Diferencia entre Parámetros e Hiperparámetros

Semillero de Neuroinformática e Inteligencia Artificial

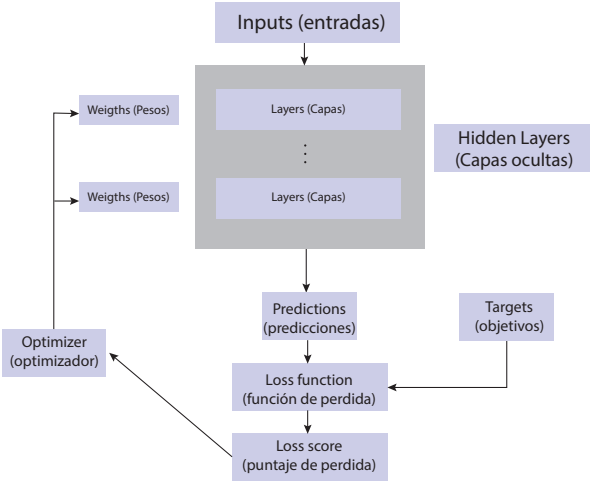
Parámetros de la arquitectura del modelo

Parámetros de optimización

Parámetros de regularización

Parámetros de inicialización

Diagrama lógico de una red neuronal



Parámetros vs Hiperparámetros

- **Parámetros:** Valores aprendidos por el modelo durante el entrenamiento.
  - Peso y sesgo en una red neuronal.
  - Coeficientes en una regresión lineal.
  - No se ajustan manualmente, sino que se actualizan automáticamente a medida que el modelo aprende.

- **Hiperparámetros:** Valores que se ajustan antes del proceso de entrenamiento.
  - Número de capas y neuronas en una red neuronal.
  - Tasa de aprendizaje en un algoritmo de optimización.
  - Seleccionados manualmente o mediante métodos de búsqueda de hiperparámetros (p.ej., Grid Search, Random Search, etc.).

Hiperparámetros en Redes Neuronales

1. **Arquitectura del modelo**
   - Número de capas.
   - Neuronas por capa.
   - Tipo de capa.
   - Funciones de activación.
2. **Optimización**
   - Número de épocas.
   - Método de optimización.
   - Tamaño del lote (batch size).
3. **Regularización**
   - Dropout.
   - L1/L2.
   - Early stopping.
4. **Inicialización**
   - Inicialización de pesos.
   - Inicialización de sesgos.

Parámetros de la arquitectura del modelo

Número de capas

El número de capas en una red neuronal tiene las siguientes implicaciones:

- **Capacidad de representación**: Más capas permiten representar funciones más complejas, pero demasiadas pueden causar overfitting.
- **Tiempo de entrenamiento**: Más capas aumentan el tiempo de entrenamiento y la dificultad por el desvanecimiento del gradiente.
- **Generalización**: Pocas capas ocultas favorecen la generalización, muchas capas tienden a memorizar el conjunto de entrenamiento.
- **Hiperparámetros**: Se deben ajustar más hiperparámetros al agregar capas, complicando su optimización.
- **Desempeño**: Agregar capas mejora el desempeño hasta cierto punto, luego se estanca y empeora por overfitting.

Número de neuronas por capa

El número de neuronas por capa implica las mismas consideraciones que el número de capas, sin embargo, el número de neuronas afecta directamente la **dimensión de los datos**. El número de neuronas determinará el número de dimensiones que se utilizan para **representar** los datos de entrada.

Desvanecimiento y explosión de gradiente

- **Exploding gradients**: Durante la propagación hacia atrás del error (backpropagation), las derivadas de las capas más profundas pueden hacerse muy grandes debido a la multiplicación repetida de números mayores que 1. Matemáticamente, si los pesos w^l están inicializados con valores altos, entonces:

$$\frac{\partial E}{\partial w^l} = \frac{\partial E}{\partial a^L} \frac{\partial a^L}{\partial z^L} \frac{\partial z^L}{\partial w^l}$$

  Donde E es el error, a la activación de la capa L, z la entrada ponderada. Si los pesos w^l son »1, entonces $\frac{\partial E}{\partial w^l}$ explota.

