

INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Introducción a las Redes Neuronales Artificiales

Dr. Juan Humberto Sossa Azuela

Reporte Práctica No.2

Acosta Rosales Jair Sebastián

INSTITUTO POLITÉCNICO NACIONAL



Introducción

Las redes neuronales profundas permiten la solución a problemas no lineales que son complejos de computar con técnicas de programación tradicionales que pueden ocupar grandes líneas de programación y aún así pueden no abarcar todos los posibles casos que puedan ser presentados para resolver un problema de clasificación.

En un primer plano tenemos las redes neuronales MLP(Multilayer perceptron), las cuales consisten en un conjunto de capas de perceptrones apiladas una detrás de otra que lo que buscan con el ajuste de pesos es encontrar la función que permita separar los elementos a clasificar en sus respectivas clases, estas redes han sido de suma importancia, sin embargo carecen de una parte importante llamada “extracción de características”, lo cual nos ayuda a que la red neuronal pueda encontrar ciertos elementos importantes dentro de los patrones de entrada y de esta forma hallar aquellos que puedan ser de utilidad para clasificar de una mejor manera cada uno de los datos de entrada, este problema fue tratado con un nuevo tipo de redes neuronales, llamadas, las CNN(Convolutional Neural Networks), las redes neuronales convolucionales permiten hacer un filtrado de la información más importante de cada uno de los datos de entrada a través de filtros, estos pueden destacar diferentes aspectos de los datos y por lo tanto la red podrá ser capaz de identificar características específicas con las cuales encontrará la forma de resolver el problema.

Para esta práctica se pretende usar ambos tipos de redes neuronales (MLP y CNN) para que se puedan clasificar los datos proporcionados por el conjunto mnist, el cual consiste en un conjunto de 60000 imágenes de números escritos en manuscrita, con ello podremos notar la diferencia entre ambos métodos al resolver el problema.

Las herramientas que se usaron para resolver esta práctica son las librerías tensorflow versión 1 y numpy, todas pertenecientes al lenguaje de programación Python en su versión 3.

Resolviendo MNIST con una arquitectura MLP

Como primera parte se importa las librerías necesarias, como lo son tensorflow, numpy y matplotlib.

Además se importa la librería necesaria para poder descargar el dataset mnist, proveniente de la librería **tensorflow.examples.tutorials.mnist**, existen otras librerías de las cuales también es posible descarga el mismo dataset, sin embargo, la que se ha descargado viene con la opción de codificación de los targets, ya sean en código binario o onehot, en este caso, se ha elegido onehot.

```
[4] import tensorflow as tf #importamos tensorflow
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
%tensorflow_version 1.x
```

Importación de datos

```
[5] from tensorflow.examples.tutorials.mnist import input_data #importamos algun dataset de la libreria mnist
mnist = input_data.read_data_sets("input/data/", one_hot = True) #lee los datasets y retorna la codificacion con one hot, puede ser representacion binaria, etc.
```

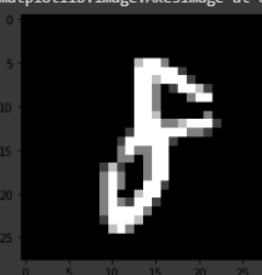
Una vez leído el conjunto de imágenes se toma una imagen de prueba y se visualiza, es importante decir que el dataset viene con cada imagen en formato de una dimensión, es decir, toda una imagen es una fila del dataset, en este caso el tamaño total de la imagen es de 784 bytes, de esta forma no es posible hacer una visualización, por lo que tenemos que proceder a realizar un reshape de la imagen,

a su dimensión original(28*28) para que con ayuda de matplotlib sea posible graficar la imagen.

```
[6] mnist.train.images.shape #obtiene todos las imagenes de entrenamiento, retorna la forma, #obtenemos 55000 imagenes de 28*28
    #print(vars(mnist.train))
    #print(mnist.train.labels)

[7] #print(mnist.train.images[0])
    imagendemo=np.reshape(mnist.train.images[30,:],(28,28)) #obtiene una de las imagenes
    #print(mnist.train.images[24,:])

[8] plt.imshow(imagendemo,cmap='gray')
```



