TP 3 - S.I.A - Grupo 1

Integrantes:

Burgos, Jose (61525)
Matilla, Juan Ignacio (60459)
Curti, Pedro (61616)
Panighini, Franco (61258)

Introducción

Neurona

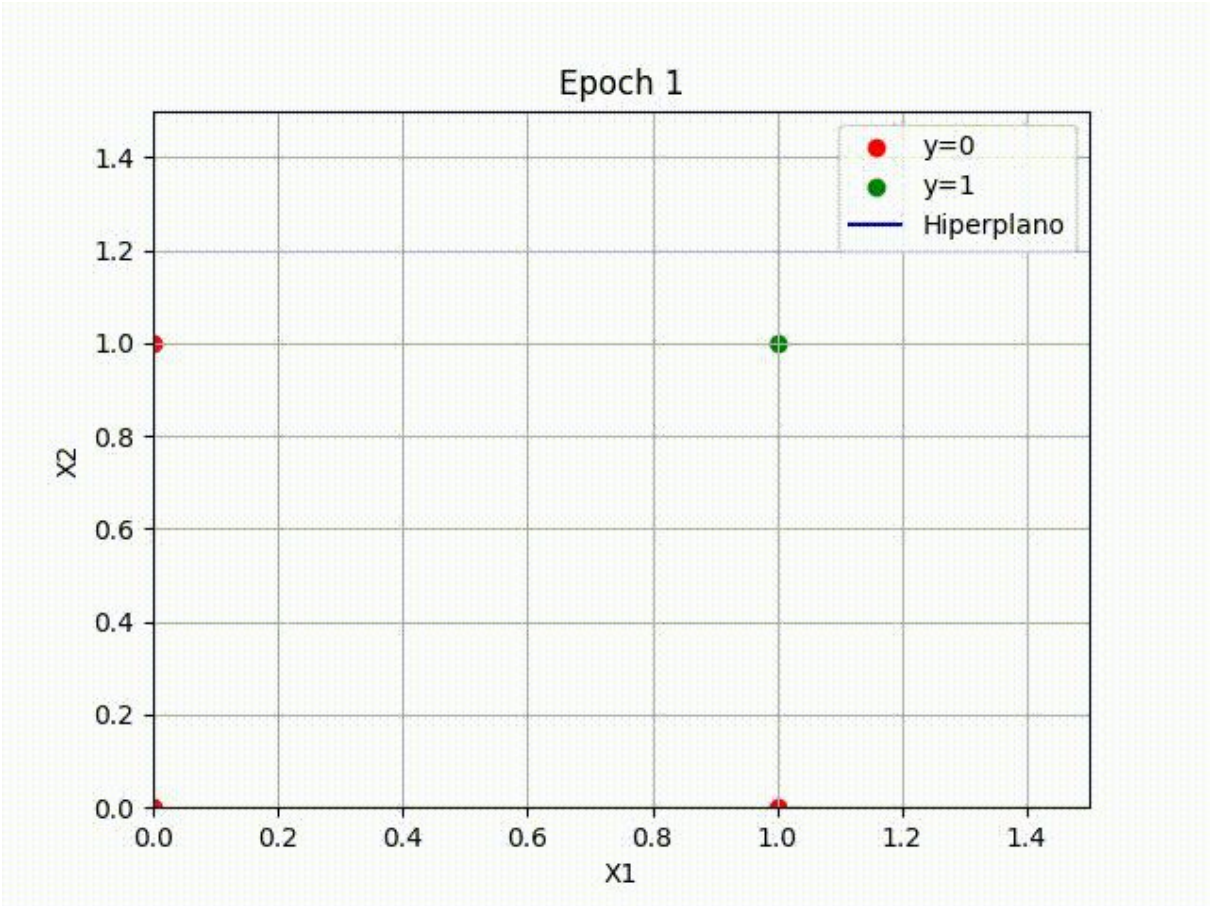
```python
class Neuron:
    def __init__(self, weights: np.ndarray, activation_function: Callable):
        self.weights = weights
        self.activation_function = activation_function

    def predict(self, inputs):
        inputs_with_bias = np.append(inputs, values: 1)
        linear_combination = np.dot(inputs_with_bias, self.weights)
        return self.activation_function(linear_combination)
```

Ej 1

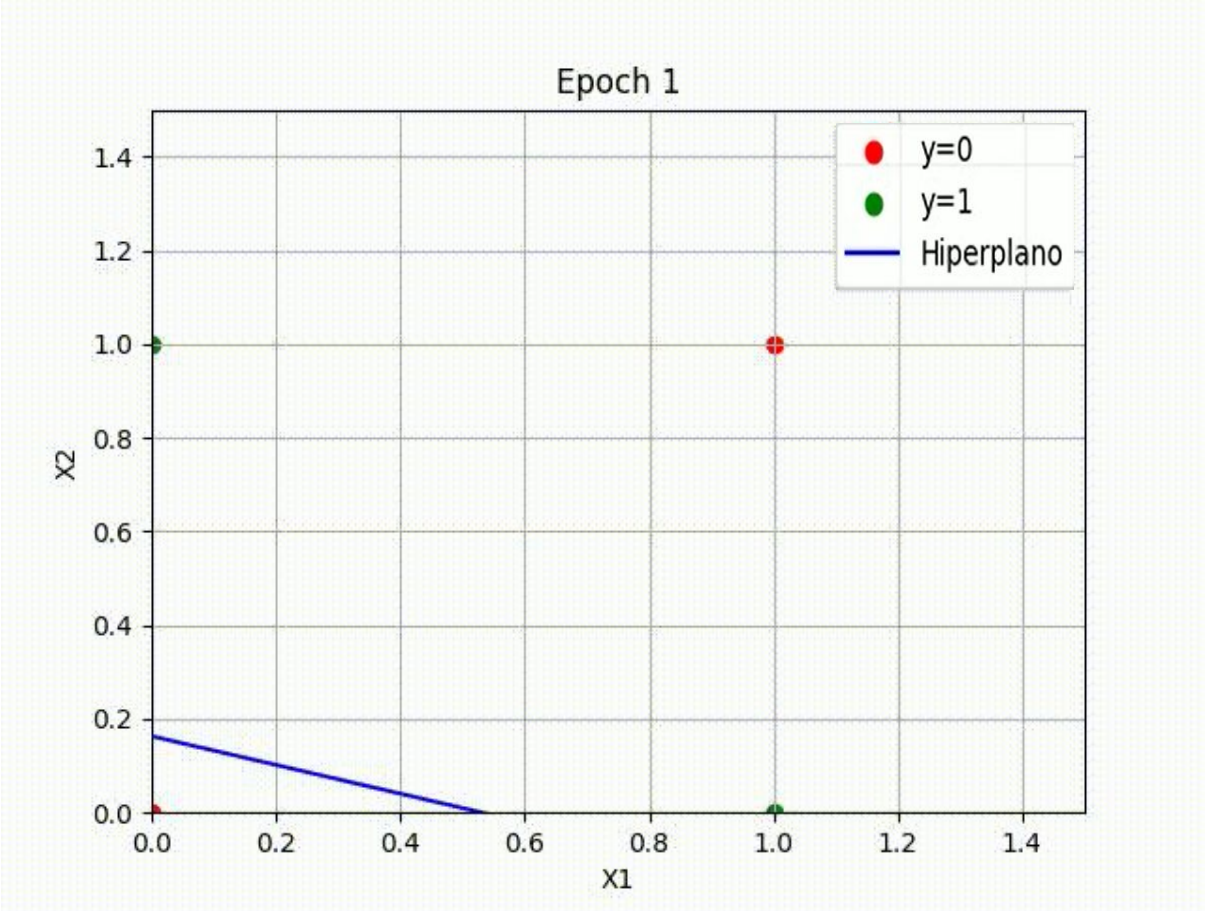
Evolución del hiperplano

AND



learning rate: 0.01

XOR



learning rate: 0.01

Conclusiones

- Al aplicar aprendizaje Hebbiano se ajustan los pesos de la neurona logrando la clasificación de la función AND

- El perceptrón no logra clasificar la función XOR al no poder separarse por una recta

Ej 2

Lineal vs No lineal

```
def weights_adjustment(self, weights, target, prediction, inputs):
    error = target - prediction
    inputs_with_bias = np.append(inputs, values: 1)
    weights += self.learning_rate * error * inputs_with_bias
    return weights
```

```
def weights_adjustment(self, weights, target, prediction, inputs):
    error = target - prediction
    adjustment = self.learning_rate * error * self.activation_function_derivate(prediction)
    inputs_with_bias = np.append(inputs, values: 1)
    weights += adjustment * inputs_with_bias
    return weights
```

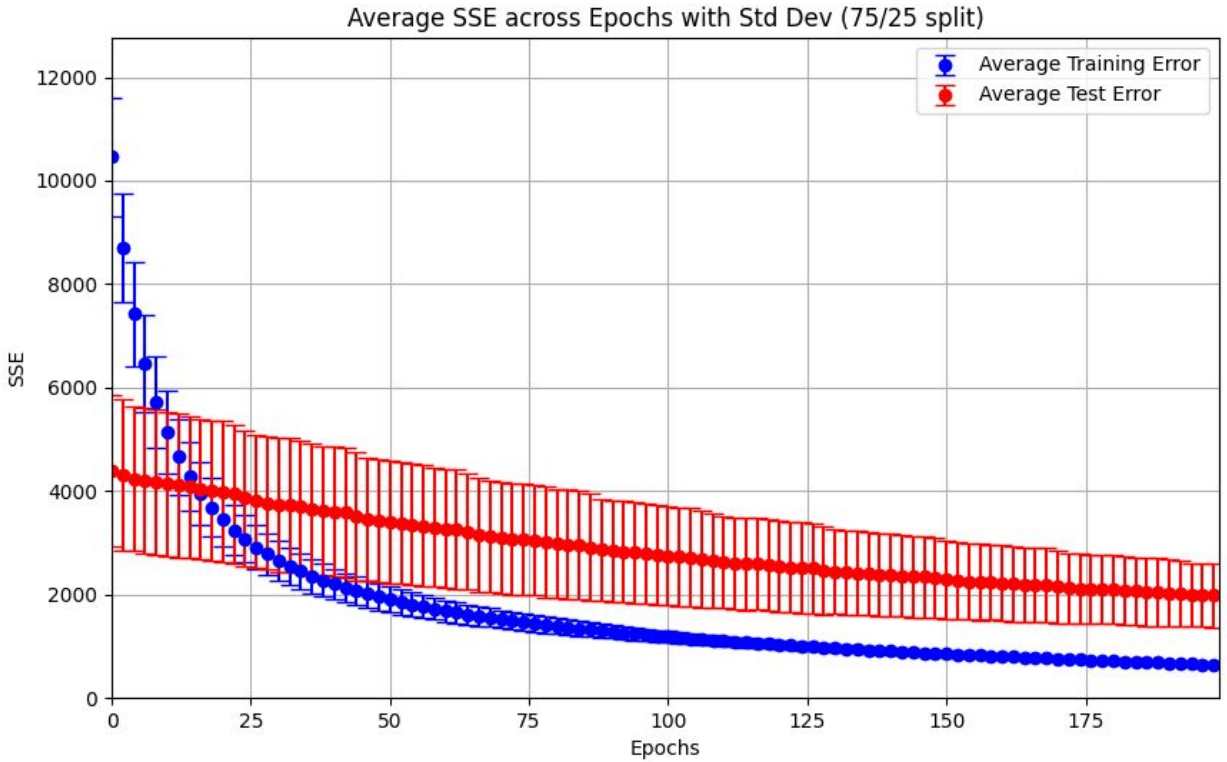
Normalización de datos

```python
def normalize_data(y, y_min, y_max):
    normalized_y = []
    for target in y:
        normalized_y.append((2 * (target - y_min) / (y_max - y_min)) - 1)
    return np.array(normalized_y)


2 usages
def denormalize_data(y_pred, y_min, y_max):
    return ((y_pred + 1) * (y_max - y_min) / 2) + y_min
```

Dataset

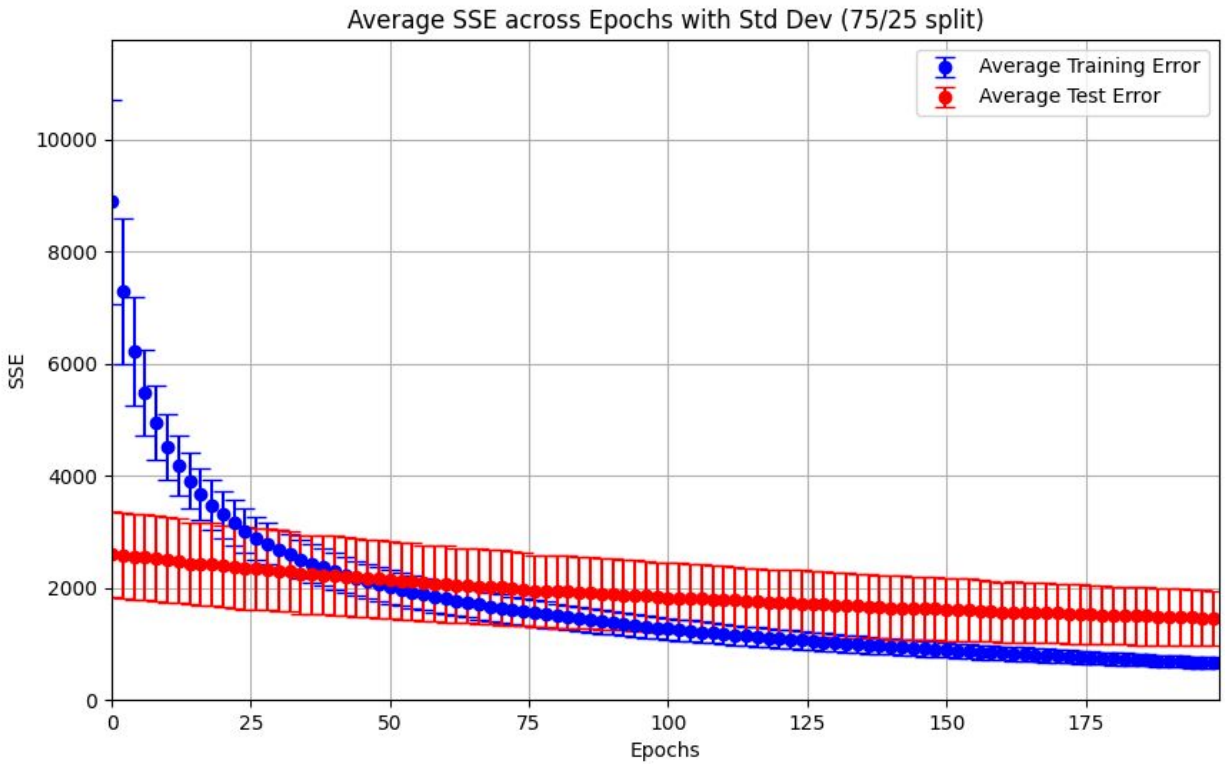
Dataset muy heterogéneo y chico



Perceptrón no lineal
Funcion de activacion tanh
Learning rate = 0.01
5 runs
75% training
25% testing