- **Vanishing gradients**: Similarmente, derivadas pequeñas (<1) se vuelven cada vez más pequeñas y se desvanecen al backpropagarse a las capas iniciales. Si los pesos w^l son «1, entonces $\frac{\partial E}{\partial w^l} \approx 0$, y las capas iniciales no aprenden.

Tipos de capas en redes neuronales

Dentro de los tipos más comunes de capas utilizadas en redes neuronales están:

- **Entrada**: Recibe los datos de entrada.
- **Ocultas**: Transforman los datos internamente mediante funciones de activación.
- **Salida**: Produce la salida final de la red.
- **Convolucionales**: Extraen características usando filtros convolucionales.
- **Pooling**: Reducen la dimensionalidad mediante operaciones como max pooling.
- **Recurrentes**: Permiten persistencia de información en redes recurrentes.
- **Dropout**: Aplican dropout para regularizar el modelo.

Funciones de activación: RELU

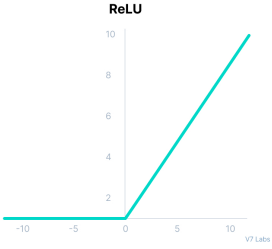


Figura: Función de activación RELU

- ReLU introduce no linealidad en la red neuronal, lo que permite aproximar funciones complejas.
- Cómputo simple: La operación de tomar el máximo es computacionalmente barata.
- Al no saturarse en 1, no sufre de problema de gradiente desvaneciente durante backpropagation.
- Puede causar neuronas muertas: Si una neurona permanece siempre negativa, se desactiva por completo.

Funciones de activación: Lineal

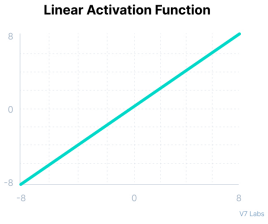


Figura: Función de activación: lineal

- Lineal: La salida es proporcional a la entrada, $f(x) = x$.
- No saturación: La salida no se limita a un rango fijo.
- Vanishing gradient: Derivada constante igual a 1, no amplifica ni atenúa el gradiente durante backpropagation.
- Al ser lineal, no puede aproximar funciones no lineales complejas.
- Se usa frecuentemente en la capa de salida para problemas de regresión.
- Rango no acotado: La salida puede tomar cualquier valor real, lo que dificulta la convergencia.

Función de activación: Sigmoid

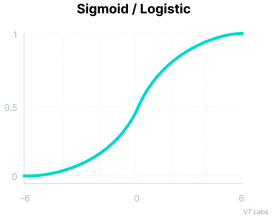


Figura: Función de activación Sigmoid

- Introduce no linealidad, permitiendo aproximar relaciones complejas entre entrada y salida.
- Acotada entre 0 y 1: $f(x) = \frac{1}{1+e^{-x}}$, la salida se satura en los límites.
- Posee derivada continua, lo que permite usarla en backpropagation.
- La derivada se hace muy pequeña en saturación, provocando desvanecimiento del gradiente.
- Valores cercanos a 0 y 1 se interpretan como falso/verdadero en clasificación.
- Requiere evaluar la exponencial, con mayor costo computacional.

Función de activación: Tanh

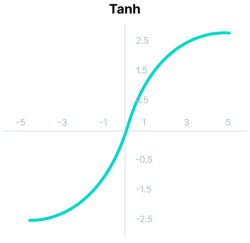


Figura: Función de activación: Tanh

- Introduce no linealidad en la red neuronal.
- Acotada entre -1 y 1: $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.
- Posee derivada continua, permitiendo su uso en backpropagation.
- La curva varía de forma suave entre saturaciones negativa y positiva.
- La derivada se hace pequeña en saturación, provocando desvanecimiento del gradiente.
- Salida centrada alrededor de 0, lo que mejora el entrenamiento.
- Computo intensivo: Requiere evaluar exponenciales, con mayor costo que ReLU.

