Universidade do Minho

Mestrado em Matemática e Computação

Classificadores e Sistemas Conexionistas

Trabalho Prático de Grupo

Eduardo Teixeira Dias (PG52249)

Hugo Filipe de Sá Rocha (PG52250)

Simão Pedro Batista Caridade Quintela (PG52257)

13 de maio de 2024

Conteúdo

Capítulo 1

Introdução

No âmbito da Unidade curricular Classificadores e Sistemas Conexionistas, do perfil de *Machine Learning* do Mestrado em Matemática e Computação da Universidade do Minho, foi realizado um projeto prático com o intuito de realizar um *benchmarking* de modelos de *Deep Learning* para visão por computador.

Para a realização dos mesmo foram selecionados dois *datasets*: o *dataset Fashion-MNIST* (atribuído pelos docentes) e um *dataset* relacionado com acordes de guitarra (criado por nós). Ambos os *datasets* serão discutidos com maior detalhe nos próximos capítulos.

1.1 Deep Learning e Computer vision

Deep Learning é um subconjunto de *Machine Learning* que utiliza redes neuronais com múltiplas camadas (*multi-layered neural networks*) para simular a capacidade de decisão e pensamento de um cérebro humano.

Estas são treinadas em conjuntos enormes de dados e com algoritmos de otimização, tal como o método do gradiente estocástico, por exemplo, para serem capazes de reconhecerem padrões, fazer predições e tomar decisões, entre outros.

Por outro lado, *Computer Vision* é uma área da inteligêngia artificial que utiliza algoritmos para tentar replicar as capacidades visuais dos seres humanos de forma a extrair informação de vídeos ou imagens digitais.

Isto inclui tarefas como identificação e classificação de objetos num vídeo, deteção de rostos em fotos e seguir o movimento de um objeto ao longo do tempo.

Para realizar estas tarefas são muitas vezes utilizadas técnicas de *Machine Learning* particularmente, e como é o caso deste projeto, *Deep neural networks*. Pois estas já demonstraram um sucesso considerável neste tipo de tarefas.

1.2 Objetivo principal

O objetivo principal deste projeto consiste averiguar qual dos diferentes modelos de *Deep Learning* aplicados obterá o melhor nível de performance para cada um dos datasets utilizados tendo em conta uma panóplia de hiperparâmetros para cada modelo e verificando o o desempenho dos mesmos através de *cross-validation*.

Capítulo 2

Benchmark - Fashion MNIST

2.1 Constituição da base de dados

A base de dados *Fashion-MNIST* é composta por imagens de peças de roupa, com dimensão 28x28, perfazendo cada imagem um total de 784 pixéis onde, cada um destes pixéis, varia no intervalo de inteiros [0,255] sendo que, um número mais próximo de 0 indica um pixél com uma cor mais clara e um número mais próximo de 255 indica um pixél mais escuro. A cada uma destas imagens está associada a respetiva *label* numérica (0 a 9) que nos indica qual é a peça de roupa em questão, ou seja, temos 10 tipos de roupa diferentes sendo elas: 'T-shirt', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag' e 'Ankle boot', estando associadas (pela respetiva ordem) aos números de 0 a 9.

Figura 2.1: Primeiras nove imagens do dataset Fashion-MNIST

2.2 Análise do *dataset*

Na análise do *dataset* procurámos analisar a dimensão dos dados de treino e de teste, bem como analisar a quantidade de cada um dos tipos de peça de roupa com vista a perceber se essas peças de roupa estavam equitativamente distribuídas ou não. Após extrair essa informação, percebemos que as peças de roupa estavam igualmente distribuídas, tanto os dados de treino como de teste. Nos dados de treino existem 60000 imagens, 6000 para cada peça de roupa e, na mesma toada, existem 10000 imagens nos dados de teste, 1000 para cada uma das peças de roupa. Além disso, procurámos imagens repetidas na base de dados ou *missing values* nos pixéis, não tendo encontrado nenhum deles. Dessa forma, o *dataset* encontrava-se perfeitamente equilibrado.

