Proyecto № 1: Redes convolucionales

Valerie Parra Cortés 20 de marzo de 2020

1. Recolección de datos

Para recolectar los datos se usaron vídeos de internet de gente que ya hubiera pasado todos los mundos de ***Mario Bros***, para esto había que asegurarnos que los vídeos tuvieran el mismo tamaño, en este caso se seleccionó que tuvieran una resolución de 720 x 1280. Sumado a esto, se tuvo en cuenta que los vídeos seleccionados no tuvieran ningún borde (o *padding*) negro, es decir que la imagen que ocupara la pantalla completa. Finalmente, los vídeos utilizados fueron los siguientes:

1. Super Mario Bros. - Full Game Walkthrough
2. Super Mario Bros. (1985) Full Walkthrough NES Gameplay
3. NES Super Mario Bros - Trucos Secretos

2. Preprocesamiento

2.1. Dependencias

Para esta parte del proyecto se utilizaron las siguientes librerías de *Python*:

1. Open-cv Versión: 4.2.0.32
2. Numpy Versión: 1.18.1

2.2. Tratamiento de los datos

Una vez se descargaron estos vídeos, estos se dividieron en *frames*, es decir en las imagenes que los componen. Para esta tarea se adaptó el código del siguiente repositorio de github. La implementación fue como sigue a continuación.

Listing 1: Ejemplo de código de python para dividir los videos en frames.

```
1 import cv2
2 import numpy as np
3 import os
4
5 cap = cv2.VideoCapture('Super Mario Bros. (1985) Full Walkthrough NES ↩
    Gameplay [Nostalgia].mp4')
```

```
6
7  try:
8      if not os.path.exists('data/video2'):
9          os.makedirs('data/video2')
10 except OSError:
11     print ('Error: Creating directory of data')
12
13 currentFrame = 0
14 while(True):
15     ret, frame = cap.read()
16     name = './data/video2/frame' + str(currentFrame) + '.jpg'
17     print ('Creating...' + name)
18     cv2.imwrite(name, frame)
19     currentFrame += 1
20
21 cap.release()
22 cv2.destroyAllWindows()
```

Una vez dividimos los videos en las imágenes que los componen, notamos que había dos tipos de imágenes que no nos servían. Las primeras aparecían en la transición entre niveles, aquí una pantalla negra con información en letra blanca del mundo, las vidas y el resumen de las monedas recogidas aparecía. Las imágenes asociadas a estas transiciones entre niveles se eliminaron. El segundo tipo de imágen que no era útil para nuestro problema, aparecía dentro del mismo nivel, cuando había transiciones y se pasaba de día a noche o de tierra a mar, aparecían imágenes ya fueran o completamente negras o completamente azules, del color del cielo o del mar del nivel, estos frames también se eliminaron. Todo este proceso se hizo de forma manual.

3. Selección de problemas

Mario Bros consiste de ocho mundos diferentes y cada uno tiene cuatro niveles. Para este proyecto se tenían que encontrar dos tipos de mundos para dos problemas de clasificación. Uno que *a priori* consideráramos fácil de clasificar y otro que consideráramos difícil.

3.1. Problema fácil

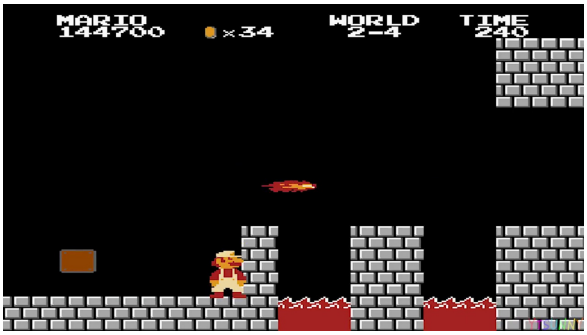
Para el problema fácil se seleccionaron los mundos dos y seis. El mundo seis tiene todos los fondos de sus niveles en fondo negro, mientras que el mundo dos tiene tres niveles en fondo azul y uno en fondo negro que corresponde al nivel cuarto. Estos mundos pueden verse se pueden ver en las Figuras 1 y 2 respectivamente. Por otro lado, vemos que el nivel 2 del mundo dos (Figura 1b) al ser el primer nivel acuático del mundo de Mario Bros tiene un color de fondo totalmente diferente al resto de los mundos, característica que nos puede ser útil a la hora de buscar características para configurar la Red Neuronal Convolucional (CNN) que se

implementará en el problema de clasificación.

(a) Nivel 1

(b) Nivel 2

(c) Nivel 3

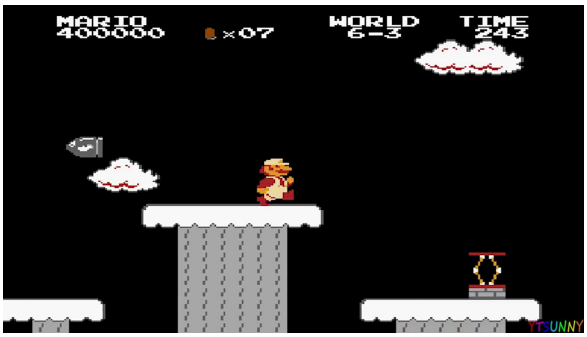


(d) Nivel 4

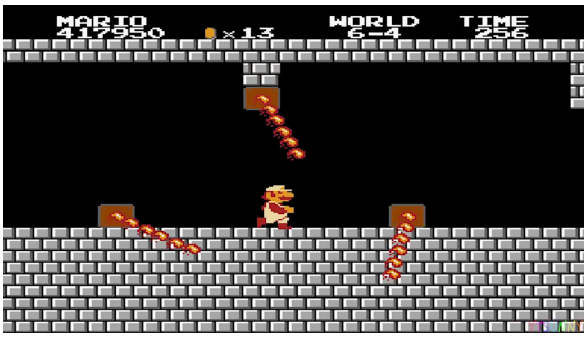
Figura 1: Diferentes niveles del mundo 2