Función de activación: Softmax

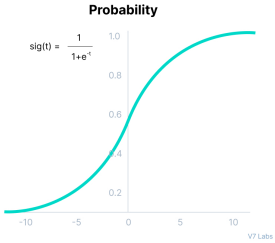


Figura: Función de activación Softmax

- La salida de la función se encuentra en el rango de 0 a 1.
- Se puede describir como una combinación de multiples sigmoides, cuyas salidas son las probabilidades , y la suma de la salida de las sigmoides es igual a 1.
- Su principal aplicación se encuentra en las capas de salida de problemas multiclase

Modificación de la arquitectura del modelo

Los parámetros de arquitectura son definidos cuando se arma el modelo. En el ejemplo que se muestra a continuación se muestra una red feedforward con 784 neuronas de entrada, 512 en la capa oculta con activación relu y una capa de salida con 10 neuronas con activación softmax.

```
model=models.Sequential()

model.add(layers.Dense(512, activation = "relu", input_shape = (28*28, )))

model.add(layers.Dense(10,activation="softmax"))
```

Universidad Tecnológica de Pereira
Facultad de Ingenierías

Parámetros de optimización

SNEIA

Número de Épocas

- **Número de épocas**: Cantidad de veces que un algoritmo recorre todo el conjunto de datos de entrenamiento.
- Cada época actualiza los parámetros del modelo para minimizar la función de pérdida.
- Épocas insuficientes pueden llevar a *underfitting*, mientras que demasiadas épocas pueden provocar *overfitting*.
- Es importante encontrar un equilibrio adecuado, utilizando técnicas como early stopping.

Modificar el Número de Épocas en Keras

El número de épocas se ajusta al momento de entrenar el modelo, de la siguiente manera:

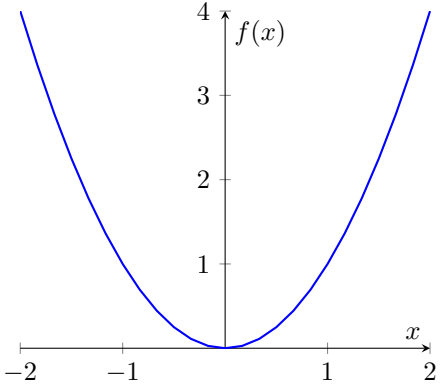
```
model.fit(X_train, y_train, epochs=n_epochs)
```

Donde `n_epochs` es el número de épocas deseado.

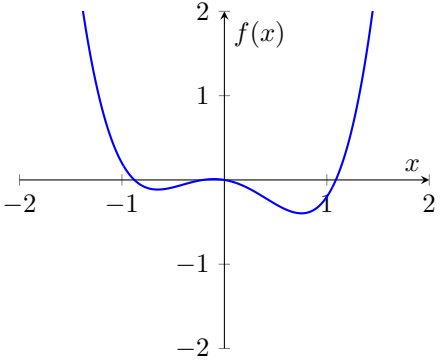
Métodos de Optimización

- Los métodos de optimización son algoritmos que ajustan los parámetros del modelo para minimizar la función de pérdida.
- Ejemplos comunes:
  - Descenso de Gradiente Estocástico (SGD): Adecuado para problemas de optimización convexos y no convexos.
  - Momentum: Mejora la convergencia del SGD al agregar un término de momento a la actualización de los parámetros.
  - RMSprop: Adecuado para problemas no convexos, ajusta la tasa de aprendizaje adaptativamente para cada parámetro.
  - Adam: Combina las ideas de Momentum y RMSprop, popular por su eficiencia y buen rendimiento en diversos problemas.
- La elección del método de optimización depende del problema y la arquitectura del modelo.

Función Convexa



Función con un mínimos Local y un Mínimo Glob



Expresiones Matemáticas de Optimizadores Comunes

Descenso de Gradiente Estocástico (SGD):

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t)$$

Momentum:

$$v_{t+1} = \gamma v_t + \eta \nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

RMSprop:

$$g_{t+1} = \rho g_t + (1-\rho)(\nabla f(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{g_{t+1} + \epsilon}} \nabla f(\theta_t)$$

Adam:

$$m_{t+1} = \alpha m_t + (1-\alpha)\nabla f(\theta_t)$$
$$v_{t+1} = \beta v_t + (1-\beta)(\nabla f(\theta_t))^2$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\alpha^{t+1}}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta^{t+1}}$$

Descenso del gradiente estocástico

Descenso de Gradiente Estocástico (SGD):

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t)$$

Esta fórmula actualiza los parámetros θ del modelo. Aquí, θ_t representa los parámetros en el tiempo t, η es la tasa de aprendizaje, que controla cuánto se actualizan los parámetros en cada paso, y $\nabla f(\theta_t)$ es el gradiente de la función de pérdida con respecto a los parámetros en el tiempo t. La idea es mover los parámetros en la dirección opuesta al gradiente para minimizar la función de pérdida.

