Perceptron Simple y Multicapa

Grupo 4

- Gastón Alasia, 61413
- Juan Segundo Arnaude, 62184
- Bautista Canevaro, 62179
- Matías Wodtke, 62098

EJ 2

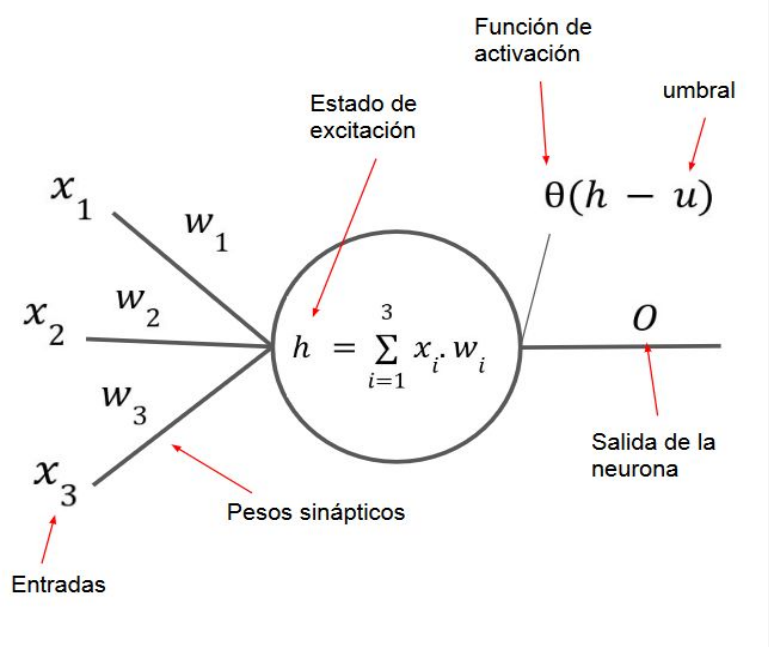
Implementamos el algoritmo del perceptrón simple lineal y no lineal. Utilizamos ambos para aprender a clasificar los datos en el archivo.

- Comparamos la capacidad de aprendizaje
- Tomaremos aquel con mayor potencial de aprendizaje y evaluaremos su capacidad de generalización

Las preguntas que buscamos responder son:

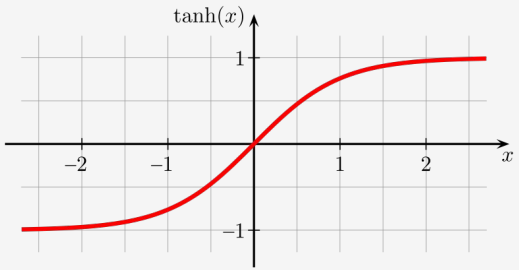
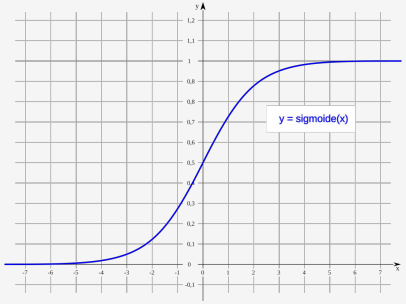
- ¿Cómo es la mejor forma de elegir el conjunto de entrenamiento?
- ¿Esta forma de selección tiene un impacto en la capacidad de generalización?

Perceptrones



- Lineal
- No-Lineal: funciones de activación tanh y logística

Normalizacion

Funcion	Imagen
	(-1 ; 1)
logística 	(0 ; 1)

Pero el rango del dataset es [0.32 ; 88.184].

Hacemos uso de **`np.interp`**, mapea el output del perceptron al rango del dataset, en caso de ser necesario.

Funciones de Activación

Tanh:

$$\theta(h) = tanh(\beta h)$$

Sigmoide:

$$\theta(h) = \frac{1}{1+exp^{-2\beta h}}$$

Lineal:

$$\theta(h) = h$$

Cálculo de Error

Se calcula el error de cada época de la siguiente manera:

Error (n° de epoca) = (Valor computado - Valor esperado) ^ 2 / Valores Totales

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE de Betas (tanh)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

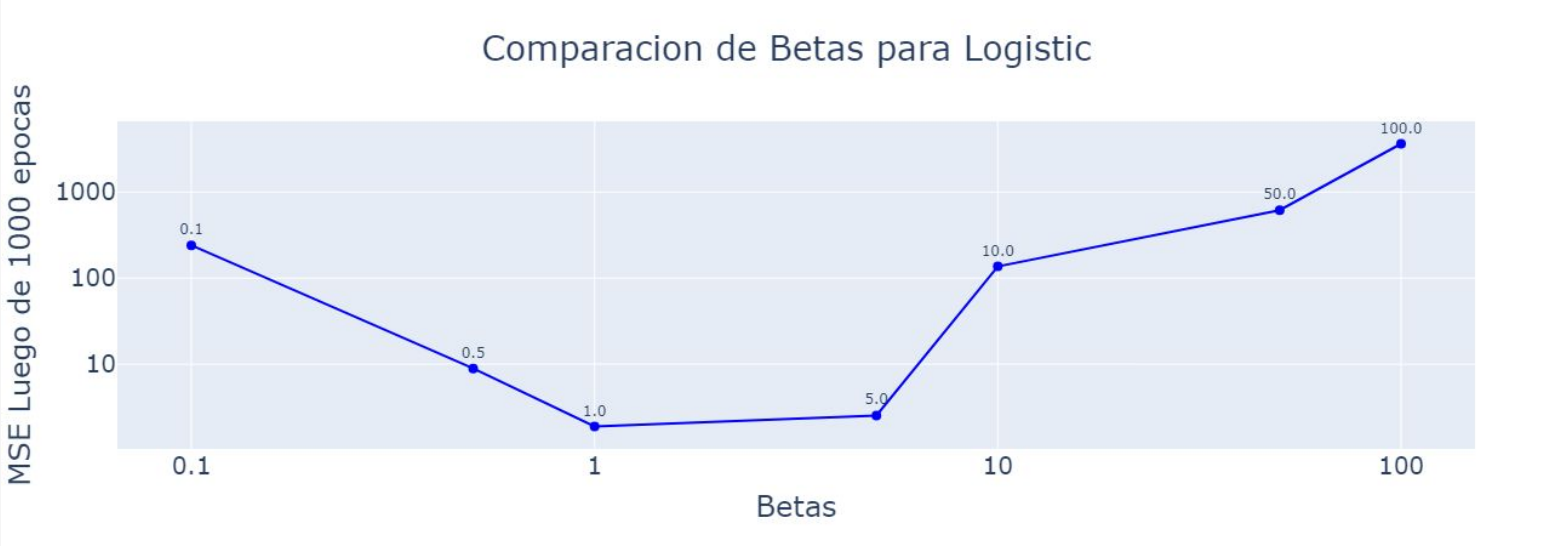
Learning rate: 0.0001

$$\theta(h) = tanh(\beta h)$$

Usaremos beta = 1

MSE de Betas (logística)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$



Learning rate: 0.0001

$$\theta(h) = \frac{1}{1+exp^{-2\beta h}}$$

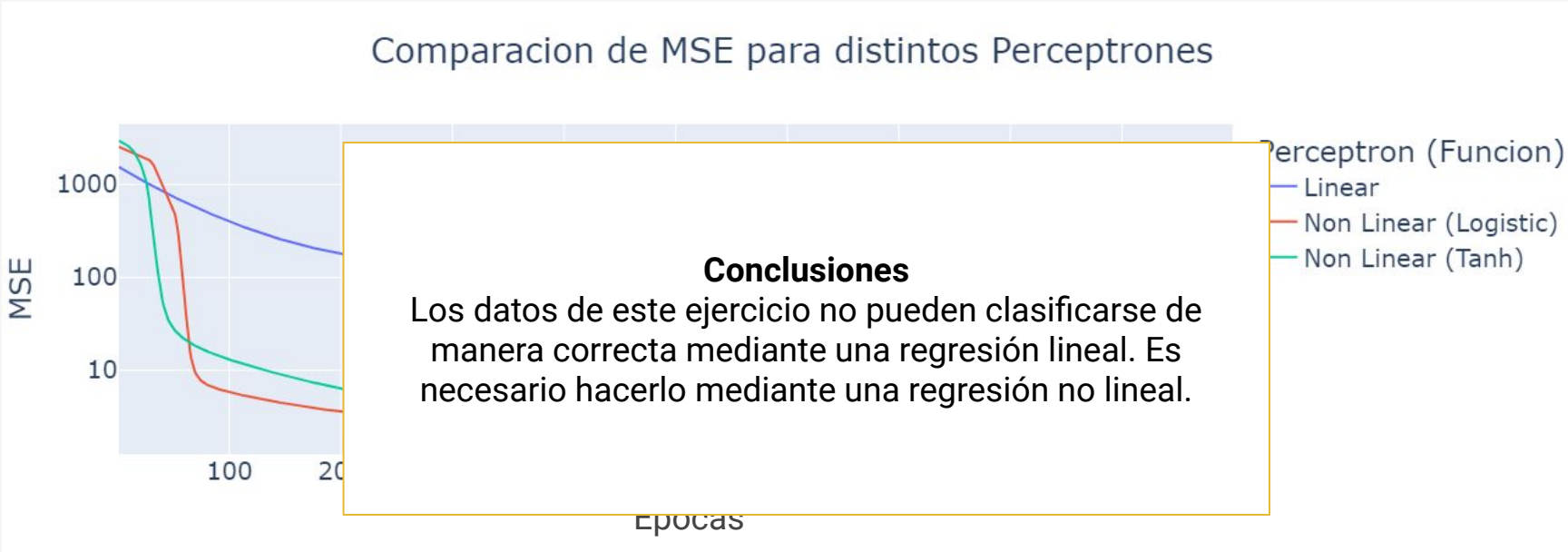
Usaremos beta = 1

MSE para distintos perceptrons

Learning rate= 0.0001
Betas = 1

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE para distintos perceptrons



Conclusiones

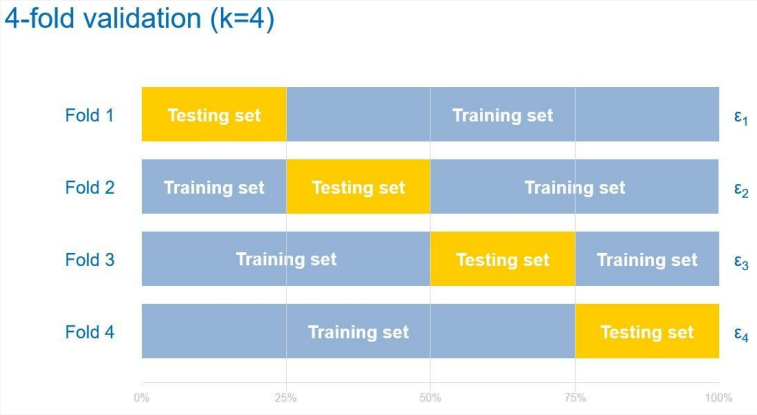
Los datos de este ejercicio no pueden clasificarse de manera correcta mediante una regresión lineal. Es necesario hacerlo mediante una regresión no lineal.

Learning rate= 0.0001
Betas = 1

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Mejor Conjunto de Entrenamiento

- Para este ejercicio, se solicita encontrar la “mejor” partición del dataset, en este caso, que brindan el menor error posible.
- Teniendo esto en cuenta, se optó por hacer uso del K-Cross-Validation.
- Se realizará el estudio con k = 2, k = 4 y k = 7.



Comparación K-Fold

K=2

Set	Training Error (MSE)	Testing Error (MSE)
1	1,4	8,69
2	4,72	3,85

K=4

Set	Training Error (MSE)	Testing Error (MSE)
1	2,79	1,18
2	0,59	6,26
3	2,25	1,14
4	2,56	1,25

Estos valores mínimos fueron obtenidos luego de correrlos 1000 épocas.
Learning rate= 0.0001
Betas = 1
Funcion = Logistic

Comparación K-Fold

K = 7

Set	Training Error (MSE)	Testing Error (MSE)
1	2,29	2,16
2	2,18	0,17
3	0,6	11
4	1,85	2,03
5	2,086	0,7
6	2,12	0,9
7	2,28	0,32

Estos valores mínimos fueron obtenidos luego de correrlos 1000 épocas.

Comparación K-Fold

K = 7

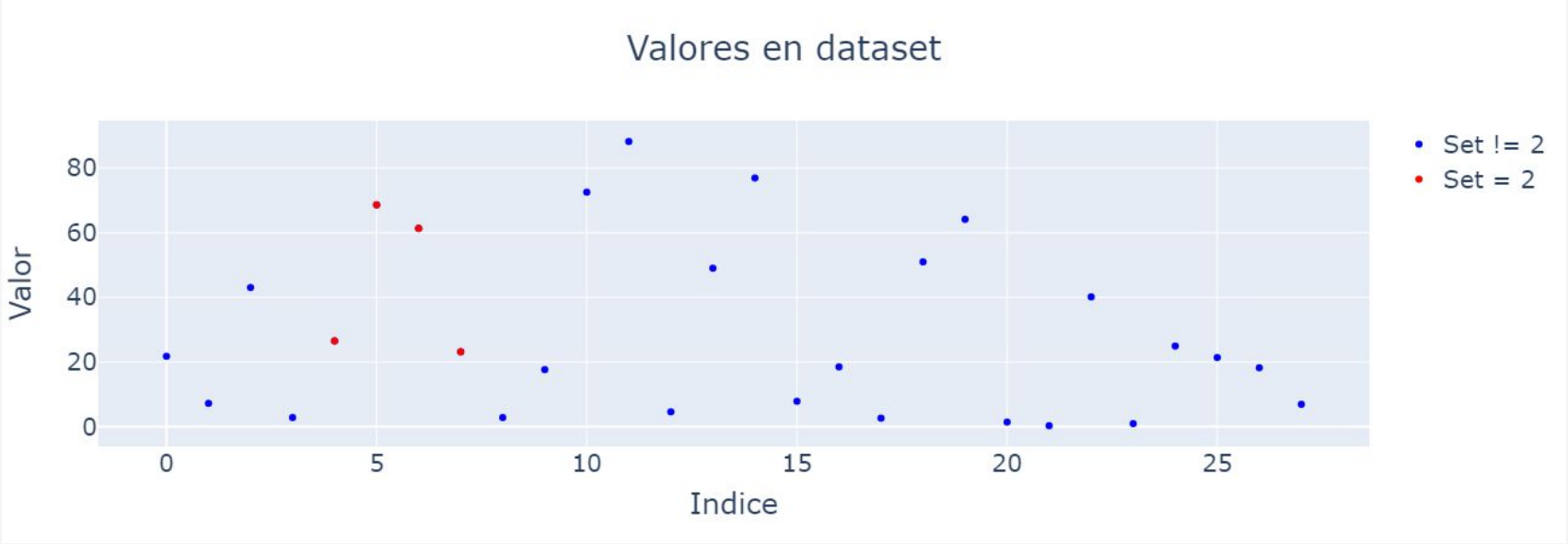
Set	Training Error (MSE)	Testing Error (MSE)
1	2,29	2,16
2	2,18	0,17
3	0,6	11
4	1,85	2,03
5	2,086	0,7
6	2,12	0,9
7	2,28	0,32

- Set n°2: Error minimo de Testing
- Set n°4: Error bajo de testing y de training.