(a) Nivel 1

(b) Nivel 2



(c) Nivel 3

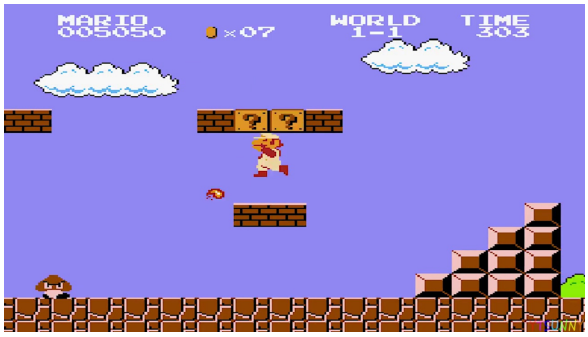


(d) Nivel 4

Figura 2: Diferentes niveles del mundo 6

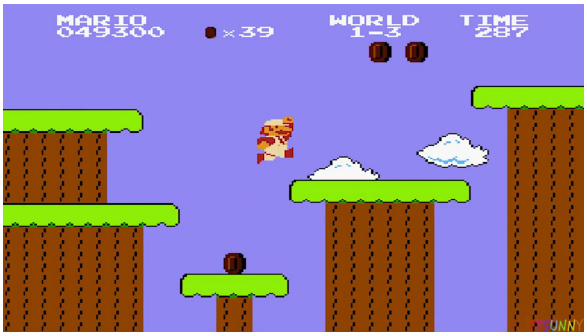
3.2. Problema difícil

Para el problema difícil se seleccionaron los mundos uno y cuatro. Estos dos mundos son tan parecidos que si la imagen careciera de la etiqueta Mundo-Nivel, que está en la parte superior de las imágenes, incluso para un ser humano sería muy difícil clasificarlos (claro, esto suponiendo que el humano no tiene un profundo conocimiento de ***Mario Bros***). La principal diferencia entre estos números en la secuencia de pasos, los enemigos (hongos para el mundo uno y tortugas y nubes en el mundo cuatro). Aún así, hay imágenes del mundo cuatro donde no aparecen enemigos que podrían ser fácilmente confundidas con una imagen del primer mundo. La similitud entre los niveles de estos mundos puede apreciarse en las Figuras 3 y 4.
Como aquí las diferencias son más sutiles nos inclinamos a pensar que una CNN entrenada para detectar paisajes podría funcionar para clasificar imágenes provenientes de este *set* de datos.

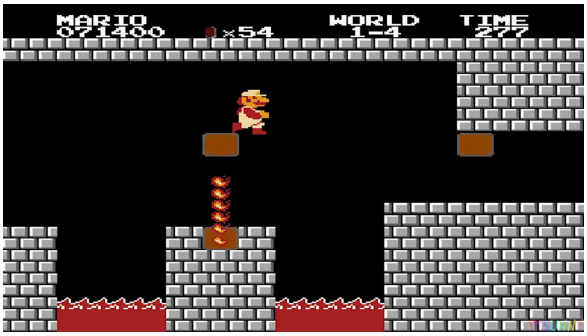


(a) Nivel 1

(b) Nivel 2

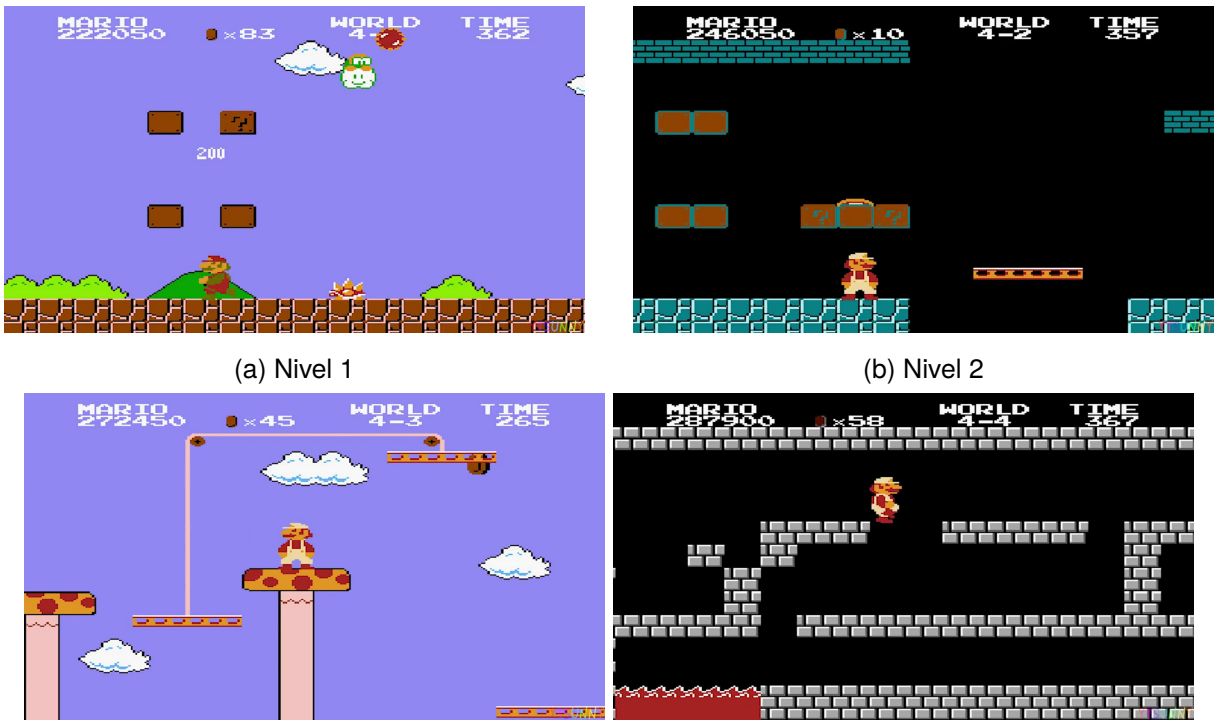


(c) Nivel 3



(d) Nivel 4

Figura 3: Imágenes de diferentes niveles del mundo 1



(a) Nivel 1

(b) Nivel 2

(c) Nivel 3

(d) Nivel 4

Figura 4: Diferentes niveles del mundo 4

4. División del conjunto de datos

4.1. Organización y recuento del *set* de datos completo

4.1.1. *Set* de prueba

Una vez seleccionados los problemas a resolver organizamos las imágenes que ya habíamos extraído de los vídeos, las agrupamos y totalizamos tanto por clase como por problema de clasificación. En total para el problema fácil tenemos un total de 30.731 imágenes mientras que para el problema difícil tenemos 25.663 imágenes. Las cantidades completas por clase de cada uno de estos problemas pueden verse en el Cuadro 1.

Mundo	Número de imágenes	Número total de imágenes
1	13.006	27.863
4	14.257	
2	16.140	30.731
6	15.191	

Cuadro 1: Número total de imágenes por clase

4.2. División entrenamiento-test

4.3. Dependencias

Las librerías de *Python* necesarias para esta etapa son:

1. *Sklearn* Versión 0.22.2
2. Os. Este modulo viene instalado con *Python* por defecto.