Descenso del gradiente estocástico

```
from tensorflow.keras.optimizers import SGD

optimizer = SGD(learning_rate=0.01)

model.compile(loss='categorical_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
```

Momentum (mejora de SGD)

Ecuaciones

$$v_{t+1} = \gamma v_t + \eta \nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

- θ_t representa los parámetros del modelo en el tiempo t.
- v_t es la "velocidad ". en el tiempo t, que acumula información sobre gradientes previos.
- γ es el factor de momentum, un valor entre 0 y 1 que controla cuánto de la velocidad anterior se mantiene en la actualización actual. Valores más altos de γ hacen que el optimizador tenga más "inercia ". y considere más la historia de los gradientes.
- η es la tasa de aprendizaje, que controla cuánto se actualizan los parámetros en cada paso.
- $\nabla f(\theta_t)$ es el gradiente de la función de pérdida con respecto a los parámetros en el tiempo t.

Momentum

```
from tensorflow.keras.optimizers import SGD

optimizer = SGD(learning_rate=0.01, momentum=0.0, nesterov=False)

model.compile(loss='categorical_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
```

La versión de Nesterov del SGD con momentum usa un término de momento mejorado para calcular los gradientes. En lugar de evaluar el gradiente en la posición actual, se evalúa en la posición que se alcanzaría aplicando el momento al gradiente anterior. Luego, se aplica el término de momento al gradiente anterior y se utiliza el gradiente calculado para actualizar los parámetros del modelo. Fue propuesta por Yurii Nesterov en 1983.

RMSprop

Ecuaciones

$$g_{t+1} = \rho g_t + (1-\rho)(\nabla f(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{g_{t+1} + \epsilon}} \nabla f(\theta_t)$$

- θ_t representa los parámetros del modelo en el tiempo t.
- g_t es una medida acumulada del cuadrado del gradiente en el tiempo t.
- β es el factor de decaimiento, un valor entre 0 y 1 que controla cuánto de la información acumulada sobre el cuadrado del gradiente anterior se mantiene en la actualización actual.
- η es la tasa de aprendizaje, que controla cuánto se actualizan los parámetros en cada paso.
- $\nabla f(\theta_t)$ es el gradiente de la función de pérdida con respecto a los parámetros en el tiempo t.
- ϵ es un pequeño valor constante para evitar la división por cero.

RMSprop

Ecuaciones

$$g_{t+1} = \rho g_t + (1-\rho)(\nabla f(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{g_{t+1} + \epsilon}} \nabla f(\theta_t)$$

En el método RMSprop, primero calculamos g_{t+1} como una combinación ponderada del cuadrado del gradiente actual y el valor acumulado anterior de g_t. Luego, actualizamos los parámetros θ dividiendo la tasa de aprendizaje por la raíz cuadrada de g_{t+1} más el término ϵ, y multiplicando esto por el gradiente actual. Al hacer esto, los parámetros se actualizan en función de la magnitud del gradiente, lo que puede ayudar a mejorar la convergencia del algoritmo en funciones de pérdida irregulares o en problemas donde los gradientes varían en diferentes direcciones.

RMSprop

```
from tensorflow.keras.optimizers import RMSprop


optimizer = RMSprop(learning_rate=0.001, rho=0.9, epsilon=1e-07)


model.compile(optimizer=optimizer, loss='categorical_crossentropy',
metrics=['accuracy'])
```

Algoritmo Adam

Ecuaciones

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla f(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla f(\theta_t))^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t
\end{aligned}
$$

1. Calcula el gradiente actual, $\nabla f(\theta_t)$.
2. Actualiza el primer momento (promedio móvil) de los gradientes
3. Actualiza el segundo momento (promedio móvil) de los cuadrados de los gradientes
4. Corrige el sesgo del primer y segundo momento
5. Actualiza los parámetros θ