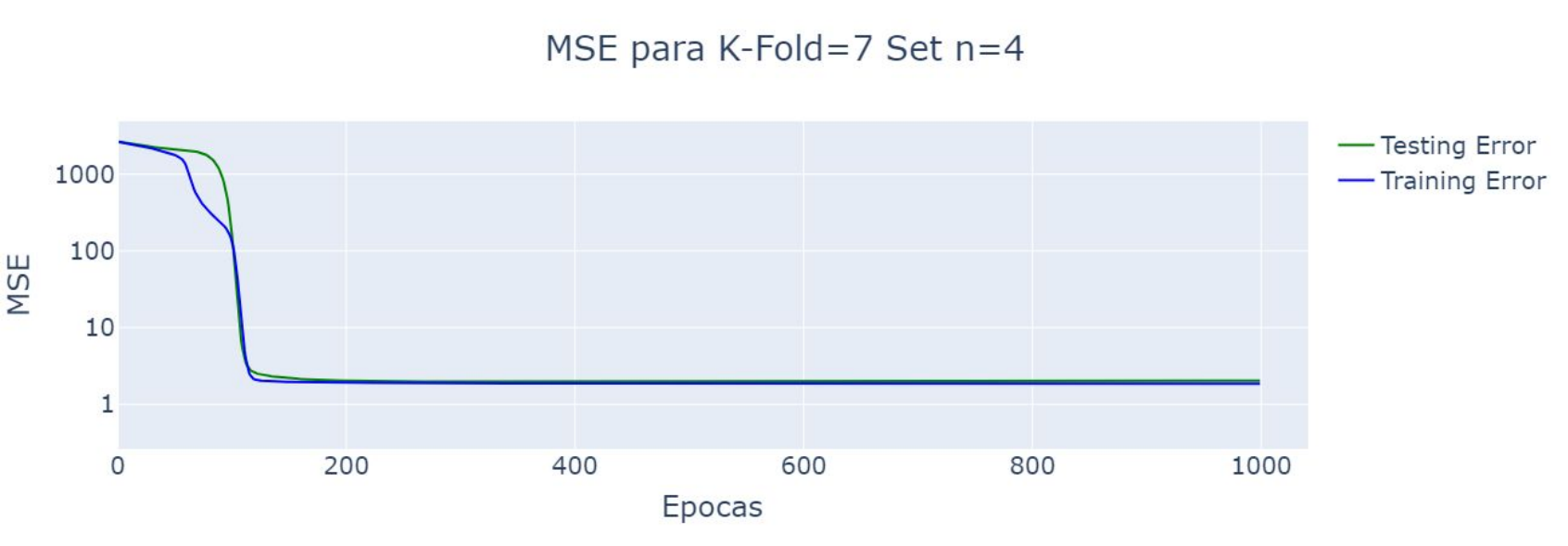
MSE – K=7 – Set = 2

Learning Rate: 0.0001
Epocas: 1000
Error: 0.005
Beta:1
Funcion: Logistic

Distribución – K=7 – Set = 2

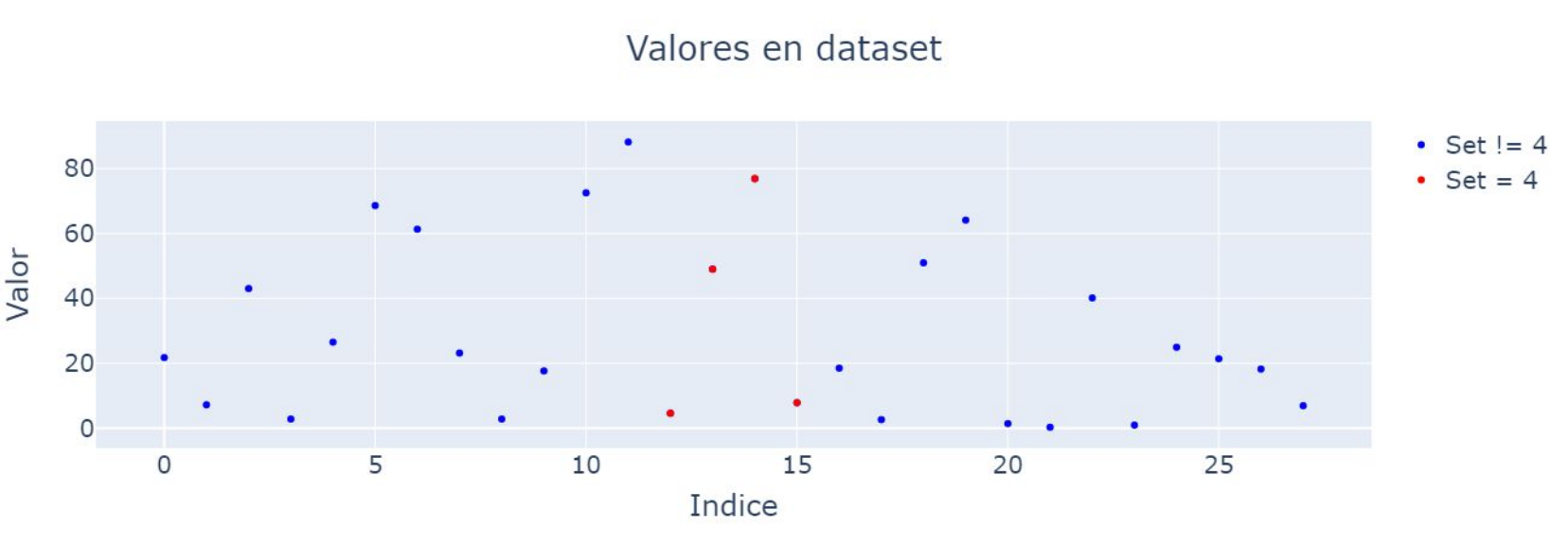


MSE – K=7 – Set = 4



Learning Rate: 0.0001
Epocas: 1000
Error: 0.005
Beta:1
Funcion: Logistic

Distribución – K=7 – Set = 4



Conclusiones

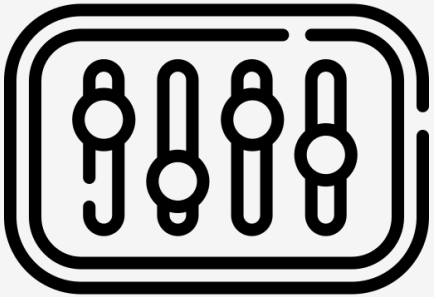
- Podemos concluir que la forma en que elegimos el dataset de training y testing tiene una gran influencia en los resultados obtenidos.
- Se puede afirmar que el perceptrón no lineal tiene capacidad de generalización.
- Set 2 y 4 con k-fold 7 son buenos conjuntos de testeo (usando el resto como training.)

EJ 3

Elección mejor conjunto de entrenamiento

Hiperparámetros relevantes

- Arquitectura
- Epoca Maxima
- Error de Corte
- Learning Rate
- Funcion de Activacion
  - Beta



Métricas

Evaluaremos los resultados de las predicciones de los modelos entrenados utilizando las métricas Accuracy, Precision, Recall y F1-Score a partir de una matriz de confusión

En las filas tenemos los valores esperados como respuesta de la predicción y en las columnas tenemos las predicciones del modelo

		Prediction	
		Positive	**Negative**
Actual	**Positive**	TP	FN
	Negative	FP	TN

Métricas

Accuracy mide el porcentaje de predicciones correctas entre todas las realizadas

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precisión indica la proporción de verdaderos positivos (TP) entre todas las instancias que el modelo predijo como positivas

$$Precision = \frac{TP}{TP + FP}$$

Métricas

Recall mide la capacidad del modelo para detectar los verdaderos positivos entre todas las instancias que realmente son positivas

$$Recall = \frac{TP}{TP + FN}$$

F1-Score es la media armónica entre precision y recall. Se necesita una buena precision y recall para tener un F1-Score alto. Es el equilibrio entre las dos métricas.

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Métodos de optimización

En el análisis de este trabajo compararemos el comportamiento de los métodos de optimización Gradient Descent, Momentum y Adam

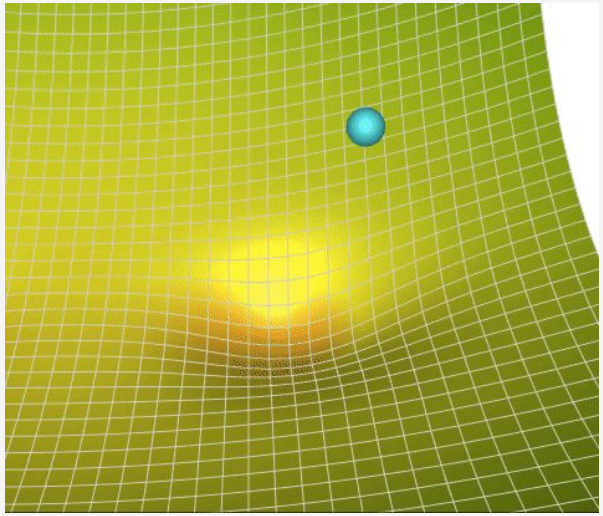
Estos son los métodos por los cuales se ajustan los pesos del perceptrón multicapa con el objetivo de reducir la función de “loss” o pérdida que en nuestro caso es el error cuadrático medio

Métodos de optimización

Gradient descent

- Calcula el gradiente de la función de error con respecto a los pesos
- Se ajustan los pesos en la dirección negativa del gradiente para minimizar dicho error

$$w \leftarrow w - \eta \cdot \nabla L(w)$$



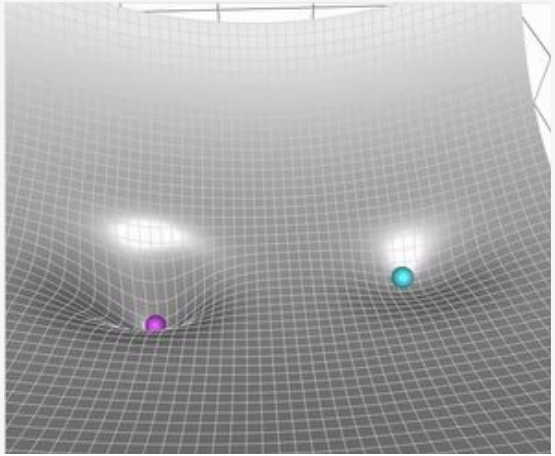
Métodos de optimización

Momentum

Para ajustar que el entrenamiento sea más rápido y evitar quedar atrapado en mínimos locales, momentum tiene en cuenta el gradiente de los pasos anteriores a la hora de actualizar los pesos

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot \nabla L(w)$$

$$w \leftarrow w - \eta \cdot v_t$$



Momentum (magenta) vs. Gradient Descent (cyan)

- Vt es la velocidad acumulada en el tiempo t.
- β es el parámetro de momentum (0 < β < 1), que controla cuánto contribuye el gradiente pasado en el paso actual.

  Siempre usamos β=0.9

Métodos de optimización

Adam(Adaptive Moment Estimation)

Adam es más complejo ya que utiliza la media y la varianza del gradiente para hacer ajustes más precisos en cada iteración.

- Media móvil de los gradientes: Simula el comportamiento del momentum

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla L(w)$$

Usaremos: β_1^t = 0.9

- Media móvil del cuadrado de los gradientes: Asegura que las actualizaciones sean proporcionales al tamaño del gradiente

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla L(w))^2$$

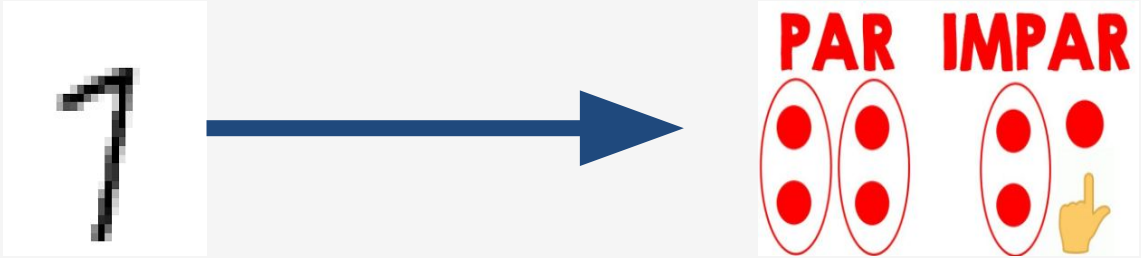
Usaremos: β_2 = 0.999

- Finalmente se ajustan los valores para corregir sesgos iniciales y se actualizan los pesos

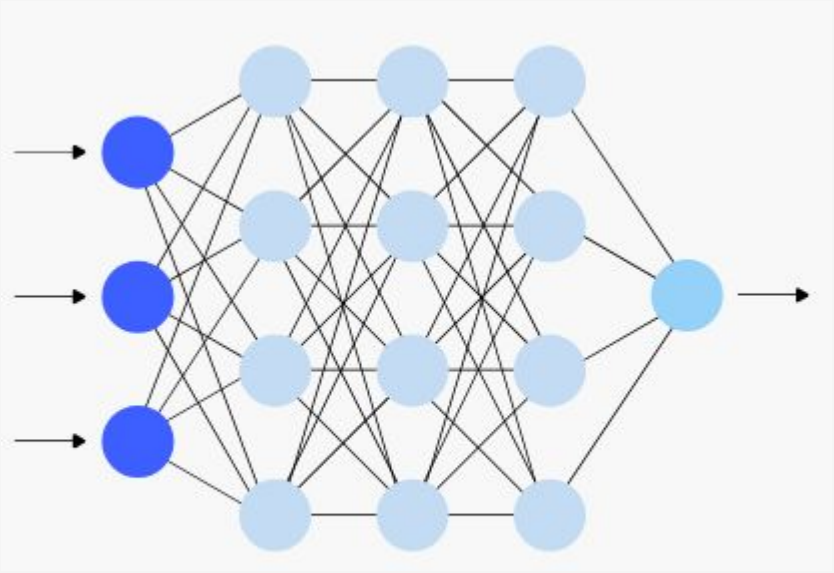
$$w \leftarrow w - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \qquad \hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Nota: ϵ es un pequeño valor para evitar divisiones por cero

EJ 3-B



Diseño



La arquitectura está implementada usando Batch.

Input: Array de 35 digitos (0,1).
Output: Valor entre (0,1).

Para clasificar al output se le aplica la siguiente función:

$$f(x) = \begin{cases} 1 & \text{si } x > 0.5 \\ 0 & \text{si } x \leq 0.5 \end{cases}$$

El 1 lo consideramos como par, el 0 como impar.

Testeamos:

- Arquitectura
- Learning Rate
- Funcion de Activacion
  - Beta
- Optimizador

→ Encontrar hiperparámetros con un buen desempeño.