4.4. Implementación

Una vez se tenían estos datos, se dividió 80 % para entrenamiento, 10 % para validación y 10 % para prueba en cada una de las clases que componía cada problema (el fácil y el difícil). Para dividir esto aleatoriamente se utilizó la librería *sklearn*, se le pasaron clases *dummy*, se tomó los nombres de archivos que la división dejaba en test y esto se pasó a otra carpeta. El código con el que se hizo esto se muestra a continuación:

Listing 2: Código utilizado para separar el conjunto de entrenamiento del conjunto de validación.

```
from sklearn.model_selection import train_test_split
import os
dummyLabels=[0 for x in range(len(os.listdir('./data/video1/W6')))]
X_train, X_test, y_train, y_test = train_test_split(os.listdir('./data/video1/W6'), dummyLabels, test_size=0.1)

for i in range(len(X_test)):
    os.rename('./data/video1/W6/'+X_test[i],'./data/video1/W6_test/'+X_test[i])
```

Con lo que finalmente el conjunto quedo dividido como se muestra en el Cuadro 2

Mundo	Total	Entrenamiento	Validación	Prueba
1	13.006	10.969	1.219	818
4	14.257	12.127	1.348	782
2	16.140	13.604	1.512	1024
6	15.191	12.613	1.402	1176

Cuadro 2: División final de los datos de entrenamiento y validación

5. Implementación de las CNN

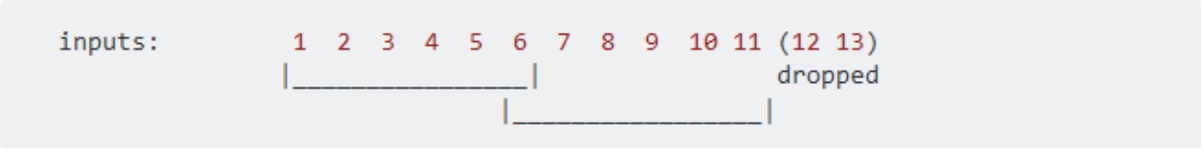
5.1. Dependencias

Las librerías de Python utilizadas para la implementación de la red neuronal son

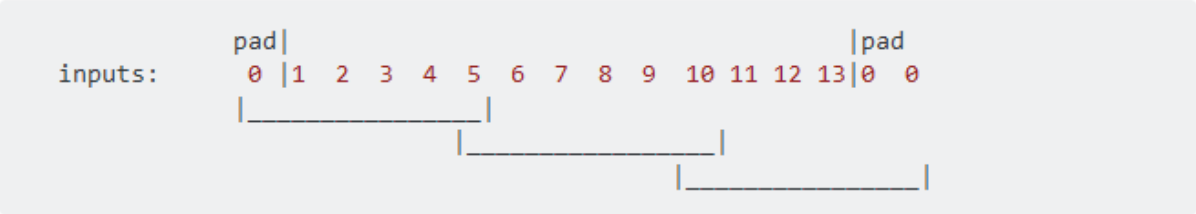
1. Numpy Versión: 1.18.1
2. *Matplotlib* (Esta es una dependencia de desarrollo, sólo se usa en caso de que se quieren visualizar los datos subidos a la red, no es necesario para correr la red neuronal)
3. *Keras* Versión 2.3.1

5.2. Solución el problema fácil

Al ver los mundos dos y seis vemos que la primera gran diferencia que se puede observar es el contraste de los colores. Con base a esta observación buscamos una red convolucional que ya fuera capaz de extraer características sobre los colores de la red. Para esto se quería implementar un modelo parecido a uno ya existente que estaba entrenado para clasificar carros de acuerdo a su color. Sin embargo este modelo era demasiado complejo y no se tenía el poder computacional para correrlo. Por lo que se trató de implementar una versión simplificada de esta red, eliminando la normalización de las salidas de las redes pero manteniendo la estructura: red convolucional seguida de una capa de *Max Pooling* donde los filtros en cada red convolucional va en aumento, se eliminaron las capas de normalización pero se mantuvo el número de filtros, los tamaño de los *pool* y del kernel. Adicionalmente todas las capas tienen un *Padding* tipo *same* igual que en el problema original (en *Keras* existen dos tipos de *padding* uno tipo *same* y otro tipo *valid o sin padding*. El primero añade ceros para completar los bits del *kernel* mientras que el segundo elimina los bits que sobran. La Figura 5 ejemplifica estos dos tipos de *padding*).



(a) *Padding* tipo *valid*



(b) *Padding* tipo *same*

Figura 5: Visualización simple de los dos tipos de *padding* que maneja *Keras* para una entrada de ancho 13, un filtro de ancho 6 y un salto de 5. Imagen tomada y modificada con fines ilustrativos de StackOverFlow

5.2.1. Arquitectura de la red implementada

La parte convolucional se mantuvo y de aquí se propusieron dos arquitecturas diferentes que compartían la red convolucional pero diferían en la red neuronal final. Se quería evaluar el

comportamiento de las arquitecturas teniendo uno o dos capas escondidas. Las arquitecturas usadas se explican a continuación:

Capas convolucionales

1. **Capa convolucional** La salida de esta capa es 720x1280x48, con un total de 192 parámetros entrenables, 48 filtros, un tamaño de *kernel* 1x1, una función de activación tipo *ReLu*

2. **Capa de Max Pooling** La salida de la cada 360x640x48 de un tamaño de *pool* de 2x2. El salto (*strides*) es de 2x2.

3. **Capa convolucional** La salida de esta capa es 360x640x64, con un total de 27.712 parámetros entrenables, 64 filtros, un tamaño de *kernel* 3x3, una función de activación tipo *ReLu* y un *padding* tipo *same*.

4. **Capa de Max Pooling**. La salida de la cada 179x319x64 de un tamaño de *pool* de 3x3. El salto o *strides* es de 2x2.

5. **Capa convolucional** La salida de esta capa es 170x319x64, con un total de 36.928 parámetros entrenables, 64 filtros, un tamaño de *kernel* 3x3, una función de activación tipo *ReLu* y un *padding* tipo *same*.

6. **Capa de Max Pooling** La salida de la cada 89x159x64 de un tamaño de *pool* de 3x3. El salto o *strides* es de 2x2.

7. **Capa convolucional** La salida de esta capa es 89x159x192, con un total de 110.784 parámetros entrenables, 192 filtros, un tamaño de Kernel 3x3, saltos de o *strides* de 1x1 y una función de activación tipo *ReLu* y un *padding* tipo *same*.

8. **Capa de Max Pooling**. La salida de la cada 44x79x192 de un tamaño de *pool* de 3x3.

9. **Capa convolucional** La salida de esta capa es 44x79x164, con un total de 110.784 parámtros entrenables, 64 filtros, un tamaño de Kernel 3x3, saltos de a 1x1 pixel y una función de activación tipo *ReLu* y un *padding* tipo *same*.

10. **Capa de Max Pooling** La salida de la capa es de 21x39x64 de un tamaño de *pool* de 3x3 El salto o *strides* de 2x2..

11. **Capa convolucional** La salida de esta capa es 21x39x64, con un total de 36.9284 parámtros entrenables, 64 filtros, un tamaño de *kernel* 3x3, saltos de a 1x1 pixel y una función de activación tipo *ReLu* y un *padding* tipo *same*.

