f}".format(res["time"]))
        print("Accuracy: {:.7f}".format(confm_diagonal.sum() / confm.sum()))
        print("mean of confusion matrix diagonal: {:.2f}".format(np.mean(confm_diagonal)))
```

```
21        print("standard deviation of confusion matrix diagonal: {:.2f}".format(np.std(confm_diagonal))
      ↪ )
22        print("-----------------------------------------")
23    plt.title("accuracy curve, sigmoid")
24    plt.legend(loc="lower right")
25    plt.ylabel("accuracy")
26    plt.xlabel("epoch")
27    plt.savefig("validation_accuracy")
28    plt.show()
29    return
```

3.4. Sigmoid v/s ReLU

Viendo la Figura 4, se decidió usar el óptimo de neuronas (39), para esta parte se creo la función p5b, que no recibe nada, como se deben probar distintas seed para el random, se creó una lista con seeds 20, 40 y 60, éstos números fueron pensados al azar y no hay ninguna razón especial para elegirlos. Para el análisis de las curvas, se harán 3 curvas con sigmoid y 3 con ReLU, y se mostrará el promedio de las curvas de cada uno, junto con los promedios de las matrices de confusión. Para esto la función parte iniciando una matriz de ceros (para ir sumando las otras matrices), también una lista de accuracy con ceros, y tiempo en 0, después, con un for, para cada seed se cargan los datos, se calcula el número de neuronas elegido para la capa oculta y se llama a la función create_and_train_perceptron, y con los resultados se le suman a las variables definidas antes, la matriz de confusión, la lista de accuracy y el tiempo. Después de realizar el proceso para las 3 seed, se dividen las variables por 3, para sacar el promedio, luego se agrega la curva promedio a un gráfico, luego se repite lo mismo, solo que al llamar a la función create_and_train_perceptron se utiliza el parámetro sigmoid=0 para utilizar ReLU en la capa oculta, finalmente se muestra el gráfico con las 2 curvas obtenidas con los promedios y se guarda en la carpeta del programa, con el nombre *test_accuracy*, también se hace un print con las matrices de confusión promedio y los datos de tiempo promedio y análisis de la diagonal. Se puede ver el gráfico obtenido en la Figura 8, y los prints en las Figuras 9 y 10.

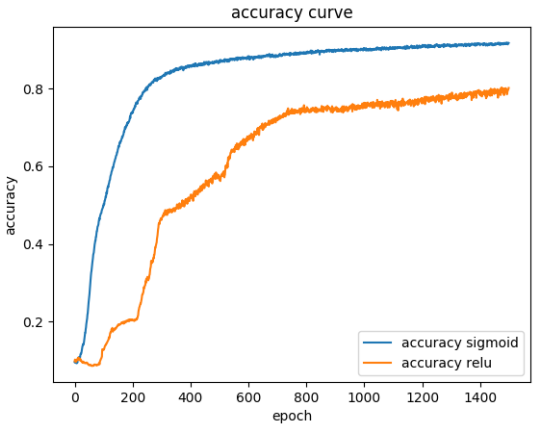

Figura 8: Comparación de accuracy obtenido con sigmoid y ReLU

```
---------------------------------------------
[[154.    0.    0.    0.    0.    7.    0.    0.    0.    0.    0. ]
 [  0.  130.3   0.    0.    0.    0.    0.    0.    0.7  21.    0. ]
 [  0.    0.  149.3   1.3   4.3   0.    0.    0.    0.    0.    0. ]
 [  0.    0.    1.  141.3   0.7   0.    0.    0.    0.    0.    0. ]
 [  0.    0.    4.    5.3 126.7   0.    0.   17.    0.    0.    0. ]
 [  5.3   0.    0.7   0.    0.  131.7   0.    0.   10.    0.3   0. ]
 [  0.    0.    0.    0.    0.    0.  154.    0.    0.    0.    0. ]
 [  0.    0.    0.    2.3  16.    0.    0.  131.7   0.    0.    0. ]
 [  1.    1.7   0.    0.    0.   17.    0.    0.  151.3   0.    0. ]
 [  0.   20.7   0.    0.    0.    0.    0.    0.    0.  118.3   0. ]
 [  0.7   0.7   0.3   0.    0.    0.3   0.    0.    0.    0.  169. ]]
training average time: 262.39
Accuracy: 0.9179091
mean of confusion matrix diagonal: 141.60
standard deviation of confusion matrix diagonal: 14.46
---------------------------------------------
```

Figura 9: Matriz de confusión y otros datos del entrenamiento, sigmoide

