Trabajo Práctico N° 2: Introducción a las Redes Neuronales

Aprendizaje Profundo y Redes Neuronales Artificiales

26 de septiembre de 2020

Alumnos :

Tomás LIENDRO tomas.liendro@ib.edu.ar

Docentes :

Ariel CURIALE
Germán MATO
Lucca DELLAZOPPA

San Carlos de Bariloche, Argentina

Índice

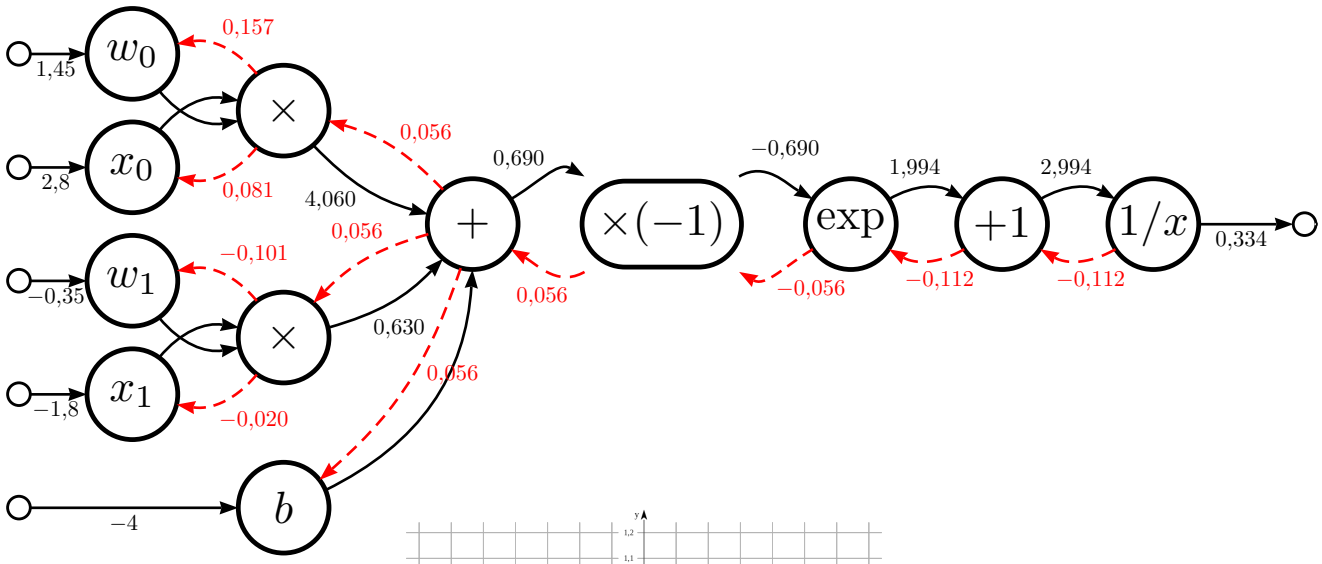
1. Ejercicio 1

En este trabajo se abordaron los conceptos de *Computational Graphs* y *Back Propagation* para entrenar una red neuronal densa. Un **grafo computacional** es una manera de representar una función matemática en el lenguaje de la teoría de grafos. En el grafo, cada nodo representa un valor de una variable o una función que combina valores. Los valores que ingresan o salen de los nodos se denominan **Tensores**.

Utilizando el concepto de **grafos computacionales** se busca implementar las funciones **Sigmoide**, **Tanh**, **ELU** y **Leaky Relu**, tomando como entrada: $x = [2{,}8, -1{,}8]$, $W = [1{,}45, -0{,}35]$, $b = -4$. En los siguientes grafos, los valores en color negro indican los resultados del proceso hacia adelante (*forward*) mientras que en rojo se colocan los resultados del *Backpropagation* recordando que el valor del gradiente a la salida del grafo es 1.

1.1. Sigmoide

$$f(x) = \frac{1}{1+e^{-x}}; \quad f'(x) = \sigma(x)(1-\sigma(x)) \tag{1.1}$$



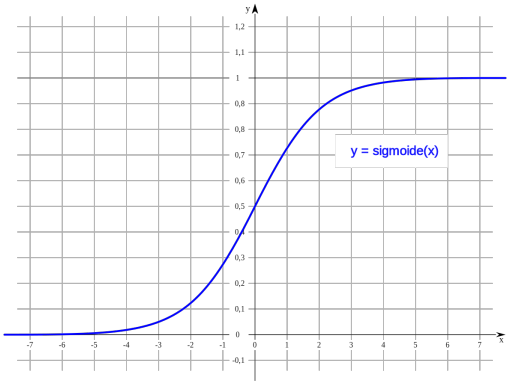


Figura 1: Función Sigmoide ($\sigma(x)$)

1.2. Tanh

$$f(x) = tanh(x) = \frac{e^{-x}-1}{e^{-x}+1}; \quad f'(x) = 1 - tanh^2(x) \tag{1.2}$$

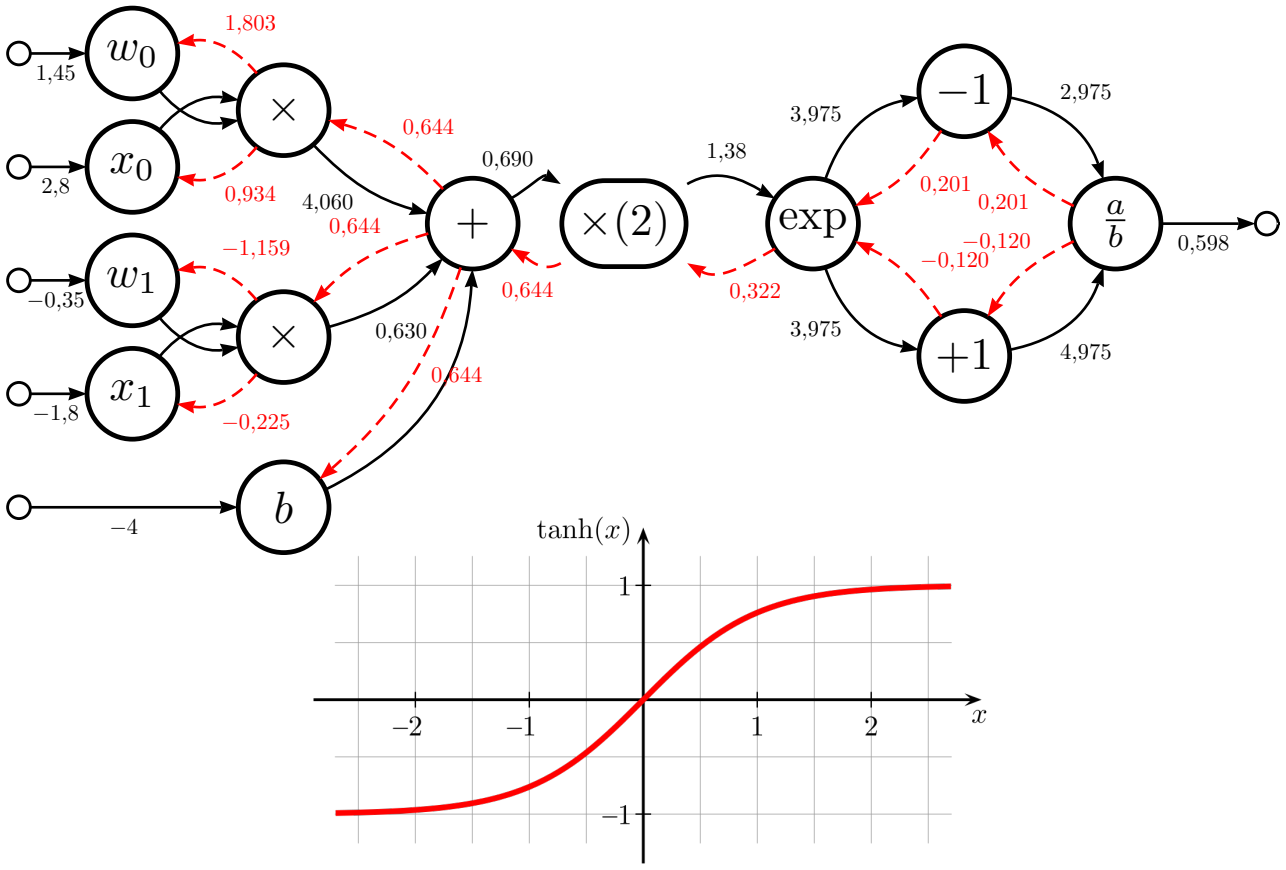
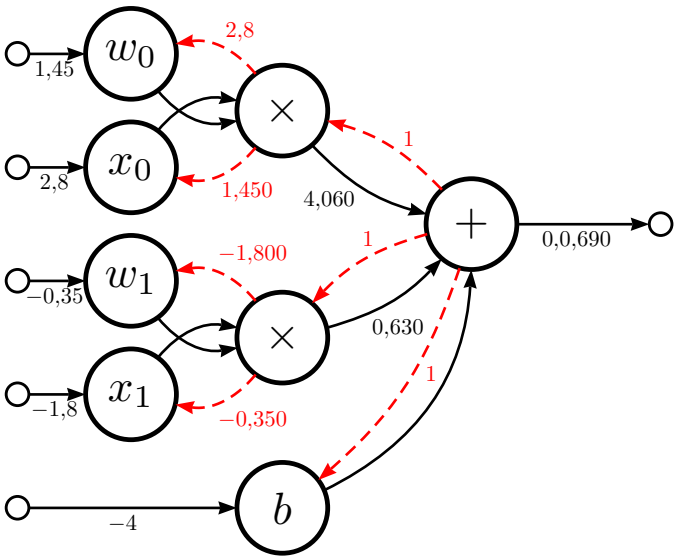