12. **Capa de Max Pooling** La salida de la capa 10x19x64 de un tamaño de *pool* de 3x3. El salto o *strides* de 2x2.

13. **Capa convolucional** La salida de esta capa es 10x19x28, con un total de 36.9284 parámtros entrenables, 64 filtros, un tamaño de *kernel* 3x3, saltos de a 1x1, una función de activación tipo *ReLu* y un *padding* tipo *same*.

14. **Capa de Max Pooling** La salida de la capa 4x19x28 de un tamaño de *pool* de 3x3. El salto o *strides* es de 2x2.

Al final de esta red se obtienen en total de 1008 *features* y esto es la entrada de la red neuronal. Para la red neuronal se probaron dos arquitecturas:

Arquitectura 1

1. Capa completamente conectada de 59 neuronas, función de activación tipo *ReLu*. La cantidad de parámetros entrenables son 59.531.
2. Esta capa cuenta con las mismas neuronas de la capa anterior, con la misma función de activación. Tiene un total de 3.540 parámetros.
3. Capa de sálida de 2 neuronas, usa *Softmax* como función de activación y en total tiene 120 parámetros entrenables

Total de parámetros entrenables para esta arquitectura: 402.547,

Arquitectura 2

1. Capa completamente conectada de 59 neuronas, función de activación tipo *ReLu*. La cantidad de parámetros entrenables son 151.350.
2. Capa de salida de 2 neuronas, usa *Softmax* como función de activación y en total tiene 302 parámetros entrenables

Total de parámetros entrenables para esta arquitectura: 491,008

5.2.2. Hiperparámetros de entrenamiento

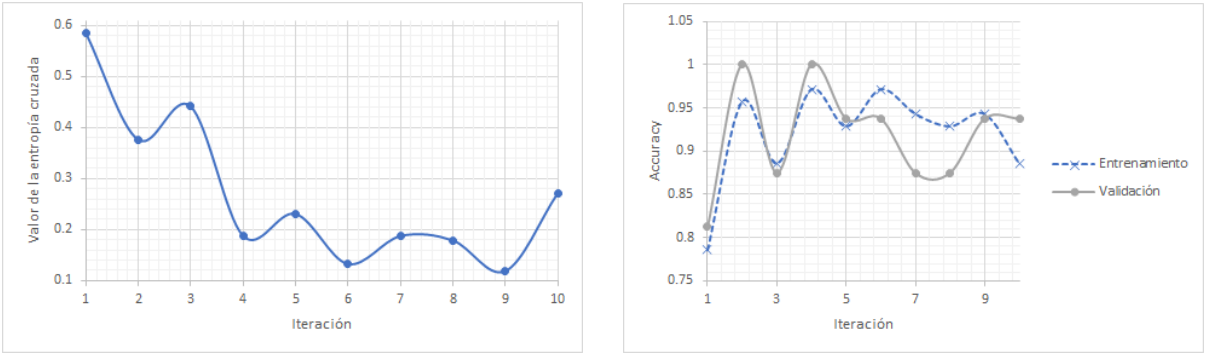
Lo primero que se hizo fue seleccionar entre las dos arquitecturas seleccionadas. Para eso se fijaron los hiperparámetros para el entrenamiento como se ve en el Cuadro 3. Estos son los mismos parámetros que la red de clasificación de vehículos.

Parámetro	**Valor**
Algoritmo de optimización	Adam
Taza de aprendizaje	1e-4
Función error	Entropía cruzada
Metricas	Exactitud
Número de iteraciones	10
Pasos por iteración	7
Cantidad de pasos en la validación	4

Cuadro 3: Hiper parámetros para entrenar la red del problema *fácil*

5.2.3. Resultados Arquitectura 1

Con esta arquitectura la exactitud final fue de 93,75 % en datos de validación y 88.57 % en datos de entrenamiento. Las gráficas del avance de las exactitudes tanto para entrenamiento como para validación, así como el valor de la entropía cruzada de pueden ver en la Figura 6. Esta gráfica nos muestra como la entropía aunque tiende a disminuir parece tener problemas para la convergencia, es más en la última iteración crece un poco, pero se logra mejorar la exactitud final de los datos. Para solucionar esto se puede o tratar de disminuir la tasa de aprendizaje o cambiar el algoritmo de actualización de pesos que se use al entrenar la red; en este caso se optó por la última opción, como ser verá más adelante. En cuanto a la Figura 6b vemos como el algoritmo predice en general igual de bien tanto para datos nuevos como para los datos con los que se esta entrenando, lo que nos indica que no hay sobreajuste en nuestra red, es más, en la iteración final se predicen mejor en los datos de validación que en los de entrenamiento.



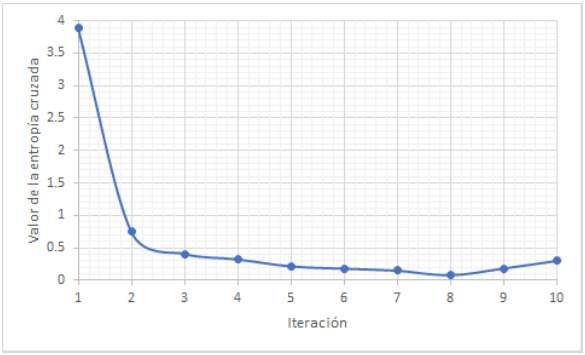
(a) Entropía vs iteración

(b) Exactitudes vs iteración

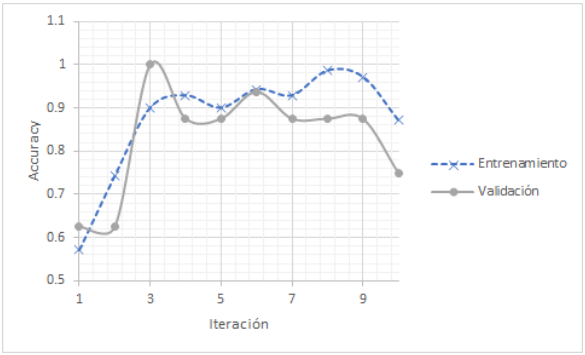
Figura 6: Valores de entropía como de la exactitud de la red en cada iteración en la

5.2.4. Resultados Arquitectura 2

Se entrenó la segunda arquitectura con los mismos parámetros propuestos en el Cuadro 3. Logramos una exactitud final del 75 %, por lo que la exactitud final se disminuyó en un 13,75 % comparado con la arquitectura anterior. Las métricas de esta red, al igual que en la anterior, se graficaron para mirar como se comportaba se comportaba la red con las iteraciones (Figura 7). Si comparamos las Figuras 7a y 6a vemos como se favorece la convergencia en esta arquitectura, a pesar de que esta tiene 88.461 parámetros más que la arquitectura anterior. Adicionalmente vemos como para este problema después de la sexta iteración la exactitud de los datos de entrenamiento siempre están por encima de los de validación, lo cual no es recomendable porque representar un sobre ajuste en los datos. Por la baja exactitud de esta arquitectura y su comportamiento a lo largo de las iteraciones se prefirió utilizar la arquitectura propuesta en el item anterior.



