Proyecto de aprendizaje profundo

Sofía Ingigerth Cañas Urbina A01173828

Profesor: Dr. Santiago Enrique Conant Pablos

Diseño de redes neuronales y aprendizaje profundo

Índice

Introducción

La base de datos Fashion-MNIST está conformada por un gran conjunto de imágenes que se encuentran en escala de grises, representando entre cada imagen 10 tipos diferentes de prendas. Este conjunto consta de entrenamiento cuenta con 60,000 ejemplos que sirven para entrenar un modelo, así como cuenta con otros 10,000 ejemplos que sirven para poner a prueba dicho modelo.

Este modelo resulta ser un nuevo conjunto de datos que es de utilidad para aprender sobre el aprendizaje profundo. Cada ejemplo que se encuentra dentro de este conjunto tiene una escala de 28x28. A partir de este conjunto de datos exploramos dos arquitecturas de redes neuronales para que puedan aprender a clasificar las distintas prendas de vestir que se encuentran contenidas en este dataset.

Con el pasar de los años, la clasificación de imágenes ha mejorado gracias a los avances tecnológicos que se han tenido y a la mejora del poder computacional, lo que nos permite realizar cosas tan maravillosas como las que buscamos realizar dentro de los fines de este proyecto de aprendizaje profundo. La Inteligencia Artificial va más allá del análisis de datos. Su evolución se dirige hacia la creación de los mismos.

Este trabajo tiene un gran objetivo: el poder explorar modelos para poder identificar entre prendas de ropa de la base de datos Fashion MNIST. Se utilizarán dos arquitecturas de redes neuronales, una puesta a disposición en clase y otra investigada. Se hará una comparación entre arquitecturas y se elegirá el modelo más adecuado.

La Inteligencia Artificial (IA) avanza más allá del puro análisis de datos. Su evolución se dirige rápidamente hacia la generación de los mismos, a medida que la ciencia redefine la capacidad de las máquinas para tomar mejores decisiones.

Modelo 1: “MNIST convnet”

Preparación de datos

La arquitectura de este modelo resulta ser una arquitectura propuesta en clase, como modelo práctico para el aprendizaje del manejo de redes convolucionales. Para llevar a cabo un adecuado uso de la arquitectura propuesta en este primer caso, se tuvo que importar previamente todas las librerías necesarias para construir el modelo convolucional. Las librerías utilizadas fueron *numpy, matplotlib* (específicamente pyplot) y *tensorflow* (específicamente keras).

Para cargar los datos de entrenamiento y de prueba, ejecutamos la siguiente línea de código:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
```

Esto permite cargar de manera efectiva el dataset fashion_mnist. Como parte de la preparación de los datos, se realizó una normalización de estos ya que es requerido trabajar con datos que se encuentren dentro del rango [0,1]. Las línea de código son:

```
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

Como se mencionó anteriormente, Fashion MNIST es un dataset con imágenes de dimensión 28x28, por lo que nos resulta de interés que las imágenes tengan un tamaño de (28, 28, 1), para ello se ejecutan las siguientes líneas de código:

```
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
```

A su vez, se requirió convertir las clases, es decir, las categorías de ropa, a una forma categórica (forma one hot). Las líneas de código son las siguientes:

```
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

Con esto es suficiente para poder comenzar a utilizar los datos de entrenamiento y prueba dentro del modelo que se presentará a continuación.

Implementación y entrenamiento

Como se mencionó anteriormente, este primer modelo fue proporcionado en clase. La arquitectura de dicho modelo es la siguiente:

```
model = keras.Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax")])
```

De esto, puede apreciarse que es un modelo que cuenta con dos convoluciones dentro de él. Para entrenar este modelo, el código original muestra las siguientes líneas de código:

```
batch_size = 128
epochs = 15
model.compile(loss="categorical_crossentropy", optimizer="adam",
              metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
              validation_split=0.1)
```

Con estos parámetros corrimos el código para tener en claro qué tan eficiente es el modelo, para saber qué expectativas tener sobre las posibles mejoras que se le pueden realizar. A continuación se muestran los resultados que resultan de entrenar con estos parámetros al modelo. Se puede apreciar que, en estas condiciones, esta arquitectura proporcionada en clase alcanza un 90.18% de precisión.