Haciendo shuffle de los datos

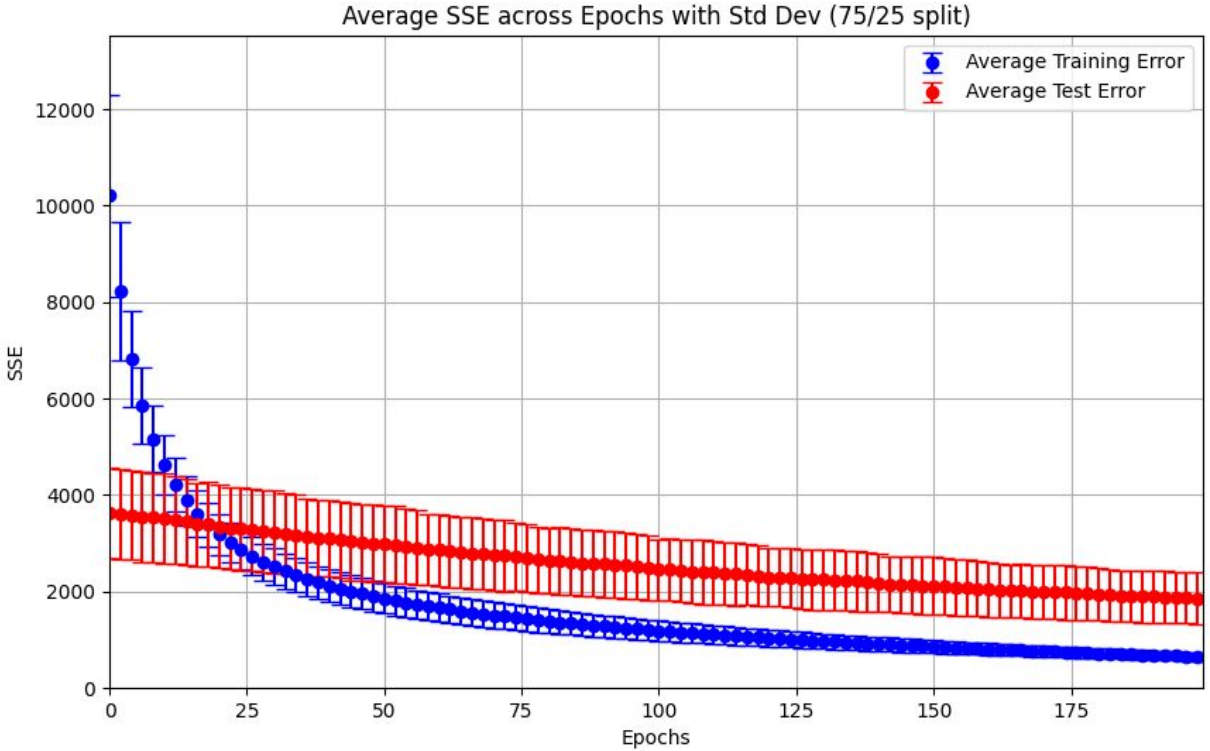
Dataset muy heterogéneo y chico



Perceptrón no lineal
Funcion de activacion tanh
Learning rate = 0.01
5 runs
75% training
25% testing

Haciendo shuffle de los datos

Dataset muy heterogéneo y chico



Perceptrón no lineal
Funcion de activacion tanh
Learning rate = 0.01
5 runs
75% training
25% testing

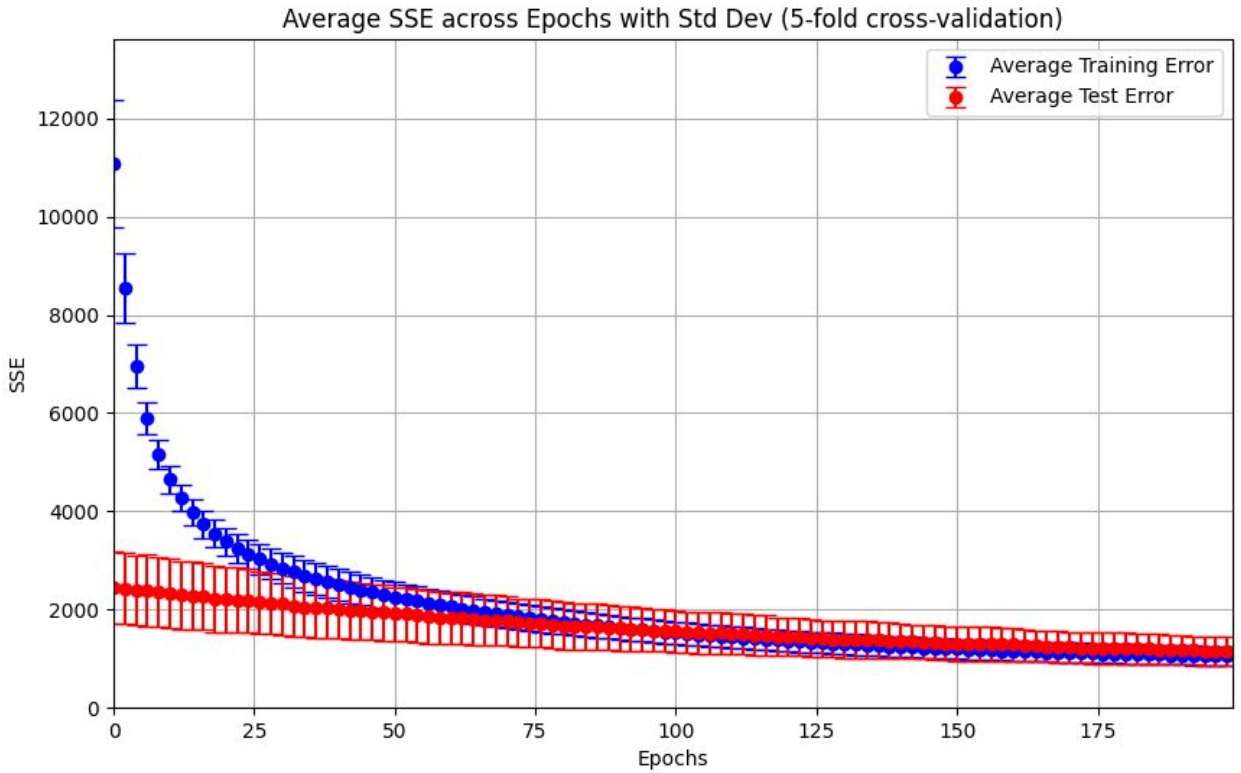
Haciendo shuffle de los datos

Conclusiones

- Varía mucho la capacidad de generalización y aprendizaje según desde qué datos del dataset se usen para entrenarlo

Cross validation

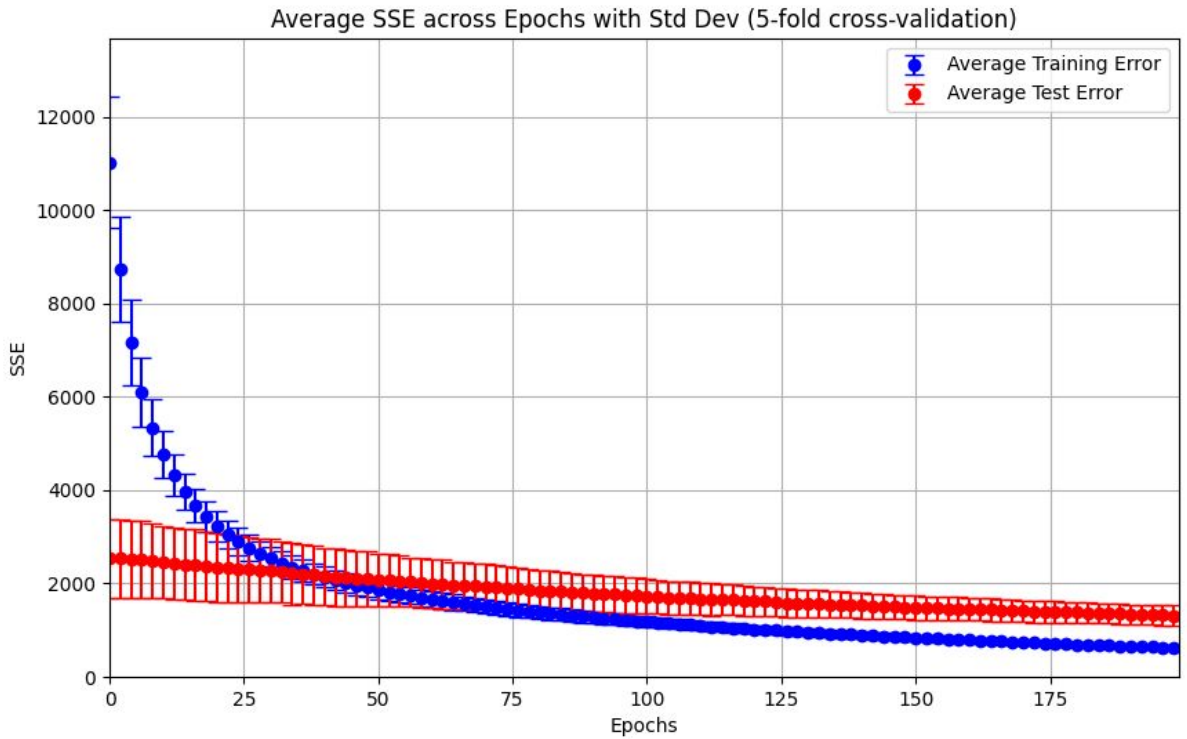
SSE del Lineal a lo largo de las epochs



Perceptrón lineal
Funcion de activacion identidad
Learning rate = 0.01
5 runs
5 folds

2000 epoch mean training SSE: 997.88 +- 616.22
2000 epoch mean testing SSE: 750.13 +- 573.76

SSE del No Lineal a lo largo de las epochs



Perceptrón no lineal
Funcion de activacion tanh
Learning rate = 0.01
5 runs
5 folds

2000 epoch mean training SSE: 248.44 +- 714.89
2000 epoch mean testing SSE: 560.54 +- 497.57

Conclusiones

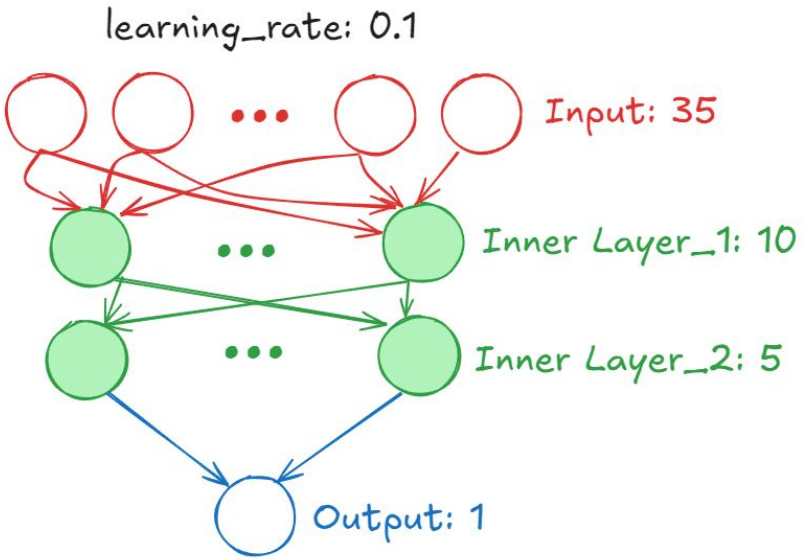
- Para estos parámetros vemos que el no lineal tiene mejor capacidad de aprendizaje y de generalización que el lineal

- En datasets muy desbalanceados es importante utilizar k-fold cross validation para realizar mediciones más realistas

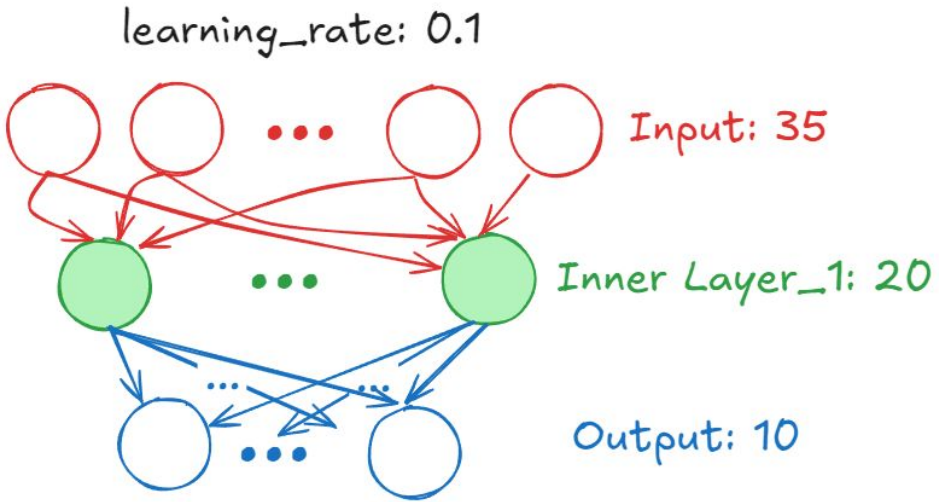
Ej 3

Arquitectura

- Ej 3.2:



- Ej 3.3:



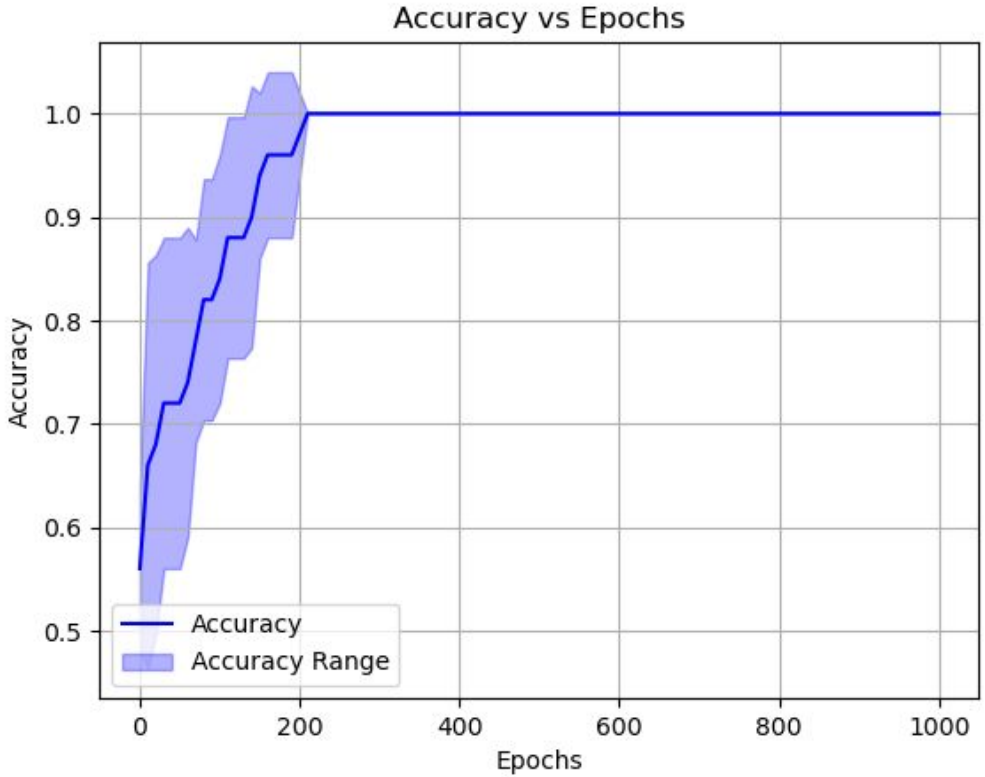
Función de activación: sigmoide

Training Set

- Al ser tan escasos los datos decidimos usar 100% de training set y 0% test set.