Definiendo la arquitectura de la red neuronal

La arquitectura es definida mediante una función, en dónde gracias al uso de tensorflow es posible declarar distintos tipos de tensores para una funcionalidad en específico, a continuación, se describe la finalidad de cada tensor declarado en la función:

- **tf.placeholder:** Este tensor se entiende como un apartado de memoria para los patrones de entrada con una dimensionalidad de 784 bytes, que es un reshape de una imagen de dos dimensiones de 28*28. Así mismo para el apartado de las salidas también se crea un tensor de este tipo.
- **tf.Variable:** Este tipo de tensor permitirá crear arrays en el que sus valores puedan ser modificados en el tiempo de ejecución, es por eso que son implementados para crear los pesos y bias de cada capa de la red neuronal.

Por lo tanto, la primer parte de la función queda de la siguiente manera:

```
def Neural_network_model(
    n_nodes_hl1=500,
    n_nodes_hl2=500,
    n_nodes_hl3=500,
    n_classes=10
):

    # Declarando las entradas y salidas, lo definimos nosotros
    x=tf.placeholder('float',[None,784]) #aparta espacio para los datos que entraran, en este caso almacenara un float, 784 = dimension del vector(de la imagen)
    #None = tamaño del batch, el cual no sabemos de que tamaño es, pero sabemos que hay uno
    y=tf.placeholder('float') #guarda espacio para los datos(etiquetas) que saldrán respecto a la entrada que se le da en cada iteración

    # Declarando las variables
    #parametro = para tensorflow el objeto Variable permitira hacer las modificaciones, es decir aquello que se puede modificar durante el entrenamiento
    #hiperparametro = Constante, todo aquello que nosotros modificamos antes del entrenamiento

    #Tamaño de las diferentes capas, aprender a definir el tamaño de las capas, como conectar las capas, funciones de activación que se pueden ocupar
    #weights se inicializan aleatoriamente con una distribución normal
    hidden_1_layer = {'weights':tf.Variable(tf.random_normal([784, n_nodes_hl1])),
                      'biases':tf.Variable(tf.random_normal([n_nodes_hl1]))}
    #print(hidden_1_layer['weights'].shape)
    #print(hidden_1_layer['biases'])
    hidden_2_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
                      'biases':tf.Variable(tf.random_normal([n_nodes_hl2]))}
    #print(hidden_2_layer['weights'].shape)
    hidden_3_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_hl3])),
                      'biases':tf.Variable(tf.random_normal([n_nodes_hl3]))}
    #print(hidden_3_layer['weights'].shape)
    output_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl3, n_classes])),
                    'biases':tf.Variable(tf.random_normal([n_classes]))}
```

Donde también se puede observar que se definen las variables `n_nodes_hlx` y `n_classes`, esto es debido a que desde esos valores nosotros manejamos el número de neuronas que tendrán las capas ocultas y la capa de salida, en este caso mnist solo maneja 10 clases por lo tanto `n_classes` debe ser obligatoriamente igual a diez.

La siguiente parte de la función es el **feedforward**, es decir la propagación hacia delante de todos los patrones de entrada que estamos recibiendo en un mismo batch, para este problema se usó la función de activación ReLu

```
# Declarando la arquitectura, conectando todo

l1 = tf.add(tf.matmul(x,hidden_1_layer['weights']), hidden_1_layer['biases'])
l1 = tf.nn.relu(l1)

l2 = tf.add(tf.matmul(l1,hidden_2_layer['weights']), hidden_2_layer['biases'])
l2 = tf.nn.relu(l2)

l3 = tf.add(tf.matmul(l2,hidden_3_layer['weights']), hidden_3_layer['biases'])
l3 = tf.nn.relu(l3)

output = tf.matmul(l3,output_layer['weights']) + output_layer['biases']
#para la salida no necesariamente se pone la función de activación
```

Una vez hecha el feedforward y obtenidos los outputs de cada uno de los patrones de entrada en la variable output, procedemos a calcular el error que hubo en cada una de las salidas que otorgo la red contra las salidas deseadas, con ayuda de la entropía cruzada, una vez calculado el error en cada uno se promedia el error de todas estas, para aplicar lo que se conoce como gradiente estocástico medio, el cual es una variación del gradiente descendiente en el cual se pretende calcular un promedio de errores de un batch para agilizar el proceso de aprendizaje de la red y no el calculo y propagación del error de cada uno de los elementos del dataset como se realizaría con el gradiente descendiente tradicional.