Adam

```
from tensorflow.keras.optimizers import Adam


optimizer = Adam(learning_rate=0.001,
beta_1=0.9, beta_2=0.999, epsilon=1e-08)


model.compile(optimizer=optimizer, loss='categorical_crossentropy',
metrics=['accuracy'])
```

Tamaño del lote (Batch size)

- El tamaño del lote se refiere al número de ejemplos de entrenamiento utilizados en cada iteración de actualización de los parámetros.
- Un lote más pequeño proporciona una estimación más precisa del gradiente, pero requiere más recursos computacionales y puede resultar en un tiempo de entrenamiento más largo.
- Un lote más grande implica menos cálculos por iteración, lo que puede acelerar el proceso de entrenamiento, pero puede resultar en actualizaciones de parámetros más ruidosas.
- Valores comunes del tamaño del lote incluyen 32, 64, 128, 256, etc. La elección óptima depende del problema, la arquitectura de la red y los recursos computacionales disponibles.

Conjunto de entrenamiento, batch size y epochs

El número de iteraciones necesarias para procesar todo el conjunto de entrenamiento una vez se puede expresar como:

```
num_iterations_per_epoch = size_of_training_set / batch_size
```

El número total de iteraciones necesarias para completar todas las epochs se puede expresar como:

```
num_iterations_total = num_iterations_per_epoch * num_epochs
```

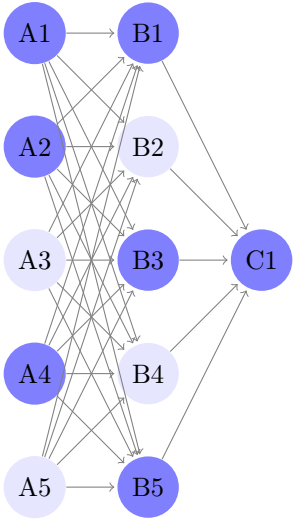
Esto muestra cómo el tamaño del conjunto de entrenamiento, el batch size y las epochs están relacionados y cómo afectan el número total de iteraciones necesarias para entrenar una red neuronal.

Universidad Tecnológica de Pereira
Facultad de Ingenierías

Parámetros de regularización

SNEIA

Técnica de Dropout



Técnica de Dropout

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten

model = Sequential([
  Flatten(input_shape=(28, 28)),
  Dense(128, activation='relu'),
  Dropout(0.5),  // Aplica dropout al 50\% de las neuronas
  Dense(10, activation='softmax')
])
```

Regularización L1 y L2 en redes neuronales

Regularización L1

Agrega un término proporcional a la suma de los valores absolutos de los pesos a la función de coste:

$$C_{L1}(\theta) = C(\theta) + \lambda \sum_i |\theta_i|$$

Induce sparsidad en los pesos, estableciendo algunos a cero.

Regularización L2

Agrega un término proporcional a la suma de los cuadrados de los pesos a la función de coste:

$$C_{L2}(\theta) = C(\theta) + \lambda \sum_i \theta_i^2$$

Reduce los pesos, pero rara vez los establece a cero.

El término de regularización se suma a la función de coste original durante el entrenamiento, penalizando los valores grandes de los pesos y ayudando a prevenir el sobreajuste y mejorar la generalización del modelo.

Regularización L1 y L2 en redes neuronales

```
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l1, l2

model = Sequential()

#Agregar una capa con L1
model.add(Dense(64, input_dim=100, activation='relu',
kernel_regularizer=l1(0.01)))

# Agregar una capa con L2
model.add(Dense(32, activation='relu', kernel_regularizer=l2(0.01)))

# Agregar una capa de salida con una neurona
model.add(Dense(1, activation='sigmoid'))

# Compilar el modelo
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Regularización Early Stopping

Early stopping es una técnica de regularización para prevenir el sobreajuste en modelos de aprendizaje automático, especialmente en redes neuronales, deteniendo el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar. Algunos aspectos clave son:

- Monitorea una métrica de rendimiento (pérdida, precisión, etc.).
- Detiene el entrenamiento si la métrica deja de mejorar durante un número específico de épocas.
- Ayuda a encontrar un equilibrio óptimo entre sesgo y varianza.
- Ahorra recursos computacionales evitando épocas innecesarias.