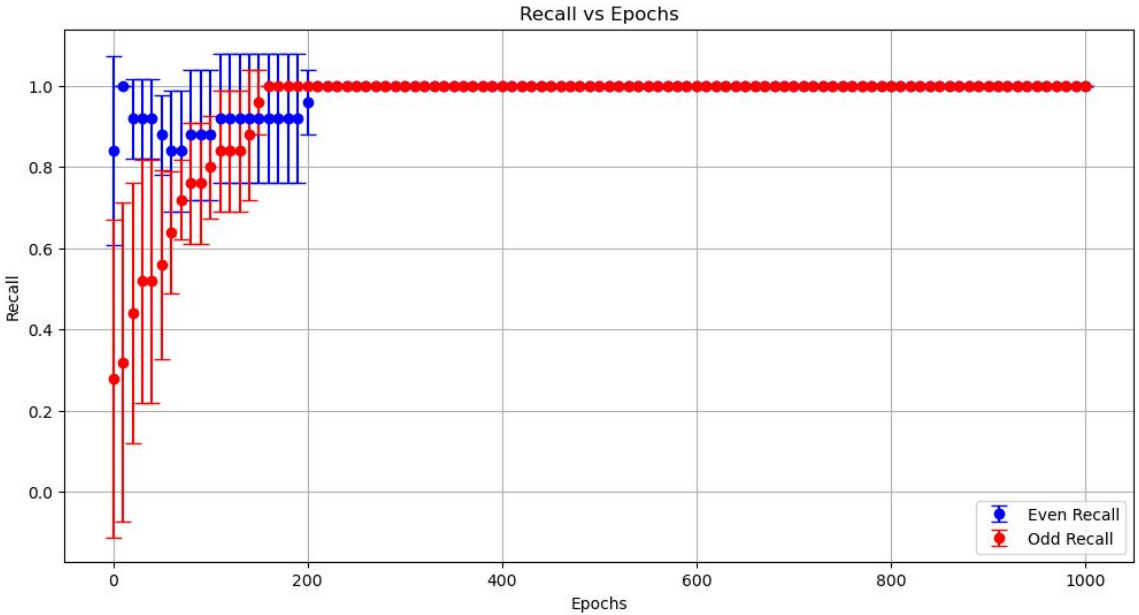
Métricas - Par e Impar

Accuracy - Par e Impar



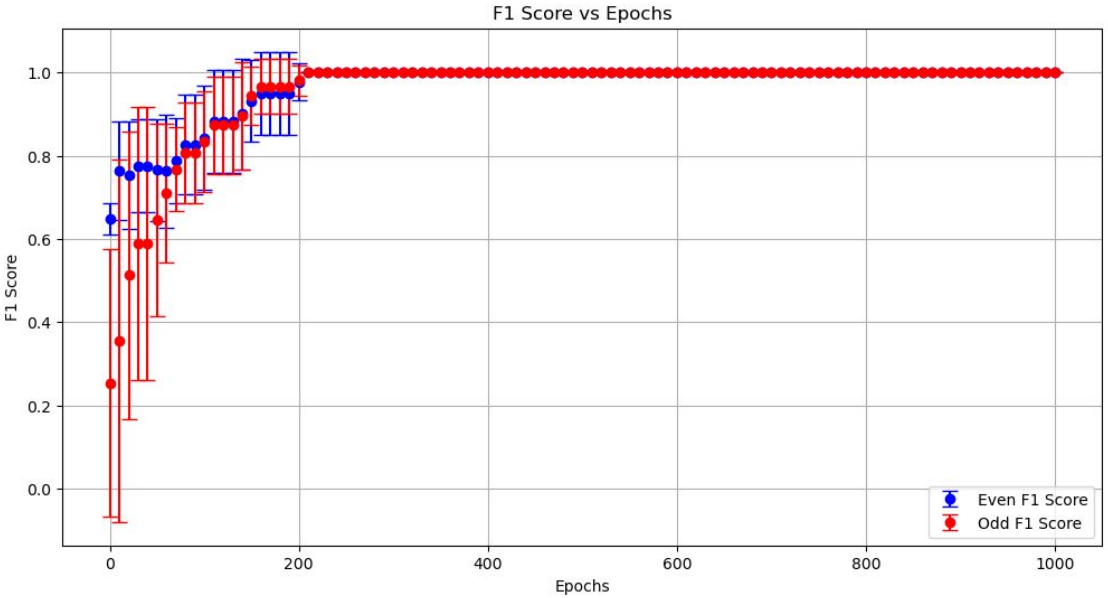
Runs: 5
Inner Layer: 10, 5
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 1000

Recall- Par e Impar



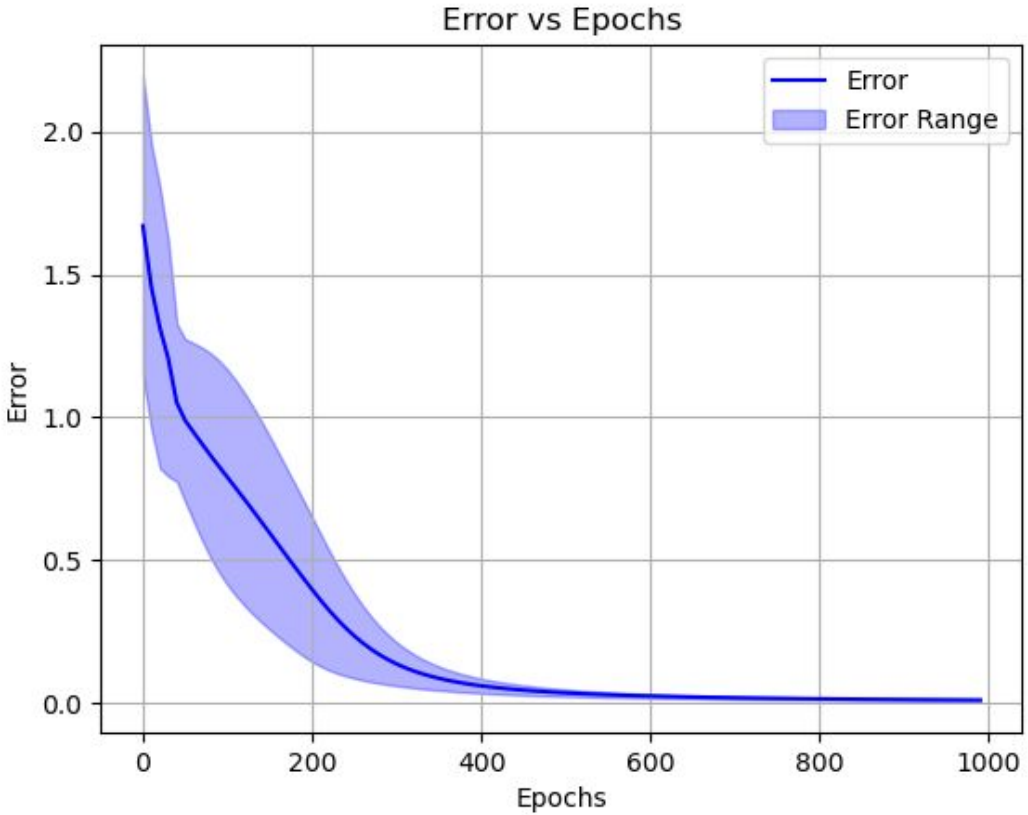
Runs: 5
Inner Layer: 10, 5
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 1000

F1- Score - Par e Impar



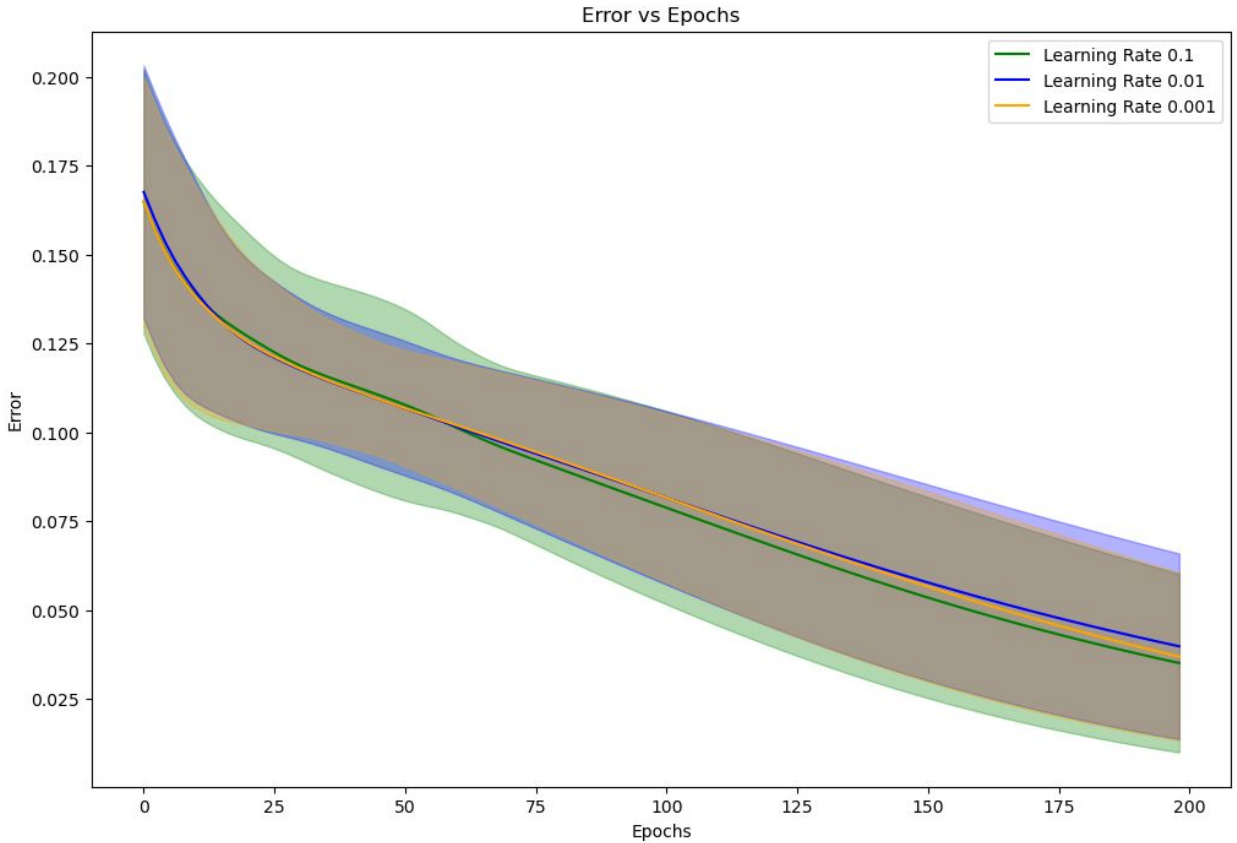
Runs: 5
Inner Layer: 10, 5
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 1000

Error - Par e Impar



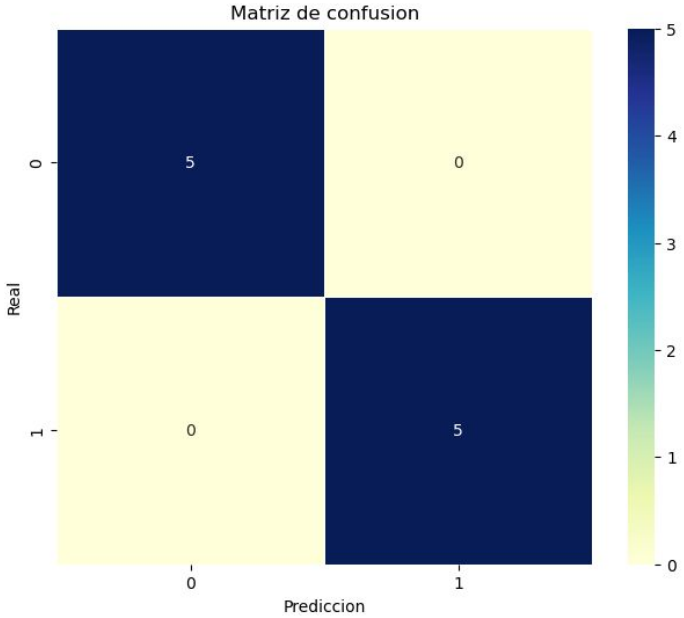
Runs: 5
Inner Layer: 10, 5
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 1000

Learning Rate - Par e Impar



Runs: 100
Inner Layer: 10, 5
Optimizer: vanilla
Epochs: 200

Matriz de confusión - Par e Impar



Runs: 5
Inner Layer: 10, 5
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 1000