Aunado a esto se hará uso del algoritmo Adam que permite el calculo de los nuevos pesos y bias con ayuda del error promedio calculado.

```
# Declarando la funcion de costo y optimizador,
cost = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(logits=output
                                                                , labels=y) )

#logits = la salida que obtengo de la red, labels = las etiquetas obtenidas c

optimizer = tf.train.AdamOptimizer().minimize(cost)#AdamOptimizer minimizara

#retorna control a cada parametro
return dict(
    x=x, #entradas
    y=y, #salidas
    output=output,#salida de la red
    cost=cost,#costo
    optimizer=optimizer
)
```

Como parte final de la operación se agrega como retorno un diccionario que nos permitirá pasar valores como lo son los respectivos patrones de entrada(x) y sus etiquetas(y) durante el tiempo de ejecución, además de poder retornar los valores de salida de la red, así como el costo o error de esa época, como apartado final la variable optimizar funcionará como un switch, en el que indicaremos si la red va a aprender o no, esto último nos sirve para poder probar la red con el conjunto de testing, de forma que no haya cambios en los parámetros de la red.

Función de entrenamiento de la red

Para la función de entrenamiento se hace uso de una función llamada Session que permite ejecutar la función anteriormente declarada, así como se puede observar definimos un número de elementos que tendrá cada batch, esto es debido a que no podemos ingresar todas las imágenes en una época al mismo tiempo dada la gran cantidad de memoria RAM que necesitaría ocupar, por ello dividimos en pequeños grupos de imágenes que se van propagando en un for anidado dentro del for de épocas

```
def train_neural_network(DNN, hm_epochs=10, batch_size=100): #DNN =Diccionario de retorno de la funcion Neural network model
    with tf.Session() as sess: #se define el espacio de código que será compilado
        sess.run(tf.global_variables_initializer())#todas las variables globales son inicializadas con los valores que nosotros declaramos

        for epoch in range(hm_epochs):
            epoch_loss = 0 #error de la epoca
            for _ in range(int(mnist.train.num_examples/batch_size)):#for que toma el numero total de elementos entre el batch
                epoch_x, epoch_y = mnist.train.next_batch(batch_size) #epoch_x = valores de entrada del batch epoch_y = etiquetas del batch actual
                feed_dict={DNN["x"]: epoch_x, #carga las entradas actuales en las entradas de la arquitectura
                           DNN["y"]: epoch_y} #carga las etiquetas correspondientes de las entradas almacenadas en epoch_x
                _, c, prediction, y = sess.run([DNN["optimizer"], DNN["cost"],
                                                DNN["output"], DNN["y"]],
                                               feed_dict=feed_dict) #sessrun hace una lista de todo lo que deseo que retorne

            epoch_loss += c

        print('Epoch', epoch, 'completed out of', hm_epochs, 'loss:', epoch_loss)
```

Características de la red MLP implementada

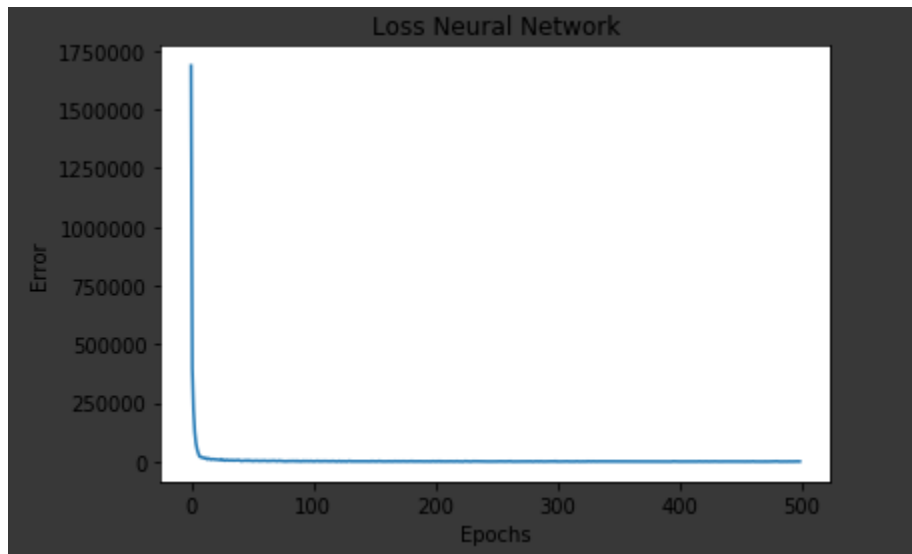
- Tres capas ocultas con 500 neuronas cada una.
- El algoritmo de entrenamiento implementado es Adam.
- El tamaño del batch seleccionado es de 100 imágenes.
- El número de épocas para el entrenamiento es de 500.
- La función de costo implementada es entropía cruzada.
- Codificación en onehot para las etiquetas.