Figura 2: Función Tangente Hiperbólica ($tanh(x)$)

1.3. ELU

$$f(a,b) = \begin{cases} \text{x} & x \geq 0 \\ \alpha(\exp(x) - 1) & x < 0 \end{cases}$$

Para la primera ecuación:



Para la segunda ecuación:

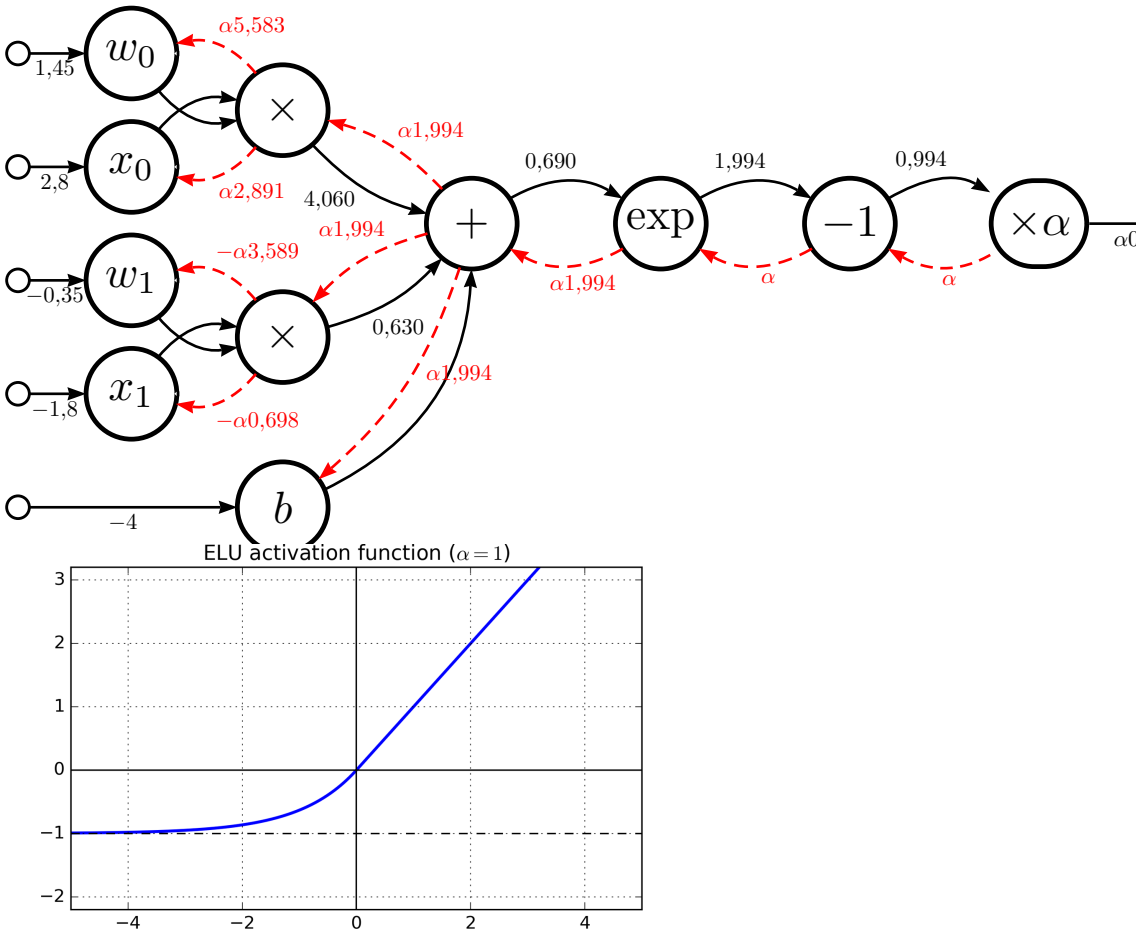
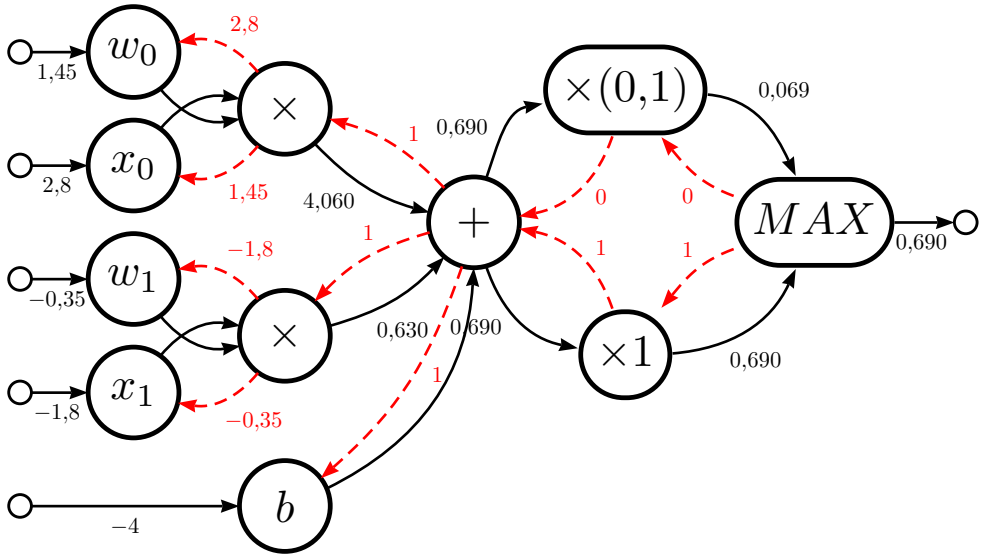


Figura 3: Función de activación ELU

1.4. Leaky Relu

$$f(x) = LeakyRelu(x) = max(0{,}1x; x) \tag{1.3}$$



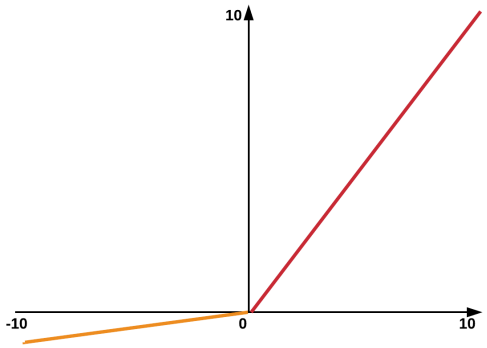


Figura 4: Función Leaky Relu

La elección de la función de activación para una capa de una red neuronal depende de los valores de salida que se requieren y de las propiedades de la función de activación utilizada. Por ejemplo, si se trabajan con probabilidades, podría pensarse en usar la función de activación **Sigmoide** dado que esta función tiene la salida acotada entre 0 y 1, si se trabaja con valores positivos se pensaría en usar la función **Relu** que anula los valores negativos y para valores generales, se pensaría en usar la función de activación **Lineal**.