Implementación de Early Stopping en Python

```
from tensorflow.keras.callbacks import EarlyStopping

# Configurar el early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3, verbose=1)

# Entrenar el modelo con early stopping
history = model.fit(x_train, y_train, epochs=20,
                    validation_split=0.2,
                    callbacks=[early_stopping])
```

Algunas recomendaciones para la funciónes de pérdida y de activación

Cuadro: Ejemplos de combinaciones de problemas, funciones de activación y funciones de pérdida en redes neuronales

Tipo de problema	**Función de activación**	**Función de pérdida**
Clasificación múltiple con one-hot	Softmax	Categorical Cross-Entropy
Clasificación múltiple con índice de categoría	Softmax	Sparse Categorical Cross-Entropy
Regresión	Linear	Mean Squared Error
Clasificación binaria	Sigmoid	Binary Cross-Entropy

Parámetros de inicialización

Parámetros de inicialización

Existen varias técnicas para inicializar los pesos y sesgos (biases) de una red neuronal artificial:

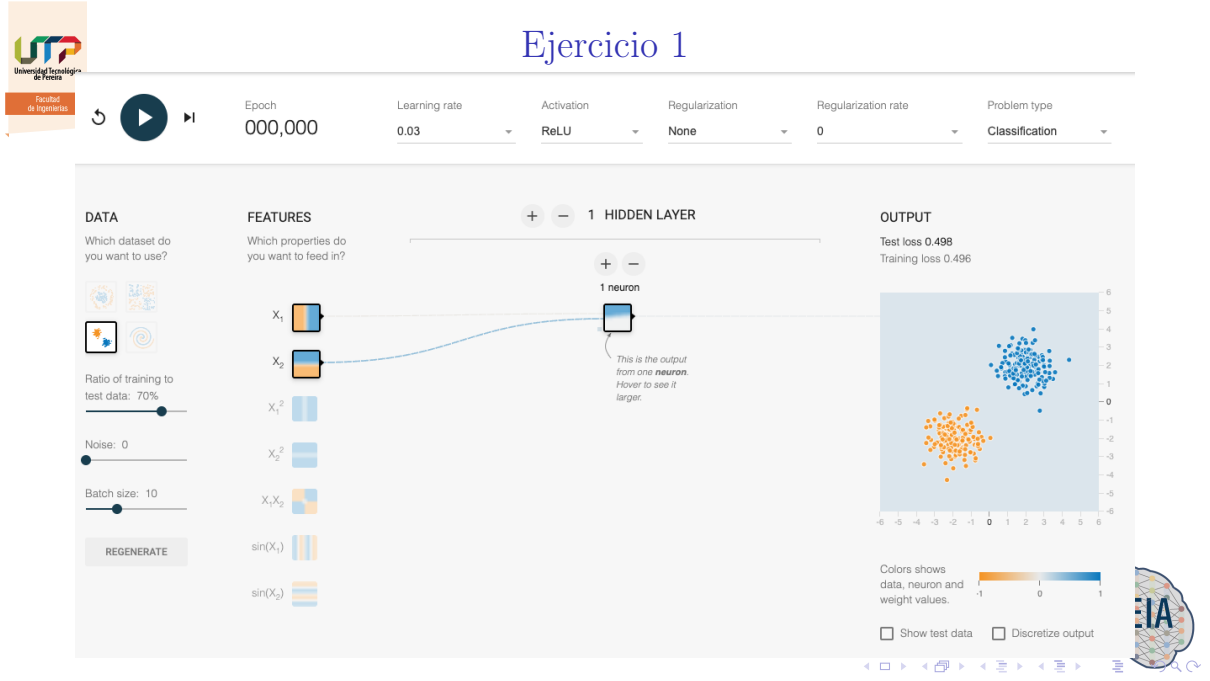
- Inicialización aleatoria: Los pesos se asignan de forma aleatoria siguiendo una distribución (normal, uniforme, etc).
- Inicialización cero: Todos los pesos empiezan en 0. Rápido pero puede causar simetría no deseada.
- Inicialización constante: Todos los pesos comparten un pequeño valor constante inicial.
- Xavier initialization: Los pesos se inicializan de acuerdo a la entrada y salida de cada capa para mantener la varianza.
- He initialization: Similar a Xavier, pero usando la raíz cuadrada del número de entradas.
- Sesgos = 0: Por lo general se inicializan los sesgos (biases) en 0.