```
422/422 [==============================] - 2s 5ms/step - loss: 0.2881 - accuracy: 0.8965 - val_loss: 0.2633 - val_accuracy: 0.9053
Epoch 12/15
422/422 [==============================] - 2s 5ms/step - loss: 0.2823 - accuracy: 0.8969 - val_loss: 0.2792 - val_accuracy: 0.8963
Epoch 13/15
422/422 [==============================] - 2s 5ms/step - loss: 0.2778 - accuracy: 0.8986 - val_loss: 0.2605 - val_accuracy: 0.9045
Epoch 14/15
422/422 [==============================] - 2s 5ms/step - loss: 0.2732 - accuracy: 0.9006 - val_loss: 0.2568 - val_accuracy: 0.9065
Epoch 15/15
422/422 [==============================] - 2s 5ms/step - loss: 0.2697 - accuracy: 0.9013 - val_loss: 0.2617 - val_accuracy: 0.9018
<keras.callbacks.History at 0x7f17d03e6e10>
```

Con esto podemos hacernos a la expectativa de que se puede mejorar aún el modelo, que resulta ser ya muy bueno.

Existen varias partes con las que se puede experimentar variaciones del modelo, tale como las siguientes:

1. Funciones de activación
2. Cantidad de épocas
3. Batch size
4. Optimizador
    a. Tipo de optimizador
    b. Razón de aprendizaje

5. Arquitectura misma de modelo propuesto

Particularmente, las funciones de activación, que actualmente son “relu” para llevar a cabo todo el proceso y “softmax” para la última capa, son funciones que no predice que no deben ser modificadas, pues estas son las que suelen ser utilizadas en una red convolucional como esta, pues dan los mejores resultados dentro de una red convolucional. De igual manera, realizamos varias pruebas como las siguiente:

- Usar “tanh” en vez de “relu”
- Usar “tanh” en vez de “softmax”
- Usar “sigmoid” en vez de “relu”
- Usar “sigmoid” en vez de “softmax”

En general, los resultados obtenidos así, empeoraba el aprendizaje de nuestro modelo, tal y como lo previsto, por lo que este cambio se descartó.

El aumentar la cantidad de épocas permite a la red tener más oportunidades de mejorar lo que ha conseguido aprender. Comenzamos a aumentar gradualmente las épocas, 30, 45, 60, etc.

El resultado más óptimo se consiguió con las primeras 60 época, en donde los resultados mejoran a un 92.23%, como se ve a continuación:

```
422/422 [==============================] - 2s 5ms/step - loss: 0.1963 - accuracy: 0.9262 - val_loss: 0.2157 - val_accuracy: 0.9237
Epoch 55/60
422/422 [==============================] - 2s 5ms/step - loss: 0.1956 - accuracy: 0.9274 - val_loss: 0.2227 - val_accuracy: 0.9187
Epoch 56/60
422/422 [==============================] - 2s 5ms/step - loss: 0.1930 - accuracy: 0.9295 - val_loss: 0.2114 - val_accuracy: 0.9260
Epoch 57/60
422/422 [==============================] - 2s 5ms/step - loss: 0.1923 - accuracy: 0.9289 - val_loss: 0.2175 - val_accuracy: 0.9227
Epoch 58/60
422/422 [==============================] - 2s 5ms/step - loss: 0.1916 - accuracy: 0.9292 - val_loss: 0.2188 - val_accuracy: 0.9202
Epoch 59/60
422/422 [==============================] - 2s 5ms/step - loss: 0.1904 - accuracy: 0.9294 - val_loss: 0.2309 - val_accuracy: 0.9187
Epoch 60/60
422/422 [==============================] - 2s 5ms/step - loss: 0.1935 - accuracy: 0.9286 - val_loss: 0.2181 - val_accuracy: 0.9223
<keras.callbacks.History at 0x7f17d0e7b110>
```

La importancia de utilizar un buen batch_size es que determinan con cuántos ejemplos trabajará cada mini lote durante cada época. Por lo general se utilizan potencias de 2 para aprovechar al máximo los recursos, por ello, variamos el tamaño utilizando 16, 32, 64 y 256, en busca de mejores resultados. Probando con 256 se obtuvo 92.18% y con 64 se obtuvo 92.08%. Para 32 y 16, se obtuvo menos del 92% y con mayor tiempo para terminar el entrenamiento (menos práctico). Se decidió probar con tamaños de batch más aleatorios pero esto no mejoró los resultados, por lo que se decidió mantener el batch_size con tamaño de 128.

La importancia de tener un buen optimizador es que a partir de él, la red neuronal ajustará sus pesos. En este modelo, de manera predeterminada se maneja Adam con una razón de aprendizaje de 0.001. Con fines de buscar mejorar nuestro 92.23% probamos con otros optimizadores: Adamax, Adadelta, Adagrad, RMSprop y SGD. Optimizadores como Adadelta, Adagrad y SGD empeoraron los resultados, disminuyendo el porcentaje a menos de 83%. Adamax, por su parte, consiguió un accuracy del 91.45%. RMSprop obtuvo un accuracy de 91.60%. Con esto en cuenta, el dejar también el optimizador Adam es nuestra mejor opción. De manera similar a esto, el cambiar la razón de aprendizaje empeoró nuestros resultados, por lo que también se tuvo que mantener este parámetro.

Respecto a la cantidad de convoluciones, hemos de mencionar como adelanto del segundo modelo a tratar que será de igual manera un modelo convolucional. La diferencia radica en el hecho de que será un modelo con tres convoluciones. El no cambiar este hecho será fundamental para nuestras arquitecturas ya que será su “esencia”.

Sabiendo esto, hemos de mencionar que dentro de la arquitectura se hará otro tipo de cambios, como agregar Batch Normalization y Dropouts, ya que estos ayudan a evitar un overfitting y permiten mejorar modelos.

Agregando sólo Batch Normalization, que comúnmente se recomienda después de cada convolución, hemos mejorado nuestros resultados, subiendo a 92.37%, como se ve en la imagen siguiente:

```
422/422 [==============================] - 3s 6ms/step - loss: 0.1694 - accuracy: 0.9372 - val_loss: 0.2481 - val_accuracy: 0.9178
Epoch 57/60
422/422 [==============================] - 3s 6ms/step - loss: 0.1693 - accuracy: 0.9364 - val_loss: 0.2574 - val_accuracy: 0.9145
Epoch 58/60
422/422 [==============================] - 3s 7ms/step - loss: 0.1655 - accuracy: 0.9393 - val_loss: 0.2285 - val_accuracy: 0.9238
Epoch 59/60
422/422 [==============================] - 3s 6ms/step - loss: 0.1653 - accuracy: 0.9380 - val_loss: 0.2179 - val_accuracy: 0.9253
Epoch 60/60
422/422 [==============================] - 2s 6ms/step - loss: 0.1680 - accuracy: 0.9375 - val_loss: 0.2232 - val_accuracy: 0.9237
```

Al aplicar Dropout, aumentamos también la cantidad de épocas al doble, ya que el Dropout lo que hace es evitar que de manera aleatoria, ciertas neuronas no se entrenen para que no exista el mencionado overfitting, por ello, resulta comprensible que la red requiera de más épocas para conseguir aprender. Después de agregar Dropouts, el accuracy aumentó a 92.53%, como se ve en la siguiente imagen:

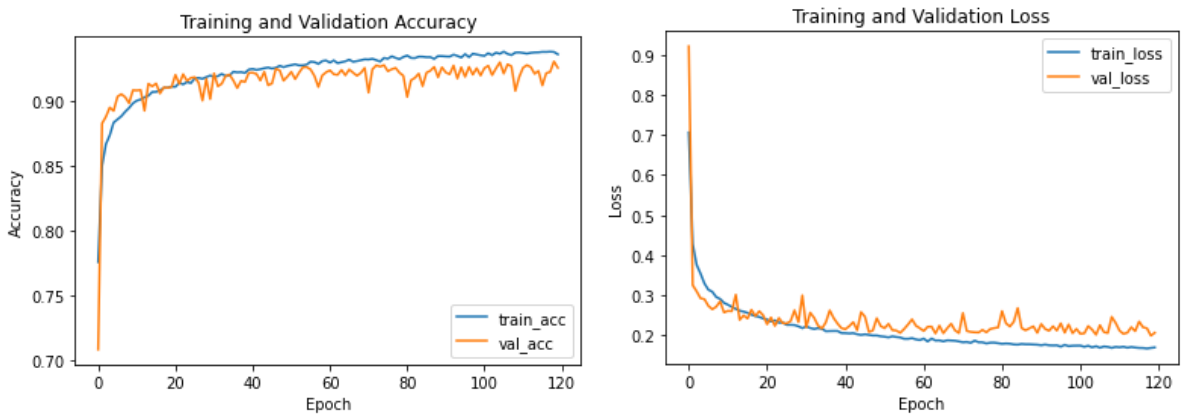
```
Epoch 116/120
422/422 [==============================] - 3s 7ms/step - loss: 0.1690 - accuracy: 0.9376 - val_loss: 0.2348 - val_accuracy: 0.9117
Epoch 117/120
422/422 [==============================] - 3s 7ms/step - loss: 0.1688 - accuracy: 0.9375 - val_loss: 0.2204 - val_accuracy: 0.9212
Epoch 118/120
422/422 [==============================] - 3s 7ms/step - loss: 0.1676 - accuracy: 0.9377 - val_loss: 0.2181 - val_accuracy: 0.9220
Epoch 119/120
422/422 [==============================] - 3s 7ms/step - loss: 0.1690 - accuracy: 0.9375 - val_loss: 0.1997 - val_accuracy: 0.9302
Epoch 120/120
422/422 [==============================] - 3s 7ms/step - loss: 0.1702 - accuracy: 0.9357 - val_loss: 0.2067 - val_accuracy: 0.9253
```

Al finalizar con las modificaciones, la nueva arquitectura quedó de la siguiente manera:

```
model = keras.Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.BatchNormalization(),
        layers.Dropout(0.1),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.BatchNormalization(),
        layers.Dropout(0.1),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax")])
```

Evaluación del modelo entrenado

Para evaluar el accuracy del modelo nos hemos basado en lo que despliega el modelo después de ser entrenado, como se vio anteriormente, que se llegó a un accuracy final del 92.53%.
También se utilizó la gráfica de la curva de aprendizaje de este entrenamiento para ver la confiabilidad que podríamos tener del modelo, puede verse a continuación:



Con esto vemos que no existe presencia alguna de overfitting por lo que podemos confiar en que consiguió predecir correctamente el 92.53% de los datos de prueba.

Conclusiones del método

Sin duda el modelo generado es muy bueno, desde el modelo inicial hasta el modelo final mejorado. En general, al ver que desde que se utilizó la arquitectura base, se tuvo la expectativa de que sería posible mejorar el modelo ya que podía apreciarse la facilidad con la que pudo aprender el 90% en un inicio.

Este modelo fue utilizado antes para predecir utilizando el dataset MNIST, y tuvo excelentes resultados, en parte por ello se tenía una alta expectativa. Por el hecho de que las redes convolucionales son muy utilizadas para aprendizaje aplicado a imágenes, es que hubo una facilidad para que este modelo consiguiera aprender.

Modelo 2: “FASHION MNIST: Convolutional Neural Network”

Nuestro segundo modelo es, al igual que el primer modelo, una red neuronal convolucional. Como mencionamos anteriormente, la “esencia” del primer modelo fue que contaba con exactamente dos convoluciones. En este caso, hemos investigado una nueva arquitectura y hemos conseguido una con una nueva “esencia”, en donde se cuenta con 3 convoluciones. El modelo fue obtenido de la plataforma *Kaggle*.

Preparación de datos

Dentro de este modelo en *Kaggle*, se utiliza un procedimiento diferente al que utilizaremos para preparar los datos. Esto es porque en *Kaggle* utilizan un procedimiento en el cual ya se tiene el dataset como csv. Por lo tanto, seguiremos el mismo procedimiento que con el modelo 1, donde las líneas de código serían las siguientes:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    x_train = np.expand_dims(x_train, -1)
    x_test = np.expand_dims(x_test, -1)
    y_train = keras.utils.to_categorical(y_train, 10)
    y_test = keras.utils.to_categorical(y_test, 10)
```