En esquemas con más de una capa, uno preferiría usar la función **Leaky Relu** en lugar de la **Relu**, la cual permite ser más permisivo al no anular completamente ciertas neuronas y por el mismo motivo, se prefiere usar la función **tanh** en lugar de la **Sigmoide**.

2. Ejercicio 2

En este ejercicio se busca calcular la activación, función de costo y todos los gradientes involucrados en una red neuronal para la arquitectura de un perceptrón compuesto por dos neuronas + bias con funciones de activación **sigmoide** y la función de costo es **MSE**. La entrada a la red neuronal es $\vec{x} = [-3{,}1, 1{,}5]$, $W = [[-0{,}2, 2], [-0{,}5, -0{,}3]]$, $b_1 = -4$ y $b_2 = -1$.

La función de costo MSE se calcula como:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2, \tag{2.1}$$

$$MSE' = \frac{1}{n}2\sum(\hat{Y}_i - Y_i), \tag{2.2}$$

donde n es el número de entradas, en este caso 2, $\hat{Y}_i$ es el vector estimado y Y_i es el vector de resultados esperado.

El grafo de este problema puede graficarse como:

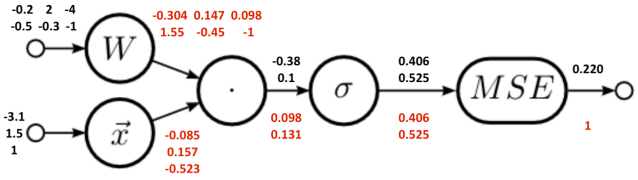


Figura 5: Grafo del ejercicio 2

donde nuevamente los valores en negro representan la evaluación hacia adelante de la red y los valores en rojo son los valores calculados durante el **backpropagation**.

3. Ejercicio 3

El ejercicio 3 solicita implementar una red neuronal (sin POO) de dos capas densas para resolver el problema de CIFAR-10. La primera capa tiene cien neuronas y tiene la función de activación sigmoidal. La segunda capa tiene diez neuronas y activación lineal. Se propone utilizar la función de costo MSE con regularización L2.

En primer lugar se arma el grafo computacional, y luego se programa el algoritmo de **backpropagation** para resolver el problema. El grafo que describe este problema es el siguiente:

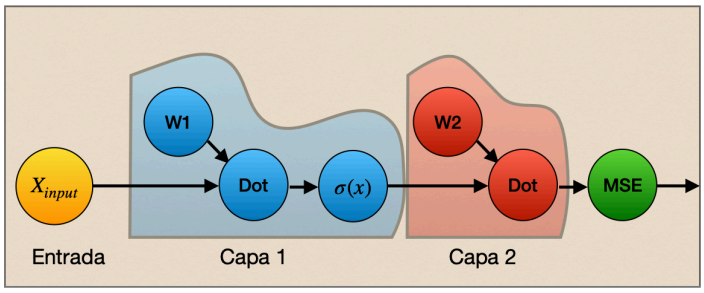


Figura 6: Grafo computacional del problema 3

El programa comienza definiendo los hiperparámetros del problema:

```
nclases = 10            # salida
nintermedia = 100       # intermedia
batch_size = 256        # Batch size
```

```
n_epocas = 100            # Cantidad de épocas
learning_rate = 1e-7      # Learning rate
reg_lambda = 1e-3         # Coeficiente de regularización
n_train_data = 5000       # Cantidad de datos que se usarán para entrenar
```

A continuación se cargan los datos de cifar y se acomodan las matrices para que puedan ser manipuladas de manera conveniente según se hizo en el Trabajo Práctico 1. Para contribuir con la estabilidad del código se resta la media de los datos de entrenamiento tanto a los datos de **testing** como de **training** y a la diferencia se la divide por la desviación estándar de los datos de training. Se definen matrices para expresar la salida de la red de manera que cada columna de la matriz de salida representa una clase del conjunto de datos, es decir para CIFAR-10, la cantidad de columnas de la matriz de salida es 10. Entonces, la información de a qué clase pertenece una imagen de CIFAR-10 (información almacenada en el vector $\mathbf{y}$) se escribiría por ejemplo: 0100000000 para representar la clase 2 y 0000000001 para representar la clase 10.

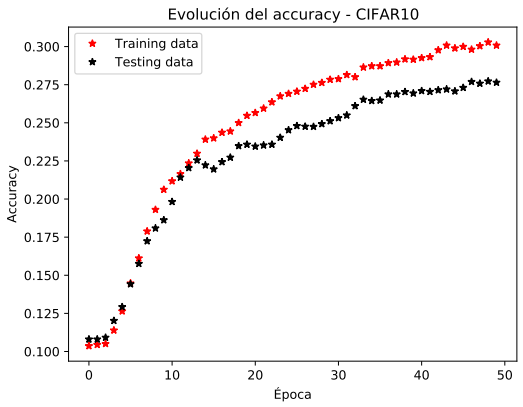
Para el entrenamiento de la red neuronal, se implementó el método de *Backpropagation* utilizando mini batches. El algoritmo de **backpropagation** se realizó siguiendo el grafo computacional anteriormente presentado y llamando a las funciones de costo, de activación y gradientes correspondientes. Además, para el cálculo de la Loss (función de costo) se aplicó la regularización L2. La función Loss se calcula haciendo:

$$Loss = \frac{1}{n} f(\hat{y}, y) + reg,$$

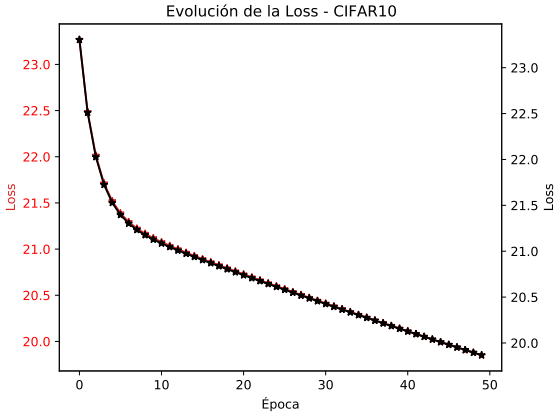
donde n es el número de mini batches, f es la función de costo, $\hat{y}$ es el vector de salida estimado, y es el vector de salida esperado y reg es la función de regularización que se calcula como:

$$reg = \sum W_1^2 + \sum W_2^2.$$

Realizando el algoritmo de *Backpropagation* y la actualización de los pesos W, se obtienen las siguientes gráficas para el Accuracy y la Loss utilizando la función de costo MSE y con la función sigmoide como función de activación en ambas capas:



(a) Accuracy para el problema de CIFAR 10 aplicando Backpropagation



(b) Loss para el problema de CIFAR 10 aplicando Backpropagation

Finalmente, se vio cómo cambia el accuracy cuando se utilizan respectivamente los métodos MSE (*Mean Squared Error*), SVM (*Support Vector Machine*)y CCE (*Categorical Cross Entropy*) como funciones de costo. En la siguiente figura se muestran los resultados.