Inicialización usando Keras

- kernel_initializer: Para inicializar los pesos en capas convolucionales y densas.
  - ‘‘glorot_uniform’’: Xavier initialization
  - ‘‘he_uniform’’: He initialization
- ‘‘random_uniform’’: Inicialización uniforme aleatoria
- bias_initializer: Para inicializar los sesgos:
  - ’zeros’: Sesgos inicializados en 0.
  - ’constant’: Sesgos con valor constante pequeño.

```
model.add(Dense(10, kernel_initializer='glorot_uniform',
bias_initializer='zeros'))
```

Ejercicio 1

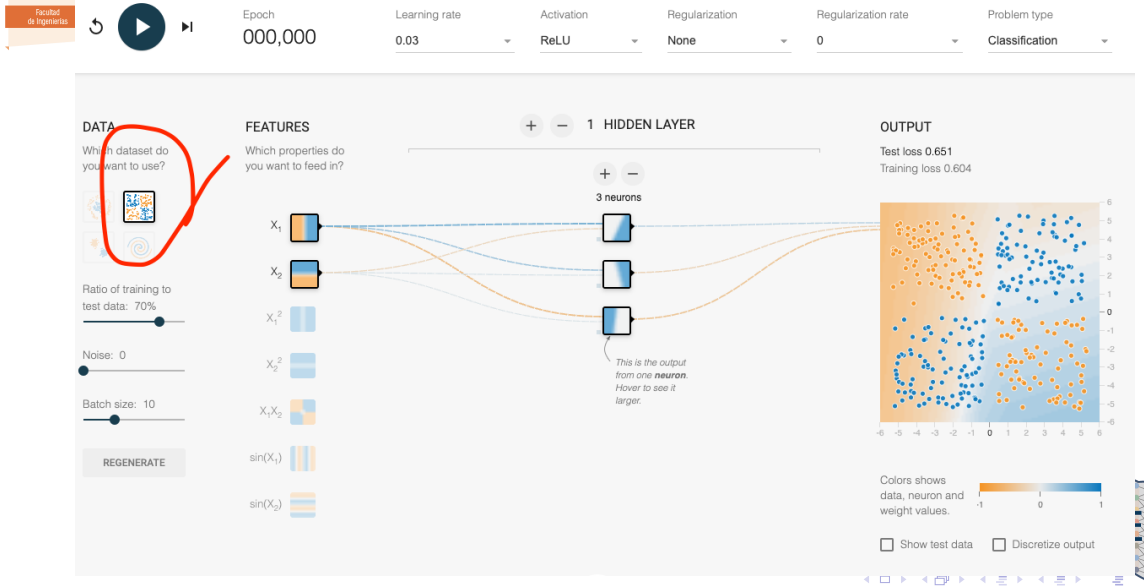

Epoch
000,000
Learning rate
0.03
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 70%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
1 neuron
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.498
Training loss 0.496
Colors shows data, neuron and weight values.
Show test data
Discretize output

Escogemos nuevo conjunto de datos

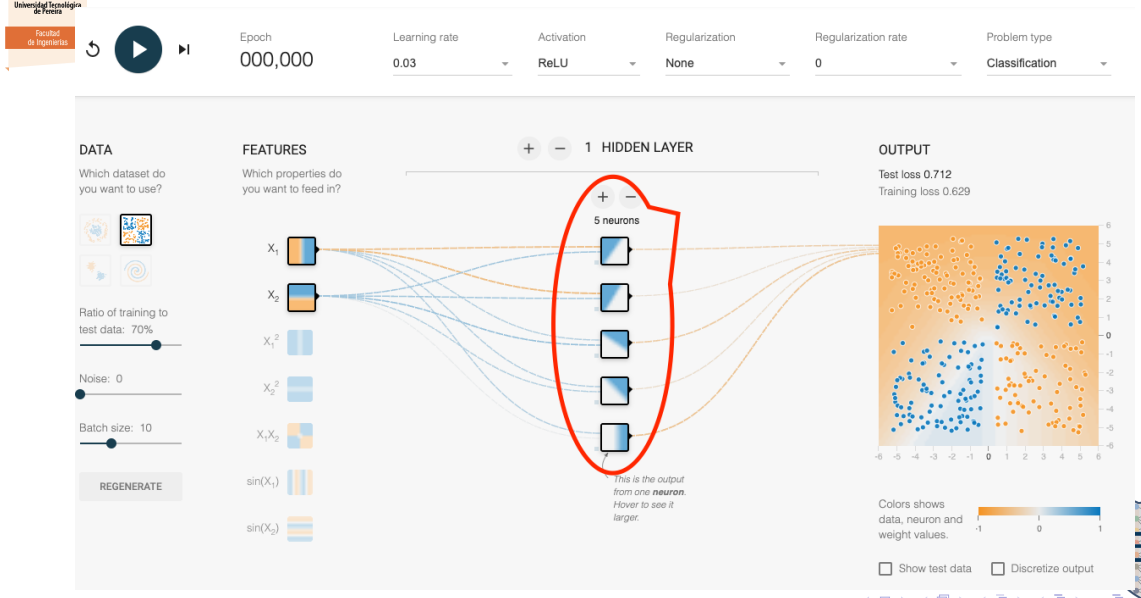
Aumentamos el número de neuronas en la capa oculta