Metricas

Qué métrica aporta más valor en este caso? Analizamos la matriz de confusión

		Prediction	
	X	1	0
Real Value	1	0	1
	0	1	1

Siendo 1 los PARES y 0 los IMPARES

Metricas

Matriz de confusión multiclase para modelo sin entrenamiento

Prediction

Real Value	X	1	0
	1	0	1
	0	1	1

Siendo 1 los PARES y 0 los IMPARES

TP=0 FN=1 FP=1 TN=1

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{0 + 1}{0 + 1 + 1 + 1} = 0.33$$

$$Precision = \frac{TP}{TP + FP} = \frac{0}{0 + 1} = 0$$

Para el caso donde el divisor es 0 definimos la métrica en 0

$$Recall = \frac{TP}{TP + FN} = \frac{0}{0 + 1} = 0$$

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 0$$

Metricas

Matriz de confusión multiclase para modelo sin entrenamiento

Prediction

Real Value	X	1	0
	1	1	0
	0	0	1

Siendo 1 los PARES y 0 los IMPARES

Vamos a usar **Accuracy** porque es el único que tiene en cuenta los true negative, es decir los que dan impar y predice que es impar.

TP=1 FN=0 FP=0 TN=1

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1 + 1}{1 + 1 + 0 + 0} = 1$$

$$Precision = \frac{TP}{TP + FP} = \frac{1}{1 + 0} = 1$$

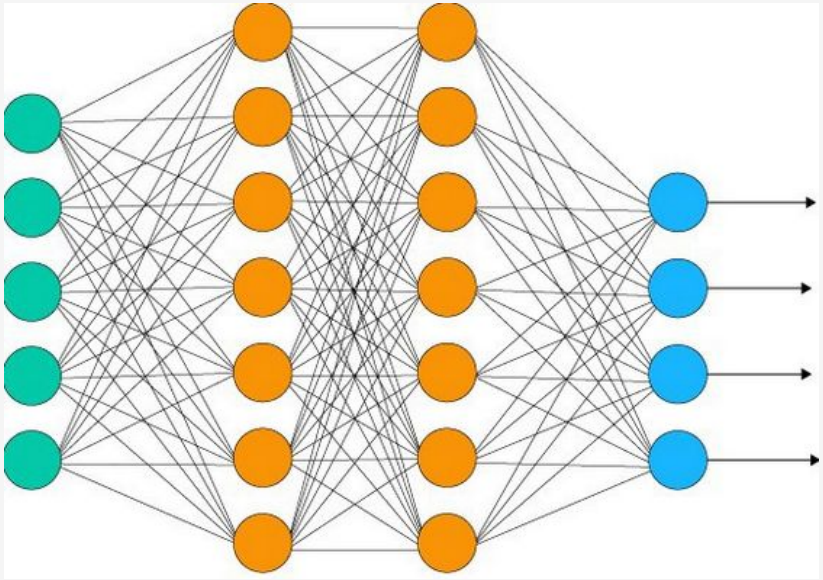
Para el caso donde el divisor es 0 definimos la métrica en 0

$$Recall = \frac{TP}{TP + FN} = \frac{1}{1 + 0} = 1$$

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 1$$

Implementacion

Utilizando la métrica de accuracy previamente definida haremos una exploración hasta encontrar una configuración cercana a la óptima.



Hiperparametros:

- Cantidad de neuronas por capa
- Cantidad de capas
- Funcion de activacion: Logistic
- Learning rate = 0.01
- Betas de funciones de activación = 1
- Optimizer= Momentum

Parametros fijos:

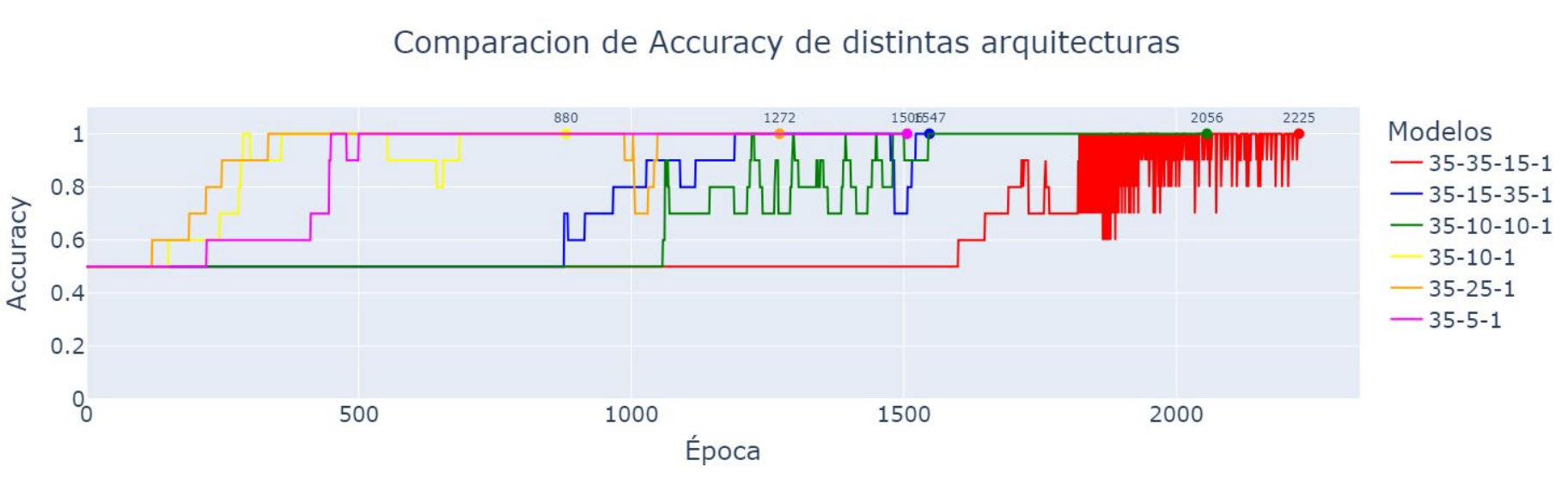
- Cantidad de épocas = 2500
- Error de corte = 0.001

Arquitecturas

Arquitecturas candidatas:

- Arq1 = [35, 35, 15, 1] “arquitectura con muchas capas y descendiente”
- Arq2 = [35, 15, 35, 1] “arquitectura con muchas capas y ascendiente”
- Arq3 = [35, 10, 10, 1] “arquitectura con muchas capas y con pocas neuronas por capa”
- Arq4 = [35, 10, 1] “arquitectura con pocas capas y pocas neuronas por capa”
- Arq5 = [35, 25, 1] “arquitectura con pocas capas y muchas neuronas por capa”
- Arq6 = [35, 5, 1] “arquitectura con pocas capas y muy pocas neuronas por capa”

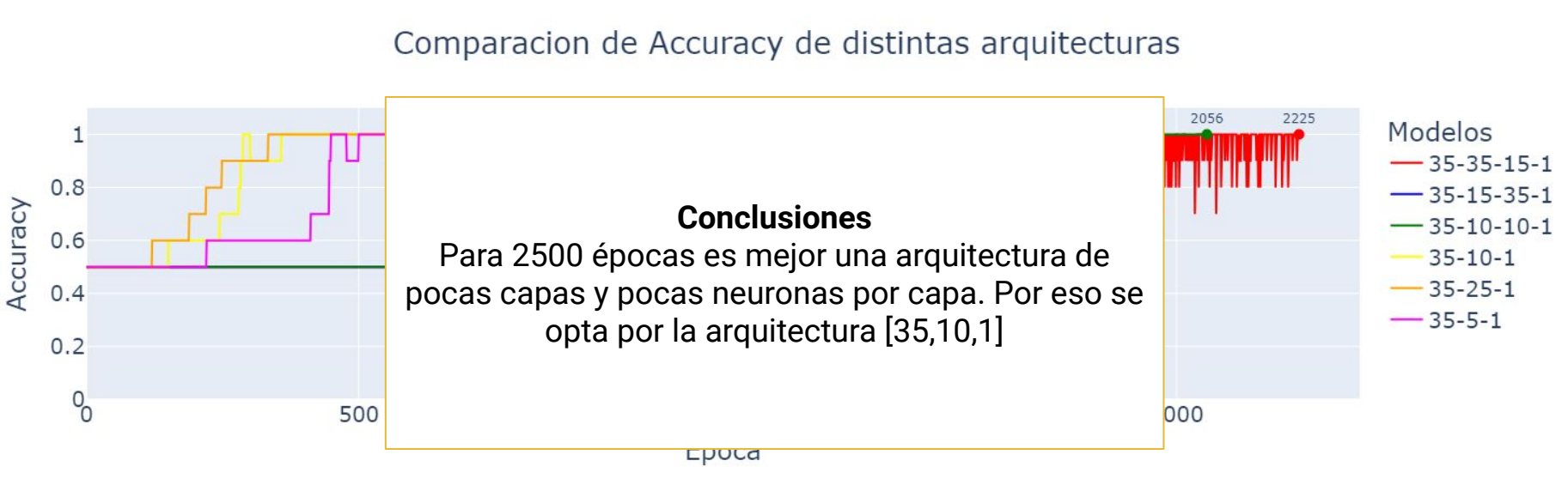
Arquitecturas



Architecture = ?
Learning Rate (η) = 0.01
A. Function = Logistic
Beta = 1
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Arquitecturas



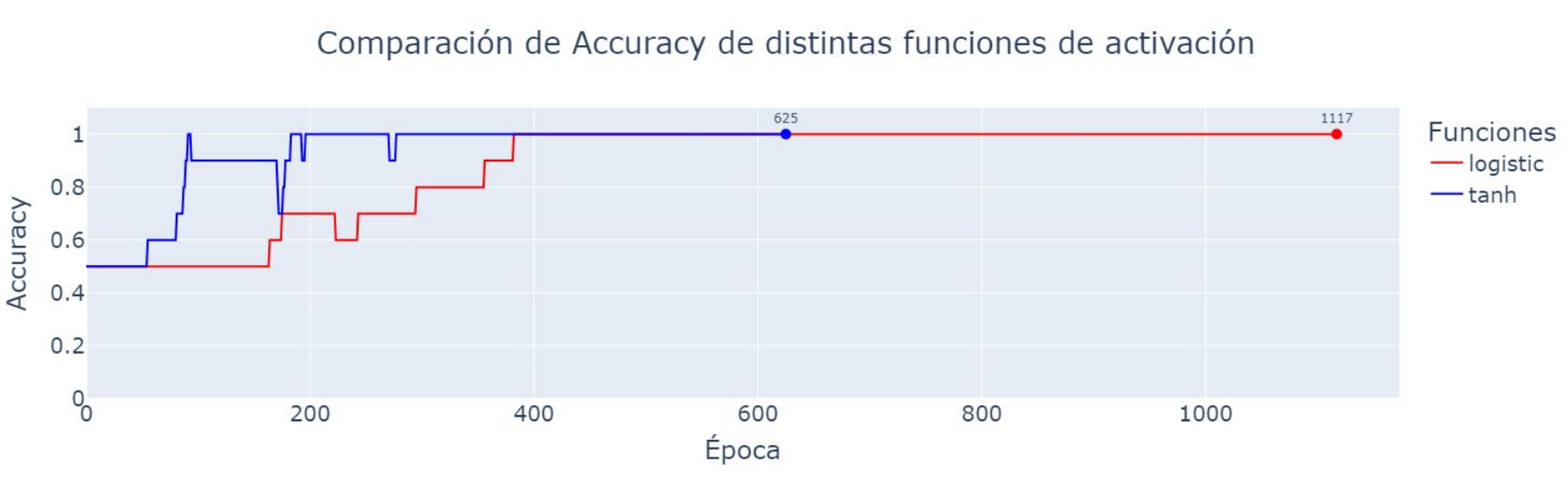
Conclusiones

Para 2500 épocas es mejor una arquitectura de pocas capas y pocas neuronas por capa. Por eso se opta por la arquitectura [35,10,1]

Architecture = ?
Learning Rate (η) = 0.01
A. Function = Logistic
Beta = 1
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

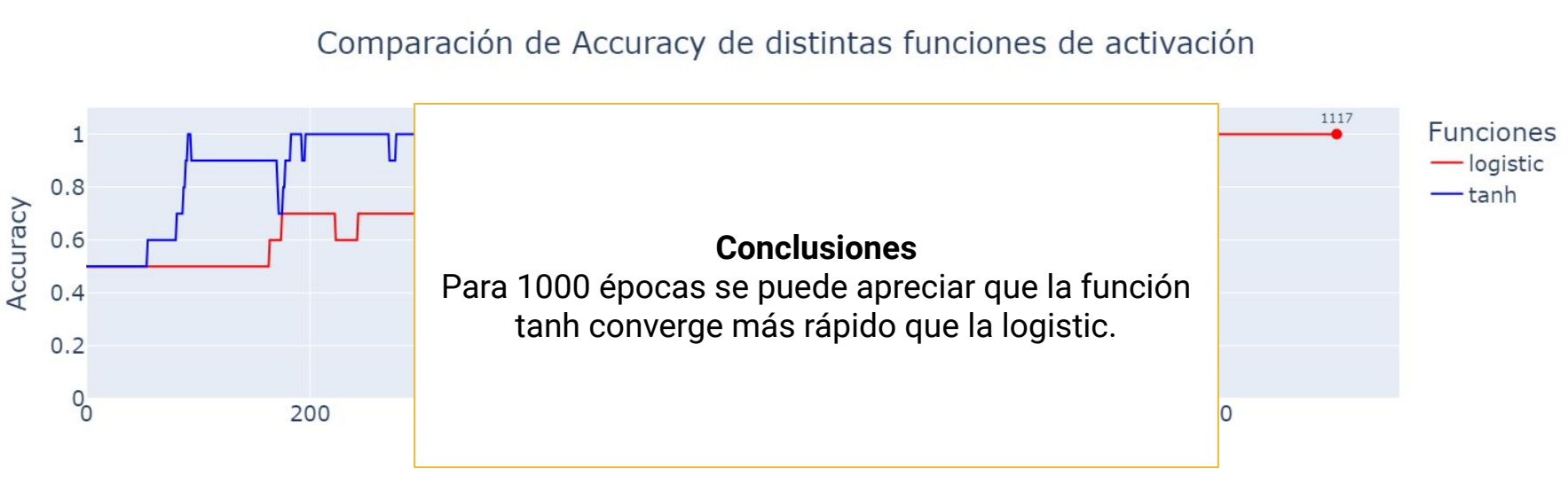
Funciones de Act.



Architecture = [35,10,1]
Learning Rate (η) = 0.01
A. Function = ?
Beta = 1
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Funciones de Act.