Implementación y entrenamiento

Para este segundo modelo, hemos tomado de *Kaggle* sólamente lo que nos interesaba, que es la arquitectura y sus parámetros, de esto que el modelo sea de la siguiente manera:

```
    #Parte 1 del modelo
    model = Sequential()
    LeakyReLU = lambda x: tf.keras.activations.relu(x, alpha=0.1)

    model.add(Conv2D(32,
                     kernel_size = (3, 3),
                     activation = LeakyReLU,
                     padding="same",
                     input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.3))

    #Parte 2 del modelo
    model.add(Conv2D(64,
                     kernel_size = (3, 3),
                     activation = LeakyReLU,
                     padding="same"))
```

```
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))

#Parte 3 del modelo
model.add(Conv2D(128, (3, 3), activation = LeakyReLU))
model.add(Flatten())                                    # Flatemos el tensor de
píxeles:
model.add(Dense(128, activation = LeakyReLU))
model.add(Dropout(0.3))
model.add(Dense(10, activation = 'softmax'))     # La última capa debe ser el
n° de lables a predecir
```

De esto, puede apreciarse que es un modelo que cuenta con tres convoluciones dentro de él. Para entrenar este modelo, el código original muestra las siguientes líneas de código:

```
model.compile(loss = keras.losses.categorical_crossentropy,
              optimizer = 'adam',
              metrics = ['accuracy'])
batch = 70
epocas = 50
train_model = model.fit(x_train, y_train,
                        batch_size = batch,
                        epochs = epocas,
                        verbose = 1,
                        validation_data = (x_test, y_test))
```

Con estos parámetros corrimos el código para tener en claro qué tan eficiente es el modelo, para saber qué expectativas tener sobre las posibles mejoras que se le pueden realizar. A continuación se muestran los resultados que resultan de entrenar con estos parámetros al modelo. Se puede apreciar que, en estas condiciones, esta arquitectura proporcionada en clase alcanza un 92.40% de precisión.

```
858/858 [==============================] - 4s 5ms/step - loss: 0.1619 - accuracy: 0.9395 - val_loss: 0.2309 - val_accuracy: 0.9218
Epoch 44/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1604 - accuracy: 0.9383 - val_loss: 0.2436 - val_accuracy: 0.9219
Epoch 45/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1620 - accuracy: 0.9385 - val_loss: 0.2276 - val_accuracy: 0.9263
Epoch 46/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1578 - accuracy: 0.9405 - val_loss: 0.2587 - val_accuracy: 0.9148
Epoch 47/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1584 - accuracy: 0.9405 - val_loss: 0.2240 - val_accuracy: 0.9273
Epoch 48/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1562 - accuracy: 0.9410 - val_loss: 0.2316 - val_accuracy: 0.9253
Epoch 49/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1549 - accuracy: 0.9405 - val_loss: 0.2367 - val_accuracy: 0.9244
Epoch 50/50
858/858 [==============================] - 4s 5ms/step - loss: 0.1551 - accuracy: 0.9407 - val_loss: 0.2320 - val_accuracy: 0.9240
```

Al igual que con el primer modelo, con esto podemos hacernos a la expectativa de que se puede mejorar aún el modelo, que resulta ser ya muy bueno.

A su vez, para este modelo de red convolucional se le pueden hacer las mismas modificaciones que el primer modelo, por lo que tomamos en cuenta los mismos 5 puntos antes mencionados para comenzar a hacer modificaciones.

En este caso, por lo mismo que ya se ha mencionado en el modelo anterior, el modificar las funciones de activación no resulta conveniente, pero de igual manera lo intentamos sustituyendo con funciones de activación “tanh” y “sigmoid”. Tal y como con el modelo 1, en general, los resultados obtenidos así, empeoraron el aprendizaje del modelo, por lo que este cambio se volvió a descartar.