(a) Entropía vs iteración *fácil*

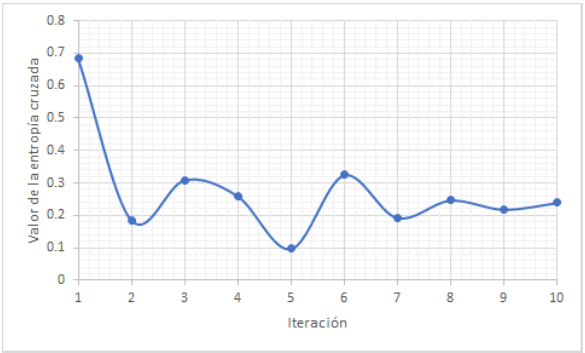


(b) Exactitudes vs iteración *fácil*

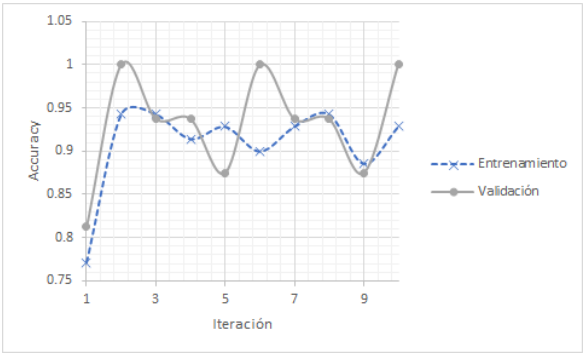
Figura 7: Valores de entropía y de la exactitud de la red en cada iteración en la arquitectura 2

5.2.5. Método de optimización

Una vez se definió completamente la arquitectura a utilizar, se buscó mejorar la convergencia del algoritmo por lo que se cambiaron los hiperparámetros. Específicamente se buscó utilizar otro algoritmo que facilitará la convergencia. Decidimos cambiar a descenso de gradiente estocastico (SGD) y probar si nos ayudaba con la convergencia de la entropía cruzada en la red neuronal. Los resultados obtenidos se ven en la Figura 8. Vemos como se mantiene la misma exactitud en la arquitectura 1 con el método de Adam (incluso mejora un poco) y que el perfil de la entropía cruzada presenta un comportamiento oscilatorio pero convergente, por lo que para este problema es mejor utilizar SGD para entrenar la red. Por otro lado es recomendable aumentar un poco el número de iteraciones para mirar la estabilización de la entropía al final y estar seguros de la convergencia del algoritmo.



(a) Entropía vs iteración *fácil*



(b) Exactitudes vs iteración *fácil*

Figura 8: Valores de entropía como de la exactitud de la red en cada iteración en la arquitectura 2

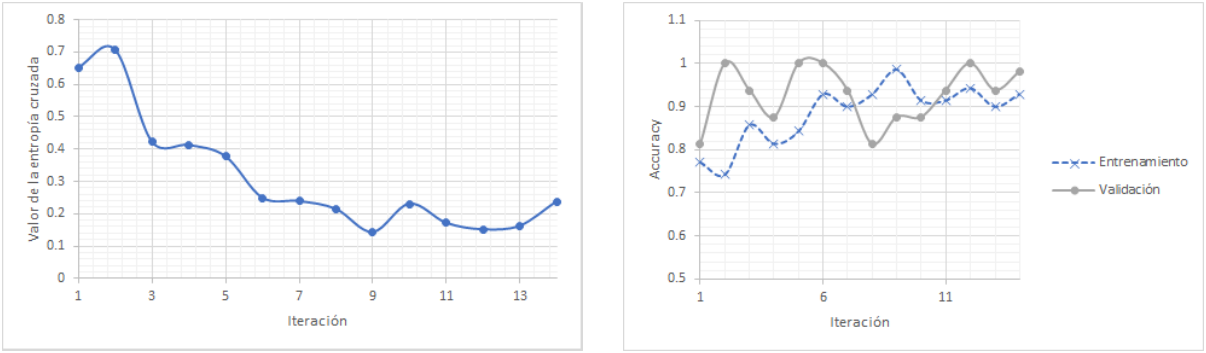
5.2.6. Hiperparámetros y resultados finales

Los hiperparámetros finales escogidos se pueden ver en el Cuadro 9, se corrió con SGD y se obtuvieron las métricas mostradas en la Figura 9, La exactitud final lograda con este modelo es 98 %. esta exactitud es alta pero hay que tener en cuenta que esta medida es relativa a los datos que se usaron para la validación y que hay que esperar a la étapa de prueba para

corroborarla. En promedio entre los datos de entrenamiento y los de validación se tiene una exactitud del 95 % lo cual es bastante bueno y puede decirse que resuelve el problema fácil.

Parámetro	Valor
Algoritmo de optimización	SDG
Taza de aprendizaje	1e-4
Función error	Entropía cruzada
Métricas	Exactitud
Número de iteraciones	14
Pasos por iteración	7
Cantidad de pasos en la validación	4

Cuadro 4: Hiper parámetros finales para entrenar la red del problema *fácil*



(a) Entropía vs iteración *fácil*

(b) Exactitudes vs iteración *fácil*

Figura 9: Valores de entropía como de la exactitud de la red en cada iteración en la arquitectura 2

5.2.7. Prueba

Al final se guardo el modelo y se utilizó para validar con los datos reservados para prueba. Se construyó la matriz de confusión (Cuadro 7). Con esta matriz nos damos cuenta que el porcentaje de aciertos para la clase mundo uno (W1) es del 91.90 % mientras que para el mundos dos (W2) es del 85.22 %. La exactitud total de la red neuronal es de 88.38 %, esto representa un decremento del 0.19 % respecto a los datos de validación y nos indica que el modelo esta siendo capaz de predecir las clases de datos nuevos manteniendo practicamente la misma exactitud mostrada durante el entrenamiento.