Conclusión

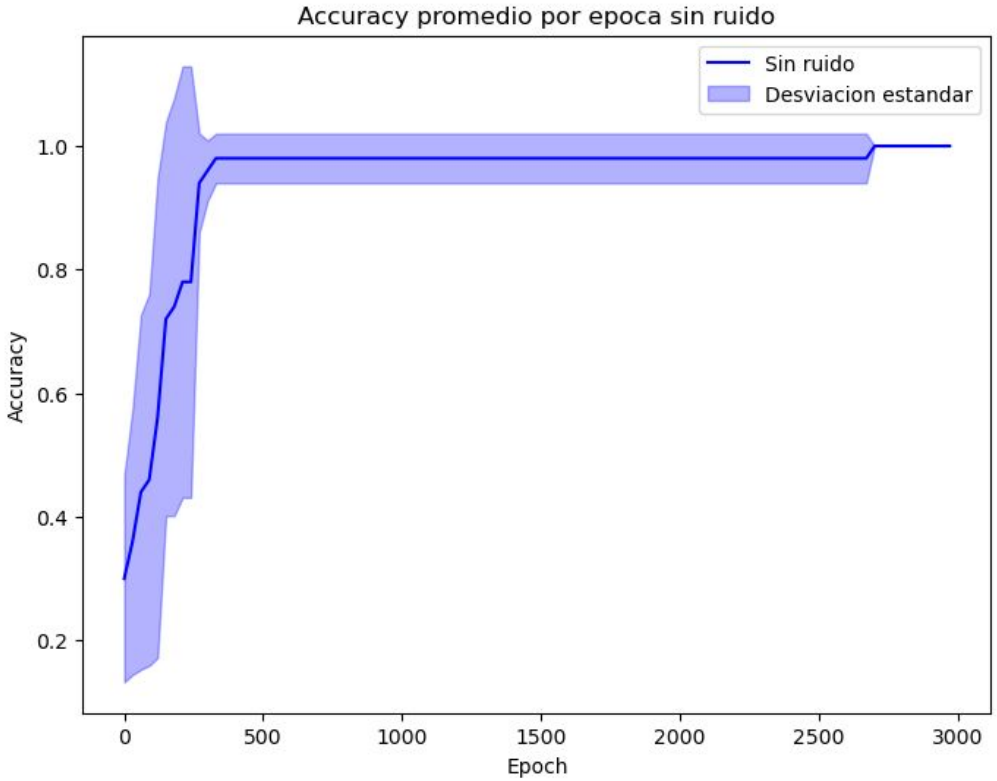
- Podemos notar gracias a las métricas que desagregan las clases (como f1-score) que el modelo tiene una mayor dificultad al comienzo para la identificacion de numeros impares, pero accuracy no me provee esta información.
- El learning rate afecta significativamente la velocidad con la que se alcanza un error deseable (cercano a 0).

Métricas - Dígitos

Train & Test Sets

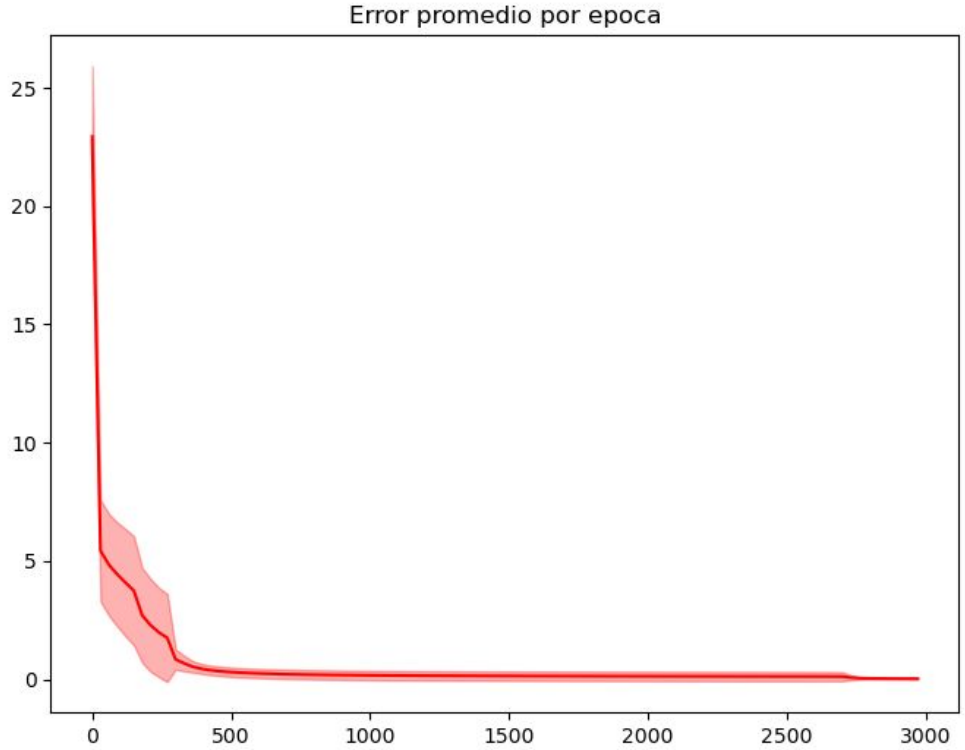
- Se entrena con el set de dígitos sin modificación provisto por la cátedra.
- Se generaliza con sets de dígitos con ruido gaussiano(α):
    - 100 Interpretaciones de cada dígito.
    - Con α tomando valores de 0.3, 0.5, 0.7.

Accuracy - Dígitos



Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 3000

S.S.E Promedio - Dígitos

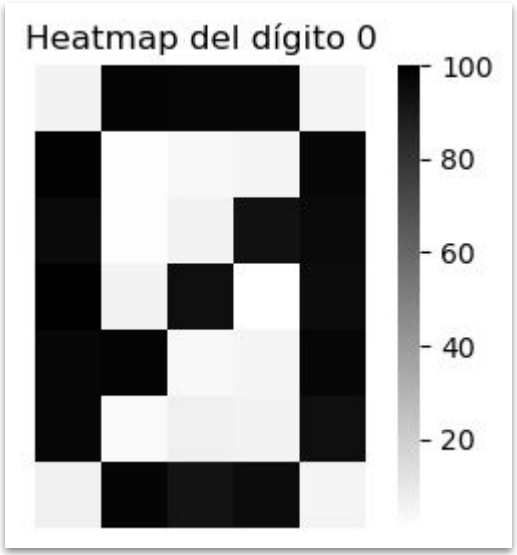


Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 3000

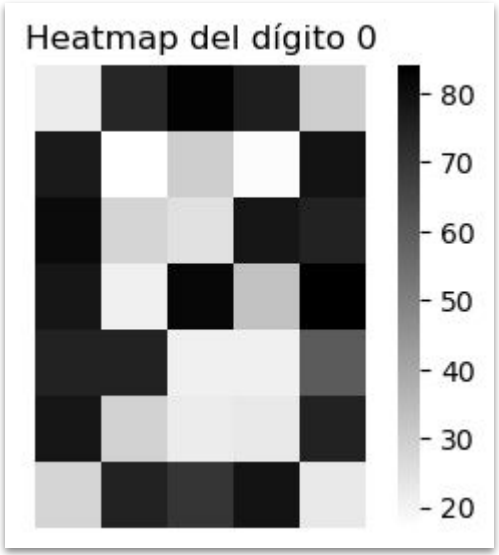
Ruido Gaussiano

Comparación Test Set - 0.3 vs. 0.7

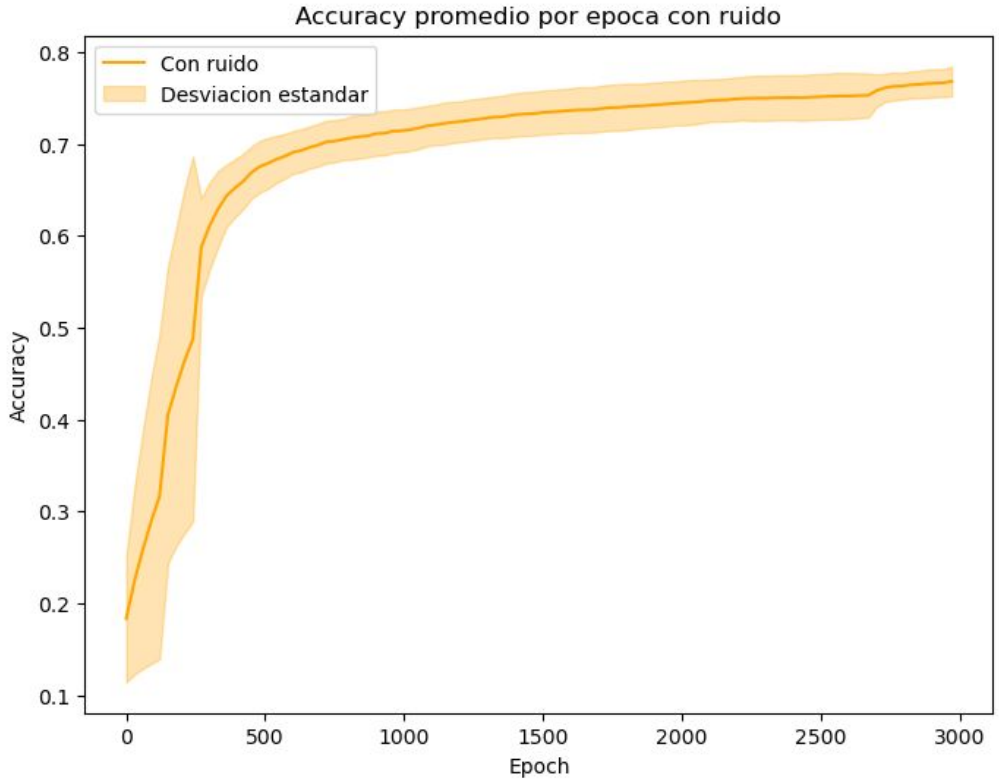
Ruido Gaussiano: 0.3



Ruido Gaussiano: 0.7

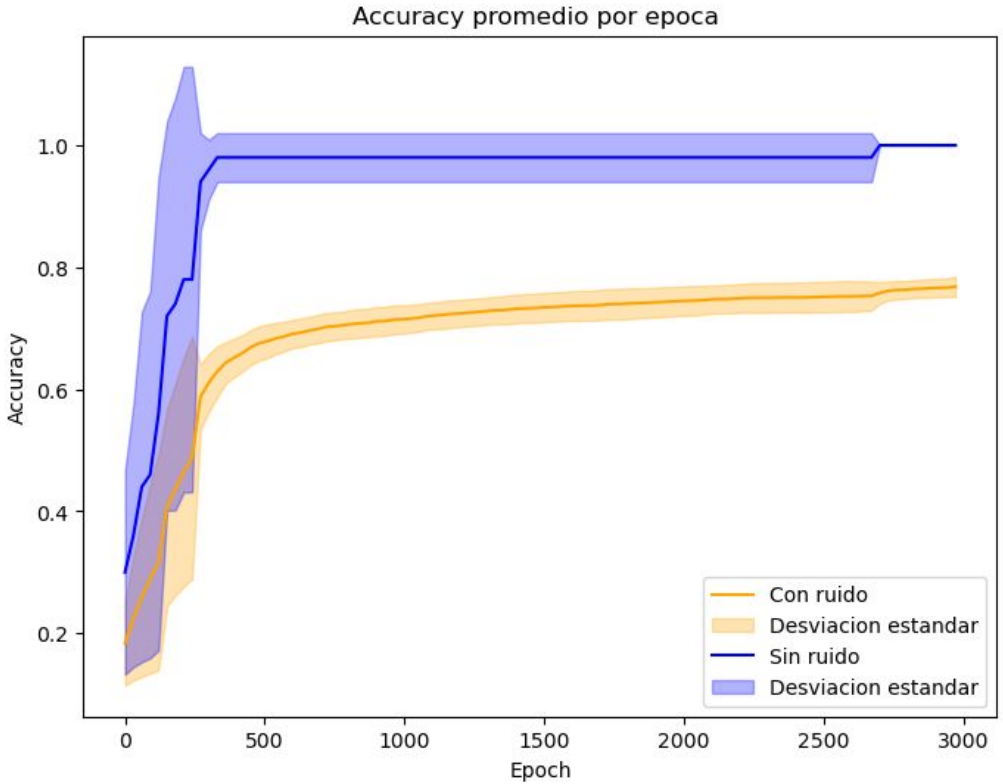


Accuracy - Dígitos



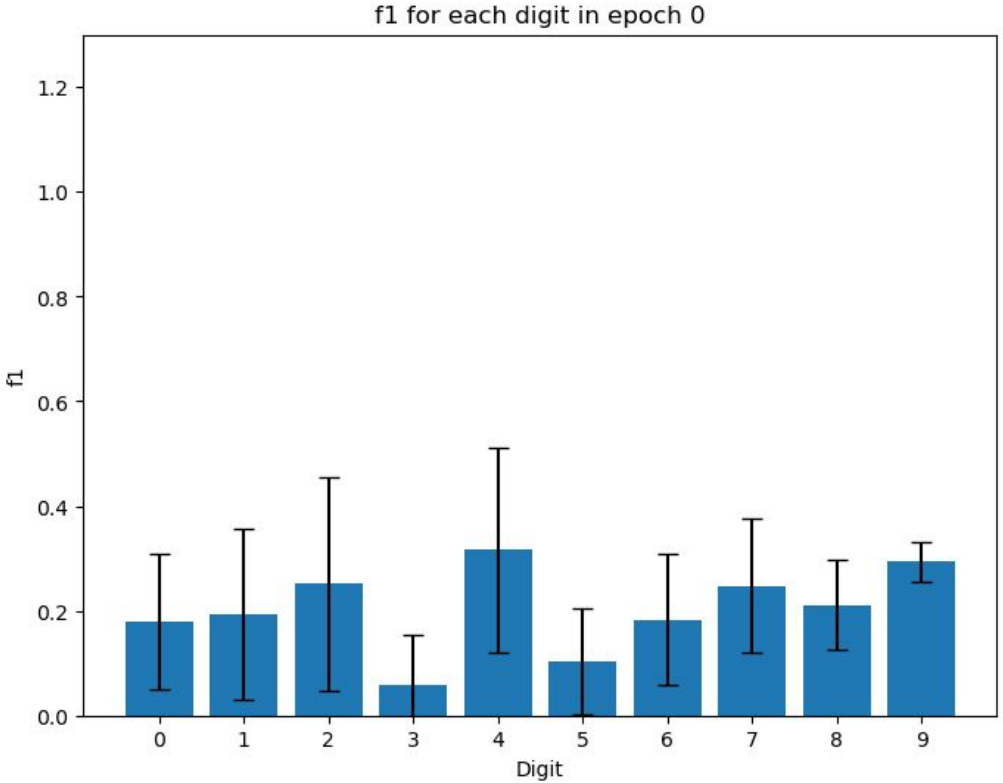
Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 3000
ruido(α): 0.5

