Entrenamiento de una red neuronal para la identificación de los parámetros de la señal que caracterizan la señal l_{cm} y σ a partir de la representación del autoencoder

- Ahora que cambió la representación en baja dimensionalidad porque cambiamos la forma en que se hacía tenemos que reentrenar una red neuronal para poder identificar los parámetros de las distribuciones de tamaños de donde provienen las señales. Para esto se hizo una exploración de hiperparámetros teniendo en cuenta distintas arquitecturas y funciones de activación. Al final se llegó a una red bastante similar a la anterior con respecto a la arquitectura pero esta vez con la función de activación tahn.

> **Multiple expressions in one cell.**
> How would you like to fix it?
>
> - Split this cell into 14 cells, or
> - Wrap all code in a *begin ... end* block.

```
# Importamos las librerias necesarias
using Flux
using Statistics
using Flux: train!
using Plots
using DataFrames
using CSV
using StatsPlots
using LaTeXStrings
using LinearAlgebra
using PlotlyJS
using CUDA
using Random
using Measures
# Importamos los parámetros del experimento
include("../1-GeneracionDatos/Parameters.jl");
```

Multiple expressions in one cell.

How would you like to fix it?

- Split this cell into 13 cells, or
- Wrap all code in a *begin ... end* block.

```
# Distribucion de probabilidad log-normal que se puede agregar a la función costo de la red neuronal, lleva mucho tiempo de entrenamiento

function Pln(lcm, σ)
    return [(exp(-(log(lc) - log(lcm))^2 / (2σ^2))) / (lc * σ * sqrt(2π)) for lc in lcs]
end

################################################################################################################################
# Funciones de pre procesamiento para escalar los datos y estandarizarlos

# Normalización Max-Min
function MaxMin(data)
    min_vals = minimum(data, dims=1)
    max_vals = maximum(data, dims=1)
    scaled_data = (data .- min_vals) ./ (max_vals .- min_vals)
    return scaled_data

end

# Estandarización Z
function Standarize(data)
    mean_vals = mean(data, dims=1)
    std_devs = std(data, dims=1)
    standardized_data = (data .- mean_vals) ./ std_devs
    return standardized_data
end

################################################################################################################################
# Metricas de validacion de la red neuronal, solo utilice RMAE

# Root Mean Squared Error
function RMSE(predicted, real)
    return sqrt(Flux.mse(predicted, real))
end

# Mean Absolute Error
function MAE(predicted, real)
    return sum(abs.(predicted .- real)) / length(predicted)
end

# R2 score
function R2_score(predicted, real)
    return 1 - sum((predicted .- real).^2) / sum((real .- mean(real)).^2)
end

```

```julia
# Realetive Root Mean Squared Error
function RRMSE(predicted, real)
    return sqrt(mean((predicted .- real).^2)) / mean(real)
end

# Relative Mean Absolute Error
function RMAE(predicted, real)
    return mean(abs.(predicted .- real)) / mean(real)
end

# Mean Absolute Percentaje Error
function MAPE(predicted, real)
    return mean(abs.((predicted .- real) ./ real))
end

################################################################################################################################

# Regularizaciones L1 y L2
pen_l2(x::AbstractArray) = Float32.(sum(abs2, x) / 2)
pen_l1(x::AbstractArray) = Float32.(sum(abs, x) / 2)

################################################################################################################################

# Funciones de pre procesamiento para escalar los datos y estandarizarlos

# Normalización Max-Min
function MaxMin(data)
    min_vals = minimum(data, dims=1)
    max_vals = maximum(data, dims=1)
    scaled_data = (data .- min_vals) ./ (max_vals .- min_vals)
    return scaled_data

end

# Estandarización Z
function Standarize(data)
    mean_vals = mean(data, dims=1)
    print(mean_vals)
    std_devs = std(data, dims=1)
    print(std_devs)
    standardized_data = (data .- mean_vals) ./ std_devs
    return standardized_data
end;
```

Multiple expressions in one cell.

How would you like to fix it?