		Predecido	
		W2	**W6**
Real	**W2**	693	61
	W6	125	721

Cuadro 5: Matriz de confusión problema fácil

5.2.8. Implementación

La implementación final considerando el análisis anterior queda entonces como sigue

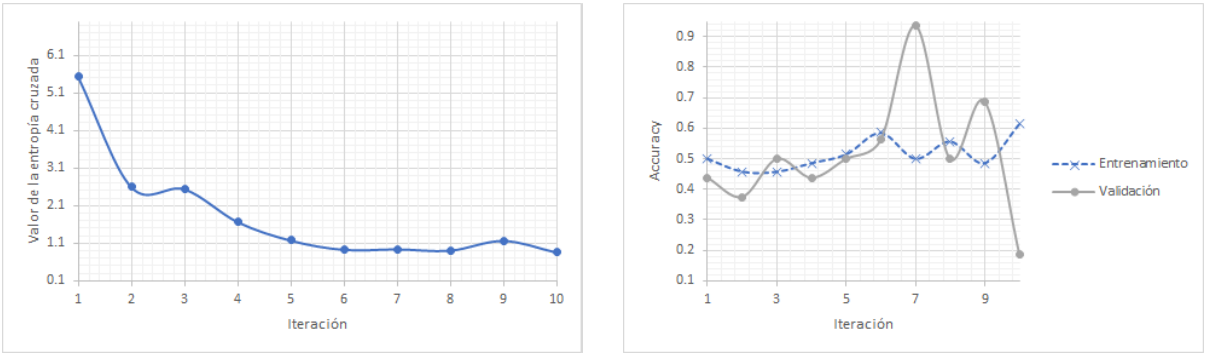
Listing 3: Código utilizado para separar el conjunto de entrenamiento del conjunto de validación.

```
import numpy as np
import keras
from keras.datasets import mnist
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers.core import Flatten, Dense
from keras.optimizers import SGD
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam

train_path = './data/video1/train_problem_one'
test_path = './data/video1/test_problem_one'

train_batch = ImageDataGenerator().flow_from_directory(train_path, ↩
    target_size=(720,1280), classes=['W2','W6'], batch_size=10)
test_batch = ImageDataGenerator().flow_from_directory(test_path, ↩
    target_size=(720,1280), classes=['W2','W6'], batch_size=4)

modelo = Sequential()
modelo.add(Conv2D(filters=48, kernel_size=(1,1), activation='relu', ↩
    input_shape=(720,1280,3)))
modelo.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

modelo.add(Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1),activation↩
    ='relu',padding='same'))
modelo.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))

modelo.add(Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1),activation↩
    ='relu',padding='same'))
modelo.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))

modelo.add(Conv2D(filters=192, kernel_size=(3, 3), strides=(1,1),↩
    activation='relu',padding='same'))
modelo.add(MaxPooling2D(pool_size=(2,2)))

modelo.add(Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1),activation↩
    ='relu',padding='same'))
modelo.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))

```

```
34 modelo.add(Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1),activation='relu',padding='same'))
35 modelo.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))
36
37 modelo.add(Conv2D(filters=28, kernel_size=(3, 3), strides=(1,1),activation='relu',padding='same'))
38 modelo.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))
39
40 modelo.add(Flatten())
41
42 modelo.add(Dense(59, activation='relu'))
43 modelo.add(Dense(59, activation='relu'))
44 modelo.add(Dense(2, activation='softmax'))
45
46 modelo.summary()
47
48 modelo.compile(SGD(lr=0.0001), loss='categorical_crossentropy', metrics=['accuracy'])
49 modelo.fit_generator(train_batch, steps_per_epoch=7,validation_data=test_batch,validation_steps=4,epochs=14,verbose=1)
```

5.3. Solución del problema difícil

Lo primero que se hizo es tratar de utilizar la red anterior a este problema y mirar los resultados que se obtienen. Al final se obtuvo una exactitud promedio de 50 %, es decir al darle un dato a la red es igual de probable que lo clasifique bien o que lo clasifique mal. Esto tiene sentido porque los colores entre los niveles son exactamente los mismo (a excepción del nivel tres) por lo que el algoritmo no puede hacer particiones puras o medianamente puras por esta características. La exactitud en la última iteración cae a menos de 20 %, en este caso es mejor clasificar todo contrario a lo que dice el algoritmo, por lo que definitivamente una red entrenada para reconocer colores no es la indicada para este problema.



(a) Entropía vs iteración *fácil*

(b) Exactitudes vs iteración *fácil*

Figura 10: Valores de entropía como de la exactitud de la red en cada iteración en la arquitectura 2

Dada la gran diferencia similitud entre estos dos mundos, con lo anterior corroboramos que no se puede hacer un clasificador tan simple como el que busque diferencias por colores. Por esta razón, se hizo la similitud de este problema con el problema de clasificación de paisajes, para esto se utilizó una red neuronal ya entrenada para clasificar paisajes en la parte de las redes convolucionales.
Al igual que en el problema anterior se propusieron dos arquitecturas en la parte de la red neuronal para ver como se reaccionaba ante una o dos capas econdidas. Las arquitecturas comparten la parte convolucional y son como siguen a continuación.

Parte convolucional

1. **Capa convolucional** de 16 filtros, con un tamaño de *kernel* de 3x3, función de activación tipo *Relu*. El tamaño de salida de esta capa es 718x1278x16. Número de parámetros entrenables de la capa: 448.
2. **Capa Max pooling** de un tamaño de 2x2. El tamaño de salida de esta capa es 359x369x16.
3. **Capa convolucional** de 32 filtros, con un tamaño de *kernel* de 3x3, función de activación tipo *Relu*. El tamaño de salida de esta capa es 357x367x32. Número de parámetros entrenables de la capa: 4.640.
4. **Capa Max pooling** de un tamaño de 2x2. El tamaño de salida de esta capa es 178x318x32.
5. **Capa convolucional** de 64 filtros, con un tamaño de *kernel* de 3x3, función de activación tipo *Relu*. El tamaño de salida de esta capa es 176x316x64. Número de parámetros entrenables de la capa: 18.496.
6. **Capa Max pooling** de un tamaño de 2x2. El tamaño de salida de esta capa es 88x158x64.
7. **Capa convolucional** de 64 filtros, con un tamaño de *kernel* de 3x3, función de activación tipo *Relu*. El tamaño de salida de esta capa es 176x316x64. Número de parámetros entrenables de la capa: 18.496.
8. **Capa Max pooling** de un tamaño de 2x2. El tamaño de salida de esta capa es 88x158x64.
9. **Capa convolucional** de 64 filtros, con un tamaño de *kernel* de 3x3, función de activación tipo *Relu*. El tamaño de salida de esta capa es 86x158x64. Número de parámetros entrenables de la capa: 36.928.
10. **Capa Max pooling** de un tamaño de 2x2. El tamaño de salida de esta capa es 43x78x64.
11. **Capa convolucional** de 64 filtros, con un tamaño de *kernel* de 3x3, función de activación tipo *Relu*. El tamaño de salida de esta capa es 41x76x64. Número de parámetros entrenables de la capa: 36.928.
12. **Capa Max pooling** de un tamaño de 2x2. El tamaño de salida de esta capa es 20x38x64.

Arquitectura 1

1. A la primera capa de la red entran un total de 48.640 *features*. Esta capa tiene 35 neuronas completamente conectadas. con una activación tipo ReLu. Hay un total de 1.702.435 parámetros entrenables
2. Capa con la misma configuración de la capa anterior. En total hay 1.260 parámetros entrenables
3. Capa de salida con dos neuronas, completamente conectada a la capa anterior. Hay un total de 72 parámetros entrenables

Esta arquitectura en suma tendría 1.801.207 parámetros, más del triple que los parámetros necesarios en la clasifiación de la tarea fácil.