2.3 Modelos desenvolvidos

No que toca a modelos de redes neuronais, optámos por fazer 3 modelos diferentes, sendo eles: uma *Multi-Layer Perceptron* (MLP), uma *Convolutional Neural Network* (CNN) e uma *Long Short-Term Memory* (LSTM) que é uma arquitetura das *Recurrent Neural Networks* (RNN).

2.3.1 Multi-Layer Perceptron

Esta rede neuronal é composta por 4 camadas, sendo elas:

- **Flatten** - transforma o *input* de dimensão 28x28 num vetor unidimensional;
- **Dense** - camada com 64 neurónios com função de ativação variável;
- **Dense** - camada com 128 neurónios com função de ativação variável;
- **Dense** - camada com 10 neurónios de saída com função de ativação *softmax*;

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0
 dense (Dense)               (None, 64)                50240
 dense_1 (Dense)             (None, 128)               8320
 dense_2 (Dense)             (None, 10)                1290
=================================================================
Total params: 59850 (233.79 KB)
Trainable params: 59850 (233.79 KB)
Non-trainable params: 0 (0.00 Byte)
```

2.3.2 Convolutional Neural Network

Esta rede neuronal tem 12 camadas distribuídas por duas secções de microarquitetura, uma de *bottleneck* e a *layer* de saída.

Na primeira secção de microarquitetura temos:

- **Duas camadas Conv2D**, ambas com 32 filtros, tamanho 3x3 e função de ativação variável;
- **Uma camada MaxPooling 2D**, com *pooling* máximo de tamanho 2x2 e um *stride* de 2x2;
- **Uma camada Dropout**, com parâmetro 0.25 (25% dos neurónios são desativados em cada passo do treino);

Na segunda secção de microarquitetura temos:

- **Duas camadas Conv2D**, ambas com 64 filtros, tamanho 3x3 e função de ativação variável;
- **Uma camada MaxPooling 2D**, com *pooling* máximo de tamanho 2x2 e um *stride* de 2x2;;

- **Uma camada Dropout**, com parâmetro 0.25 (25% dos neurónios são desativados em cada passo do treino);

Na secção de *bottleneck* temos:

- **Uma camada Flatten**, para transformar as imagens num vetor unidimensional;
- **Uma camada Dense**, com 128 neurónios e função de ativação variável;
- **Uma camada Dropout**, com parâmetro 0.50 (50% dos neurónios são desativados em cada passo do treino);

Por fim, temos uma camada de saída com 10 neurónios de *output* com função de ativação *softmax*.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320
 conv2d_1 (Conv2D)           (None, 28, 28, 32)        9248
 max_pooling2d (MaxPooling2  (None, 14, 14, 32)        0
 D)
 dropout (Dropout)           (None, 14, 14, 32)        0
 conv2d_2 (Conv2D)           (None, 14, 14, 64)        18496
 conv2d_3 (Conv2D)           (None, 14, 14, 64)        36928
 max_pooling2d_1 (MaxPoolin  (None, 7, 7, 64)          0
 g2D)
 dropout_1 (Dropout)         (None, 7, 7, 64)          0
 flatten (Flatten)           (None, 3136)              0
 dense (Dense)               (None, 128)               401536
 dropout_2 (Dropout)         (None, 128)               0
 dense_1 (Dense)             (None, 10)                1290
=================================================================
Total params: 467818 (1.78 MB)
Trainable params: 467818 (1.78 MB)
Non-trainable params: 0 (0.00 Byte)
```

2.3.3 Long Short-Term Memory

Por último foi construida uma RNN (Recurrent Neural Network). Mais especificamente uma LSTM (Long Short-Term Memory), uma arquitetura desenvolvida para colmatar os problemas das RNNs tradicionais.