Gráficas del entrenamiento

Las últimas cinco épocas del entrenamiento, notamos que los saltos del error son grandes debido al gradiente estocástico medio.

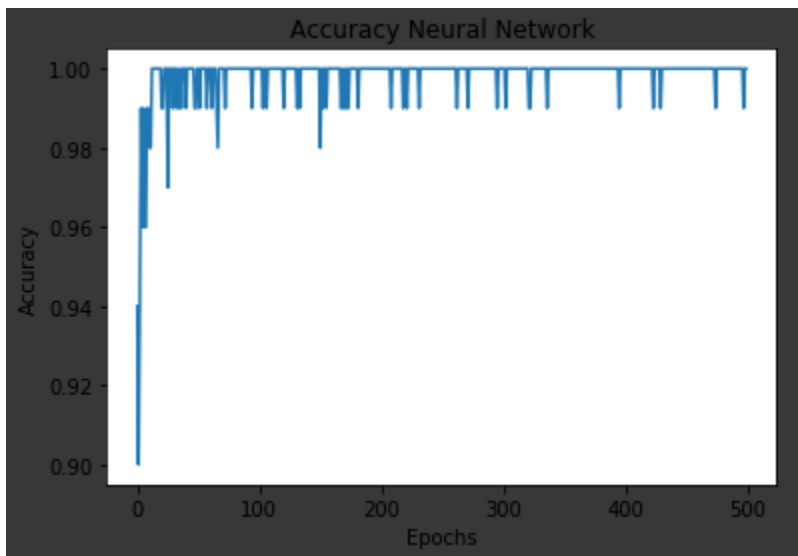
```
Epoch 495 completed out of 500 loss: 696.6259822845459
Epoch 496 completed out of 500 loss: 1237.4930067062378
Epoch 497 completed out of 500 loss: 623.6661670207977
Epoch 498 completed out of 500 loss: 759.6828036308289
Epoch 499 completed out of 500 loss: 1097.4555967093038
Accuracy Test: 0.9786
```

Costo de la red neuronal por época



Accuracy de la red neuronal por época

Podemos notar los saltos del accuracy, a pesar de esto se observa que va subiendo notablemente de precisión.



Resolviendo MNIST con una arquitectura CNN

El preprocesamiento de los datos es exactamente el mismo que para la arquitectura MLP, en este caso resolveremos el problema aplicando capas convolucionales al inicio de la arquitectura, las cuales permitirán extraer las características más importantes que la red neuronal pueda detectar.

```
n_nodes_hl1=500,
n_nodes_hl2=500,
n_nodes_hl3=500,
keep_rate = 0.8,
n_classes=10
):
# Declarando las entradas y salidas
x=tf.placeholder('float',[None,784])
y=tf.placeholder('float')

#Imágenes
img = tf.reshape(x, shape=[-1, 28, 28, 1]) #Dimensiones de la imagen
# Declarando las variables
weights = {'W_conv1':tf.Variable(tf.random_normal([5,5,1,32])), #Pr
          'W_conv2':tf.Variable(tf.random_normal([5,5,32,64])),
          'W_conv3':tf.Variable(tf.random_normal([5,5,64,128])),
          'W_fc':tf.Variable(tf.random_normal([4*4*128,1024])),
          'out':tf.Variable(tf.random_normal([1024, n_classes]))}

biases = {'b_conv1':tf.Variable(tf.random_normal([32])),
          'b_conv2':tf.Variable(tf.random_normal([64])),
          'b_conv3':tf.Variable(tf.random_normal([128])),
          'b_fc':tf.Variable(tf.random_normal([1024])),
          'out':tf.Variable(tf.random_normal([n_classes]))}
```