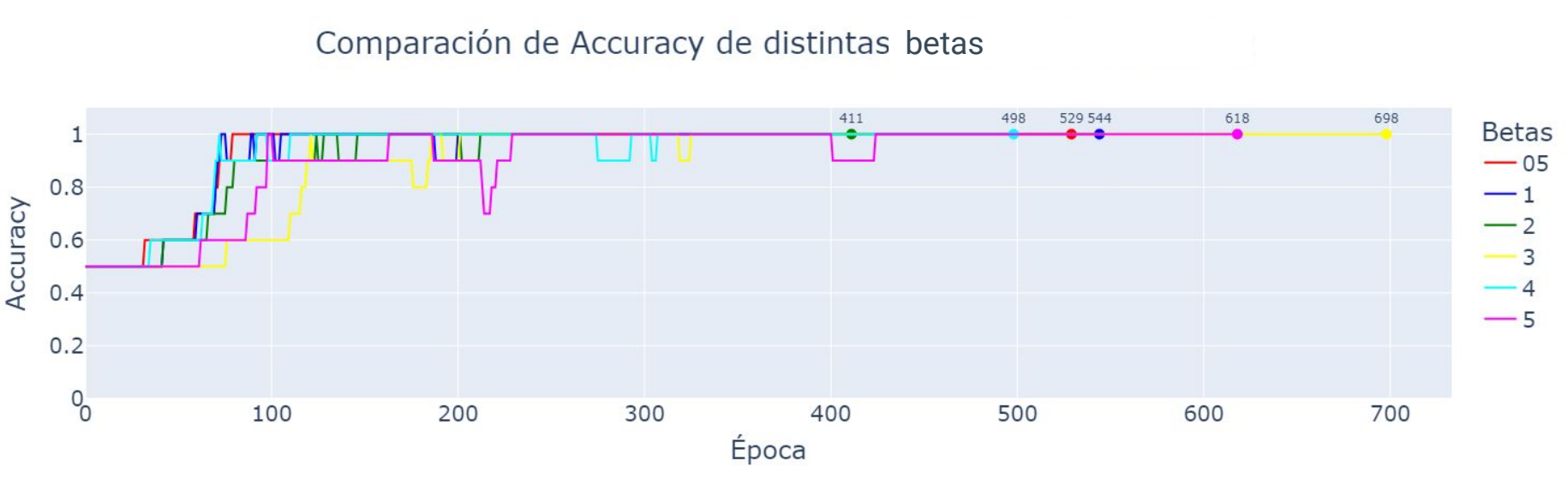
Conclusiones

Para 1000 épocas se puede apreciar que la función tanh converge más rápido que la logistic.

Architecture = [35,10,1]
Learning Rate (η) = 0.01
A. Function = ?
Beta = 1
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

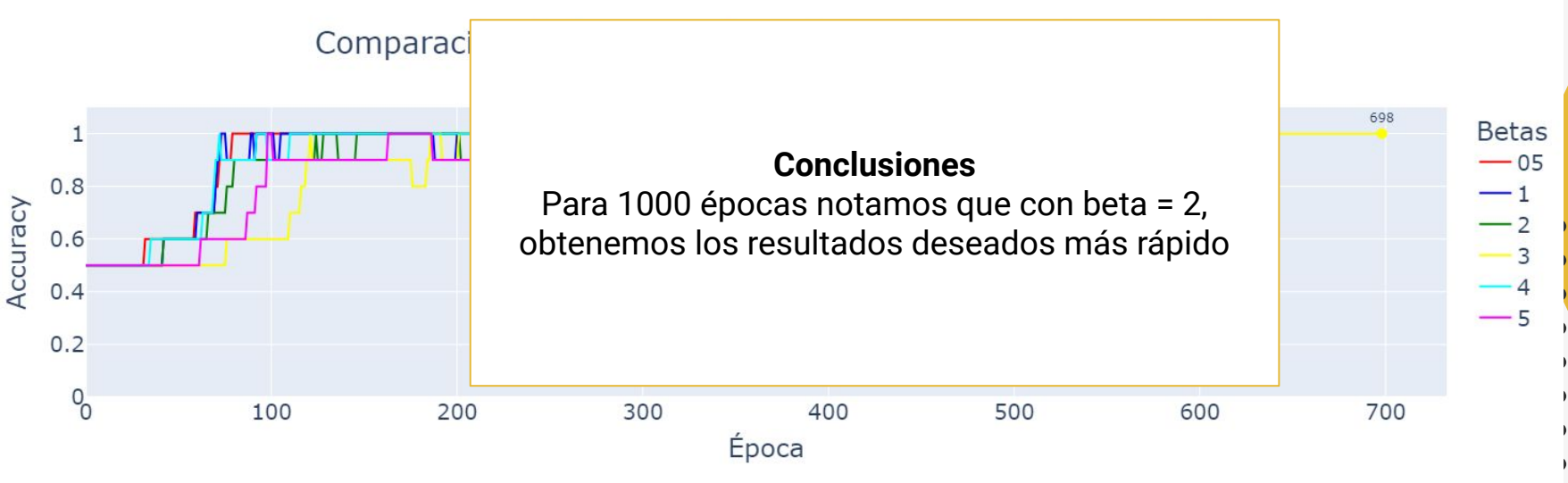
Betas



Architecture = [35,10,1]
Learning Rate (η) = 0.01
A. Function = tanh
Beta = ?
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Betas



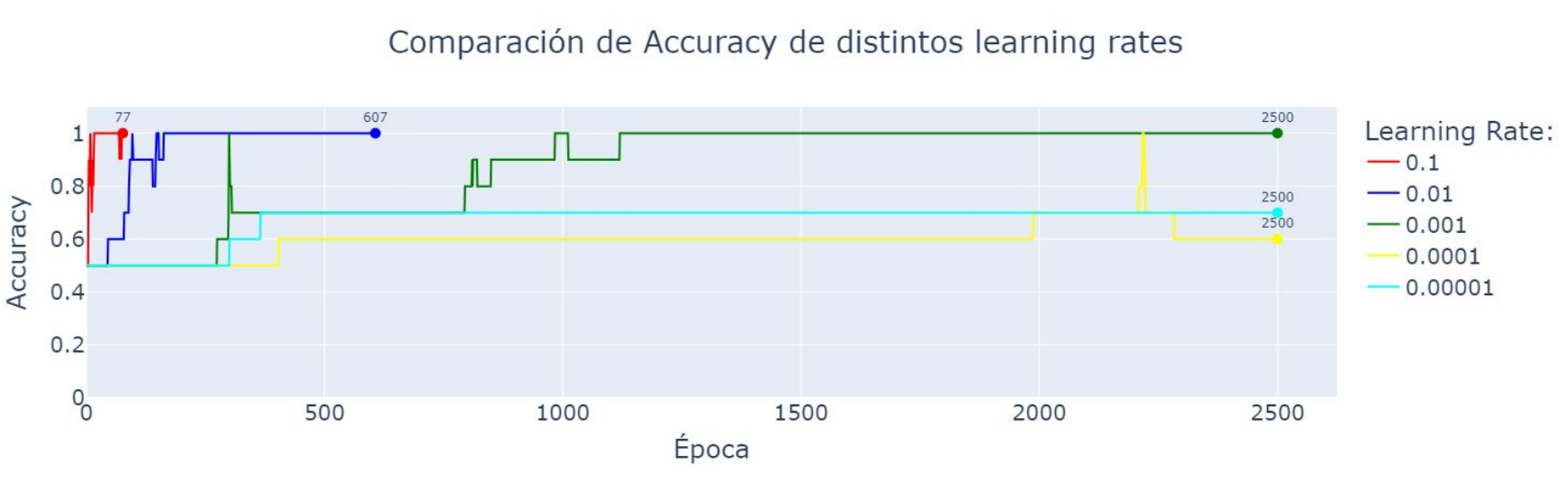
Conclusiones

Para 1000 épocas notamos que con beta = 2, obtenemos los resultados deseados más rápido

Architecture = [35,10,1]
Learning Rate (η) = 0.01
A. Function = tanh
Beta = ?
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

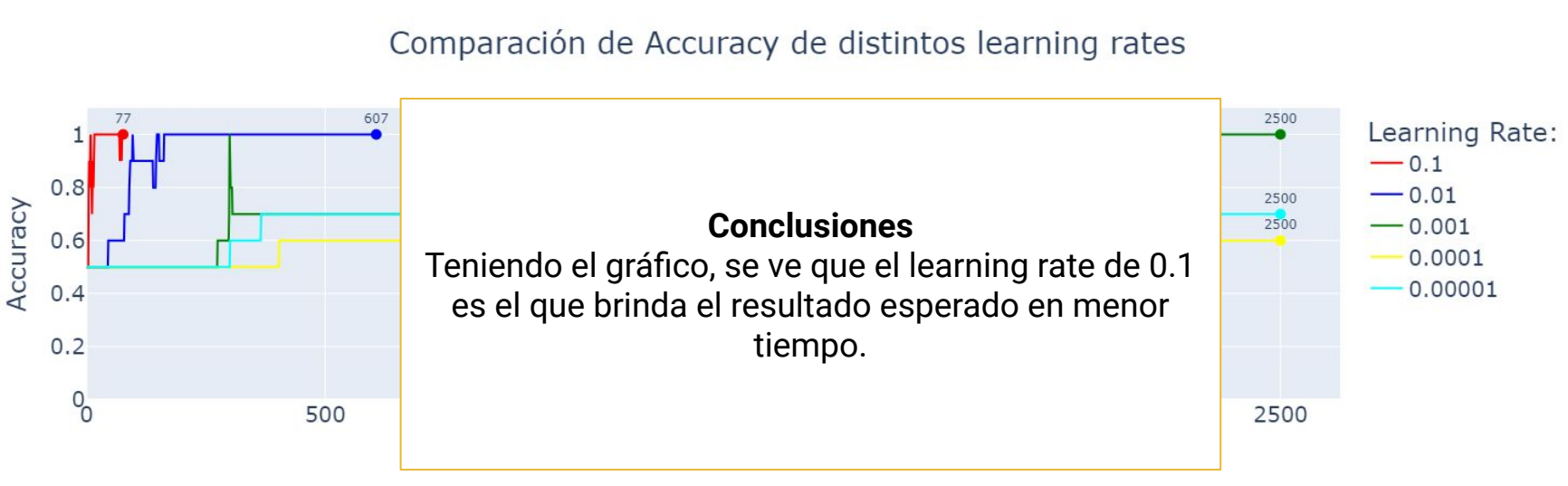
Learning Rate



Architecture = [35,10,1]
Learning Rate (η) = ?
A. Function = tanh
Beta = 2
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Learning Rate



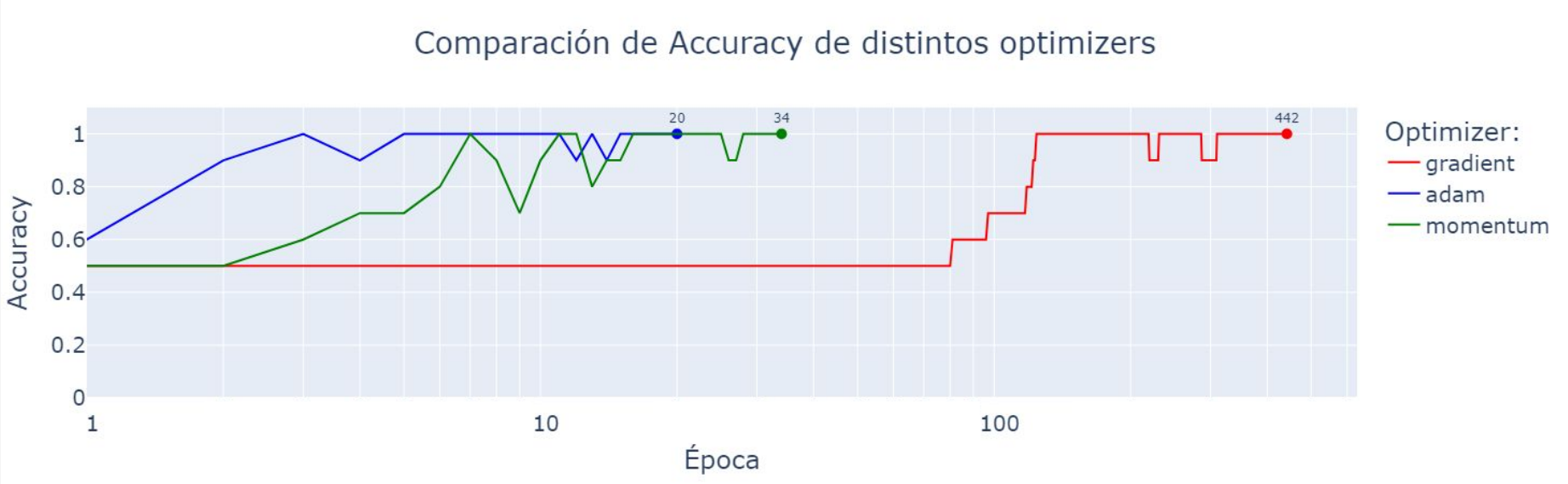
Conclusiones

Teniendo el gráfico, se ve que el learning rate de 0.1 es el que brinda el resultado esperado en menor tiempo.

Architecture = [35,10,1]
Learning Rate (η) = ?
A. Function = tanh
Beta = 2
Optimizer = Momentum

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

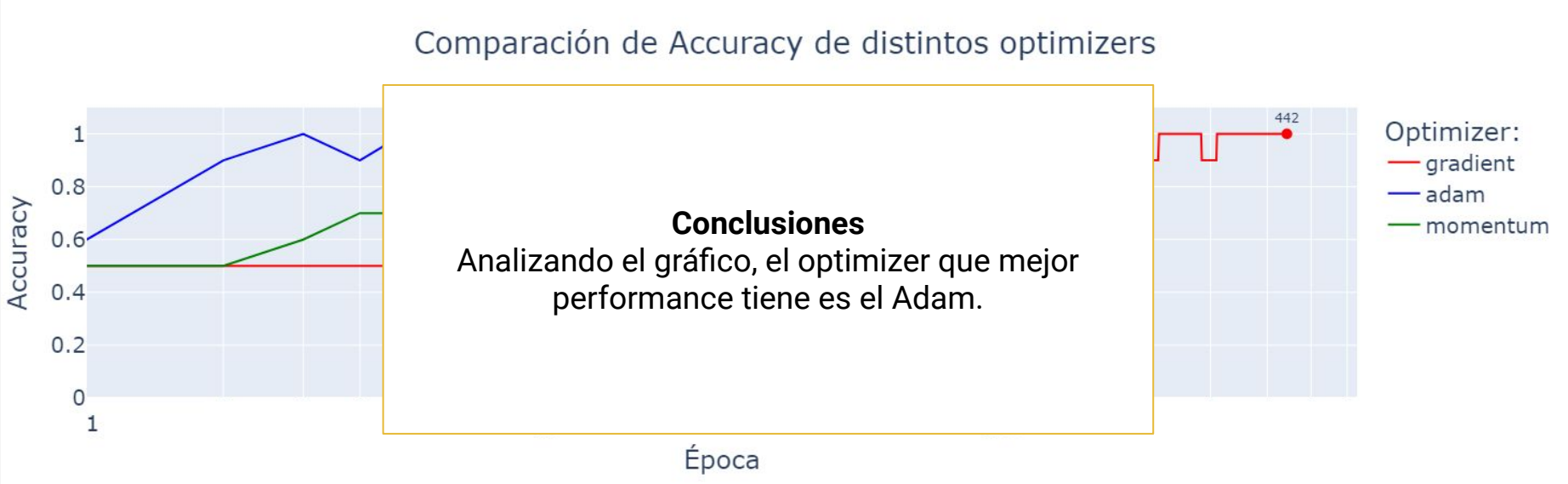
Optimizer



Architecture = [35,10,1]
Learning Rate (η) = 0.1
A. Function = tanh
Beta = 2
Optimizer = ?

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Optimizer



Conclusiones
Analizando el gráfico, el optimizer que mejor performance tiene es el Adam.

Architecture = [35,10,1]
Learning Rate (η) = 0.1
A. Function = tanh
Beta = 2
Optimizer = ?