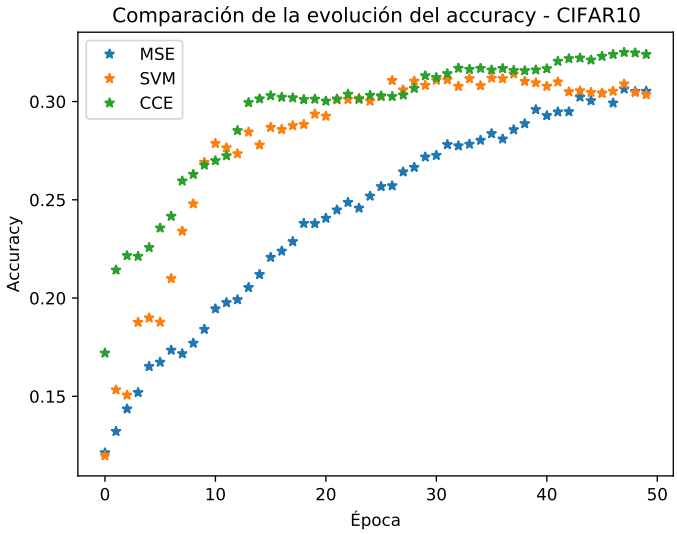


Figura 8: Comparación de las accuracy de la red para las funciones de costo MSE, SVM y CCE.

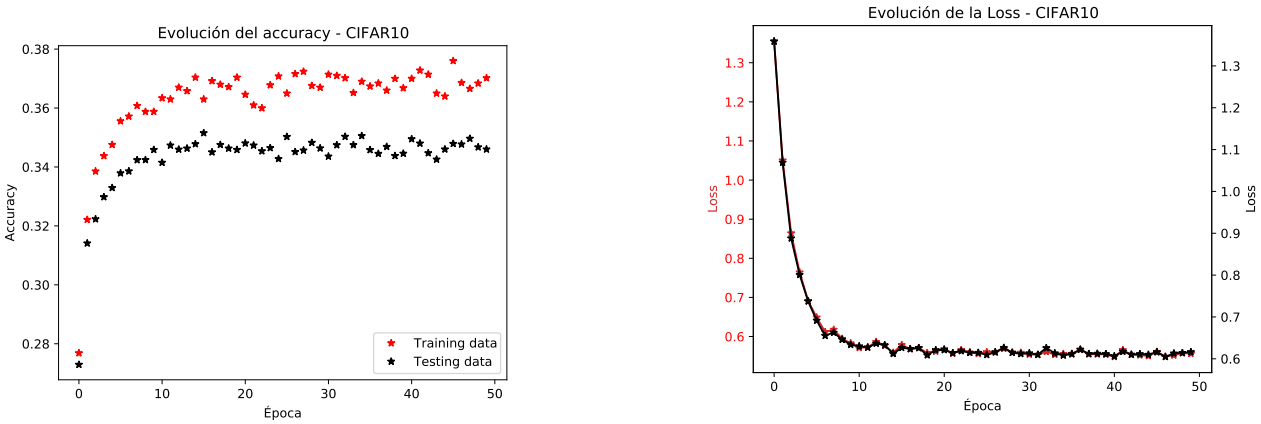
De esta figura se ve que el método que alcanza un accuracy más alto es el CCE de SoftMax con un máximo de 38 % para 50 épocas y el método que usa MSE igualó al SVM en las 50 épocas en un valor de 30 %.

4. Ejercicio 4

Este ejercicio propone realizar lo mismo que el ejercicio anterior pero en vez de usar MSE para calcular la pérdida se utilzó CCE (*Categorical Cross Entropy*).

El grafo computacional es el mismo que el obtenido para el ejercicio 3, con la salvedad que la función de costo (último bloque del grafo) es la función CCE.

El resultado obtenido para este problema es:



(a) Accuracy para el usando la función de costo CCE

(b) Loss para el usando la función de costo CCE

Figura 9: Resultados obtenidos del ejercicio 4

donde los parámetros del problema son:

```
nclases = 10  # neuronas capa de salida
nintermedia = 100  # neuronas capa intermedia
batch_size = 128  # tamaño del batch
```

```
n_epocas = 50  # cantidad de épocas para entrenar la red
learning_rate = 1e-3  # tasa de aprendizaje
reg_lambda = 1e-3  # factor de regularización
n_train_data = 5000  # cantidad de datos tomados para el entrenamiento
```

La comparación solicitada en este punto es la misma que la del inciso 3 (ver figura 8)

5. Ejercicio 5

Este ejercicio es similar al anterior y lo que se quiere ver es como evoluciona la loss y el accuracy cuando se tienen las funciones de activación RELU en la primera capa y Sigmoide o Lineal para la segunda. El grafo computacional correspondiente a este problema es el que se muestra en la siguiente figura:

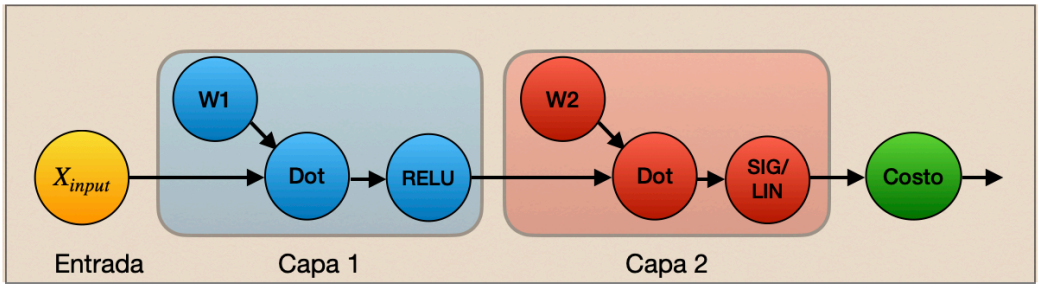
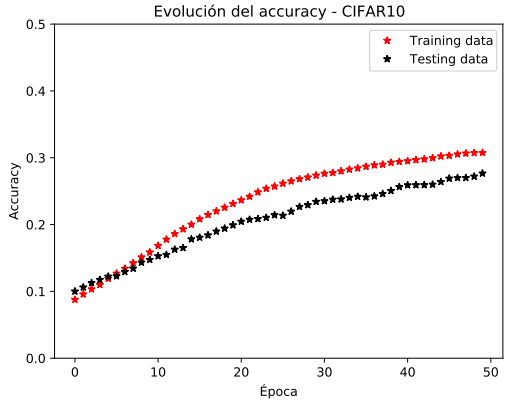


Figura 10: Grafo computacional del Ejercicio 5

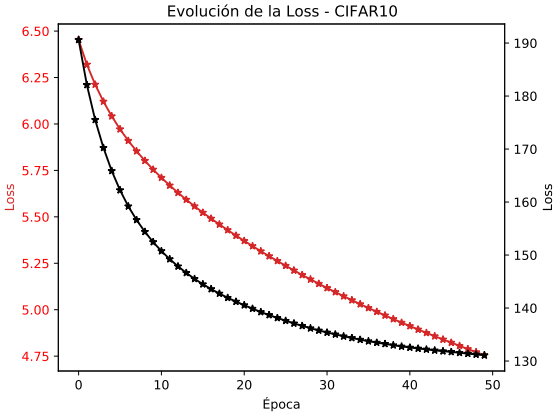
Para la primera combinación RELU+Sigmoide utilizando como hiperparámetros:

```
nclases = 10  # Cantidad de neuronas de la 2ž capa
nintermedia = 100  # Cantidad de neuronas de la 1ž capa
batch_size = 128  # Tamaño del Batch
n_epocas = 50  # Cantidad de épocas para entrenar
learning_rate = 1e-6  # Tasa de aprendizaje
reg_lambda = 0.1  # Factor de regularización
n_train_data = 10000  # Cantidad de datos utilizados para el entrenamiento de la red
```

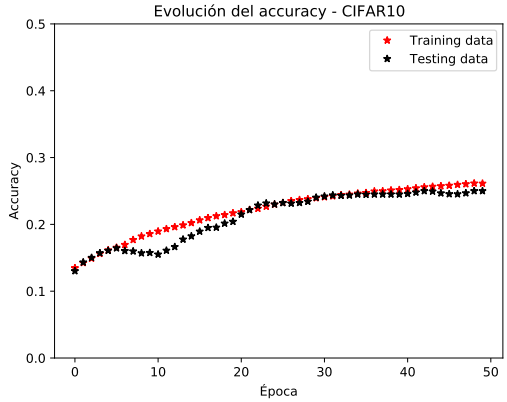
se obtuvieron las siguiente gráficas:



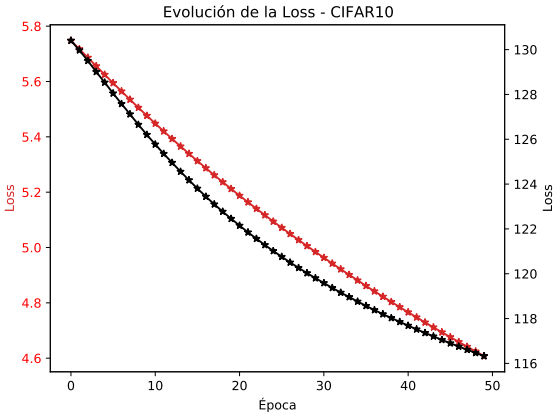
(a) Accuracy vs. Épocas para los datos de CIFAR-10 utilizando RELU+Sigmoide como funciones de activación y MSE para el cálculo de la Loss



(b) Loss vs. Épocas para los datos de CIFAR-10 utilizando RELU+Sigmoide como funciones de activación y MSE para el cálculo de la Loss



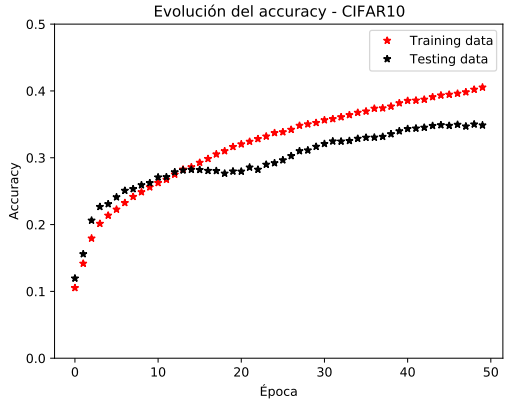
(c) Accuracy vs. Épocas para los datos de CIFAR-10 utilizando RELU+Sigmoide como funciones de activación y CCE para el cálculo de la Loss



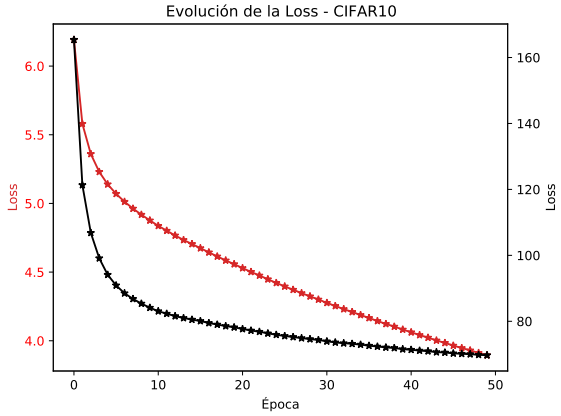
(d) Loss vs. Épocas para los datos de CIFAR-10 utilizando RELU+Sigmoide como funciones de activación y CCE para el cálculo de la Loss

De los resultados obtenidos se ve que para la combinación RELU+Sigmoide como funciones de activación se alcanzó una mejor precisión para la función de costo de MSE, con la cual el accuracy alcanzó 32 % contra un accuracy del 27 % para el CCE.

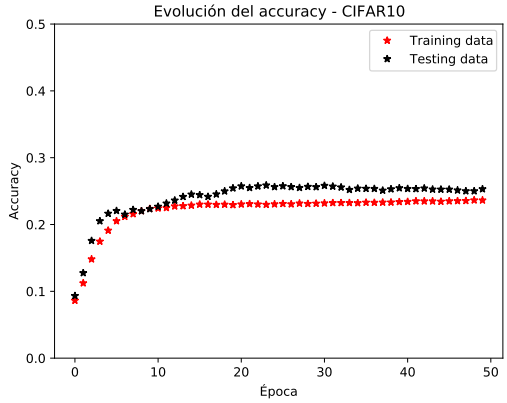
Si ahora se cambia la función de activación de la segunda capa pos la función lineal (es decir, solo se se hace el producto de las matrices de entrada de la segunda capa con la matriz W2) y para los mismos hiperparámetros se obtienen las siguientes gráficas:



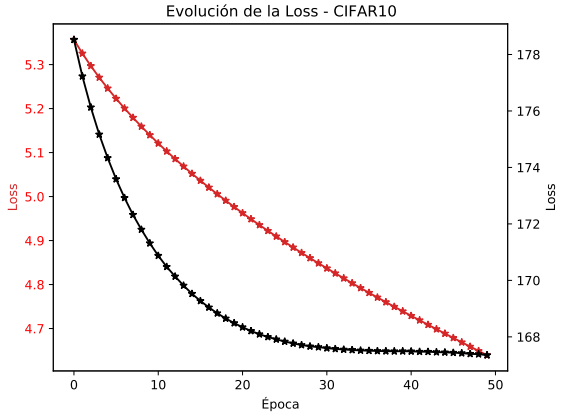
(a) Accuracy vs. Épocas para los datos de CIFAR-10 utilizando RELU+Lineal como funciones de activación y MSE para el cálculo de la Loss
- RELULIN-SMAX.pdf - RELULIN-SMAX.bb



(b) Loss vs. Épocas para los datos de CIFAR-10 utilizando RELU+Lineal como funciones de activación y MSE para el cálculo de la Loss
- SMAX - RELULIN.pdf - SMAX - RELULIN.bb



(c) Accuracy vs. Épocas para los datos de CIFAR-10 utilizando RELU+Lineal como funciones de activación y CCE para el cálculo de la Loss



(d) Loss vs. Épocas para los datos de CIFAR-10 utilizando RELU+Lineal como funciones de activación y CCE para el cálculo de la Loss

De estos resultados se ve que la precisión para ambas la función de costo MSE se alcanza un 35 % y para la función de costo CCE se logra un 25 % de precisión.

Analizando los cuatro casos anteriores, se ve que la función de costo MSE es la que consigue una mejor precisión para ambos esquemas de la red (RELU+Sigmoide y RELU+Lineal). En cuanto a si es mejor utilizar la activación Sigmoidal o Lineal para la segunda capa, se podría decir que se obtienen resultados similares para ambas funciones de activación. Sin embargo, debería esperarse que la función de activación Sgimoidal funcione mejor para este problema dado que las etiquetas de cada imagen fue reacondicionada para indicar que si la imagen pertenece a la clase por ejemplo 3, entonces la etiqueta se expresa como 0001000000, es decir, colocando un 1 en la posición correspondiente a la clase de la imagen y 0 en el resto de las clases. Esto tiene relevancia porque como la función Sigmoide toma valores entre 0 y 1, uno esperaría que sea adecuado usar esta función para esta manera de etiquetar los objetos a clasificar. Con la función lineal como activación, la salida podría tomar valores negativos, que para el criterio de clasificación propuesto no tiene sentido un valor negativo. La función lineal debería funcionar mejor para un conjunto de valores no acotados entre 0 y 1. Notar que si el conjunto de valores que puede tomar la salida es mayor que 1 y/o menor que 0, la función de activación sigmoidal tendería a saturar las salidas entre los valores límites 0 y 1.

6. Ejercicio 6

En este ejercicio la idea es entrenar una red neuronal para que aprenda la regla XOR que tiene dos entradas (1 y -1) y sigue la regla: "Si las entradas son iguales la salida es -1 y si son distintas entonces la salida es 1". Para ello se propone utilizar los conceptos de programación orientada a objetos y dos arquitecturas como las que se muestran a continuación:

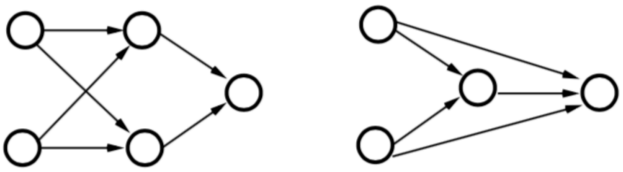


Figura 13: Arquitecturas propuestas para resolver el problema de aprendizaje de la regla XOR.

Ambas arquitecturas cuentan con una capa de entrada de datos, una capa oculta y una capa de salida. La diferencia subyace en que en la primera arquitectura todos la salida se obtiene directamente de los resultados de la capa intermedia u oculta, mientras que en la segunda arquitectura se propone que los datos de salida se obtengan utilizando tanto de los resultados de la capa intermedia como los datos de entrada.