Accuracy - Con Ruido vs. Sin Ruido



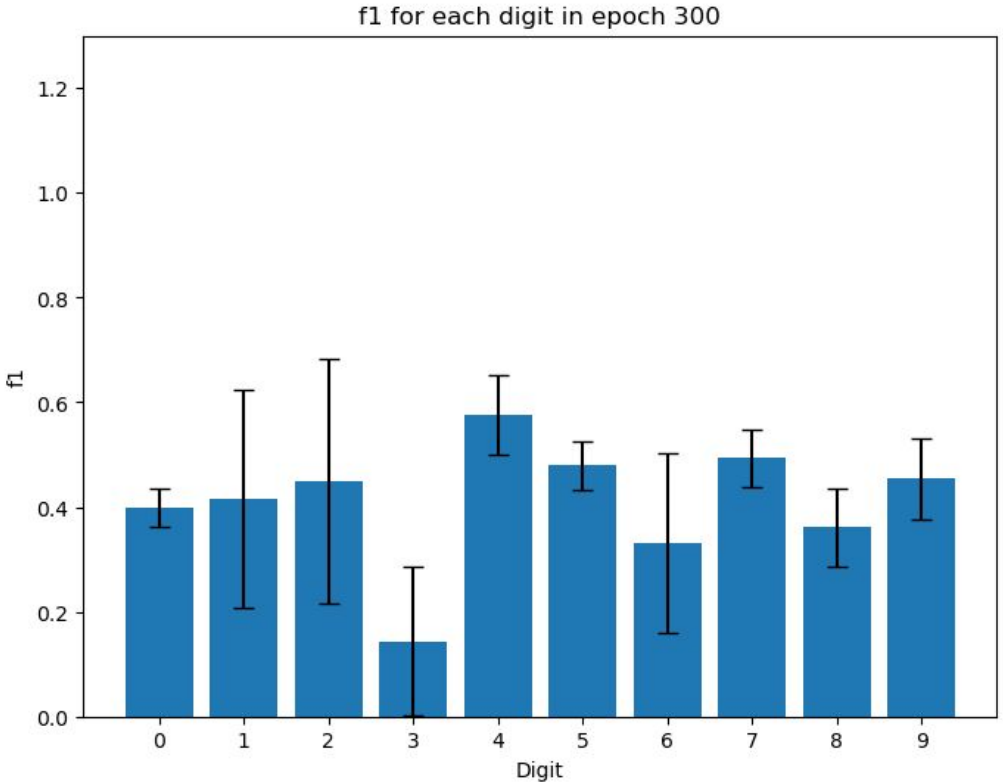
Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 3000
ruido(α): 0.5

F1-Score (Epoch 0) - Dígitos



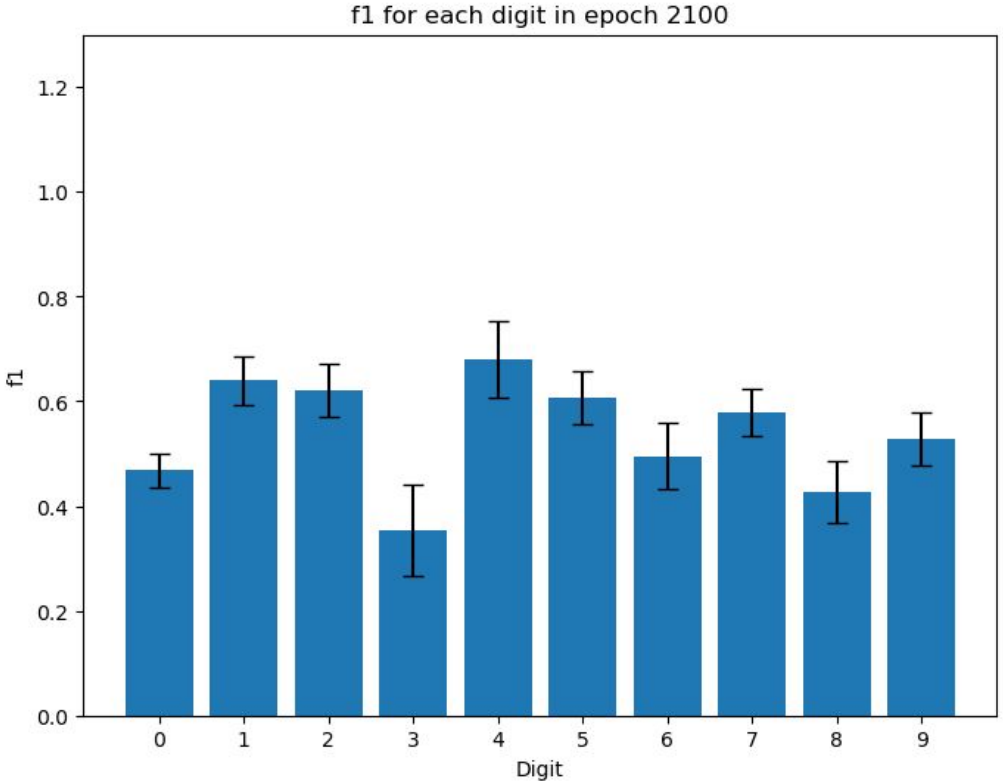
Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 5000
ruido(α): 0.7

F1-Score(Epoch 300) - Dígitos



Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 5000
ruido(α): 0.7

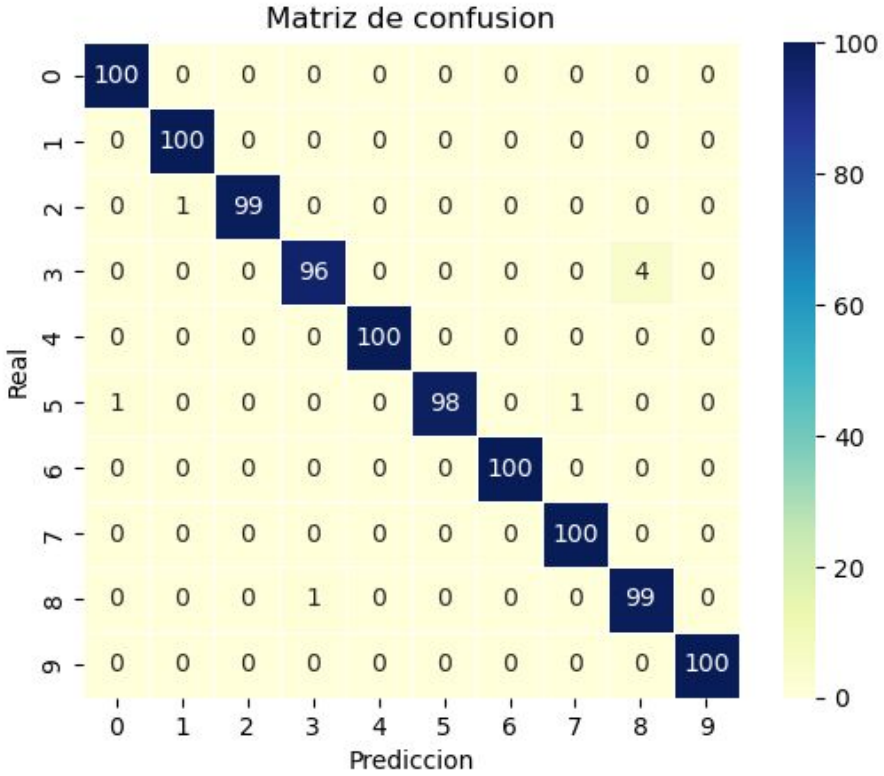
F1-Score (Epoch 2100) - Dígitos



Runs: 5
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 5000
ruido(α): 0.7

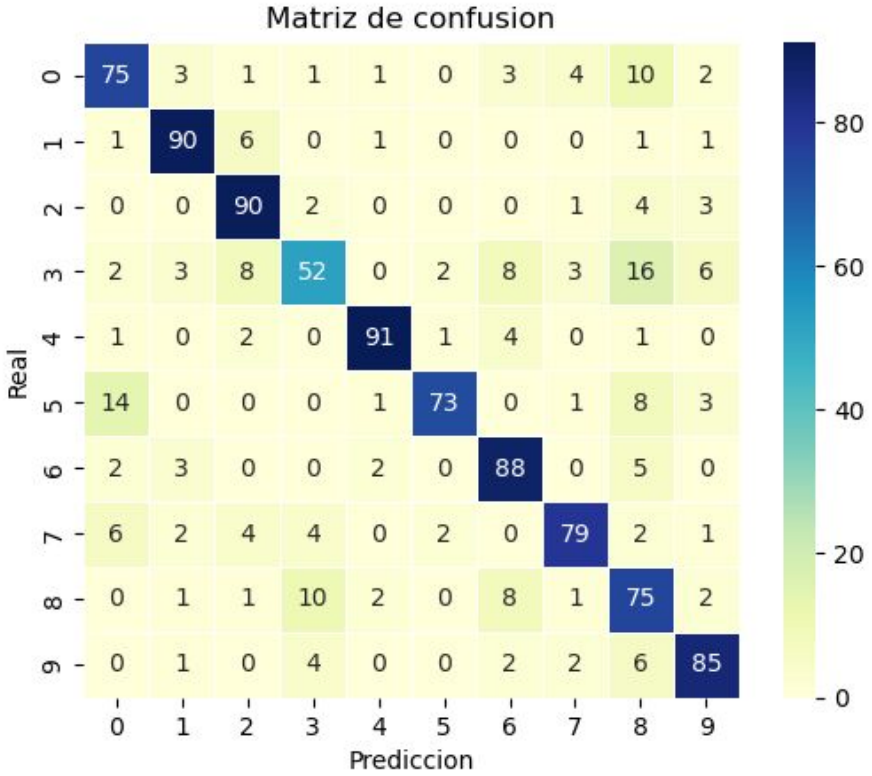
Análisis con distintos ruidos

Matriz de confusión - Dígitos



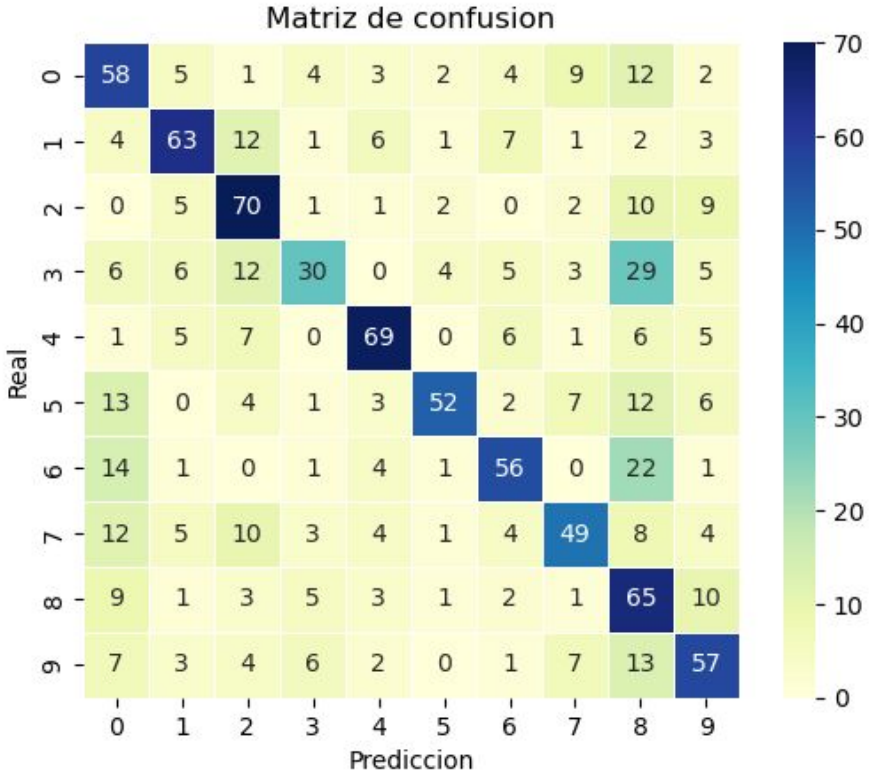
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 5000
ruido(α): 0.3

Matriz de confusión - Dígitos



Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 5000
ruido(α): 0.5

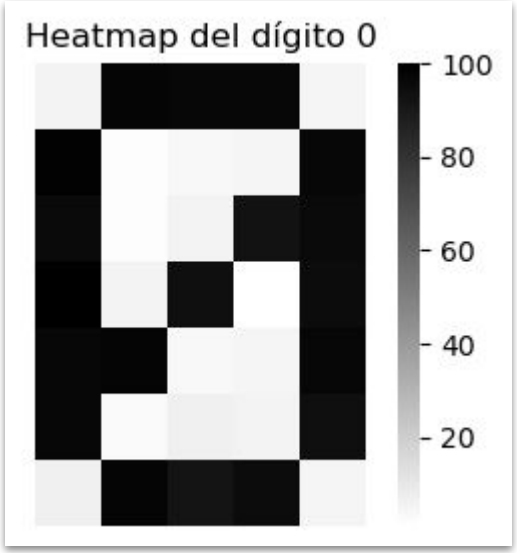
Matriz de confusión - Dígitos

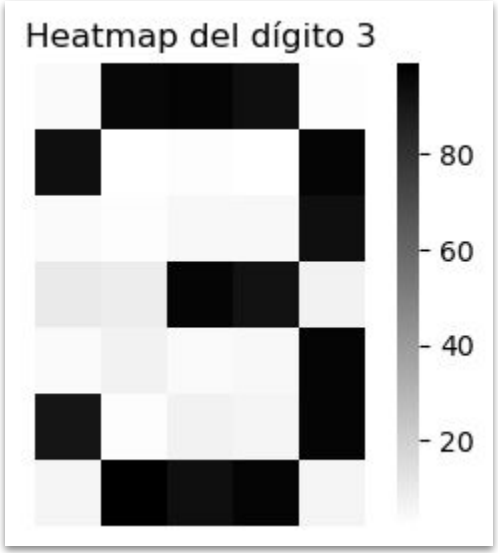


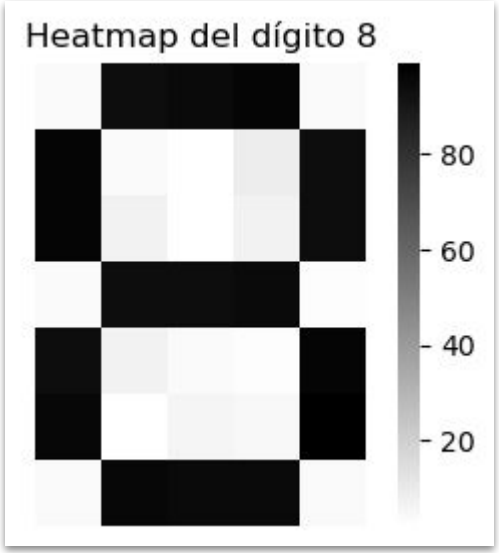
Inner Layer: 20
Learning Rate: 0.1
Optimizer: vanilla
Epochs: 5000
ruido(α): 0.7