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Generalization

Architecture = [35,10,1]
Learning Rate (η) = 0.1
A. Function = tanh
Beta = 2
Optimizer = Adam

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Conclusión

En este caso el perceptrón multicapa no tiene capacidad de generalización. Es decir, en base a la representación matricial de algunos dígitos utilizados para su entrenamiento no puede inferir la paridad de otros números.

EJ 3-C

1

Clasificación de predicciones

[0.02 0.01 0.03 0.76 0.89 0.01 0.00 0.24 0.32 0.07]

Qué número predijo?

Clasificación de predicciones

[0.02 0.01 0.03 0.76 **0.89** 0.01 0.00 0.24 0.32 0.07]

Qué número predijo?

Normalizamos tomando el mayor número del conjunto

[0 0 0 0 **1** 0 0 0 0 0]

Predijo el número 4

Arquitectura

Qué métrica aporta más valor en este caso? Analizamos matriz de confusión

Real value

0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0
5	0	0	0	1	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	0
7	0	0	0	1	0	0	0	0	0	0
8	0	0	0	1	0	0	0	0	0	0
9	0	0	0	1	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Prediction

Arquitectura

Matriz de confusión multiclase para modelo sin entrenamiento

0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0
5	0	0	0	1	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	0
7	0	0	0	1	0	0	0	0	0	0
8	0	0	0	1	0	0	0	0	0	0
9	0	0	0	1	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Tomamos la clase del 0

TP=0 FN=1 FP=0 TN=9

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{0 + 9}{0 + 9 + 0 + 1} = 0.9$$

$$Precision = \frac{TP}{TP + FP} = \frac{0}{0 + 0} = 0$$

Para el caso donde el divisor es 0 definimos la métrica en 0

$$Recall = \frac{TP}{TP + FN} = \frac{0}{0 + 1} = 0$$

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 0$$

Arquitectura

Matriz de confusión multiclase para modelo entrenado

0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	1
	0	1	2	3	4	5	6	7	8	9

Tomamos la clase del 0

TP=1 FN=0 FP=0 TN=9

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1 + 9}{1 + 9 + 0 + 0} = 1$$

$$Precision = \frac{TP}{TP + FP} = \frac{1}{1 + 0} = 1$$

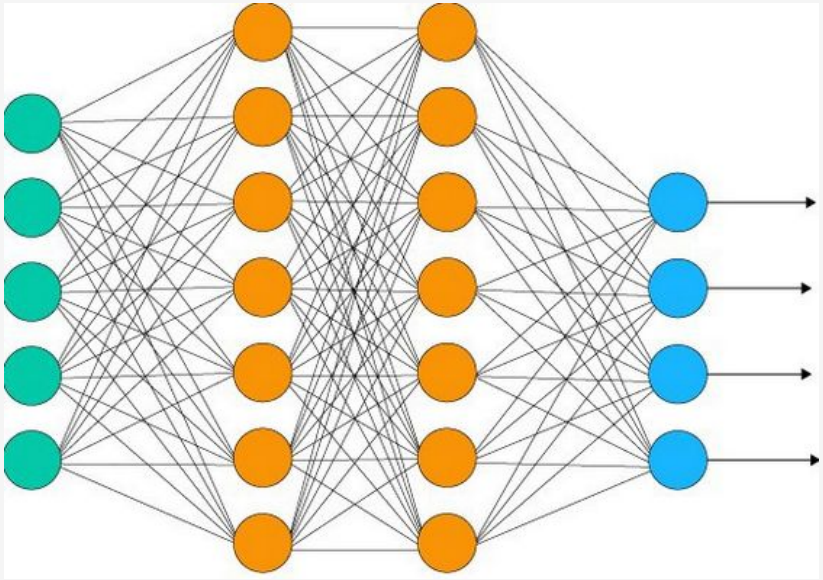
$$Recall = \frac{TP}{TP + FN} = \frac{1}{1 + 0} = 1$$

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 1$$

Dada la naturaleza de los datos, usaremos F1-Score porque es la métrica que nos aporta mayor información y a diferencia de Accuracy permite ver el progreso del modelo a través del entrenamiento más claramente.

Arquitectura

Utilizando la métrica de F1-Score previamente definida haremos una exploración intuitiva de la cantidad de cantidad de capas y de neuronas por capa. En este caso fijamos la función de activación y elegimos tanh.



Parametros variables:

- Cantidad de neuronas por capa
- Cantidad de capas
- Funcion de activacion: tanh
- Learning rate = 0.01
- Betas de funciones de activación = 1
- Optimizer= Gradient Descent

Parametros fijos:

- Cantidad de épocas = 1000
- Error de corte = 0.01

Arquitectura

Arquitecturas candidatas

- Arq1 = [35, 35, 15, 10] “arquitectura con muchas capas y descendiente”
- Arq2 = [35, 15, 35, 10] “arquitectura con muchas capas y ascendiente”
- Arq3 = [35, 10, 10, 10] “arquitectura con muchas capas y con pocas neuronas por capa”
- Arq4 = [35, 10, 10] “arquitectura con pocas capas y pocas neuronas por capa”
- Arq5 = [35, 25, 10] “arquitectura con pocas capas y muchas neuronas por capa”

Arquitectura

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Architecture = ?, Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 1, Optimizer = Gradient Descent

Arquitectura

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

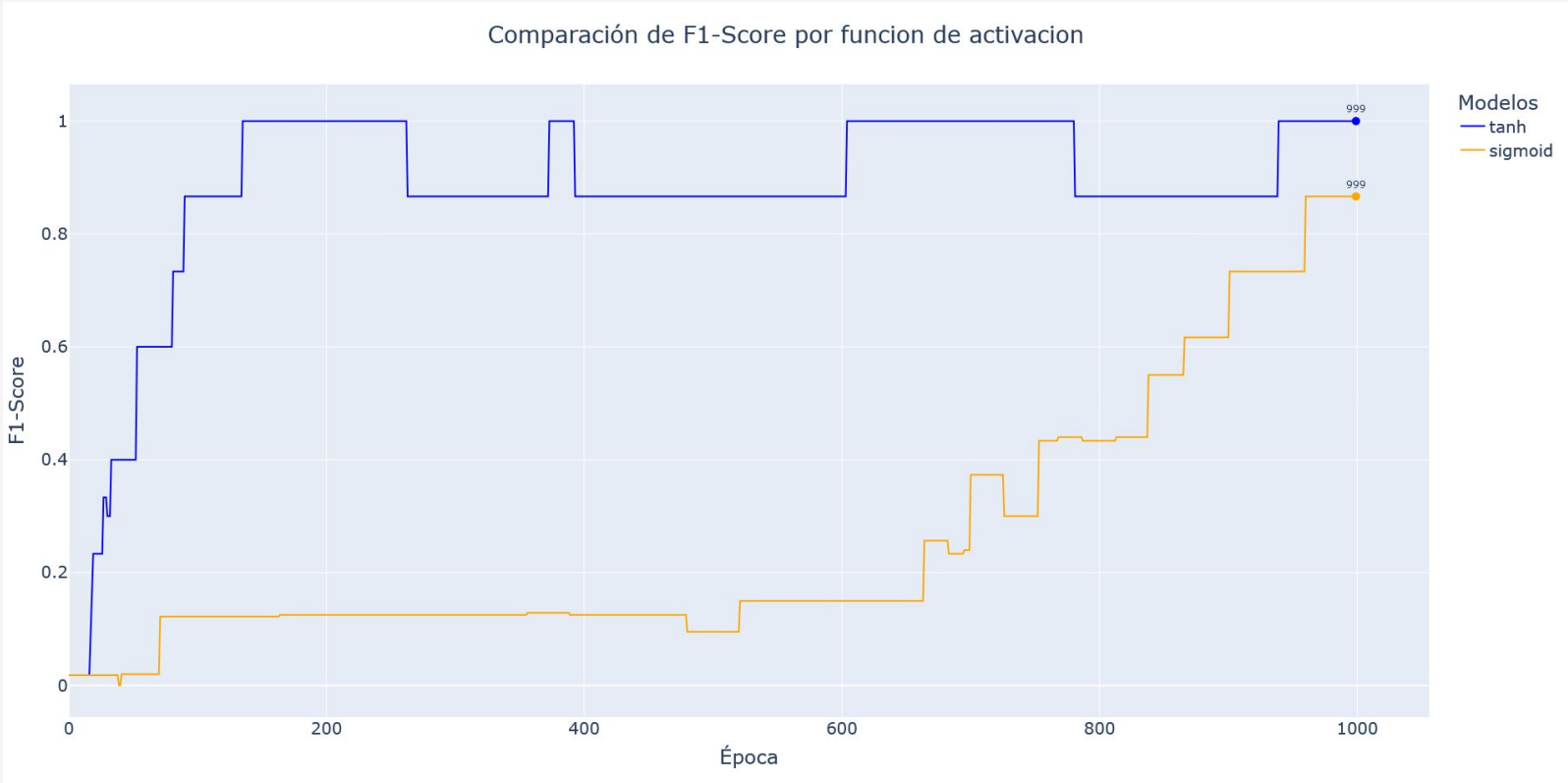
Conclusiones

Para 1000 épocas es mejor una arquitectura de pocas capas y muchas neuronas por capa

Architecture = ?, Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 1, Optimizer = Gradient Descent

Arquitectura

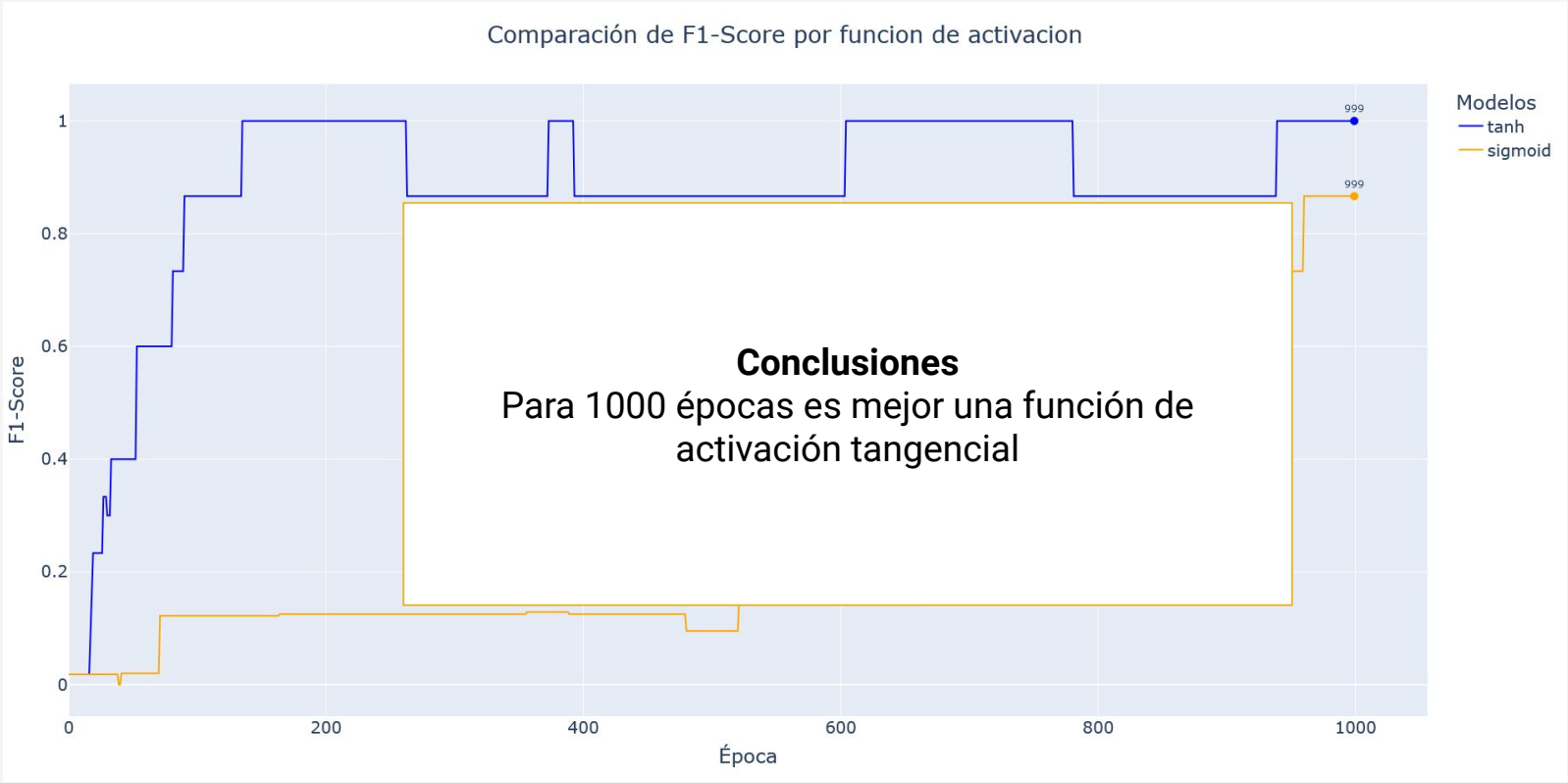
$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = ? ,Beta = 1, Optimizer = Gradient Descent

Arquitectura

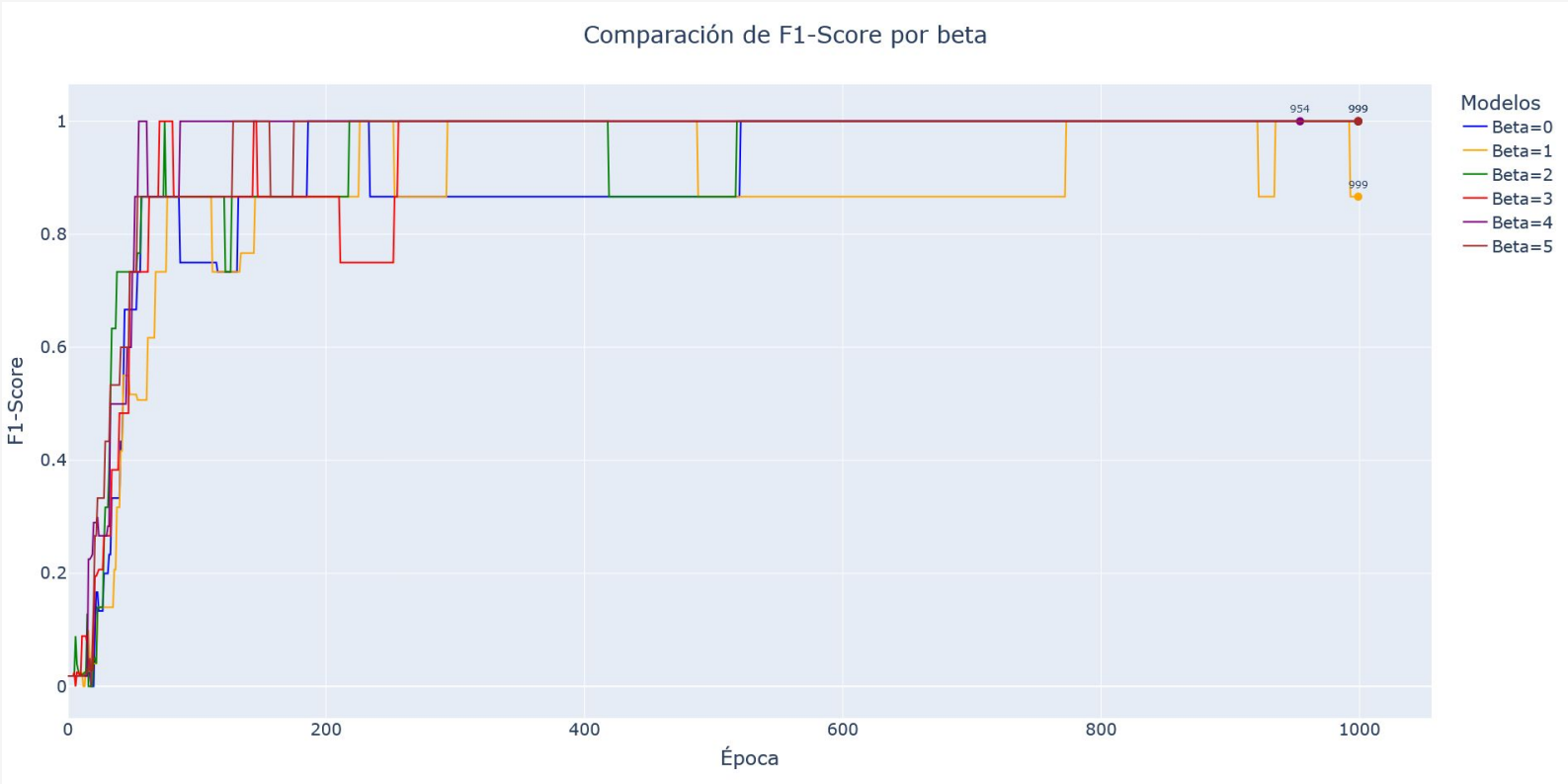
$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = ? ,Beta = 1, Optimizer = Gradient Descent