- Split this cell into 2 cells, or
- Wrap all code in a *begin ... end* block.

```
# Leemos los datos de las señales en PCA
path_read = "../1-GeneracionDatos/Data/"
df_datasignals =
CSV.read("C:/Users/Propietario/OneDrive/Escritorio/ib/Tesis_V1/MLonNMR/2-2-
Autoencoders/Models/ReducedDataNLData.csv", DataFrame)
```

K-folds

Usamos la técnica de validación cruzada K-folds para prevenir el sobreajuste de la red neuronal. Este método divide el conjunto de datos en K subconjuntos, y el modelo se entrena K veces, utilizando en cada iteración un subconjunto diferente como conjunto de validación y el resto como conjunto de entrenamiento. Al final, los resultados obtenidos en cada iteración se promedian, esto nos permite obtener una mejor estimación de cómo funcionará el modelo con datos no vistos previamente.

El proceso de separación de los datos es el siguiente:

Los datos se dividen en K subconjuntos de igual tamaño. Cada subconjunto requiere de una representación adecuada de los datos, eligiendo señales de validación basadas en diferentes valores de σ y l_{cm} de la siguiente manera: Se comienza con $l_{cm} = 0.5$, pero en cada fold se utiliza un valor distinto de σ. Posteriormente, se seleccionan valores equiespaciados de l_{cm} y σ de modo que el 20% de los datos se destinen a la validación. Así, al comenzar cada fold con un valor diferente de σ, se obtienen K conjuntos disjuntos para validar. Después de entrenar el modelo K veces, cambiando el conjunto de validación en cada iteración, se calcula el rendimiento promedio de la red. Para esto, se obtienen la media y la desviación estándar de los errores de validación RMSE de los K folds.

Multiple expressions in one cell.

How would you like to fix it?

- Split this cell into 11 cells, or
- Wrap all code in a *begin ... end* block.

```
# Utilizamos la técnica k-fold de validación cruzada para prevenir el overfitting de la red neuronal
# Definimos el número de folds
folds = 5
percent_valid = 0.2
step_valid = Int(1 / percent_valid)
num_datos = Int(size(df_datasignals, 1))

# Guardamos los datos de validacion de cada NN en cada fold
out_of_sample_data = []
out_of_sample_pred = []

# Guardamos la metrica de validación de cada NN en cada fold
scores_RMSE = []

# Definimos desde donde empezamos los datos de testing
idx_startTest = 6

# Primero sacamos los datos de testing de los datos de señales, estos seran un 3er conjunto de datos que no se usara para entrenar ni validar la red
df_datasignals_out = df_datasignals[idx_startTest:step_valid:num_datos,:]
# Datos de Testing
df_datasignals_minus_out = df_datasignals[setdiff(1:num_datos, idx_startTest:step_valid:num_datos),:]
# Nuevo numero de datos que tenemos para entrenamiento + validacion
num_datos_new = Int(size(df_datasignals_minus_out, 1))
```

UndefVarError: `folds` not defined

Stack trace

Here is what happened, the most recent locations are first:

1. *This cell: line 3*

```
2
3 for k in 1:folds
4     # Usamos 5 conjuntos disjuntos de datos de validación del 5% total de
los datos para cada fold
```

```
# Método de K-Folds

for k in 1:folds
    # Usamos 5 conjuntos disjuntos de datos de validación del 5% total de los datos
para cada fold

    datasignals_valid = Float32.(Matrix(df_datasignals_minus_out[k^2 +
10:step_valid:num_datos_new,1:3])')
    datasignals = Float32.(Matrix(df_datasignals_minus_out[setdiff(1:num_datos_new,
k^2 + 10:step_valid:num_datos_new),1:3])')

    σ_valid = df_datasignals_minus_out[k^2 + 10:step_valid:num_datos_new,4]
    lcm_valid = df_datasignals_minus_out[k^2 + 10:step_valid:num_datos_new,5]

    σ_col = df_datasignals_minus_out[setdiff(1:num_datos_new, k^2 +
10:step_valid:num_datos_new),4]
    lcm_col = df_datasignals_minus_out[setdiff(1:num_datos_new, k^2 +
10:step_valid:num_datos_new),5]

    dataparams = hcat(lcm_col, σ_col)'
    dataparams_valid = hcat(lcm_valid, σ_valid)'

    # Uno de los modelos que dio buenos resultados en la exploración anterior
    # 17,"[3, 32, 64, 32, 16,
2]",tanh,ADAM,0.0,None,0,0.017810457567638147,0.01427131790632415

    id = 17

    # Definimos la red neuronal
    model = Chain(
        Dense(3, 32),
        Dense(32, 64, tanh_fast),
        Dense(64, 32, tanh_fast),
        Dense(32, 16, tanh_fast),
        Dense(16, 2, softplus),
    )

    # Cargamos los parámetros del modelo
    path = "C:/Users/Propietario/OneDrive/Escritorio/ib/Tesis_V1/MLonNMR/2-
NNIdentificaHahn/G_8.73e-7_TE_1_AE/Params/017_params.csv"
```

```
34
35      p, re = Flux.destructure(model)
36
37      theta = CSV.read(path, DataFrame)
38      p = @views Float32.(theta[:,1])
39
40      # Función de loss
41      function loss(x,y)
42          return Flux.mse(re(p)(x), y)
43      end
44
45      # Definimos el metodo de aprendizaje y la tasa de aprendizaje
46      η = 1e-4
47      opt = ADAM(η)
48
49      # Definimos el número de épocas
50      epochs = 5000
51
52      # Definimos el tamaño del batch
53      batch_size = 100
54
55      # Usamos dataloader para cargar los datos
56      data = Flux.DataLoader((datasignals, dataparams), batchsize = batch_size,
    shuffle = true)
57      data_valid = Flux.DataLoader((datasignals_valid, dataparams_valid), batchsize =
    batch_size, shuffle = true)
58
59      # Definimos una funcion de callback para ver el progreso del entrenamiento
60      global iter = 0
61      cb = function()
62          global iter += 1
63          if iter % length(data) == 0
64              epoch = iter ÷ length(data)
65              if epoch % 500 == 0
66                  actual_loss = loss(data.data[1], data.data[2])
67                  actual_valid_loss = loss(data_valid.data[1], data_valid.data[2])
68                  println("Epoch $epoch || Loss = $actual_loss || Valid Loss =
    $actual_valid_loss")
69              end
70          end
71      end;
72
73      # Entrenamos la red neuronal con el loss mse variando la tasa de aprendizaje
    cada 500 épocas
74      for epoch in 1:epochs
75          Flux.train!(loss, Flux.params(p), data, opt, cb=cb)
76          if epoch % 500 == 0
77              η = η * 0.2
78              opt = ADAM(η)
79          end
80      end
81
82      # Predicción de la red en la validacion
83      predictions_valid = re(p)(datasignals_valid)
84
85      # Métricas de validación de la red
86      RMSE_valid = RMSE(predictions_valid, dataparams_valid)
87
```

```
    push!(scores_RMSE, RMSE_valid)

    # Guardamos los datos de validación y las predicciones de la red
    push!(out_of_sample_data, dataparams_valid)
    push!(out_of_sample_pred, predictions_valid)

    println("Fold $k terminado con score de validación RMSE = $RMSE_valid")

```

El promedio de la metrica RMSE en los conjuntos 5 folds de validación es 0.014 y el desvio estandar es 0.0011

- Al haber testeado en todos estos datos y seguir obteniendo valores similares podemos asegurar que el modelo no está sobreajustando
- Podemos reentrenar el modelo con todos los datos y obtener un modelo más robusto

> **Multiple expressions in one cell.**
>
> How would you like to fix it?
>
> - Split this cell into 8 cells, or
> - Wrap all code in a *begin ... end* block.

```
# Convertimos las señales de test y train a una matriz de Float32
datasignals_test = Float32.(Matrix(df_datasignals_out[:,1:3])')
datasignals = Float32.(Matrix(df_datasignals_minus_out[:,1:3])')

# Extraemos las columnas objetivo de test y train
σ_test = df_datasignals_out[:,4]
lcm_test = df_datasignals_out[:,5]

σ_col = df_datasignals_minus_out[:,4]
lcm_col = df_datasignals_minus_out[:,5]

# Las concatenamos en dataparams
dataparams = hcat(lcm_col, σ_col)'
dataparams_test = hcat(lcm_test, σ_test)';
```

Multiple expressions in one cell.

How would you like to fix it?