Lo segundo que hemos probado modificar es la cantidad de épocas. En el modelo base utilizan 50, por lo que hemos probado con valores mayores a eso, ya que así el modelo tendrá más oportunidades de mejorar.

Tras aumentar las épocas a 120, el modelo pudo subir a 92.60% de accuracy, como se ve a continuación:

```
Epoch 115/120
858/858 [==============================] - 4s 5ms/step - loss: 0.1315 - accuracy: 0.9497 - val_loss: 0.2428 - val_accuracy: 0.9286
Epoch 116/120
858/858 [==============================] - 4s 5ms/step - loss: 0.1324 - accuracy: 0.9497 - val_loss: 0.2561 - val_accuracy: 0.9226
Epoch 117/120
858/858 [==============================] - 4s 5ms/step - loss: 0.1328 - accuracy: 0.9497 - val_loss: 0.2501 - val_accuracy: 0.9265
Epoch 118/120
858/858 [==============================] - 4s 5ms/step - loss: 0.1288 - accuracy: 0.9522 - val_loss: 0.2544 - val_accuracy: 0.9247
Epoch 119/120
858/858 [==============================] - 4s 5ms/step - loss: 0.1294 - accuracy: 0.9510 - val_loss: 0.2555 - val_accuracy: 0.9240
Epoch 120/120
858/858 [==============================] - 4s 5ms/step - loss: 0.1299 - accuracy: 0.9502 - val_loss: 0.2554 - val_accuracy: 0.9260
```

Dejamos el estándar en esta cantidad de épocas. Después de aumentar esta cantidad, parecía que el modelo mejoraba con los datos de entrenamiento pero se mantenía en 92% para los datos de entrenamiento, una señal de que podría estar ocurriendo overfitting, por ello, optamos por establecer este valor.

Dentro del modelo anterior expresamos la importancia de tener un buen batch_size. Por lo general se utilizan potencias de 2 para aprovechar al máximo los recursos, por ello, variamos el tamaño utilizando 16, 32, 64, 128 y 256, en busca de mejores resultados. Probando con un tamaño de 256 se consiguió mejorar a 92.64% y con 128 mejoró a 92.93%. Para 64, 32 y 16, se obtuvo menos del 91% y con mayor tiempo para terminar el entrenamiento (menos práctico). A continuación se aprecia una imagen de nuestro mejor resultado.

```
Epoch 116/120
469/469 [==============================] - 3s 7ms/step - loss: 0.1145 - accuracy: 0.9564 - val_loss: 0.2502 - val_accuracy: 0.9316
Epoch 117/120
469/469 [==============================] - 3s 7ms/step - loss: 0.1107 - accuracy: 0.9581 - val_loss: 0.2457 - val_accuracy: 0.9316
Epoch 118/120
469/469 [==============================] - 3s 7ms/step - loss: 0.1095 - accuracy: 0.9584 - val_loss: 0.2503 - val_accuracy: 0.9290
Epoch 119/120
469/469 [==============================] - 3s 7ms/step - loss: 0.1116 - accuracy: 0.9587 - val_loss: 0.2542 - val_accuracy: 0.9307
Epoch 120/120
469/469 [==============================] - 3s 7ms/step - loss: 0.1103 - accuracy: 0.9583 - val_loss: 0.2532 - val_accuracy: 0.9293
```

Como un batch_size de 128 nos dió mejores resultados, nos quedamos con esta opción.

Igual hemos mencionado la importancia de los optimizadores antes. En este modelo, de manera predeterminada se maneja Adam con una razón de aprendizaje de 0.001. Probamos con otros Adamax, Adadelta, Adagrad, RMSprop y SGD para ver si nos permitirían mejorar. De la misma manera que en el modelo anterior, Adadelta, Adagrad y SGD empeoraron los resultados, disminuyendo el porcentaje a menos de 88%. Adamax y

RMSprop obtuvieron accuracies mayores al 90% pero no superaron a Adam, por ello, permanecimos con este optimizador.

Algo que podemos mencionar, es que la arquitectura de este modelo investigado ya cuenta con Dropouts que permiten un mejor desempeño en el entrenamiento de un modelo y ayuda a evitar overfitting, lo que no tiene el modelo es Batch Normalization, por lo que decidimos aplicarlo para analizar si esto nos puede dar mejores resultados.