Como se puede observar en el cuadro rojo, la declaración de las capas convolucionales se hace dando cuatro argumentos a la inicialización del tensor, los argumentos son:

- **arg1:** tamaño del kernel o del filtro sobre el eje x de la imagen
- **arg2:** tamaño del kernel o del filtro sobre el eje y de la imagen
- **arg3:** profundidad del kernel, es decir si la imagen cuenta con más capas de valores, en este caso las imágenes están binarizadas, por lo tanto solo es un canal de valores, si fuese una imagen rgb, entonces serían tres canales y la profundidad debería ser igualmente de tres.

- **arg4:** Número de filtros que se aplicarán en esa capa, es decir, el número de características extraídas de esa capa convolucional.

Para el apartado de **feedforward** se tiene que aplicar una convolución en las primeras 3 capas y si lo deseamos un maxpooling el cual permite resaltar todavía más los detalles más importantes y reducir el tamaño de la imagen y por ende el procesamiento que puede tardar si mantenemos la imagen original, finalmente la parte de feedforward de convolución queda:

```
#Recibe imagen de 28*28*1
#Primer convolución
"""
stride[0] = marca saltos entre las imagenes de un batch

stride[1] y stride[2] son los strides espaciales

stride[3] = stride de profundidad, marca los saltos entre cada capa del kernel, en este caso va a ir de 1 en 1
lo que indica que ira recorriendo cada kernel 1 a 1
"""
l1 = tf.nn.conv2d(img, weights['W_conv1'], strides=[1,1,1,1], padding='SAME')

l1 = tf.add(l1, biases['b_conv1'])
l1 = tf.nn.relu(l1)
l1 = tf.nn.max_pool(l1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
#Sale imagen de 14*14*32

#Segunda convolución
#Recibe imagen de 14*14*64
l2 = tf.nn.conv2d(l1, weights['W_conv2'], strides=[1,1,1,1], padding='SAME')
l2 = tf.add(l2, biases['b_conv2'])
l2 = tf.nn.relu(l2)
l2 = tf.nn.max_pool(l2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
#Sale imagen de 7*7*64
#Tercer convolución
#Recibe imagen de 7*7*64
l3 = tf.nn.conv2d(l2, weights['W_conv3'], strides=[1,1,1,1], padding='SAME')
l3 = tf.add(l3, biases['b_conv3'])
l3 = tf.nn.relu(l3)
l3 = tf.nn.max_pool(l3, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
#Sale imagen de 4*4*128
```

Aplicando una función de activación en cada capa, finalmente el resultado es un batch de imágenes de 4*4, con 128 características filtradas gracias a las convoluciones, por lo que la entrada a la capa oculta de la red será de 4*4*128, esto queda:

```
#Sale imagen de 4*4*128

fc = tf.reshape(l3,[-1, 4*4*128])
fc = tf.nn.relu(tf.matmul(fc, weights['w_fc'])+biases['b_fc'])
fc = tf.nn.dropout(fc, keep_rate)

output = tf.matmul(fc, weights['out'])+biases['out']

# Declarando la funcion de costo y entrenamiento
cost = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(logits=output
                                                                , labels=y) )
optimizer = tf.train.AdamOptimizer().minimize(cost)
```

Podemos notar que se hace un reshape de la imagen de 4*4*128, para poder ingresarla como un vector de entrada a nuestra red.