- Split this cell into 2 cells, or
- Wrap all code in a *begin ... end* block.

```
# Función de loss
function loss(x,y)
    y_hat = re(p)(x)
    return Flux.mse(y_hat, y)
end

# Loss compuesto para hacer un fine tune comparando la predicción de la red con la distribución log-normal
function composed_loss(x,y)
    y_hat = model(x)
    Pln_predicted = Pln.(y_hat[1,:], y_hat[2,:])
    Pln_real = Pln.(y[1,:], y[2,:])
    return mean(Flux.mse.(Pln_predicted,Pln_real)) + Flux.mse(y_hat, y)
end
```

Multiple expressions in one cell.

How would you like to fix it?

- Split this cell into 15 cells, or
- Wrap all code in a *begin ... end* block.

```
# Definimos la red neuronal
model = Chain(
    Dense(3, 32),
    Dense(32, 64, tanh_fast),
    Dense(64, 32, tanh_fast),
    Dense(32, 16, tanh_fast),
    Dense(16, 2, softplus),
)

# Cargamos los parámetros del modelo
path = "C:/Users/Propietario/OneDrive/Escritorio/ib/Tesis_V1/MLonNMR/2-NNIdentificaHahn/G_8.73e-7_TE_1_AE/Params/017_params.csv"

p, re = Flux.destructure(model)

# Definimos el tamaño de los mini batchs para entrenar
batch_size = 100

# Usamos dataloader para cargar los datos
data = Flux.DataLoader((datasignals, dataparams), batchsize = batch_size, shuffle = true)

# Parámetros de la red neuronal
params = Flux.params(model)

# Definimos el vector donde guardamos la pérdida
losses = []

# Definimos una funcion de callback para ver el progreso del entrenamiento
iter = 0
cb = function()
    global iter += 1
    if iter % length(data) == 0
        epoch = iter ÷ length(data)
        actual_loss = loss(data.data[1], data.data[2])
        if epoch%100 == 0
            println("Epoch $epoch || Loss = $actual_loss")
        end
        push!(losses, actual_loss)
    end
end;

losses_composed = []
cb2 = function()
    global iter += 1
    epoch = iter ÷ length(data)
    actual_loss = composed_loss(data.data[1], data.data[2])
    println("Epoch $epoch || Loss = $actual_loss")
```

```
    push!(losses_composed, actual_loss)
end;

# Definimos el modo de aprendizaje y la tasa de aprendizaje
η = 1e-4
opt = ADAM(η)

# Definimos el número de épocas
epochs = 5000;

println("Se va a entrenar una red con \n batch_size: $(batch_size) \n η: $(η) \n Optimizer: $(opt) \n Epochs: $(epochs)")
```

UndefVarError: `epochs` not defined

Stack trace

Here is what happened, the most recent locations are first:

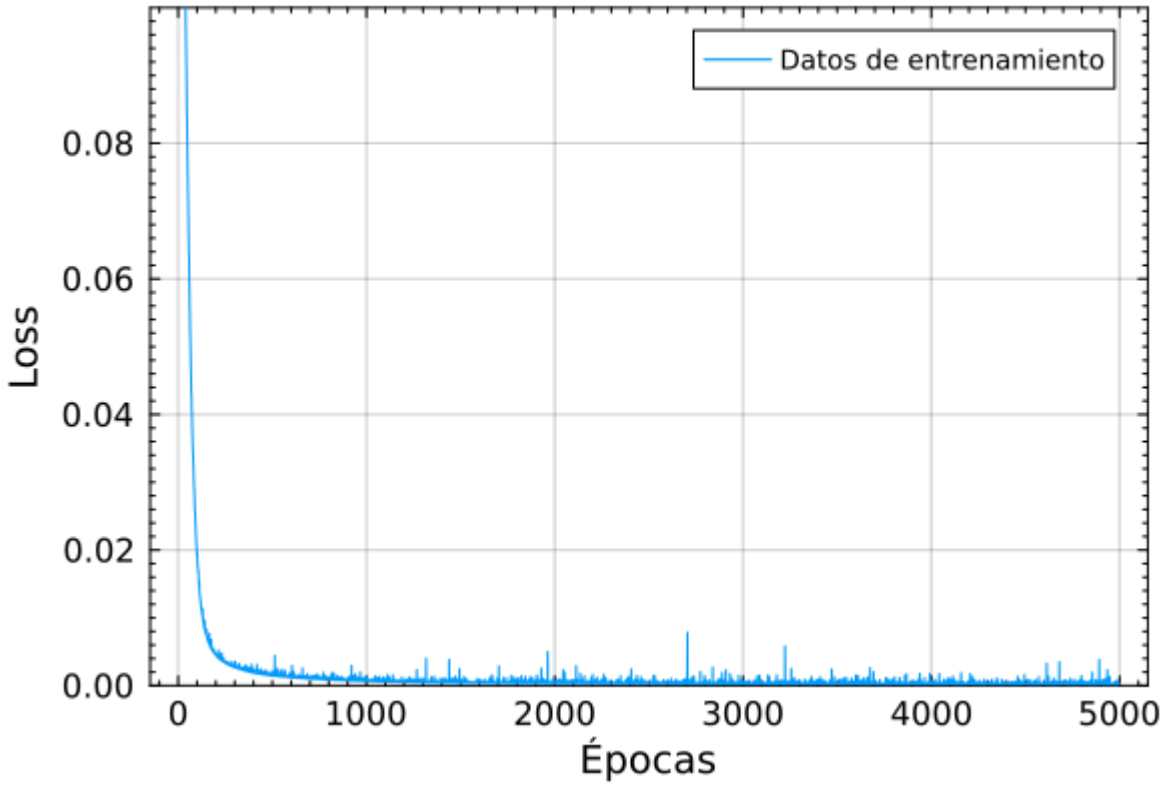
1. *This cell: line 2*

```
# Entrenamos la red neuronal con el loss mse
for epoch in 1:epochs
    Flux.train!(loss, Flux.params(p), data, opt, cb=cb)
```

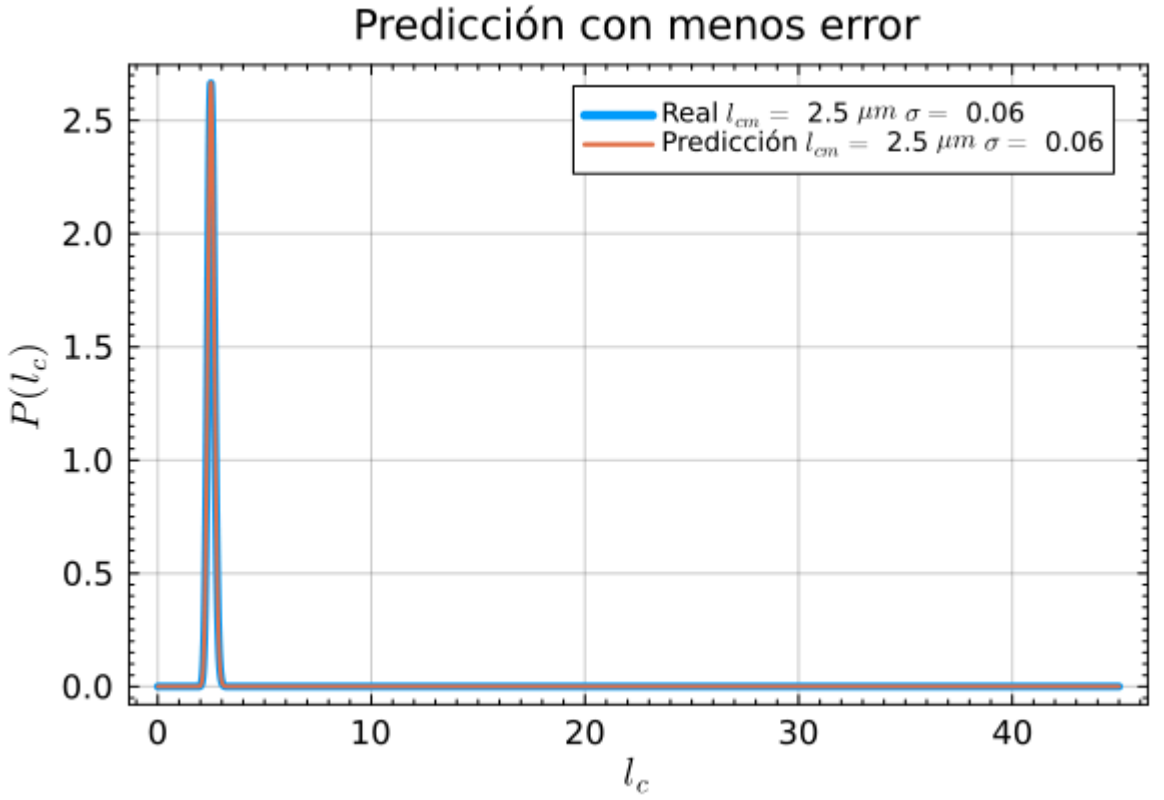
```
# Entrenamos la red neuronal con el loss mse
for epoch in 1:epochs
    Flux.train!(loss, Flux.params(p), data, opt, cb=cb)
end

# Entrenamiento con loss compuesto
# for epoch in 1:1
#     Flux.train!(composed_loss, Flux.params(model, opt), data, opt, cb=cb2)
# end
```