```
---------------------------------------------
[[137.7   0.    0.3   0.    0.   21.    0.    0.    2.    0.    0. ]
 [  0.  119.    0.    0.    0.    0.    0.    0.    0.3  32.7   0. ]
 [  0.7   0.  144.3   2.7   5.7   1.    0.    0.3   0.    0.3   0. ]
 [  0.    0.    2.7 124.7   7.    0.    8.3   0.3   0.    0.    0. ]
 [  2.    0.   12.7  13.7  88.    0.3   0.3  35.7   0.    0.3   0. ]
 [ 44.7   0.3   1.    0.7   0.   82.7   0.    0.   18.3   0.    0.3]
 [  0.    0.    1.3  12.    0.3   0.  140.3   0.    0.    0.    0. ]
 [  2.7   0.    1.    5.3  26.7   0.    1.  113.    0.    0.3   0. ]
 [  7.    5.    0.3   0.3   0.   31.3   0.    0.  125.7   1.3   0. ]
 [  0.   21.3   0.    0.    0.    0.    0.    0.    0.3 117.3   0. ]
 [  0.    1.3   0.    0.3   0.    0.    0.    0.    0.    1.  168.3]]
training average time: 278.41
Accuracy: 0.8021926
mean of confusion matrix diagonal: 123.73
standard deviation of confusion matrix diagonal: 23.40
---------------------------------------------
```

Figura 10: Matriz de confusión y otros datos del entrenamiento, ReLU

El código es el siguiente:

```
def p5b():
    random = [20, 40, 60]
    div = len(random)
    training_epochs = 1500

    confusion_matrix_sigmoid = np.zeros((11, 11))
    accuracy_sigmoid = np.zeros(training_epochs)
    time_sigmoid = 0
    for seed in random:
        data = load_data(seed)

        x_train, y_train = data["x_train"], data["y_train"]
        x_test, y_test = data["x_test"], data["y_test"]

        n_input = x_train.shape[1]
        n_classes = y_train.shape[1]
        n_hidden = int((n_input + n_classes) * 2 / 3)

        res = create_and_train_perceptron(x_train, y_train, x_test, y_test, n_hidden, sigmoid=1)
        confusion_matrix_sigmoid += np.array(res["confm"])
        accuracy_sigmoid += np.array(res["accuracy"])
```