Cada capa tiene como función de activación la función **tangente hiperbólica** y se utiliza la función MSE como función de Costo. Para resolver el problema se propone una estructura de código que no se explicará en este informe pero que puede consultarse en [**?**].

Una vez implementado el código que resuleve el problema, se utilizan los métodos **add** de la clase *Models* y **Dense** y **Concatenate** ambos de la clase *layers* para generar cada arquitectura. Notar que solo se trabajarán con capas densas en este trabajo.

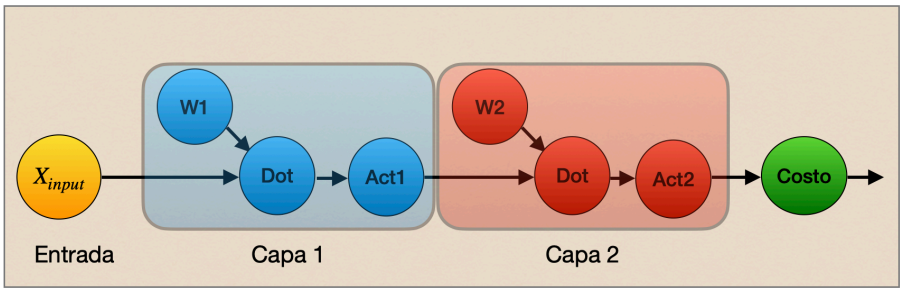


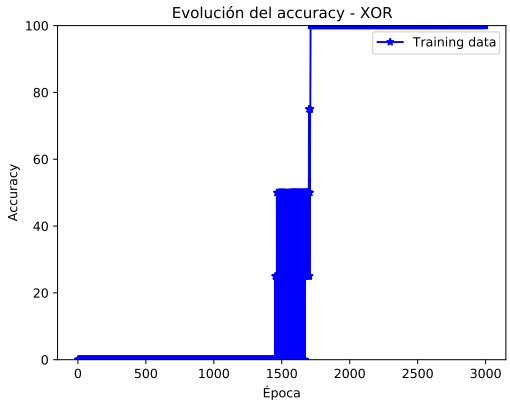
Figura 14: Grafo computacional de la primera arquitectura

Entonces, para crear la primera arquitectura se arma el grafo computacional de la Fig. **??** se realiza el siguiente algoritmo:

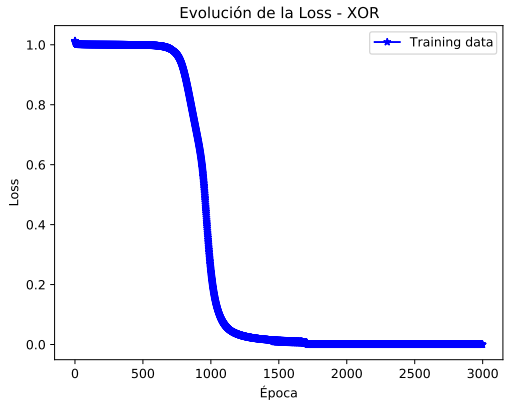
```
# Dataset: se crean los datos de entrada
x_train = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
y_train = np.array([[1], [-1], [-1], [1]])

# Create model: 1st architecture
model = models.Network() # se crea la red vacía
model.add(layers.Dense(units=2, activation=activations.Tanh(),
    input_dim=x_train.shape[1]))  # agregamos la primera capa
model.add(layers.Dense(units=1, activation=activations.Tanh()))# agregamos la segunda capa

# Train Network: entrenamiento de la red
```

```
model.fit(x_train, y_train, test_data=None, epochs=3000, loss=losses.MSE(),
    opt=optimizers.BGD(lr=0.01, bs=x_train.shape[0]), name=problem_name)
```

Con esta arquitectura y con los parámetros mencionados los resultados obtenidos para el Accuracy y la Loss son los siguientes:



(a) Accuracy para el problema de aprendizaje de la regla XOR para la primera arquitectura presentada



(b) Loss para el problema de aprendizaje de la regla XOR para la primera arquitectura presentada

Figura 15: Resultados para la primera arquitectura presentada para resolver el problema de la regla XOR con dos capas densas

Para implementar la segunda arquitectura se utilizaron las siguientes líneas de código:

```
    # Create model: 2nd architecture
model = models.Network()
# se crea la primera capa densa
model.add(layers.Dense(units=2, activation=activations.Tanh(),input_dim=x_train.shape[1]))
# se concatena la salida de la capa creada con los datos de entrada
model.add(layers.Concatenate(x_train.shape[1]))
# se crea la capa densa de salida
model.add(layers.Dense(units=1, activation=activations.Tanh()))
# Train Network
model.fit(x_train, y_train, test_data=None, epochs=3000, loss=losses.MSE(),
    opt=optimizers.BGD(lr=1e-2, bs=x_train.shape[0]), name=problem_name, reg=None)
    #regularizers.L2(lam=1e-6))
```

donde *Concatenate* genera una pesudo capa oculta que permite concatenar la salida de la primera capa oculta con la entrada del sistema. El grafo computacional por el que se puede respresentar esta arquitectura se muestra en la Fig. 16.

Los resultados del accuracy y la loss para esta segunda arquitectura resultó:

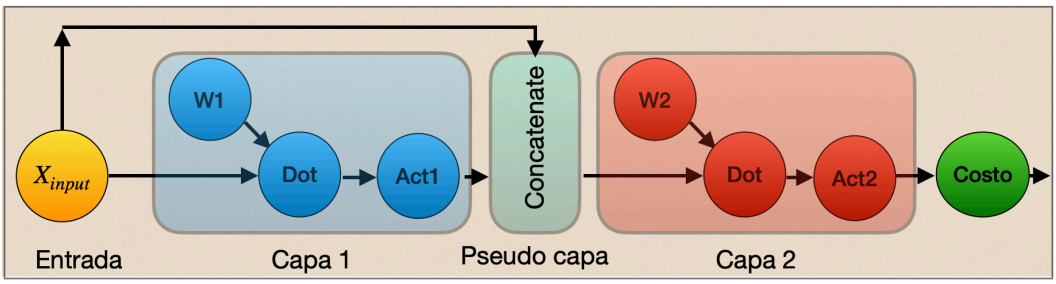
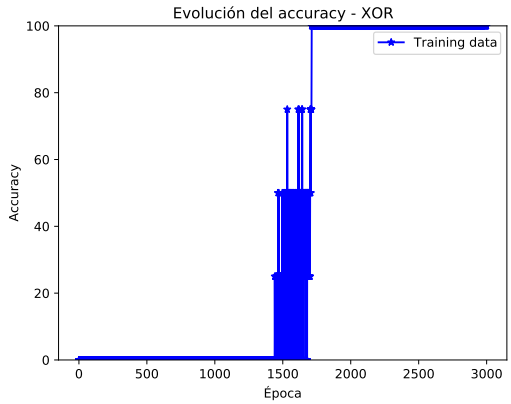
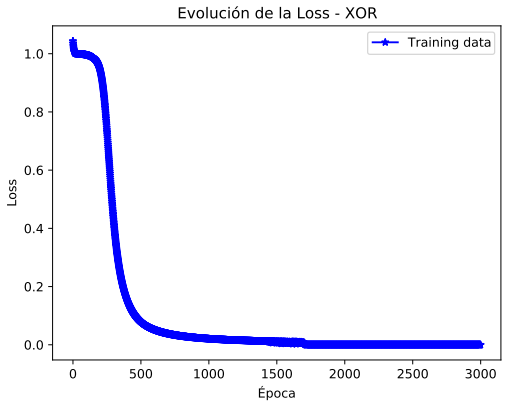


Figura 16: Grafo computacional de la segunda arquitectura



(a) Accuracy para el problema de aprendizaje de la regla XOR para la segunda arquitectura presentada



(b) Loss para el problema de aprendizaje de la regla XOR para la segunda arquitectura presentada

Figura 17: Resultados para la segunda arquitectura presentada para resolver el problema de la regla XOR con dos capas densas