Agregando el Batch Normalization, conseguimos mejorar nuestros resultados, subiendo a 93.38%, como se ve en la imagen siguiente:

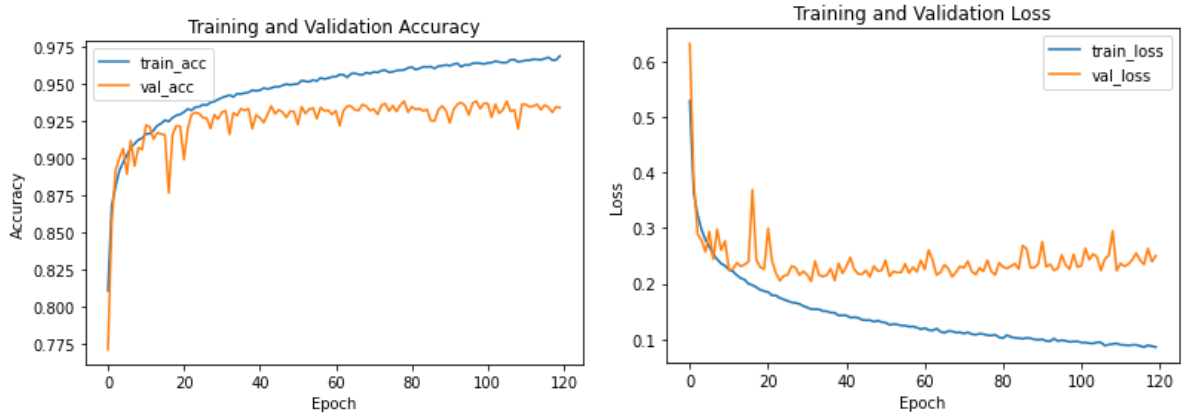
```
469/469 [==============================] - 4s 8ms/step - loss: 0.0896 - accuracy: 0.9660 - val_loss: 0.2550 - val_accuracy: 0.9324
Epoch 116/120
469/469 [==============================] - 4s 8ms/step - loss: 0.0877 - accuracy: 0.9668 - val_loss: 0.2424 - val_accuracy: 0.9356
Epoch 117/120
469/469 [==============================] - 4s 8ms/step - loss: 0.0855 - accuracy: 0.9676 - val_loss: 0.2346 - val_accuracy: 0.9340
Epoch 118/120
469/469 [==============================] - 4s 8ms/step - loss: 0.0890 - accuracy: 0.9657 - val_loss: 0.2636 - val_accuracy: 0.9307
Epoch 119/120
469/469 [==============================] - 4s 8ms/step - loss: 0.0871 - accuracy: 0.9659 - val_loss: 0.2398 - val_accuracy: 0.9344
Epoch 120/120
469/469 [==============================] - 4s 8ms/step - loss: 0.0858 - accuracy: 0.9686 - val_loss: 0.2503 - val_accuracy: 0.9338
```

Al finalizar con las modificaciones, la nueva arquitectura quedó de la siguiente manera:

```
model = Sequential()
LeakyReLU = lambda x: tf.keras.activations.relu(x, alpha=0.1)
model.add(Conv2D(32,
                 kernel_size = (3, 3),
                 activation = LeakyReLU,
                 padding="same",
                 input_shape=(28, 28, 1)))
model.add(keras.layers.BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.3))
model.add(Conv2D(64,
                 kernel_size = (3, 3),
                 activation = LeakyReLU,
                 padding="same"))
model.add(keras.layers.BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(128, (3, 3), activation = LeakyReLU))
model.add(keras.layers.BatchNormalization())
model.add(Flatten())
model.add(Dense(128, activation = LeakyReLU))
model.add(Dropout(0.3))
model.add(Dense(10, activation = 'softmax'))
```

Evaluación del modelo entrenado

Para evaluar el accuracy del modelo nos hemos basado en lo que despliega el modelo después de ser entrenado, como se vio anteriormente, que se llegó a un accuracy final del 93.38%.
También se utilizó la gráfica de la curva de aprendizaje de este entrenamiento para ver la confiabilidad que podríamos tener del modelo, puede verse a continuación:



En estas gráficas podemos observar una pequeña separación entre las curvas de aprendizaje, pero basta con observar las escalas de los datos en los ejes de accuracy y loss, para ver que no es malo el modelo.