Arquitectura 2

1. A la primera capa de la red entran un total de 48.640 *features*. Esta capa tiene 75 neuronas completamente conectadas. con una activación tipo ReLu. Hay un total de 3.648.075 parámetros entrenables.
2. Capa de salida con dos neuronas, completamente conectada a la capa anterior. Hay un total de 72 parámetros entrenables

Esta arquitectura en suma tendría 3.745.667 parámetros, alrededor de 8 veces más parámetros que los necesarios en la clasificación del problema fácil.

5.3.1. Hiperparámetros iniciales

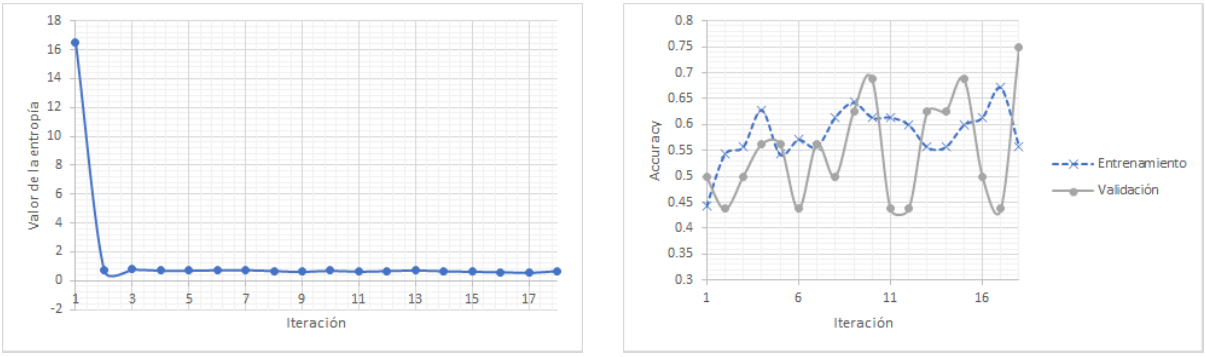
Se utilizaron los hipeparámetros iniciales propuestos en la clasificación de paisajes pero se cambió el algoritmo a SGD porque en el problema fácil tuvo una mejor convergencia, si se veía que el algoritmo tenía problemas para converger se buscaría otro método pero esto no ocurrió. Además se aumentaron el número de iteraciones porque este problema a 18 y se procedieron a evaluar las dos arquitecturas propuestas.

Parámetro	Valor
Algoritmo de optimización	SDG
Taza de aprendizaje	1e-4
Función error	Entropía cruzada
Métricas	Exactitud
Número de iteraciones	18
Pasos por iteración	7
Cantidad de pasos en la validación	4

Cuadro 6: Hiper parámetros para entrenar la red del problema *fácil*

Resultados arquitectura 1

Se graficaron las métricas utilizadas anteriormente (Figura 11). La exactitud final en los datos de validación fue de 75 % y de 55 % en los datos de entrenamiento. Vemos como no hay problema para la convergencia pero si tenemos una exactitud extremadamente variable con el número de las iteraciones, particularmente con los datos de validación, en general el algoritmo parece adecuarse más a los datos con los que esta entrenando que con los que esta validando, hecho que como ya se dijo antes se quiere evitar.



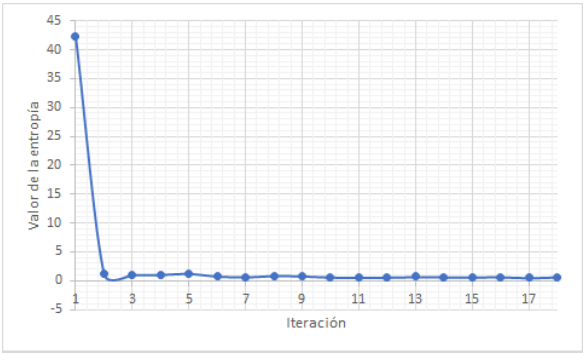
(a) Entropía vs iteración *fácil*

(b) Exactitudes vs iteración *fácil*

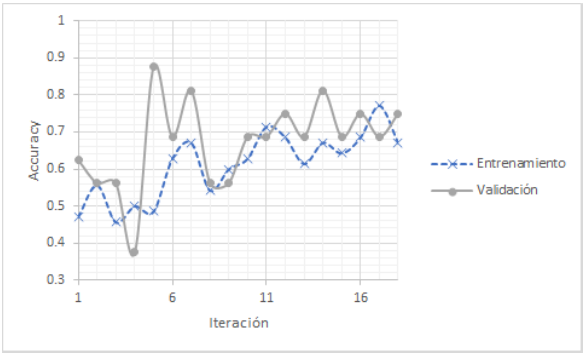
Figura 11: Valores de entropía como de la exactitud de la red en cada iteración en la arquitectura 2

Resultados Arquitectura 2

Las métricas de esta arquitectura se pueden observar en la Figura 12. La exactitud final del modelo fue de 75 % para los datos de validación, igual que en la arquitectura anterior, y 67,5 % para los de entrenamiento, 13,5 % por encima del logrado en la arquitectura anterior. En los datos de validación este problema está 23 % por debajo en los datos de validación de la exactitud lograda en el problema fácil. En esta arquitectura dos tenemos una convergencia igual de buena que en la arquitectura anterior, pero la exactitud de modelo, aunque variable si parece en general si parece aumentar y converger con el número de iteraciones, por lo que esta arquitectura se prefirió sobre la anterior.



(a) Entropía vs iteración *fácil*



(b) Exactitudes vs iteración *fácil*

Figura 12: Valores de entropía como de la exactitud de la red en cada iteración en la arquitectura 2

5.3.2. Pruebas

Al final se guardo el modelo y se utilizó para validar con los datos reservados para prueba. Se construyó la matriz de confusión (Cuadro 7). Con esta matriz nos damos cuenta que el porcentaje de aciertos para la clase mundo dos (W2) es del 65.31 % mientras que para el mundos seis (W6) es del 74 %. La exactitud total de la red neuronal es de 69.45 %, esto representa un decremento del 0.06 % respecto a los datos de validación y nos indica que el modelo esta siendo capaz de predecir las clases de datos nuevos. Aún así, la exactitud de este modelo está muy por debajo del problema fácil.

		Predecido	
		W2	**W6**
Real	**W2**	751	399
	W6	273	777

Cuadro 7: Matriz de confusión problema fácil

Implementación

A continuación se muestra la implementación de la red en *Python*

Listing 4: Red neuronal para el problema fácil.