7. Ejercicio 7

En este ejercicio se busca generalizar el problema anterior para N entradas, tomando como regla que si el producto de las entradas es 1 entonces la salida es 1 y si el producto de las entradas es -1 entonces la salida es -1. Se propone usar el mismo esquema de la primera arquitectura y se quiere ver como cambian los resultados frente a distintas combinaciones de cantidad de dimensión de datos de entrada N y cantidad de neuronas de la capa oculta N:

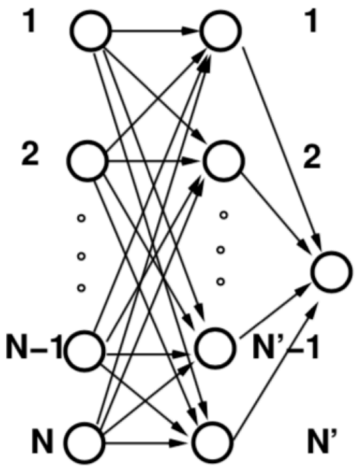
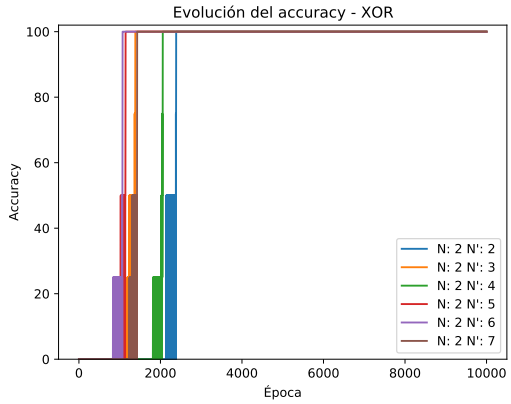
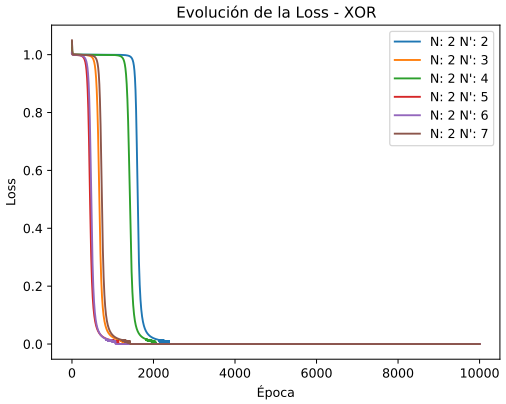


Figura 18: Problema 7: generalización de la regla XOR

Cuando se mantiene fija la dimensión de los datos de entrada y se varían las neuronas de la capa oculta, se obtienen los siguientes resultados para el accuracy y la loss:



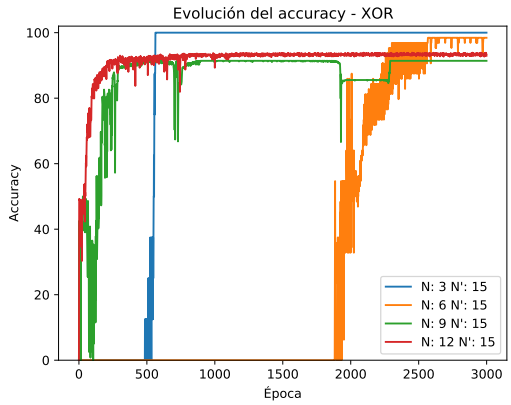
(a) Accuracy para el problema de aprendizaje de la regla XOR para dimensión de los datos de entrada de 2 variando la cantidad de neuronas de la capa oculta



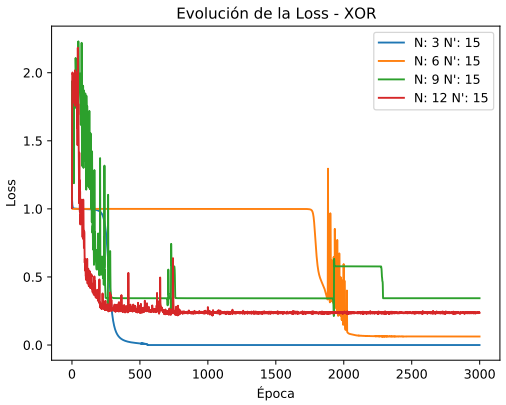
(b) Loss para el problema de aprendizaje de la regla XOR para dimensión de los datos de entrada de 2 variando la cantidad de neuronas de la capa oculta

Lo que se ve de estos resultados es que para la dimensión de los datos de entrada N fijos (en 2 en este caso) a medida que aumenta la cantidad de neuronas de la capa oculta N' es más rápido el aprendizaje de la regla XOR. De acá surge que si uno quisiera lograr un aprendizaje más acelerado, uno preferiría usar más neuronas en las capas ocultas, aunque es claro que a mayor cantidad de neuronas, mayor es el costo computacional de este tipo de problemas.

Si ahora se mantiene fijo N' (por ejemplo en 15) y se varía la dimensión de los datos de entrada, se obtienen los siguientes resultados:

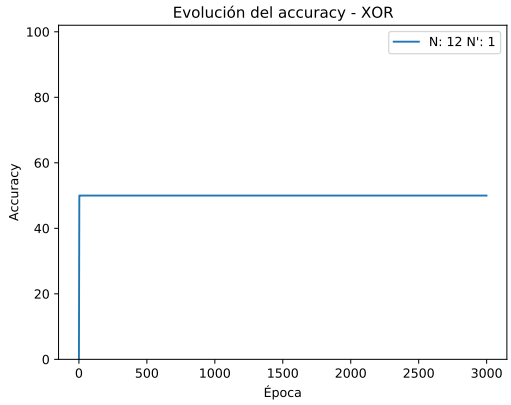


(a) Accuracy para el problema de aprendizaje de la regla XOR para dimensión de los datos de entrada de variable y con la cantidad de neuronas de la capa oculta fija en 15

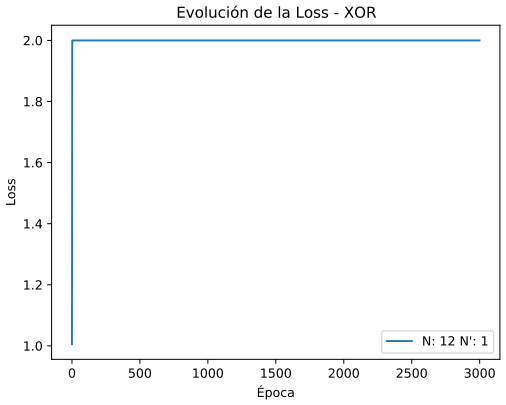


(b) Loss para el problema de aprendizaje de la regla XOR para dimensión de los datos de entrada de variable y con la cantidad de neuronas de la capa oculta fija en 15

Finalmente, para el caso en que $N' << N$ se tiene que el método no logra el aprendizaje en las 3000 épocas como se ve en la siguiente figura:



(a) Accuracy para el problema de aprendizaje de la regla XOR cuando N$\acute{}$<<N



(b) Loss para el problema de aprendizaje de la regla XOR cuando N$\acute{}$<<N

8. Ejercicio 8

En este ejercicio se busca resolver el problema del ejercicio 3 aplicando una capa intermedia de 100 neuronas con la función de activación **tanh**. El grafo computacional correspondiente a este problema es:

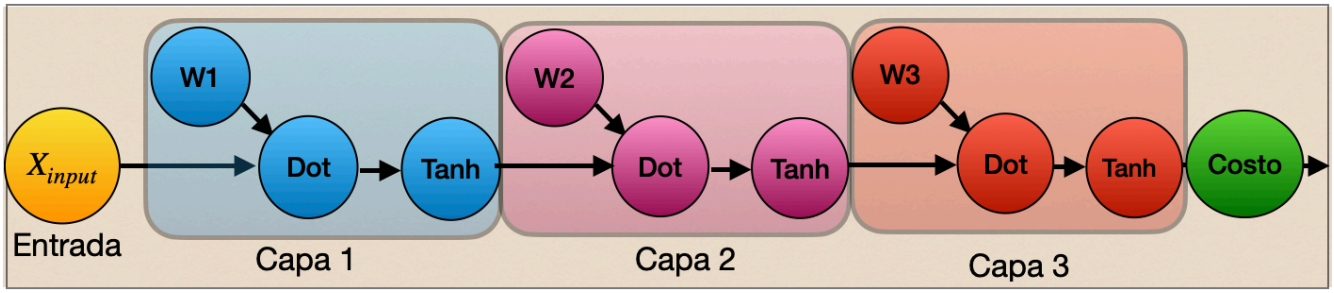


Figura 22: Grafo computacional del ejercicio 8

Para comenzar, se cargaron los datos de CIFAR-10 y se creó la estructura solicitada de dos capas con 100 neuronas en la primera capa (original del problema 3), 100 neuronas en la segunda capa (agregada en este problema) y 10 neuronas en la capa de salida. Nuevamente los datos fueron reacondicionados para expresar la clase de cada imagen como un vector que contiene un 1 en la posición de la clase correcta y el resto de los elementos 0.