Arquitectura

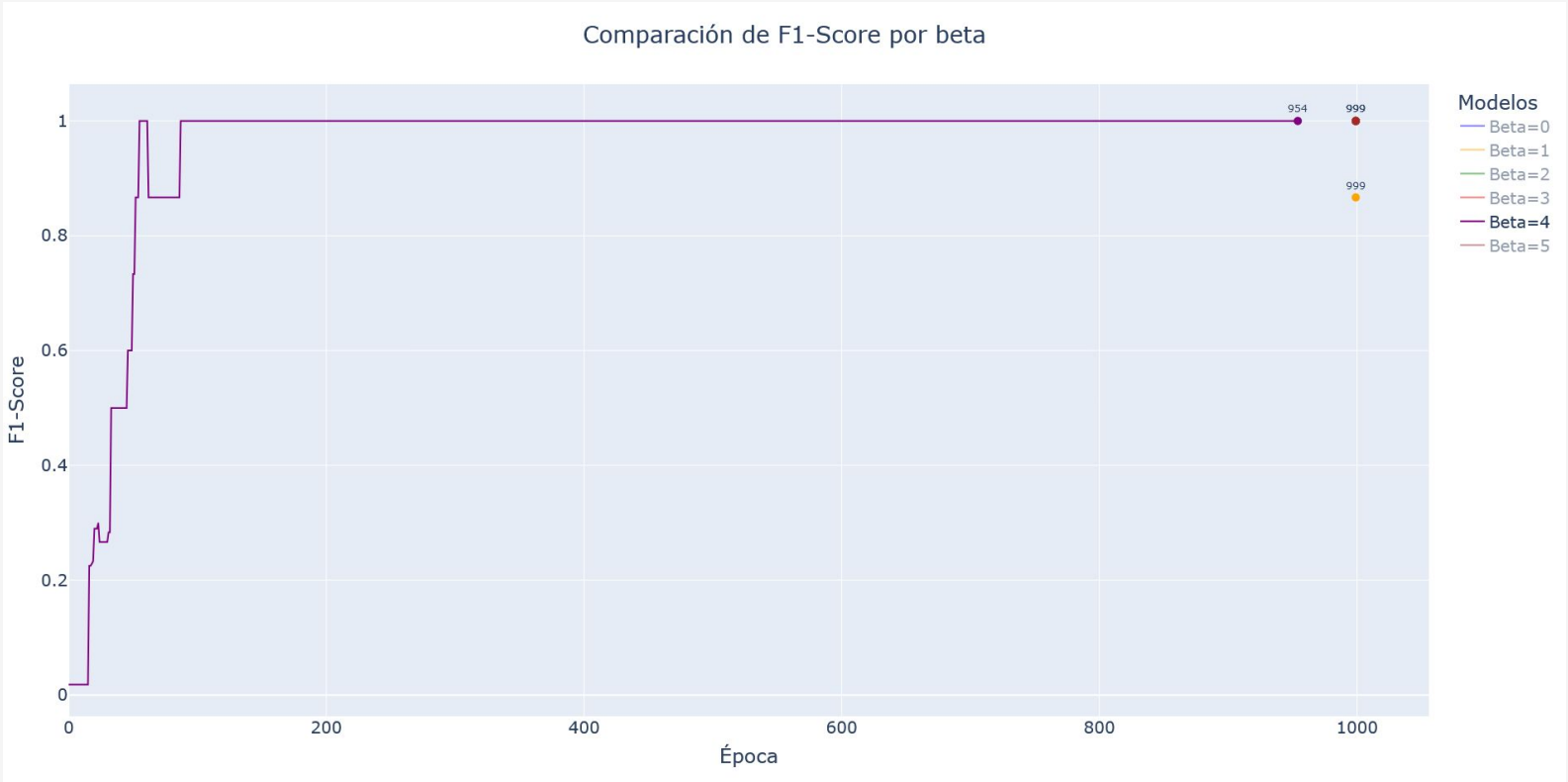
$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = ?, Optimizer = Gradient Descent

Arquitectura

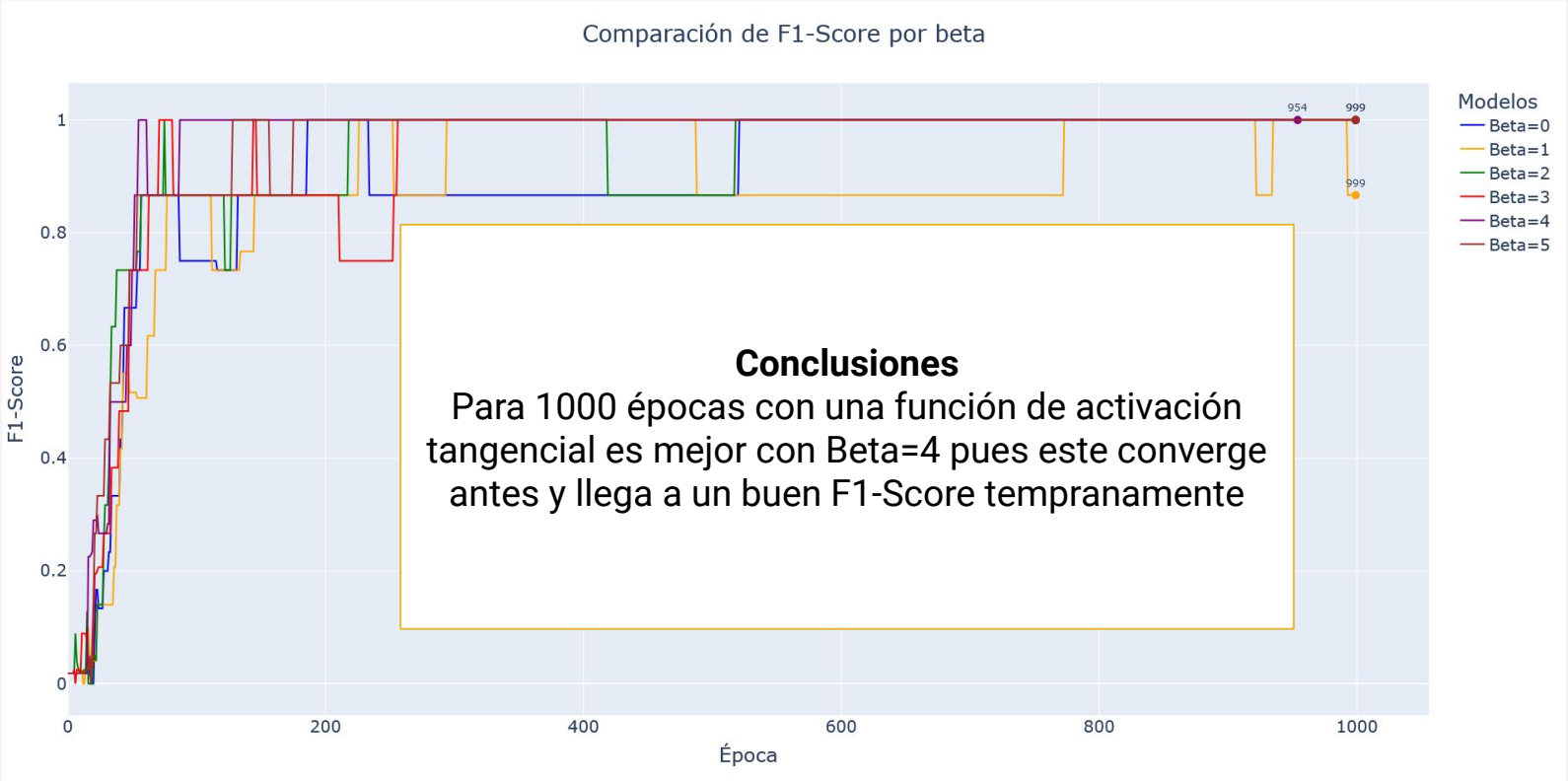
$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = ?, Optimizer = Gradient Descent

Arquitectura

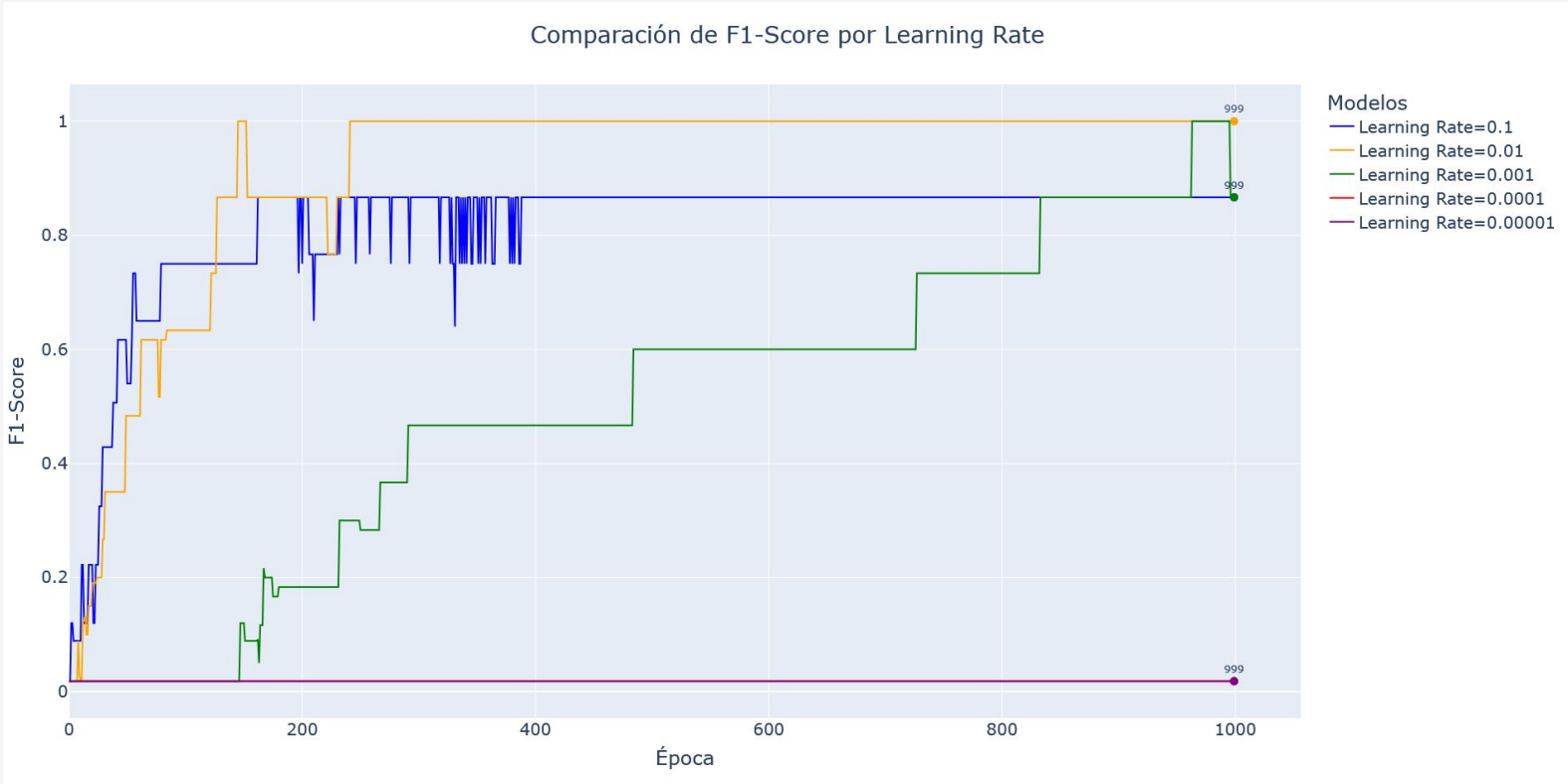
$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = ?, Optimizer = Gradient Descent