```
22      time_sigmoid += res["time"]
23
24  confusion_matrix_sigmoid = np.around(confusion_matrix_sigmoid / div, 1)
25  accuracy_sigmoid = accuracy_sigmoid / div
26  time_sigmoid = time_sigmoid / div
27  print("-----------------------------------------------")
28  confm_diagonal_sigmoid = np.diag(confusion_matrix_sigmoid)
29  print(confusion_matrix_sigmoid)
30  print("training average time: {:.2f}".format(time_sigmoid))
31  print("Accuracy: {:.7f}".format(confm_diagonal_sigmoid.sum() / confusion_matrix_sigmoid.sum
    ↪ ()))
32  print("mean of confusion matrix diagonal: {:.2f}".format(np.mean(confm_diagonal_sigmoid)))
33  print("standard deviation of confusion matrix diagonal: {:.2f}".format(np.std(
    ↪ confm_diagonal_sigmoid)))
34  print("-----------------------------------------------")
35  plt.plot(range(training_epochs), accuracy_sigmoid, label="accuracy sigmoid")
36
37  confusion_matrix_relu = np.zeros((11, 11))
38  accuracy_relu = np.zeros(training_epochs)
39  time_relu = 0
40  for seed in random:
41      data = load_data(seed)
42
43      x_train, y_train = data["x_train"], data["y_train"]
44      x_test, y_test = data["x_test"], data["y_test"]
45
46      n_input = x_train.shape[1]
47      n_classes = y_train.shape[1]
48      n_hidden = int((n_input + n_classes) * 2 / 3)
49
50      res = create_and_train_perceptron(x_train, y_train, x_test, y_test, n_hidden, sigmoid=0)
51      confusion_matrix_relu += np.array(res["confm"])
52      accuracy_relu += np.array(res["accuracy"])
53      time_relu += res["time"]
54
55  confusion_matrix_relu = np.around(confusion_matrix_relu / div, 1)
56  accuracy_relu = accuracy_relu / div
57  time_relu = time_relu / div
58
59  print("-----------------------------------------------")
60  confm_diagonal_relu = np.diag(confusion_matrix_relu)
61  print(confusion_matrix_relu)
62  print("training average time: {:.2f}".format(time_relu))
63  print("Accuracy: {:.7f}".format(confm_diagonal_relu.sum() / confusion_matrix_relu.sum()))
64  print("mean of confusion matrix diagonal: {:.2f}".format(np.mean(confm_diagonal_relu)))
65  print("standard deviation of confusion matrix diagonal: {:.2f}".format(np.std(confm_diagonal_relu
    ↪ )))
66  print("-----------------------------------------------")
67  plt.plot(range(training_epochs), accuracy_relu, label="accuracy relu")
68  plt.title("accuracy curve")
69  plt.legend(loc="lower right")
70  plt.ylabel("accuracy")
```

```
71    plt.xlabel("epoch")
72    plt.savefig("test_accuracy")
73
74    plt.show()
```

4. Conclusiones y Análisis de Resultados

Con respecto al cambio en el número de neuronas en la capa oculta, se puede ver que el desempeño es parecido en los 3 casos, pero el óptimo se muestra por encima de los otros por un pequeño rango, analizando las matrices de confusión, se puede ver que para las 3 diagonales el promedio es parecido, y la desviación estándar de la diagonal es menor para el óptimo de neuronas.
Con respecto al tiempo de entrenamiento, la que más se demoró fue la red con mayor cantidad de neuronas en la capa oculta, mientras que las otras 2 curvas tuvieron un tiempo muy parecido (menos de 1 segundo de diferencia), pero no se podría decir que una red es mejor que la otra por si tiempo, acá se puede ver en la curvas que tenían 39 y 59 neuronas, que tuvieron un desempeño muy parecido, mientras que sus tiempos difieren en casi 13 segundos.

Con respecto a la no-linealidad de la capa oculta, se puede ver que la curva utilizando sigmoide tiene mucho mejor desempeño al clasificar que con ReLU, esto puede deberse a que, al tener solo 1 capa oculta, ReLU inhabilita ciertas neuronas, ya que su input es negativo, y esto puede resultar en varias neuronas que no afectan al output, por lo que en vez de tener el número óptimo de neuronas en la capa oculta, se tiene un número menor, además, ReLU normalmente se utiliza para redes que utilizan muchas capas ocultas (deep neural networks).
Con respecto al tiempo de entrenamiento, las curvas con ReLU se demoraron más en entrenar, aunque esto no dice que sean mejores, porque en este caso, aunque estuvieron más tiempo entrenándose, tuvieron un peor desempeño.

Para finalizar, se puede decir que para trabajar con redes neuronales artificiales hay que fijarse en varias cosas que pueden afectar al clasificador, tales como ver cual es el número de neuronas de la capa oculta, o incluso el número de capas ocultas, así como que función de no-linealidad se utiliza, ya que dependiendo de la función pueden dar resultados muy diferentes, por lo que para poder encontrar la mejor configuración, se puede entrenar muchas veces probando distintas configuraciones, y quedarse con la mejor. También que hay que tener tiempo para entrenar ya que, dependiendo de como se programe la red, el entrenamiento puede tomar bastante tiempo.