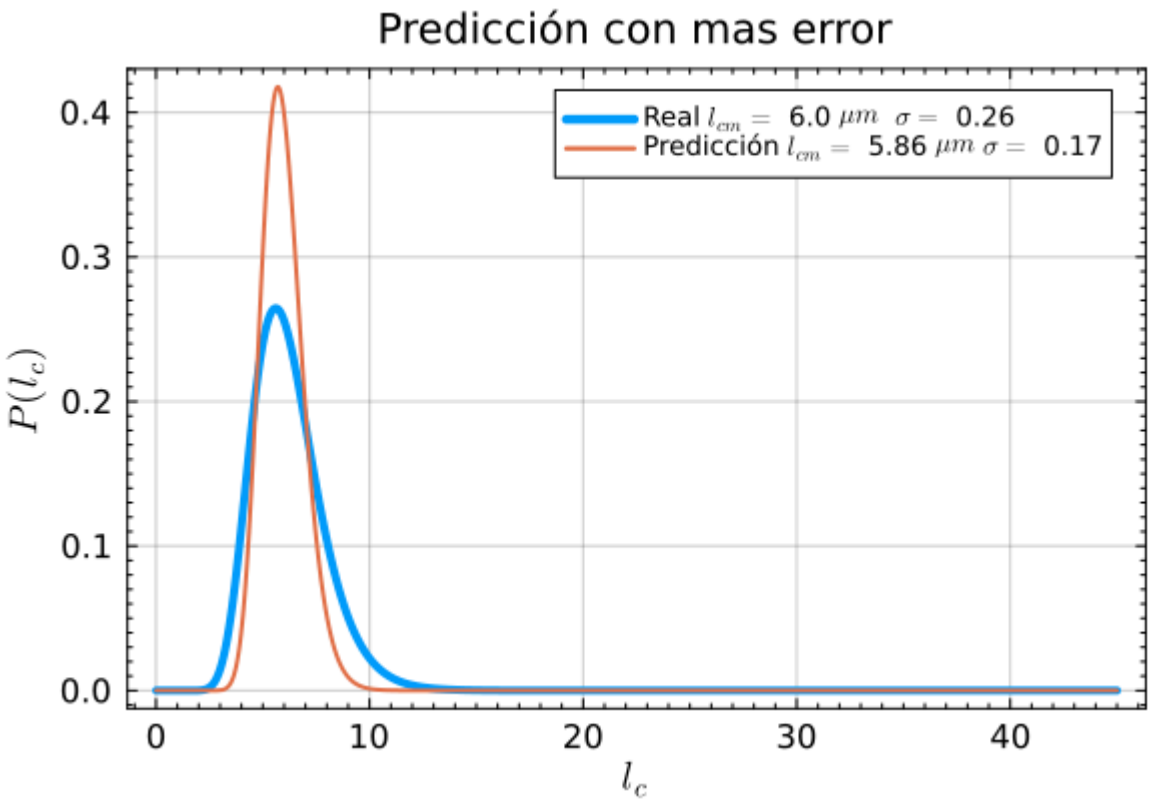
Loss luego de reentrenar con los datos de validación sabiendo que no hay overfitting

```
using PlutoUI
```