Comparación Test Set - 0.3 vs. 0.7

Ruido Gaussiano: 0.3

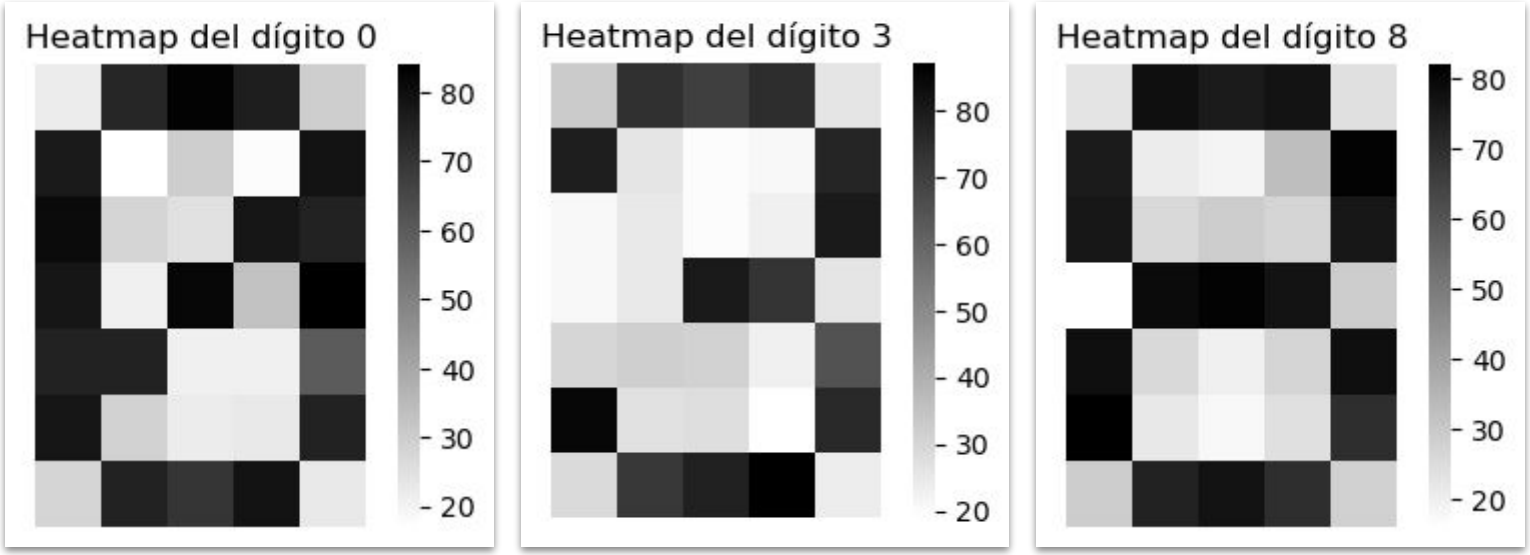






Comparación Test Set - 0.3 vs. 0.7

Ruido Gaussiano: 0.7



Conclusión

- El nivel de ruido genera patrones similares en algunos grupos de números haciendo que tengan peor desempeño.
- La evolución de f1-score nos permite ver que numeros son los que peor se generalizan(Vemos que 0 3 y 8 son los de peor desempeño).
- Existe una correlación directa entre ruido y error, a mayor ruido peores resultados.

Ej 4

MNIST Dataset & Persistencia

MNIST Dataset

```python
def load_preprocessed_mnist():
    # Load and preprocess the MNIST dataset
    mnist = tf.keras.datasets.mnist
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Normalize the data to [0, 1] range
    x_train = x_train.astype('float32') / 255
    x_test = x_test.astype('float32') / 255

    # Flatten the 28x28 images into 1D arrays of 784 pixels
    x_train = x_train.reshape(60000, 784)
    x_test = x_test.reshape(10000, 784)

    # One-hot encode the labels (convert labels to 10-dimensional vectors)
    y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
    y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

    return (x_train, y_train), (x_test, y_test)
```

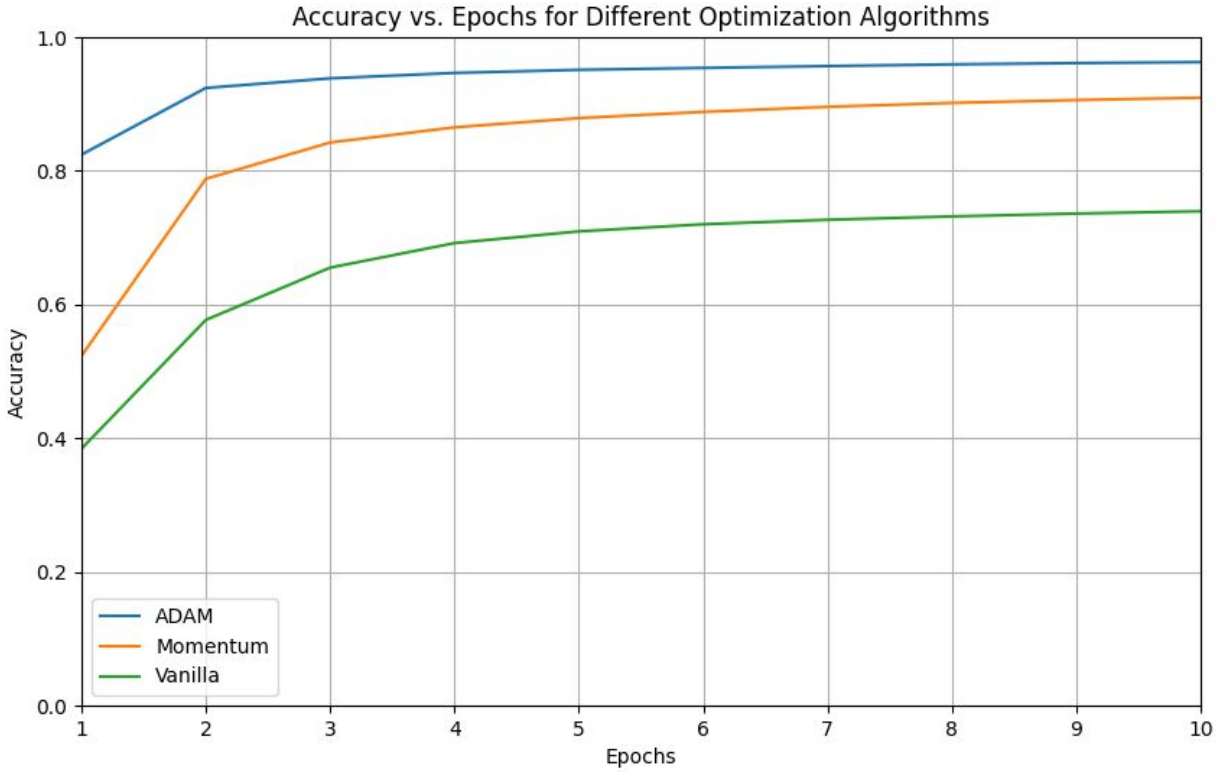
Persistencia del modelo con DILL

```python
def store_model(mlp, model_filename):
    # Save the trained model to a file
    with open(model_filename, 'wb') as model_file:
        dill.dump(mlp, model_file)


8 usages
def load_model(model_filename):
    # Load the trained model from a file
    with open(model_filename, 'rb') as model_file:
        return dill.load(model_file)
```

Hiperparametros

Optimizadores



Tiempos:

- ADAM: 1350.63s
- Momentum: 639.84s
- Vanilla: 550.91s

Learning Rate:

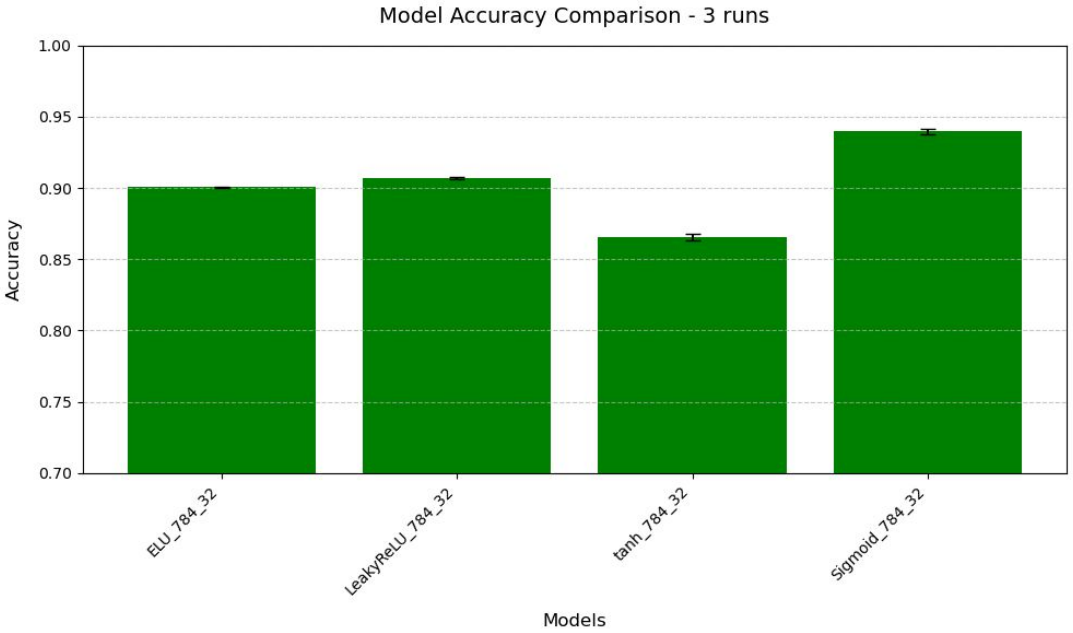
- Vanilla, Momentum: 0.01
- ADAM: 0.001

Función de activación Sigmoide
ADAM: beta1=0.9, beta2=0.999, epsilon=1e-8
Momentum: alpha=0.9
Inner layer: 32
Training

Conclusiones

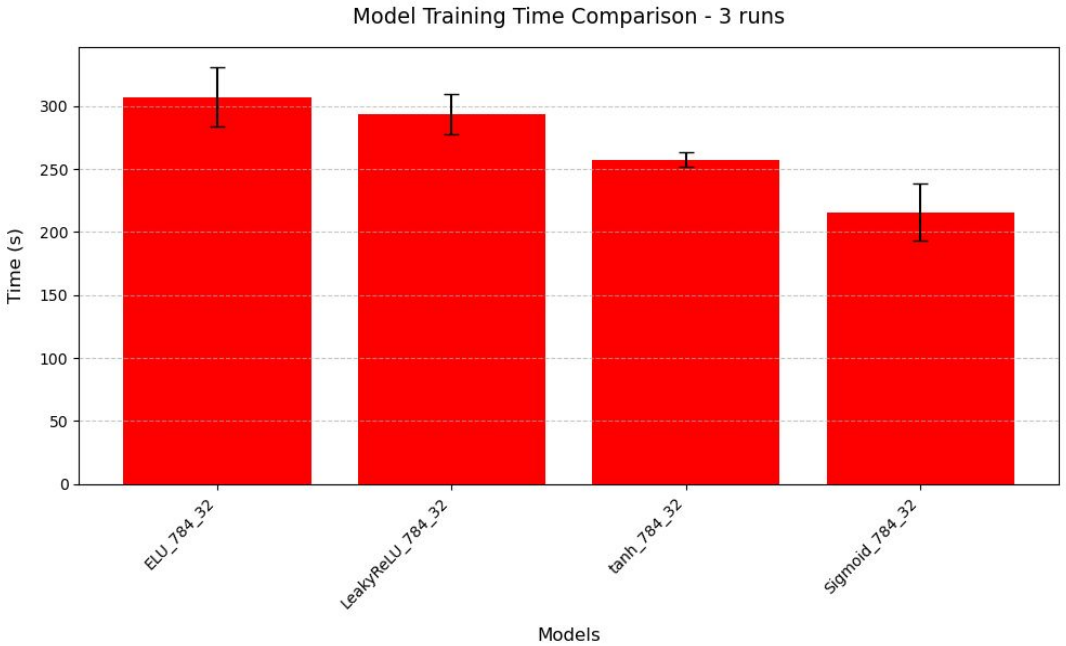
- ADAM mejora mucho más por epoch debido a su optimización de dirección y magnitud en el cálculo gradiente
- Estas optimizaciones son costosas y hacen que tarde mucho más el training para la misma cantidad de epochs
- Utilizar GPU para paralelizar los algoritmos es una manera eficiente de reducir los tiempos debido a la naturaleza del algoritmo

Funciones de activación



- Cantidad de epochs: **3**
- Cantidad de runs: **3**
- Optimizador: **ADAM** *(default config.)*
- Estructura de la red: **784 - 32 - 10**

Funciones de activación



- Cantidad de epochs: **3**
- Cantidad de runs: **3**
- Optimizador: **ADAM** *(default config.)*
- Estructura de la red: **784 - 32 - 10**