Para cerciorarnos de que el modelo es bueno, adicionalmente usamos la función *evaluate* para que probara el modelo. Los resultados son los siguientes:

```
a = model.evaluate(x_test,y_test)
print("Precisión: ", a[1])
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.2503 - accuracy: 0.9338
Precisión:  0.9337999820709229
```

Confirmando el 93.38% de precisión.

Conclusiones del método

Una vez más, vemos que el modelo generado es muy bueno, desde el modelo inicial hasta el modelo final mejorado. En general, al ver que desde que se utilizó la arquitectura base, se tuvo la expectativa de que sería posible mejorar el modelo ya que podía apreciarse la facilidad con la que pudo aprender el 92% en un inicio.

Como se mencionó antes, por el hecho de que las redes convolucionales son muy utilizadas para aprendizaje aplicado a imágenes, es que se piensa que ocurrió esta facilidad para que este modelo consiguiera aprender de los datos.

Comparación general de los 2 modelos

Después de haber dedicado el tiempo necesario a este proyecto de aprendizaje profundo, podemos hablar sobre el desempeño que han tenido estos modelos. Ambos proyectos requirieron su tiempo para poder conseguir excelentes resultados. Podríamos decir que el tiempo que requirieron fue aproximadamente el mismo, pues más o menos fue de ese modo el tiempo que se les proporcionó. Por su parte, el más latoso podría considerar que fue el primer modelo, pues es menos complejo por el hecho de tener sólo 2 convoluciones. Esto ha hecho que el conseguir mejorar los resultados fueran más tardados.
El modelo investigado consiguió ser mejor entrenado que el proporcionado en clase, al ser un modelo que cuenta con 3 convoluciones consiguió aprender de los datos de una mejor manera.
De la misma manera, acorde a las métricas seleccionadas, en este caso, el accuracy, el segundo modelo convolucional alcanzó un total de 93.38%, consiguiendo ser el mejor resultado.
Algo que puede mencionarse respecto a los recursos que requirieron ambos modelos, es que en general, pareciera que no necesitaron de tanto, pues en su mayoría, conforme se entrenaba el modelo, el tiempo que le tomaba cada época fue muy corto.

Conclusiones

Sin duda alguna, un trabajo como este se debe reconocer, no por el tiempo dedicado, sino que permitió aprender un poco más sobre las redes neuronales, y particularmente, sobre las redes convolucionales.

Al ser ambos modelos redes convolucionales, la posible mejora que podrían tener es aumentar convoluciones o capas densas después del flatten. A su vez, como próximos avances, podría probarse con modelos más avanzados como las redes neuronales con memoria a corto plazo o quizás una GAN.

En esta ocasión, la red convolucional con tres convoluciones resultó victoriosa al ser la mejor red entre nuestros modelos, por ello, esta sería la red que elegiría para predecir.

Gracias a este proyecto pude aprender un poco más sobre la importancia de cada parte de una red neuronal, es decir, la importancia de una época, o de un batch size, o de seleccionar un adecuado optimizador, etc.

Lo que más me gustó de este proyecto fue el poder probar con modelos diferentes, permitiendo que podamos ver de una mejor manera cómo pueden variar nuestros resultados dependiendo de cómo ataquemos un problema.

Lo que menos me agradó de este proyecto fue el hecho de que este tipo de datasets busca una red que pueda aprender a identificar entre imágenes, por ende, lo más óptimo para atacar el problema, son redes convolucionales. Sería más interesante haber probado con otro tipo de dataset que al escucharlo no viniera a la mente una convolucional como único modelo.

Referencias bibliográficas

Gilabert, O. (2020). FASHION MNIST: Convolutional Neural Network (CNN). Recuperado de https://www.kaggle.com/code/oriolgilabertlpez/fashion-mnist-convolutional-neural-network-cnn