Arquitectura

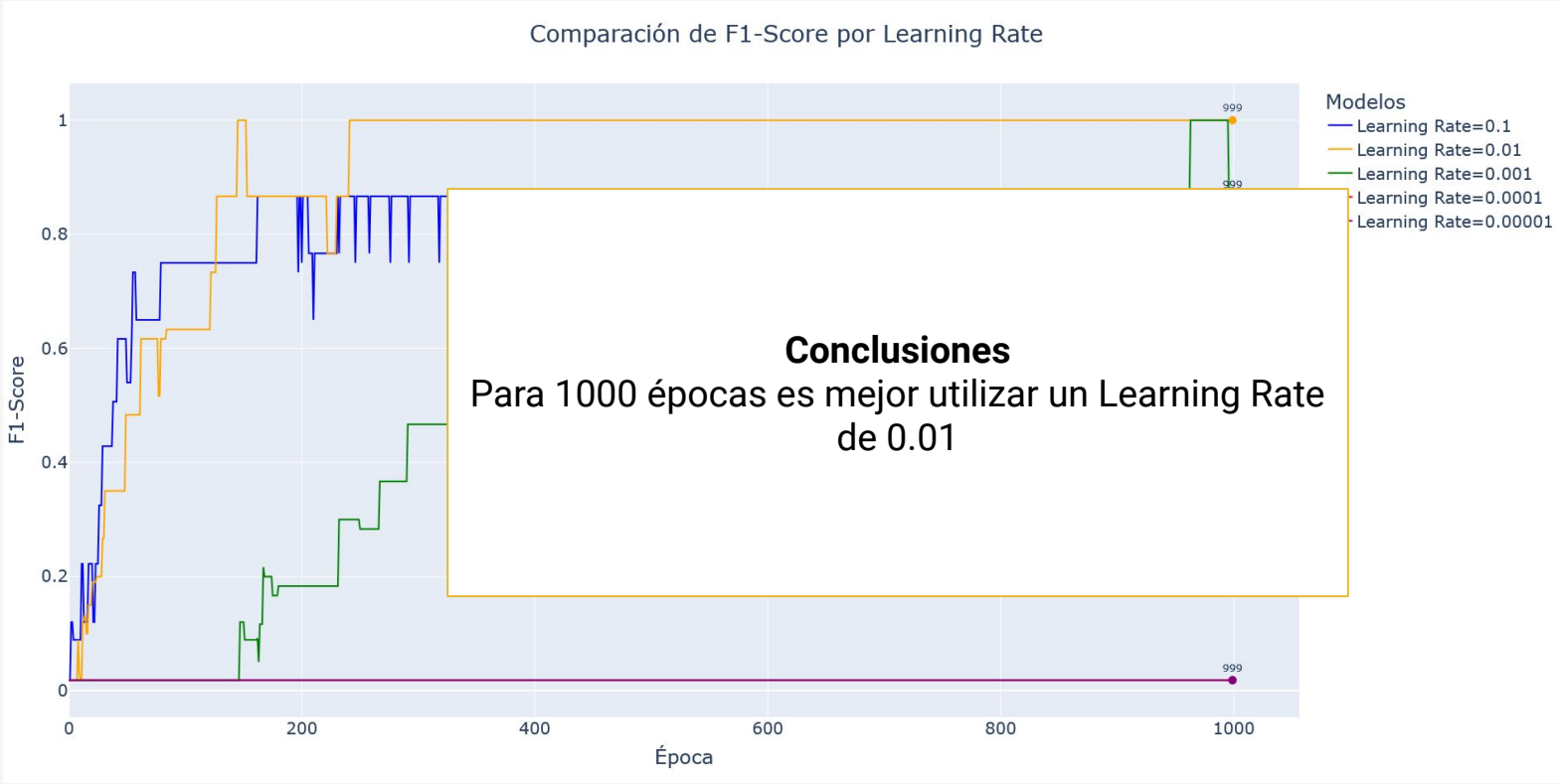
$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Architecture = [35,25,10], Learning Rate (η) = ? , A. Function = tanh ,Beta = 4 , Optimizer = Gradient Descent

Arquitectura

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Conclusiones

Para 1000 épocas es mejor utilizar un Learning Rate de 0.01

Architecture = [35,25,10], Learning Rate (η) = ? , A. Function = tanh ,Beta = 4 , Optimizer = Gradient Descent

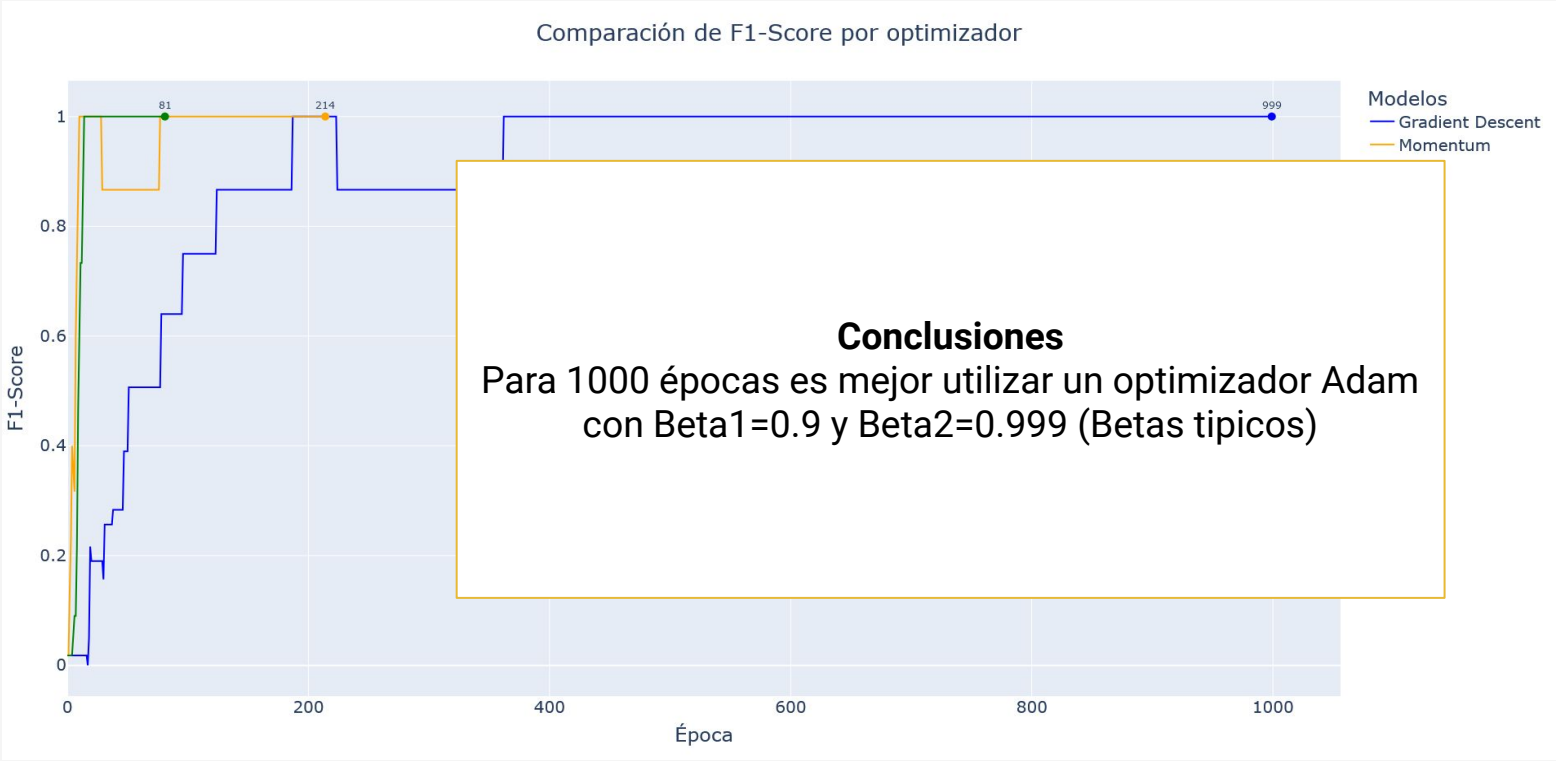
Arquitectura

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 4 , Optimizer = ?

Arquitectura

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



Conclusiones

Para 1000 épocas es mejor utilizar un optimizador Adam con Beta1=0.9 y Beta2=0.999 (Betas tipicos)

Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 4, Optimizer = ?

Arquitectura

En base a lo estudiado tomaremos la siguiente arquitectura para los próximos análisis:

- Estructura de layers: [35, 25, 10]
- Función de activación: tanh
  - Beta: 4
- Learning Rate: 0.01
- Optimizador: Adam
  - Beta_1: 0.9
  - Beta_2: 0.999

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 4, Optimizer = Adam

Capacidad de generalización

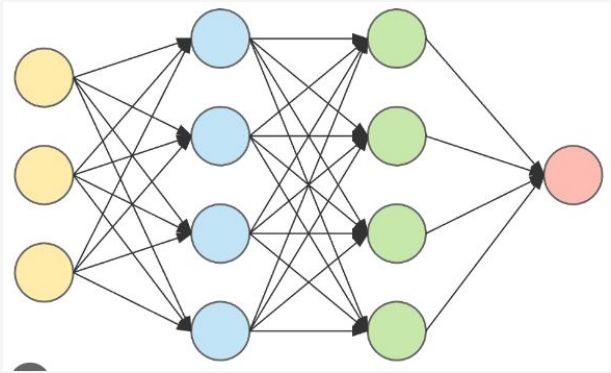
En este análisis buscaremos responder la siguiente pregunta:
¿El perceptrón tiene capacidad de generalización dados los datos de entrenamiento?

Para el siguiente experimento usaremos la arquitectura analizada previamente. El conjunto de entrenamiento serán los dígitos 4, 5 y 6 (utilizamos más de un número para evitar el sesgo) y en el conjunto de testeo incluímos todos los dígitos del 0 al 9 sin ruido y con ruido gaussiano de media 0 y desvíos estándar 0.05, 0.1, 0.15 y 0.2.

1	1	1	1	1
1	0	0	0	0
1	1	1	1	0
0	0	0	0	1
0	0	0	0	1
1	0	0	0	1
0	1	1	1	0

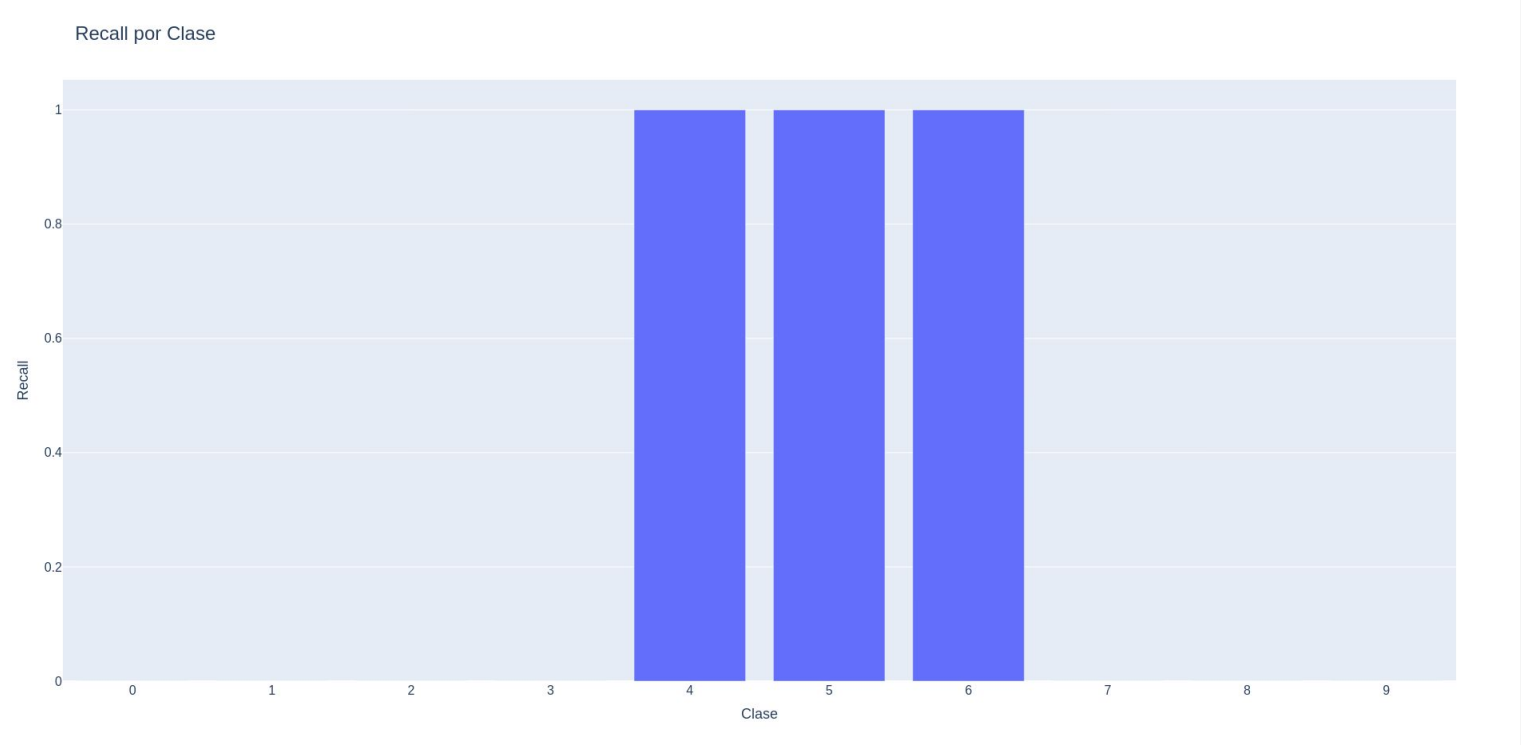
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	1	1
0	0	0	1	0
0	0	0	1	0

0	0	1	1	0
0	1	0	0	0
1	0	0	0	0
1	1	1	1	0
1	0	0	0	1
1	0	0	0	1
0	1	1	1	0



Capacidad de generalización

Evaluaremos utilizando la métrica de **Recall** ya que buscaremos ver si el modelo puede clasificar los dígitos con los cuales no fue entrenados cuando realmente le enviamos esos dígitos.



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 4, Optimizer = Adam

Capacidad de generalización

Conclusión

El modelo no tiene capacidad de generalización en este caso. Es decir, si solo lo entrenamos con 4, 5 y 6 nunca podrá predecir el 0, 1, 2, 3, 7, 8 y 9 ya que nunca aprendió cómo se ven sus “figuras”, es decir, sus representaciones matriciales de 0s y 1s y no puede inferir nada sobre estas a partir de las representaciones de los elementos del conjunto de entrenamiento

Ruido Gaussiano

La idea del ruido gaussiano es formar una matriz de ruido con valores aleatorios que se obtienen de una distribución gaussiana con una cierta media μ y desvío estándar σ

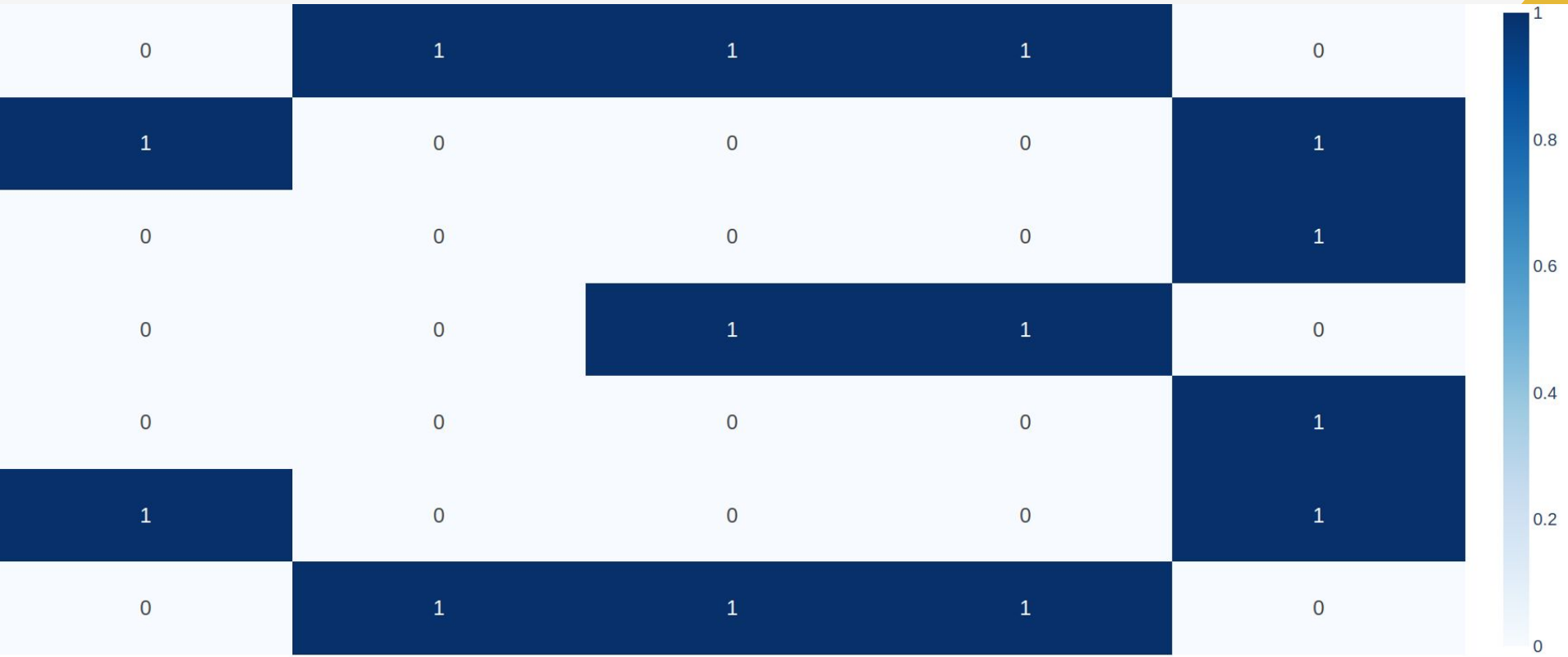
Esto nos permitirá obtener distintos conjuntos de prueba luego del entrenamiento formando nuevas matrices según la ecuación