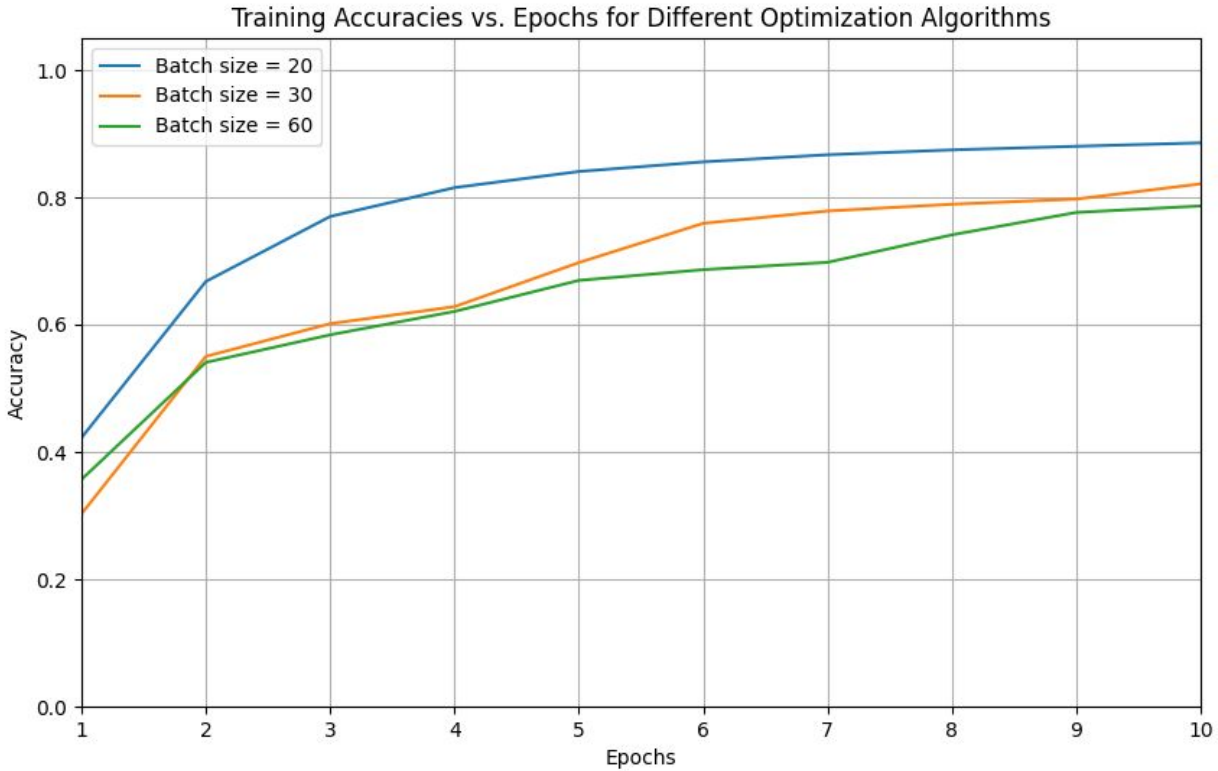
Conclusiones

- Notar que no usamos F1, usamos accuracy ya que el dataset de MNIST es variado y relativamente grande (tiene 60000 entradas en el training set)

- La mejor función de activación en este caso fue ‘Sigmoid’. Fue la que menos tiempo tardo y mayor accuracy alcanzó

Resultados

Batching



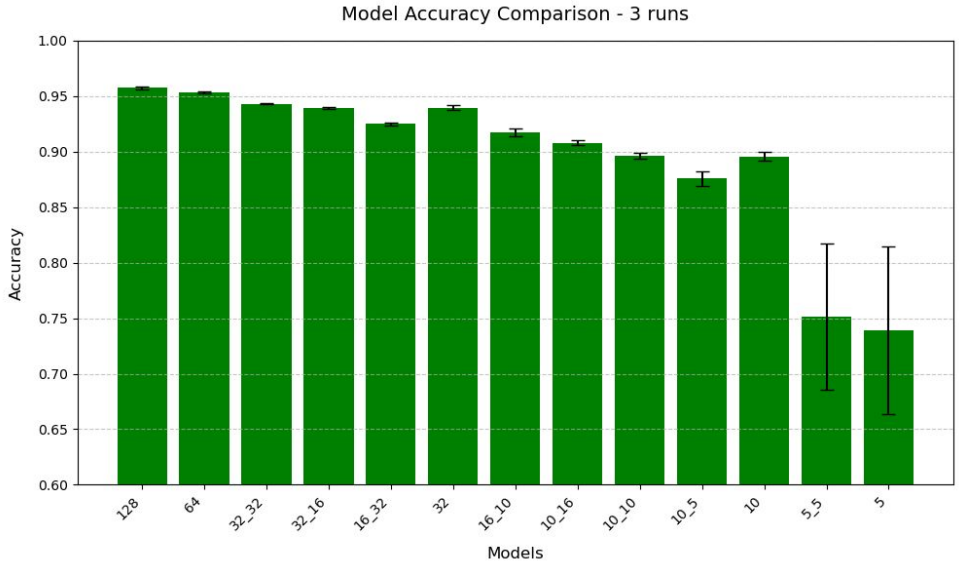
Optimizador: Vanilla
Learning rate: 0.01
Inner layer: 32
Función de activación sigmoide

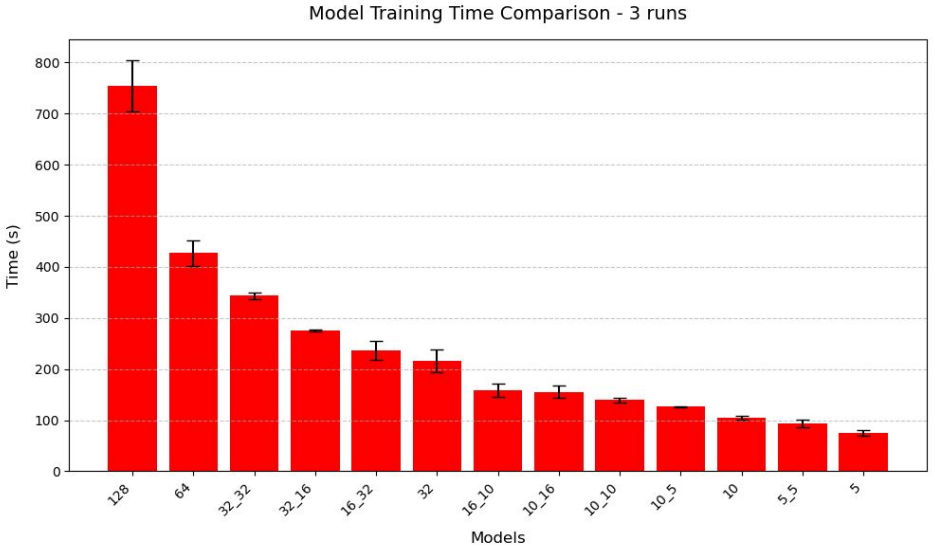
Conclusiones

- Un batch size más pequeño acelera la convergencia del aprendizaje del perceptrón

Estructuras de redes - Capas y Nodos

- Optimizador: **ADAM** *(default config.)*
- Función de activación: **Sigmoide**
- Cantidad de epochs: **3**
- Cantidad de runs: **3**

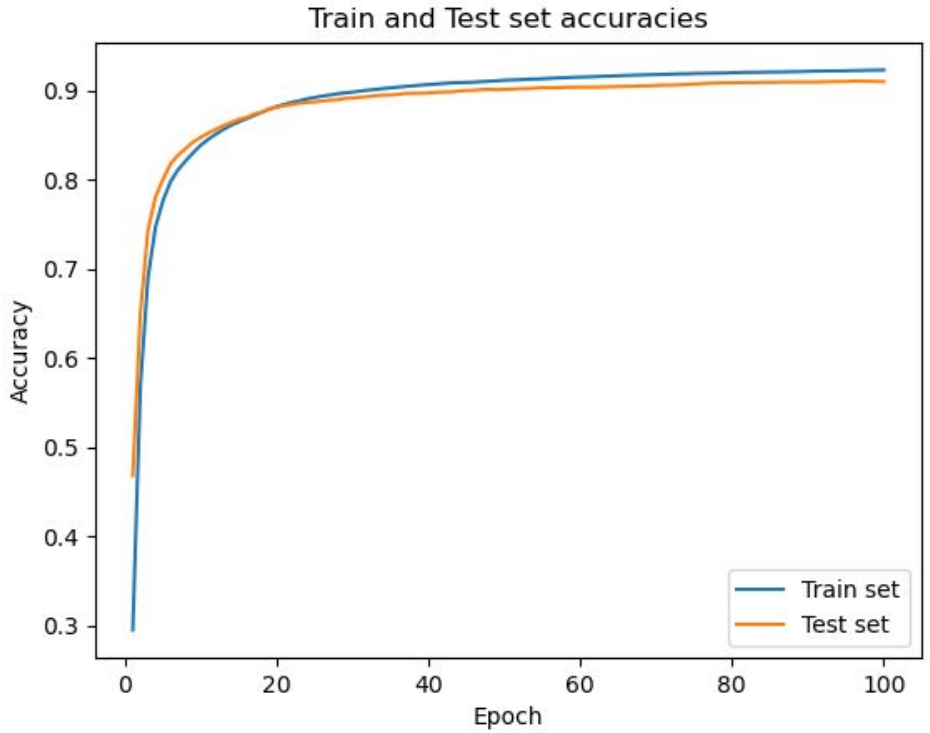




Conclusiones

- Cuantas más neuronas hay más tarda en entrenarse el modelo y mayor accuracy tiene
- La diferencia en accuracy no es proporcional al tiempo
- La cantidad de capas no parece afectar los resultados, pero si la cantidad de neuronas de la capa inicial

Training set vs. Testing set - Accuracy

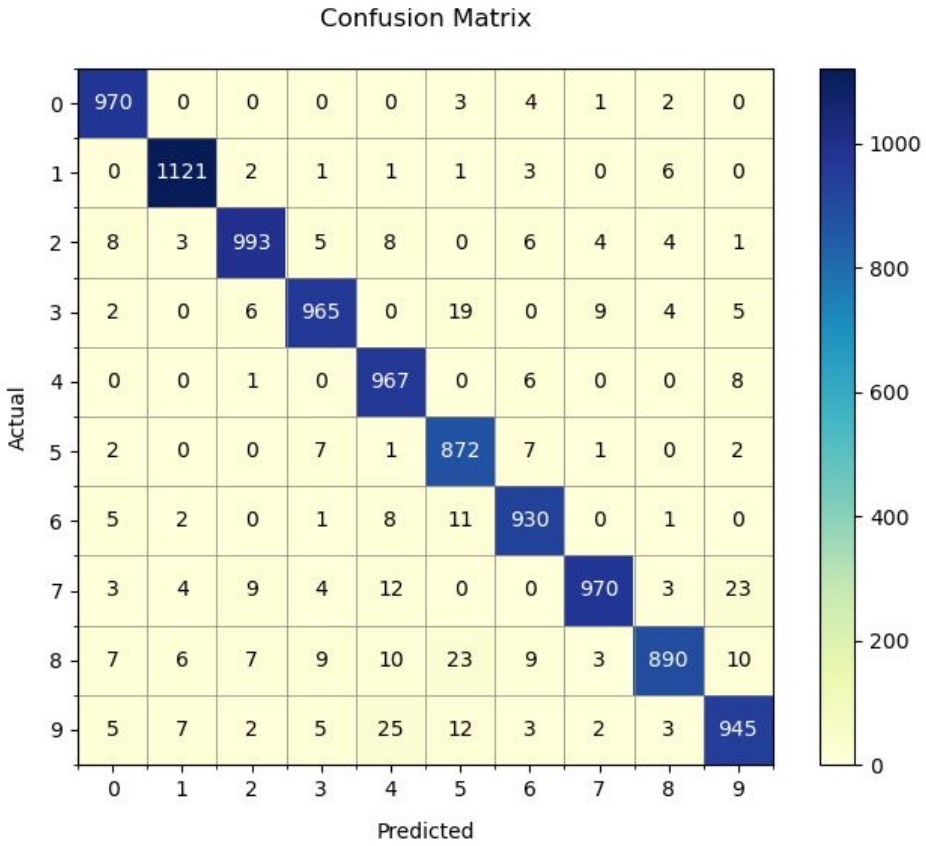


- Optimizador: **VANILLA**
- Función de activación: **Sigmoide**
- Learning rate: **0.01**
- Estructura de red: **784 - 10 - 10**
- Cantidad de epochs: **100**
- Training set: **60000**
- Testing set: **10000**

Conclusiones

- El modelo funciona correctamente
- Parece evitarse el overfitting ya que el conjunto de datos de entrenamiento es grande
- Se muestra VANILLA porque ADAM tarda muchisimo por epoch

Matriz de confusión - Mejor modelo

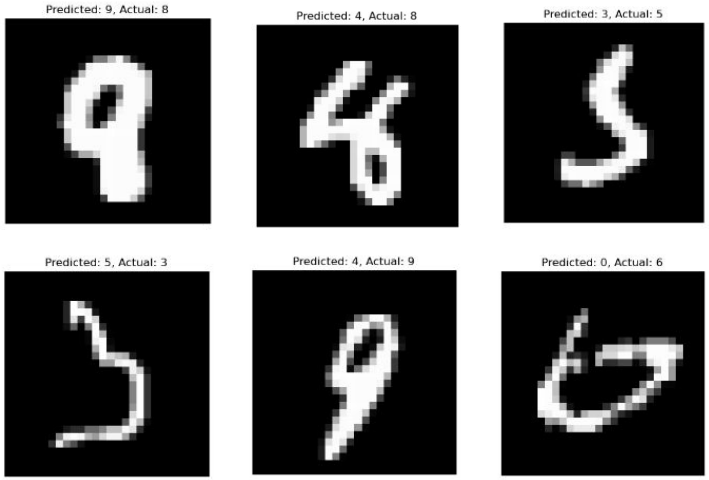


- Optimizador: **ADAM** *(default config.)*
- Estructura de red: **784 - 256 - 128 - 64 - 10**
- Función de activación: **Sigmoide**
- Tiempo de entrenamiento: **18983.45s ≈ 5hs 16m**
- Cantidad de epochs: **10**