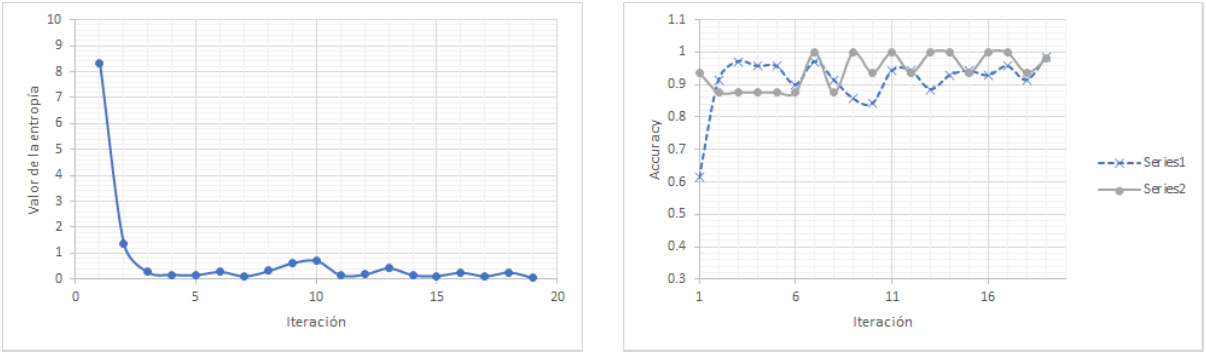
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import keras
4 from keras.datasets import mnist
5 from keras.utils import np_utils
6 from keras.models import Sequential
7 from keras.layers.convolutional import Conv2D, MaxPooling2D
8 from keras.layers.core import Flatten, Dense
9 from keras.optimizers import SGD
```

```
10 from keras.preprocessing.image import ImageDataGenerator
11 from keras.optimizers import Adam
12
13 train_path = './data/video1/train_problem_one'
14 test_path = './data/video1/test_problem_one'
15
16 train_batch = ImageDataGenerator().flow_from_directory(train_path, ↩
       target_size=(720,1280), classes=['W2','W6'], batch_size=10)
17 test_batch = ImageDataGenerator().flow_from_directory(test_path, ↩
       target_size=(720,1280), classes=['W2','W6'], batch_size=4)
18
19 modelo = Sequential()
20 modelo.add(Conv2D(filters=16, kernel_size=(3,3), activation='relu', ↩
       input_shape=(720,1280,3)))
21 modelo.add(MaxPooling2D(pool_size=(2,2)))
22
23 modelo.add(Conv2D(32, (3, 3), activation='relu'))
24 modelo.add(MaxPooling2D(pool_size=(2,2)))
25
26 modelo.add(Conv2D(64, (3, 3), activation='relu'))
27 modelo.add(MaxPooling2D(pool_size=(2,2)))
28
29 modelo.add(Conv2D(64, (3, 3), activation='relu'))
30 modelo.add(MaxPooling2D(pool_size=(2,2)))
31
32 modelo.add(Conv2D(64, (3, 3), activation='relu'))
33 modelo.add(MaxPooling2D(pool_size=(2,2)))
34
35 modelo.add(Flatten())
36
37 modelo.add(Dense(75, activation='relu'))
38 modelo.add(Dense(2, activation='softmax'))
39
40 modelo.compile(Adam(lr=0.0001), loss='categorical_crossentropy', metrics=[↩
       'accuracy'])
41 modelo.fit_generator(train_batch, steps_per_epoch=7,validation_data=↩
       test_batch,validation_steps=4,epochs=18,verbose=1)
```

5.4. Implementación de la red difícil, en el problema fácil

Por último se probó la arquitectura de la red final utilizada en el problema difícil en el problema fácil para ver si las dos dos problemas podían resolverse con una sola red neuronal. Los resultados obtenidos son los mostrados en la Figura 13. Observamos que la exactitud final en validación es exactamente igual (98 %), sin embargo la exactitud en los datos de entrenamiento si aumentó en un 5.18 % . No obstante esto se logra en el doble de iteraciones, en un número

de iteraciones igual al usado en el problema fácil (10 iteraciones). Si usáramos este número de iteraciones lograríamos apenas un 70 %, esto puede deberse a que esta red, como ya se dijo ants, entrena alrededor de 3 veces más parámetros que la red anterior (alrededor de 3.5 MM de parámetros) entones requiere más iteraciones para entrenarse. En la figura 13b vemos como la exactitud del algoritmo también es mucho más estable con las iteraciones que en la red original implementada para este problema. En cuanto a la convergencia, en el Figura 13b vemos que no hay mayor problema para la convergencia de este algoritmo, esto se esperaba ya que se esta usando el mismo algoritmo que el seleccionado en la red original de este problema.



(a) Entropía vs iteración *fácil*

(b) Exactitudes vs iteración *fácil*

Figura 13: Valores de entropía como de la exactitud de la red en cada iteración en la arquitectura 2

5.5. Conclusiones

Al final, se lograron resolver los dos problemas de clasificación planteados, el resumen de los resultados obtenidos se muestra en el Cuadro 8. Aquí podemos ver que, como esperamos, logramos una mayor exactitud el el problema fácil. Otra cosa interesante es que para el entrenamiento de la red difícil es el doble de rápido que entrenar la red de fácil a pesar de que tenga casi 10 parámetros más que la otra red. Esto se debe a que esta arquitectura tiene más capas y es más demorado calcular otra linea de derivadas, calcular una sola neurona para un computador es sólo una multiplicación.

Otra cosas que podemos ver es el número de características que entra a la red neuronal. El problema fácil sólo requiere 1.008 características mientras que el problema dificil requiere 48.648, es decir 40 veces más y aún así se obtiene 23 % menos de exactitud. Al final para ambas redes la exactitud en prueba cae con respecto a entrenamiento lo que nos reafirma la importancia de probar las redes que implementemos con datos que el algoritmo no conozca.

Características	**Problema fácil**	**Problema difícil**
Mundos de Mario Bros	Mundos 2 y 6	Mundos 2 y 4
Número de capas totales	17	14
Número de capas en la parte convolucional	14	12
Número de capas en la red neuronal	3	2
Número total de características que entran a la red neuronal	1008	48.648
Número total de características que entran a la red neuronal	402.547	3.745.667
Tiempo total ajuste [minutos]	45:21:0	17:39:0
Exactitud final en los datos de validación	93,75 %	75 %
Exactitud final en los datos de entrenamiento	98 %	67,5 %
Exactitud final en los datos de prueba	88,38 %	69,45 %

Cuadro 8: Resumen de los modelos implementados

Para ambos problemas se lograron exactitud buena, incluso para el problema difícil por lo que se puede decir que se cumplió con el objetivo del problema de clasificación aunque entrenar las redes sea demorado. Si se quisiera generalizar el problema podría utilizarse usando la red del problema difícil ya que es más rápido. Finalmente, se aprendió a testear e implementar una red convolucinal completa y usarla para un problema aplicado.