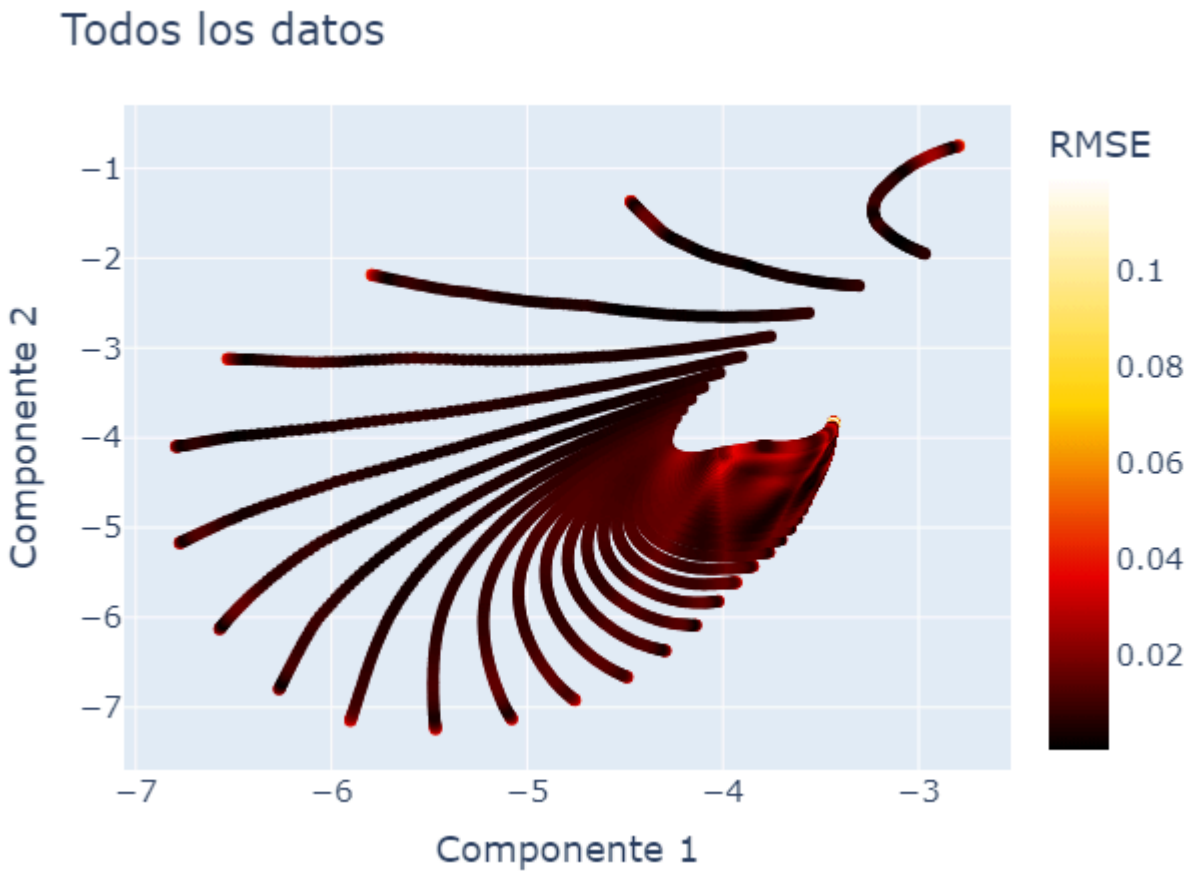


- El error raíz cuadrática media de la red neuronal en el conjunto de entrenamiento es 0.02





- Este modelo funciona aún mejor que el anterior quizas porque esta vez la representación en baja dimensionalidad de donde entran los datos a esta red proviene de un autoencoder.
- El error RMSE en el conjunto de test es 0.02.



Recordar que la represantación aunque cambió sigue siendo similar en el hecho que σ y l_{cm} pequeños es donde mas separadas y van aumentando a medida que se acercan a la zona donde hay mas puntos acumulados. Dejo un HTML en el trello con el gráfico que indica cuando pasas el mouse el sigma y en color el lcm