Conclusiones

- El modelo predice correctamente en la mayoría de los casos

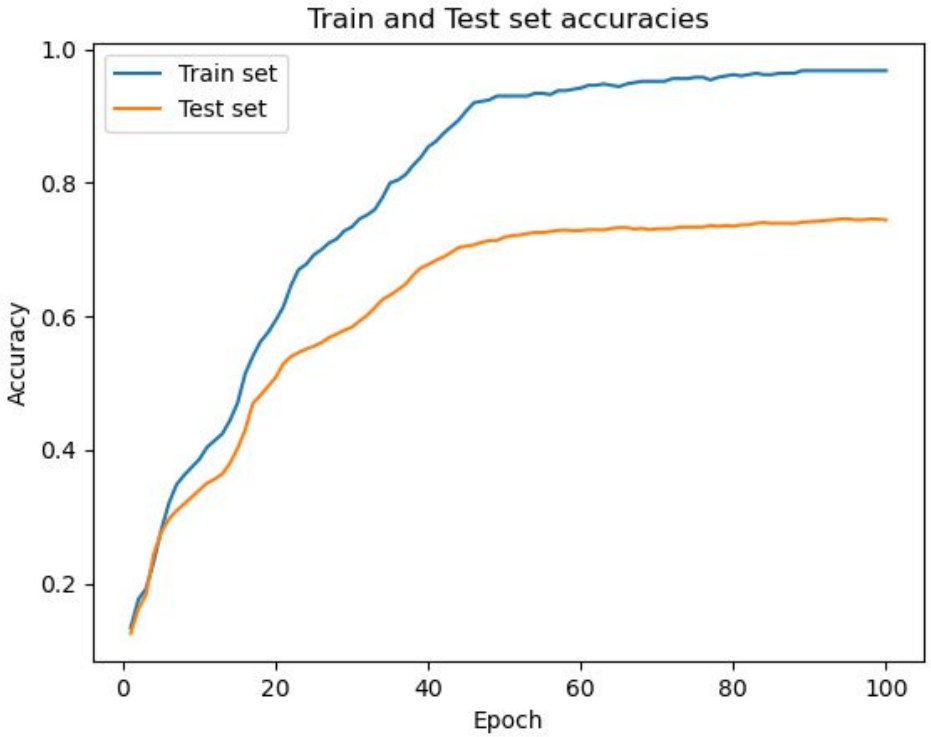
- Se confunde entre números que son “parecidos”
  - 4 y 9,
  - 5 y 8,
  - 9 y 7,
  - 5 y 3



- Tiene accuracy de 0.9758 en el training set y 0.9623 en el test set

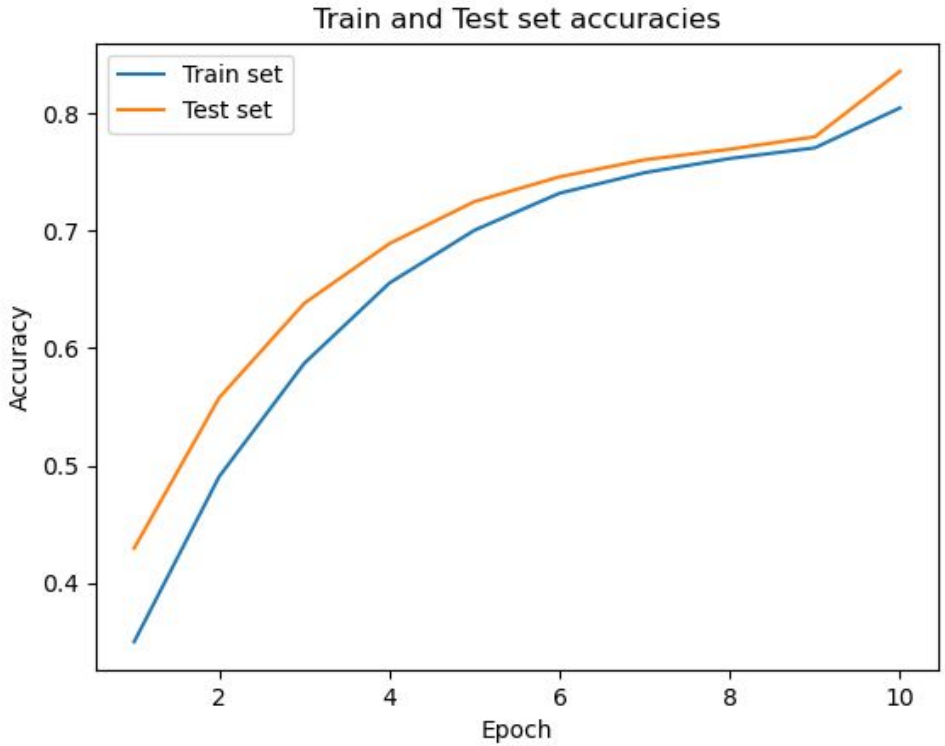
Over & Underfitting

Overfitting 'artificial'



- Optimizador: **VANILLA**
- Función de activación: **Sigmoide**
- Learning rate: **0.01**
- Estructura de red: **784 - 10 - 10**
- Tiempo de entrenamiento: **284.27s**
- Cantidad de epochs: **100**
- Training set: **500**
- Testing set: **10000**

Underfitting 'artificial'



- Optimizador: **VANILLA**
- Función de activación: **Sigmoide**
- Learning rate: **0.01**
- Estructura de red: **784 - 10 - 10**
- Tiempo de entrenamiento: **190.66s**
- Cantidad de epochs: **10**
- Training set: **60000**
- Testing set: **10000**

Conclusiones

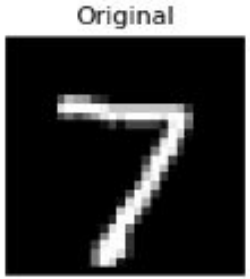
- El modelo queda ‘overfitted’ cuando se lo entrena con demasiadas epochs y un training set suficientemente chico para que no pueda predecir datos que no estén en su training set

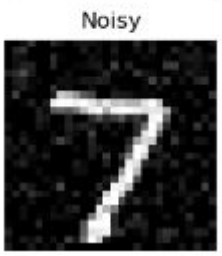
- El modelo queda ‘underfitted’ cuando se lo entrena con muy pocas epochs y un training set suficientemente grande tal que el modelo no termina de ‘aprender’

- Ambos causan que la red neuronal entrenada no alcance su máximo potencial

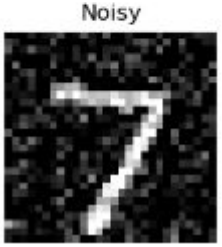
Error Gaussiano

Error Gaussiano

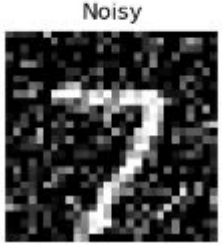




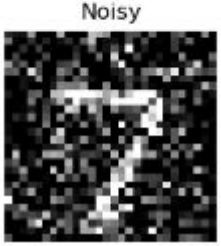
$\alpha = 0.1$



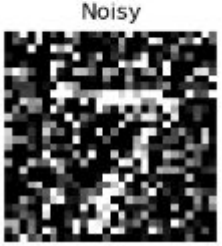
$\alpha = 0.2$



$\alpha = 0.3$



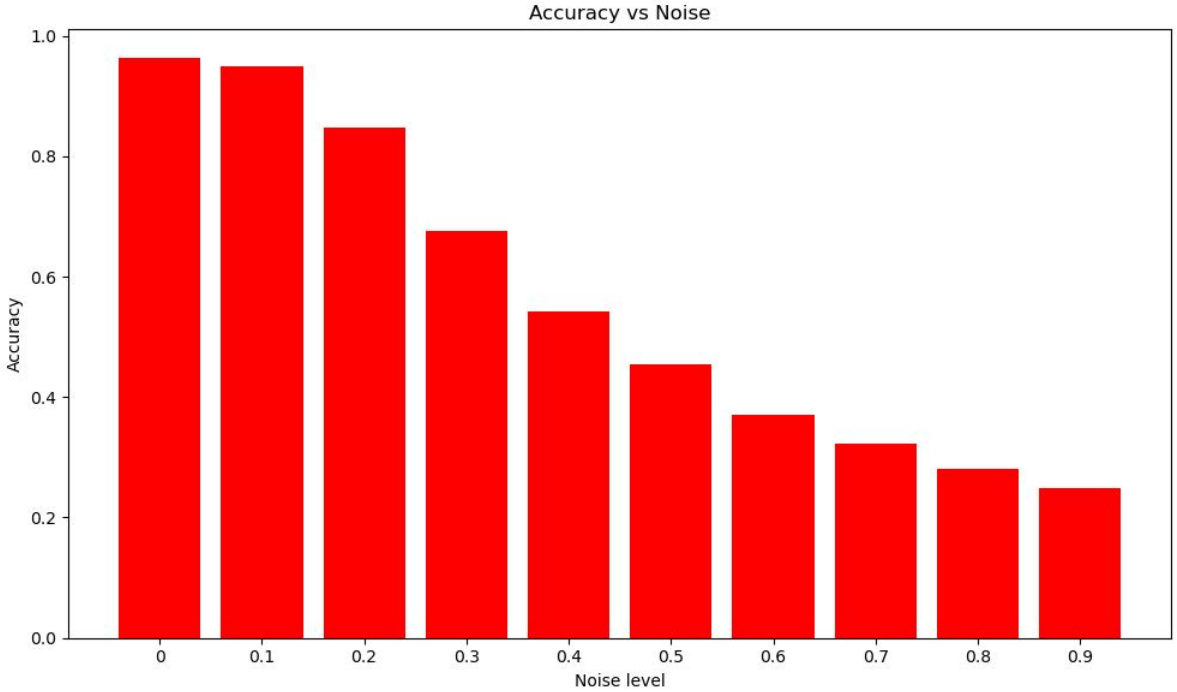
$\alpha = 0.4$



$\alpha = 0.6$

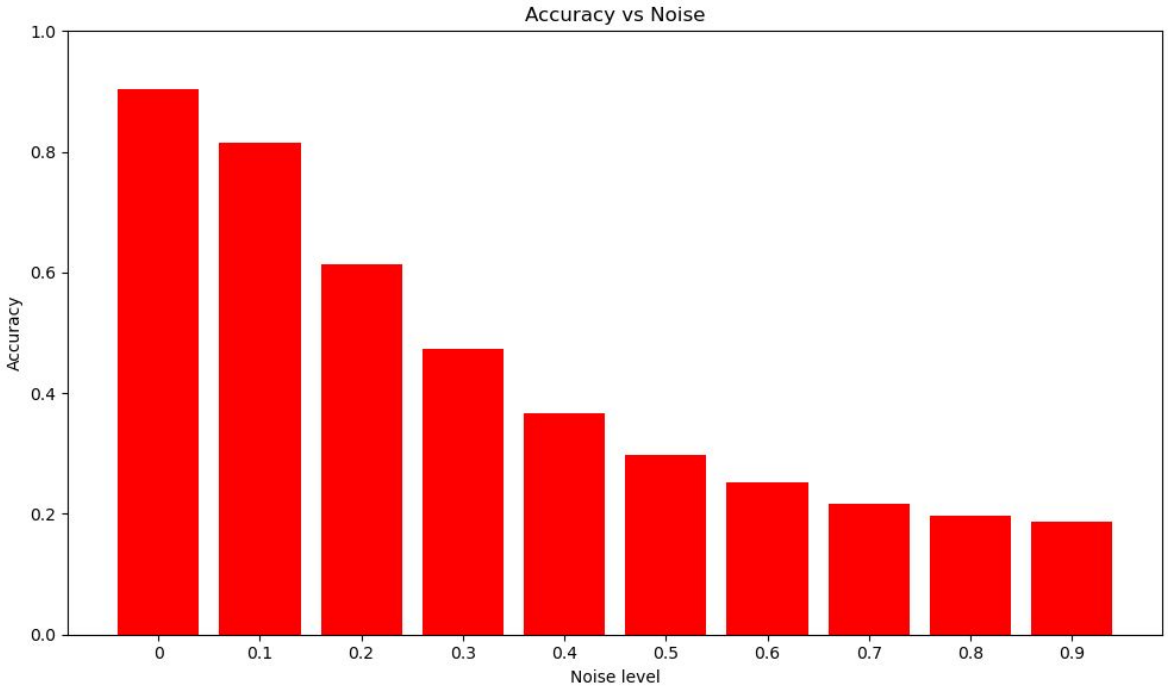
$\alpha = 0.8$

Error Gaussiano



- Optimizador: **ADAM** *(default config.)*
- Estructura de red: **784 - 256 - 128 - 64 - 10**
- Función de activación: **Sigmoide**
- Tiempo de entrenamiento: **18983.45s = 5hs 16m**
- Cantidad de epochs: **10**

Error Gaussiano

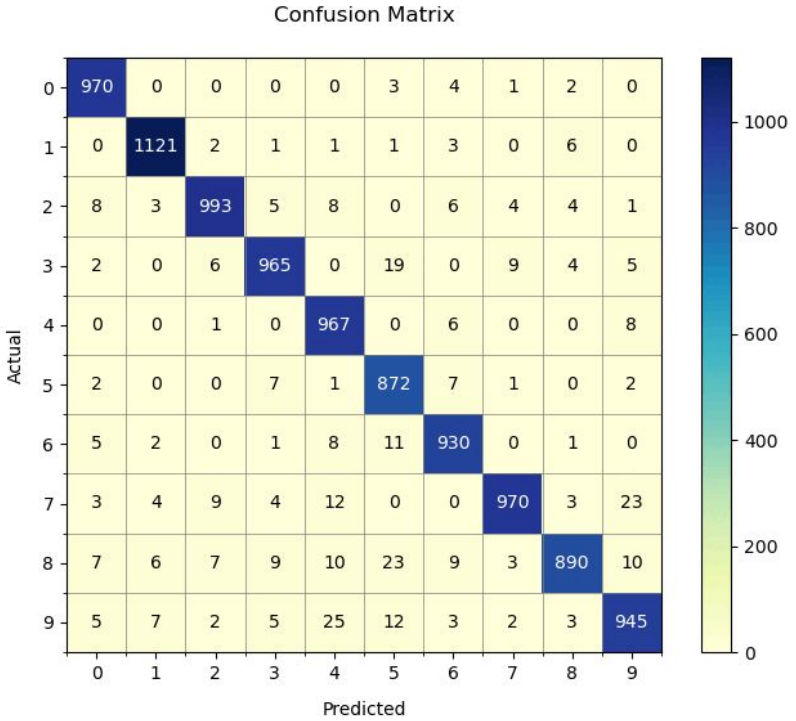


- Optimizador: **ADAM** *(default config.)*
- Estructura de red: **784 - 10 - 10**
- Función de activación: **Sigmoide**
- Tiempo de entrenamiento: **107.75s**
- Cantidad de epochs: **3**

Conclusiones

- La red neuronal con estructura **784 - 256 - 128 - 64 - 10**, tiene mayor resistencia al ruido comparado a la red neuronal con estructura **784 - 10 - 10**

- Cuanto más aumenta el ruido (α) menor es el accuracy



Gracias por su atención :)