Función de entrenamiento

La función de entrenamiento aplicada resulta ser la misma de la MLP.

```
def train_neural_network(DNN, hm_epochs=10, batch_size=100):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        for epoch in range(hm_epochs):
            epoch_loss = 0
            for _ in range(int(mnist.train.num_examples/batch_size)):
                epoch_x, epoch_y = mnist.train.next_batch(batch_size)
                feed_dict={DNN["x"]: epoch_x,
                           DNN["y"]: epoch_y}
                _, c, prediction, y = sess.run([DNN["optimizer"], DNN["cost"],
                                                DNN["output"], DNN["y"]],
                                                feed_dict=feed_dict)
                epoch_loss += c

            print('Epoch', epoch, 'completed out of', hm_epochs, 'loss:', epoch_loss)

        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
```

En resumen, tenemos:

Características de la red CNN implementada

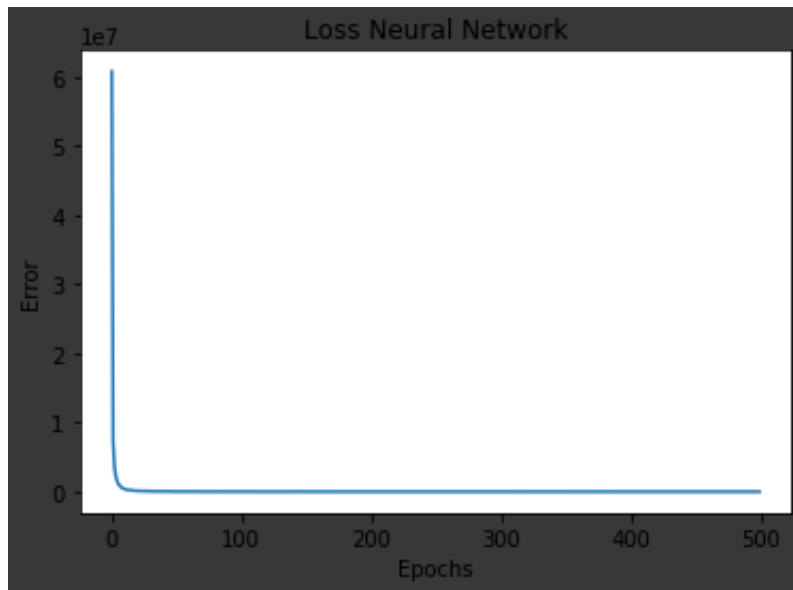
- Tres capas ocultas convolucionales donde:
 1. Primera capa con kernel de 5×5 , profundidad de 1 (la imagen solo tiene un canal (gris), y 32 filtros o características a la salida, además cuenta con un maxpool que reduce el tamaño de la imagen original (28×28) a la mitad es decir 14×14 .
 2. Segunda capa con kernel de 5×5 , profundidad de 32 (los filtros de salida de la capa anterior) y 64 filtros o características a la salida, además cuenta con un maxpool que reduce el tamaño de la imagen recibida (14×14) a la mitad es decir 7×7 .
 3. Tercera capa con kernel de 5×5 , profundidad de 64 (los filtros de salida de la capa anterior), y 128 filtros o características a la salida además cuenta con un maxpool que reduce el tamaño de la imagen recibida (7×7) a la mitad es decir 4×4 (se redondea al valor más grande).
- Una capa oculta completamente conectada de 2048 neuronas, dado que la imagen obtenida resultante es de tamaño 4×4 con 128 filtros, por lo tanto, estas tres dimensiones se convierten en una para que pueda ser un único vector de entrada a la capa oculta.
- El algoritmo de entrenamiento implementado es Adam.
- El tamaño del batch seleccionado es de 100 imágenes.
- El número de épocas para el entrenamiento es de 500.
- La función de costo implementada es entropía cruzada.
- Codificación en onehot para las etiquetas.

Gráfica de error

Últimas cinco épocas de entrenamiento y accuracy en el dataset de testing con un 99.25% de precisión correcta para la clasificación del conjunto.

```
Epoch 495 completed out of 500 loss: 3166.645309448242
Epoch 496 completed out of 500 loss: 3136.1243743896484
Epoch 497 completed out of 500 loss: 809.6068840026855
Epoch 498 completed out of 500 loss: 3906.882152557373
Epoch 499 completed out of 500 loss: 5012.188045501709
Accuracy Test: 0.9925
```

Costo de la red neuronal por época



Accuracy de la red neuronal por época

En esta gráfica es igualmente visible los saltos de accuracy que se tienen debido al gradiente estocástico medio, sin embargo, con el paso de las épocas se puede observar que los saltos van disminuyendo, cosa que con la arquitectura MLP no sucedía.