Aumentamos el número de neuronas en la capa oculta

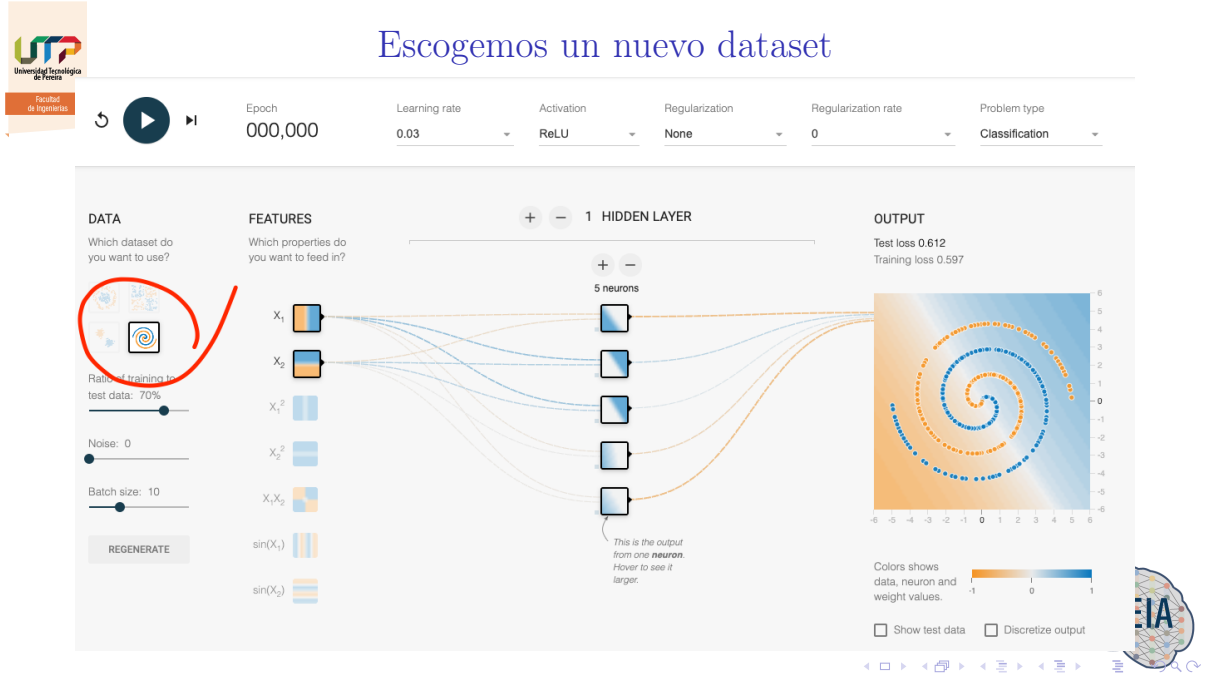
Escogemos un nuevo dataset



Aumentamos el número de neuronas en la capa oculta



Escogemos un nuevo dataset



Aumentamos el número de neuronas en la capa oculta

Creamos el modelo mas complejo para resolver el problema

Un modelo mas simple e interesante