Matriz original + Matriz de ruido = Matriz ruidosa

Luego se evalúa cuán resistente es el modelo entrenado con matrices originales que representan números a las matrices ruidosas

Gaussian noise

Original matrix

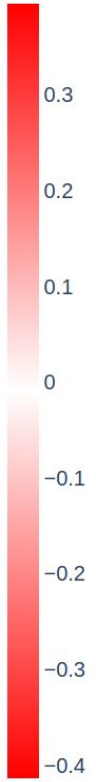


Gaussian noise

Noise matrix

- Usamos una distribución Gaussiana de media 0 y desvío estándar 0.2 para obtener los valores

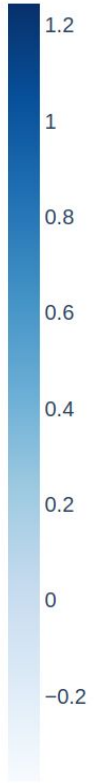
-0.16	-0.41	0.15	0.11	-0.11
0.05	0.38	-0.23	0.04	-0.03
0.2	0.05	0	-0.01	-0.08
0.11	-0.06	-0.15	0.01	0.04
0.19	0.14	-0.21	0.29	0.08
-0.05	-0.38	0.04	0.4	0.25
-0.09	0.15	-0.06	0.08	0.05



Gaussian noise

Noisy number matrix

-0.16	0.59	1.15	1.11	-0.11
1.05	0.38	-0.23	0.04	0.97
0.2	0.05	0	-0.01	0.92
0.11	-0.06	0.85	1.01	0.04
0.19	0.14	-0.21	0.29	1.08
0.95	-0.38	0.04	0.4	1.25
-0.09	1.15	0.94	1.08	0.05



Gaussian noise

Para el siguiente análisis utilizaremos un modelo con el siguiente conjunto de hiperparámetros:

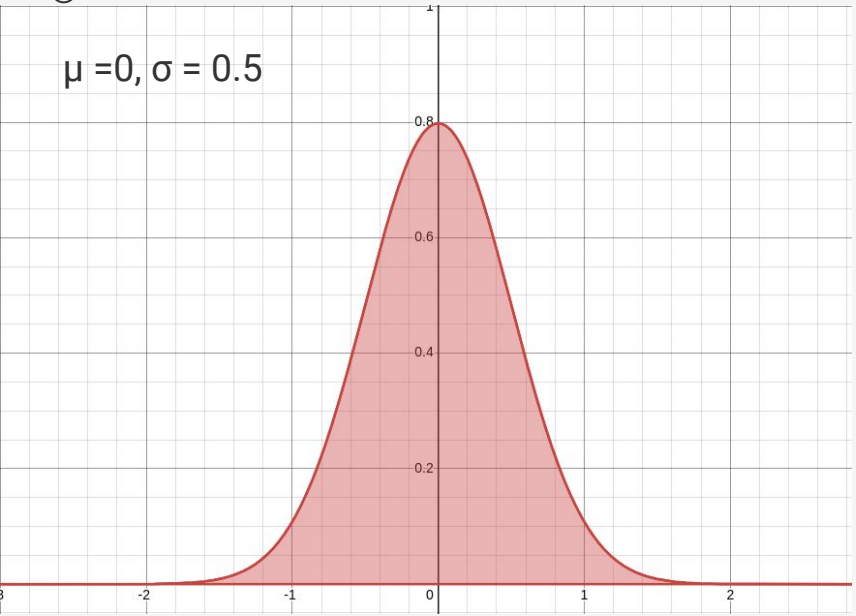
- Estructura de layers: [35, 25, 10]
- Función de activación: tanh
  - Beta: 4
- Learning Rate: 0.01
- Épsilon: 0.01
- Optimizador: Adam
  - Beta_1: 0.9
  - Beta_2: 0.999

El modelo utilizó 64 épocas para su entrenamiento. Una vez entrenado generamos distintos conjuntos de prueba con ruido gaussiano de media μ=0 y donde variamos el desvío estándar σ. Cada conjunto de prueba tiene 100 matrices con el mismo desvío estándar, 10 por cada número. A partir de estos conjuntos de prueba tomamos la media de las métricas F1-Score de cada clase.

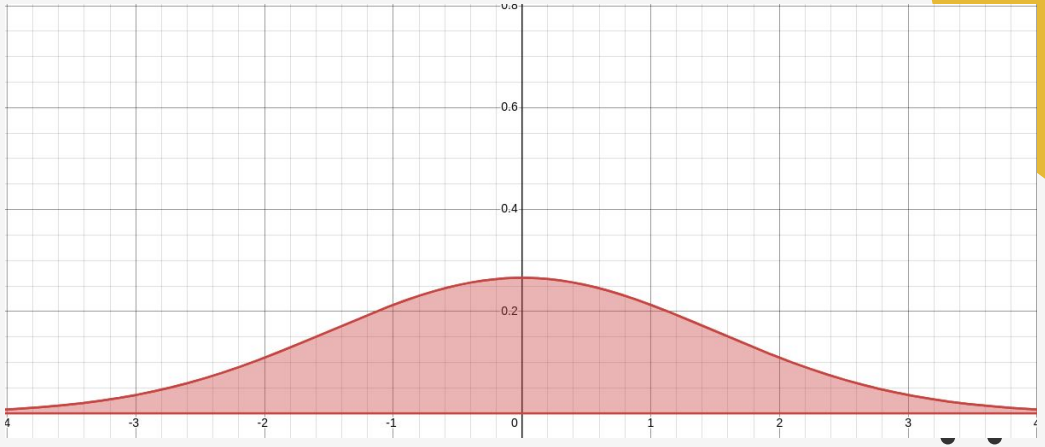
Gaussian noise

Por qué buscaremos variar el desvío estándar?

μ =0, σ = 0.5



μ =0, σ = 1.5

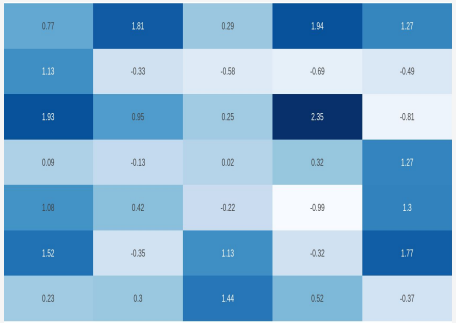


A mayor desvío estándar, mayor será la probabilidad de tomar números alejados de la media para formar la matriz de ruido y de esta forma afectaremos en mayor medida la matriz que representa al número.

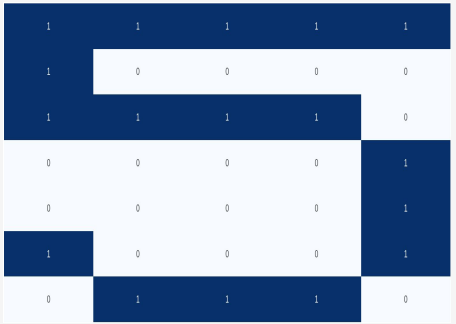
Gaussian noise

Cómo se refleja el ruido de dichas distribuciones en las matrices?

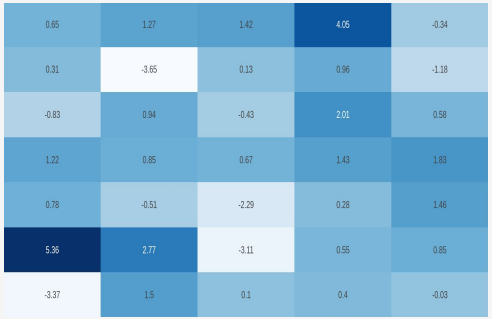
μ =0, σ = 0.5



original

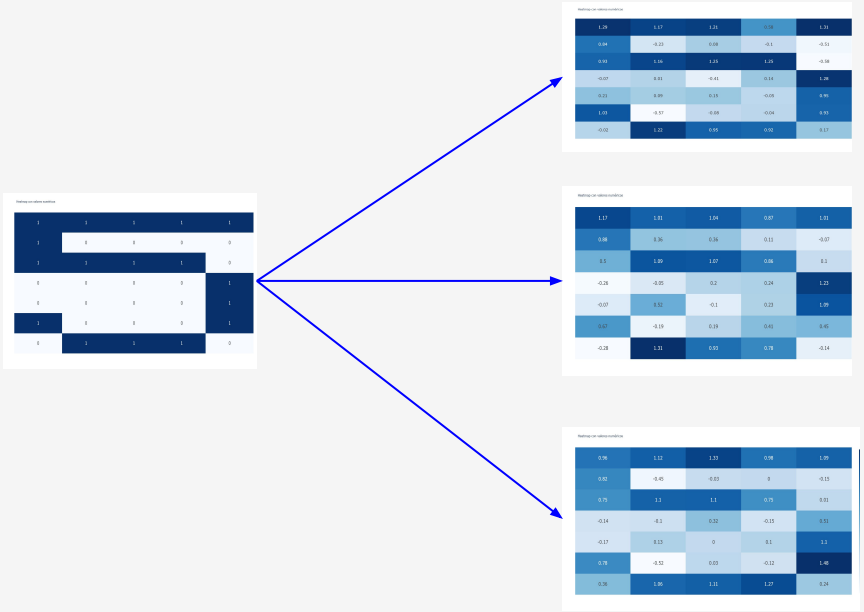


μ =0, σ = 1.5



Dataset Expansion

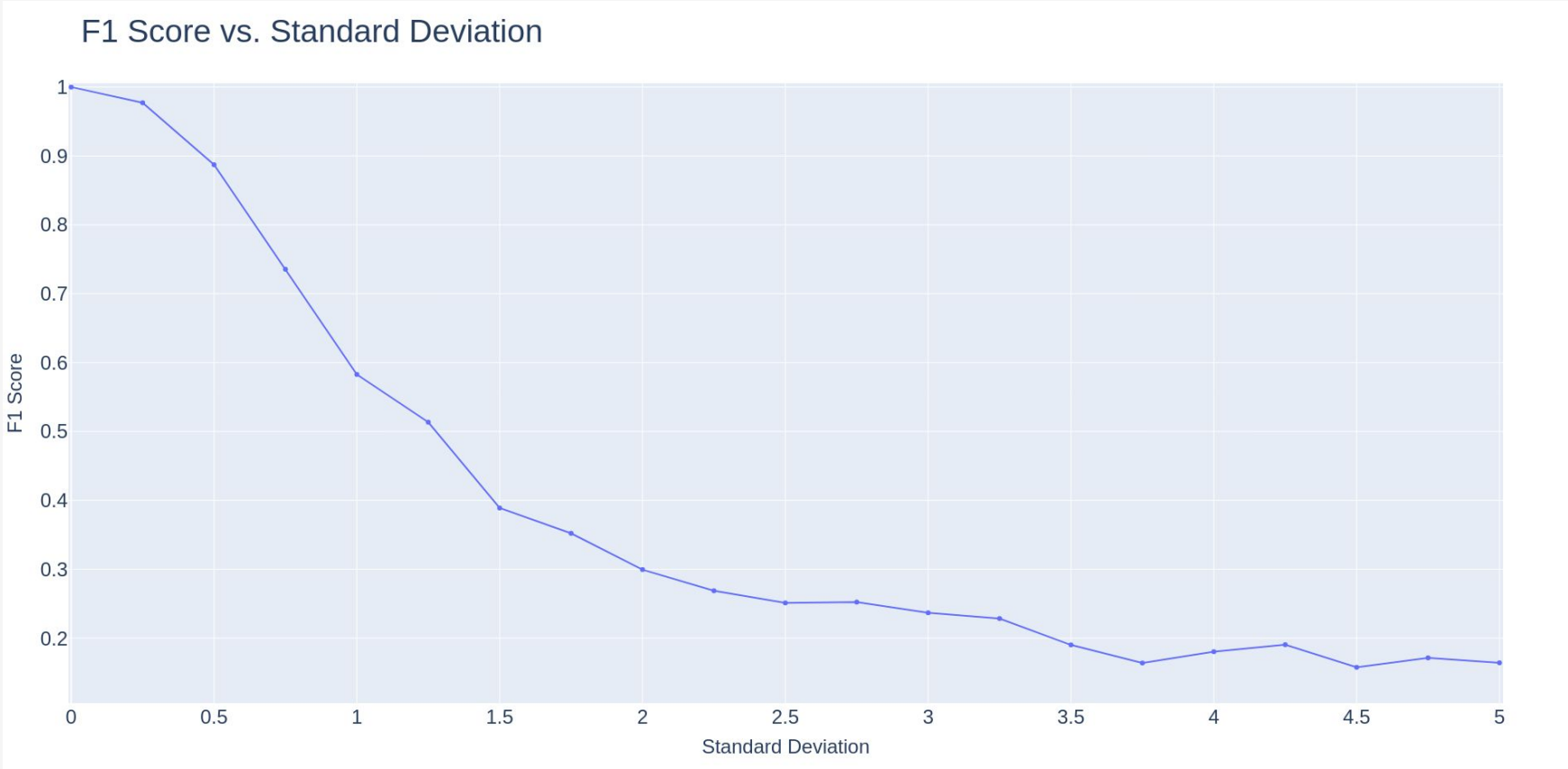
Para tener más datos sobre los que testear, al conjunto de testing se lo extendió.



Ejemplo de expansión del dataset al triple de su tamaño original.

Nosotros lo expandimos x100.

Gaussian noise



Architecture = [35,25,10], Learning Rate (η) = 0.01, A. Function = tanh ,Beta = 4, Optimizer = Adam

Gaussian noise

Conclusión

El modelo tiene una resistencia al ruido sorprendente si vemos las representaciones de los dígitos con ruido. Presenta estos resultados hasta llegar a un desvío superior a 1 donde a partir de ese momento empieza a acertar la mitad del conjunto y va disminuyendo rápidamente para los desvíos posteriores. Finalmente queda anclado en un 10% de aciertos.

¡Muchas gracias!