Se inicializa la red neuronal con el siguiente fragmento de código:

```
# Creo la NN vacía:
model = models.Network()
# Creo la primera capa densa
model.add(layers.Dense(units=100, activation=activations.Sigmoid(),
input_dim=np.prod(x_train.shape[1:]), factor=1e-2))
# Creo la segunda capa densa:
model.add(layers.Dense(units=100, activation=activations.Sigmoid(), factor=1e-3))
# Creo la tercera capa densa:
model.add(layers.Dense(units=10, activation=activations.Linear(), factor=1e-2))
# se entrena la NN:
model.fit(x_train, y_train, test_data=test_data, epochs=100, loss=losses.MSE(),
opt=optimizers.BGD(lr=1e-4, bs=64), name=problem_name, reg=regularizers.L2(lam=1e-3))
```

De acá se ve que se le pasan todos los hiperparámetros del problema al momento del entrenamiento de la red. Se compararon los casos en que la función de costo es MSE, SVM y CCE y se obtuvo la siguiente gráfica:

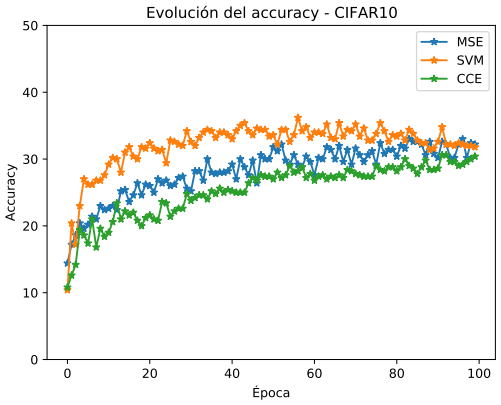
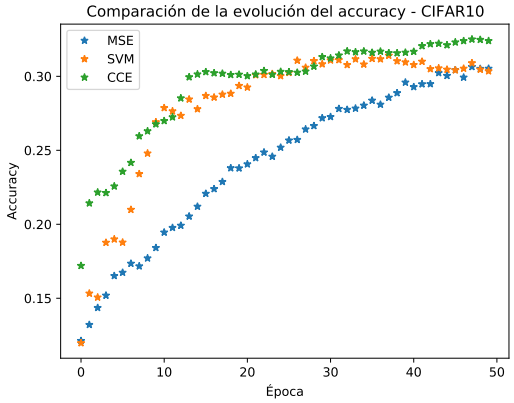


Figura 23: Accuracy vs. Época del ejercicio 8. Se muestran los resultados de haber usado las funciones de costo MSE, SVM y CCE.

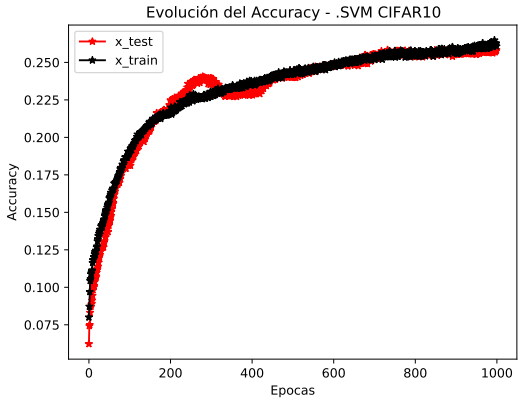
Las figuras de abajo muestran los resultados obtenidos para el problema 3 de este trabajo práctico:



(a) Accuracy para el problema 3 del TP2 usando las funciones de costo MSE, SVM y CCE

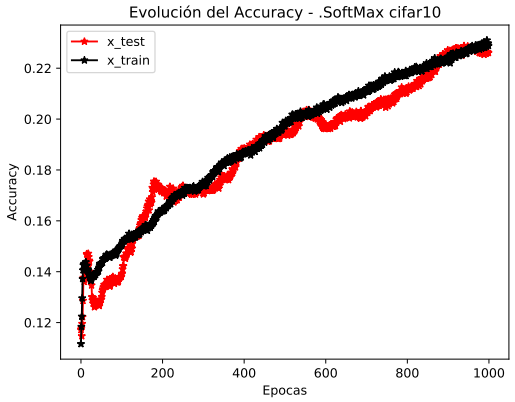
y los resultados del ejercicio 5 del TP1:

CIFAR10.pdf CIFAR10.bb



(a) Accuracy para el problema 5 del TP1 usando la función de costo SVM

cifar10.pdf cifar10.bb



(b) Loss para el problema 5 del TP1 usando la función de costo CCE

Figura 25: Resultados obtenidos del ejercicio 5 del TP1 donde no se utilizaban capas ocultas intermedias

Para el problema 8 se usaron 2 capas ocultas de 100 neuronas y una capa de salida de 10 neuronas, para el problema 3 de este trabajo práctico se usó una arquitectura de una capa oculta densa de 100 neuronas y una capa de salida de 10 neuronas y para el problema 5 del TP1 no se usó ninguna capa oculta y la salida tiene una sola neurona. De comparar los cuatro gráficos, el resultado más llamativo es la cantidad de épocas necesarias para entrenar las redes. Para el ejercicio 8 y 3 se ve que con 50 épocas se alcanza un 'estacionario' mientras que para la arquitectura usada en el problema 5 del TP1 se requieren al menos 500 épocas de entrenamiento para lograr el 'estacionario'. En cuanto a las precisiones alcanzadas, en el ejercicio 8 se alcanzó una precisión máxima de 33 % para MSE, SVM y CCE, para la arquitectura de solo 2 capas del problema 3 se alcanzaron precisiones de 33 % para el CCE y de 30 % para SVM y MSE. Finalmente para la arquitectura de solo una capa de salida con una neurona (problema 5 del TP1) se alcanzaron despues de 1000 épocas, precisiones del orden de 25 % para SVM y de 23 % para CCE.

En todos los casos se utilizaron 5000 datos para el entrenamiento y los hiper parámetros se ajustaron un poco tratando de que sean lo más parecidos posibles para esta comparación.

En conclusión, se pudo observar el efecto 'acelerador' que tiene la incorporación de capas ocultas con una gran cantidad de neuronas frente a arquitecturas más sencillas como las del problema 5 del TP1. Además de

requerir menos épocas y menos tiempo de ejecución, también se logra aumentar la precisión alcanzada por los métodos.

Referencias

[1] Julien Jacques, Linear Regression in High Dimension and/or for Correlated Inputs. Article in EAS Publications Series ůi January 2015. https://www.researchgate.net/publication/276249836

[2] Ariel Curiale, German Matos, *Notas de la cátedra: Aprendizaje Profundo y Redes Neuronales Artificiales*. https://classroom.google.com/u/2/c/MTQxNDcxMTM5NjEx

[3] Fei-Fei Li, *Convolutional Neural Networks for Visual Recognition*. http://cs231n.stanford.edu/2017/

[4] Códigos del TP1: https://github.com/TomasLiendro/Deep_Learning/blob/master/TP1_Introduction%20to%20ML/

[5] Códigos del TP2: https://github.com/TomasLiendro/Deep_Learning/tree/master/TP2_Introduction%20to%20Neural%20Networks