A LSTM criada é composta por 3 camadas, sendo elas:

- **Uma camada Reshape**, para remodelar os dados de entrada para a forma 28x28;
- **Uma camada LSTM**, com 128 neurónios e função de ativação variável;
- **Uma camada de saída, Dense**, com 10 neurónios;

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 reshape (Reshape)           (None, 28, 28)            0
 lstm (LSTM)                 (None, 128)               80384
 dense (Dense)               (None, 10)                1290
=================================================================
Total params: 81674 (319.04 KB)
Trainable params: 81674 (319.04 KB)
Non-trainable params: 0 (0.00 Byte)
```

2.4 Otimização dos hiperparâmetros *(tuning)*

Depois de criados os nossos modelos decidimos otimiza-los em função de 5 hiperparâmetros:

- **Número de *epochs*** - número de iterações completas do *dataset* de treino;
- **Função de ativação** - função que transforma o *input* de um nodo num *output*, de forma a introduzir não-linearidade ao modelo;
- ***Data augmentation*** - técnica utilizada para fazer variar os dados de um *dataset* através da criação de versões modificadas dos mesmos.
- ***Batch size*** - Número de observações dos dados que são utilizados numa iteração (equivalente a um *forward pass* e um *backward pass*) no processo de treino de uma rede neuronal;
- ***Learning Rate*** - hiperparâmetro que determina o tamanho do "passo"dado durante o processo de otimização, de forma a definir quão depressa o modelo aprende com os dados de treino, ;

Para tal foi utilizada a técnica de *Grid Search* a qual testa exaustivamente todos os valores de hiperparâmetros, neste caso, 2 valores para cada um dos 5 hiperparâmetros indicados acima que nós fornecemos numa "grelha"de forma a avaliar o desempenho de todas as combinações possíveis e determinar a combinação ótima (Tabela 2.1). Uma para cada modelo de *Deep Learning* criado (MLP, CNN, LSTM)

Tabela 2.1: Tabela dos Hiperparâmetros

Hiperparâmetros	**Valores**
Número de epochs	5, 10
Função de ativação	relu, sigmoid
Data augmentation	True, False
Batch size	64, 128
Learning Rate	0.001, 0.01

2.5 Cross-Validation

Simultaneamente fomos realizando *cross-validation*, mais especificamente a metodologia *k-folds cross-validation*, de forma a realizar uma validação de cada modelo. Esta divide os nossos dados em k subgrupos. Seguidamente, o modelo com determinada combinação de hiperparâmetros é treinado k vezes usando um dos subgrupos como dados de teste e os restantes como dados de treino, sendo que a cada uma das k iterações um subgrupo diferente dos dados é utilizado como dados de teste e os restantes como dados de treino até todos os subgrupos terem sido utilizados como dados de teste uma vez.

No final, é calculada a média da *accuracy* obtida para cada um dos modelos treinados de forma a obter uma avaliação mais precisa e confiável da performance do modelo do que se tivessemos utilizado apenas um único grupo de treino e teste, para além de também permitir mitigar o risco de *overfitting*.

Por convenção os valores utilizados para k costumam ser entre 5 e 10. Tendo em conta o tamanho do nosso *dataset* e de forma a ter uma eficácia computacional superior, optamos por utilizar o valor mínimo desse intervalo, ou seja, k=5.

2.6 Benchmarking dos modelos

2.6.1 Multi-Layer Perceptron

epochs	batch_size	data_aug	activation_function	learning_rate	mean_score	time
10	64	False	relu	0.001	0.917	1:33
10	128	False	relu	0.001	0.913	0:51
10	64	False	sigmoid	0.001	0.909	1:35
10	128	False	sigmoid	0.001	0.903	0:52
5	64	False	relu	0.001	0.901	0:40
5	128	False	relu	0.001	0.898	0:30
5	64	False	sigmoid	0.001	0.898	0:41
5	128	False	sigmoid	0.001	0.892	0:32
10	128	False	sigmoid	0.01	0.891	0:53
10	128	False	relu	0.01	0.885	0:49
5	128	False	sigmoid	0.01	0.884	0:32
10	64	False	sigmoid	0.01	0.878	1:42
5	128	False	relu	0.01	0.877	0:29
5	64	False	sigmoid	0.01	0.872	0:41
5	64	False	relu	0.01	0.871	0:41
10	64	False	relu	0.01	0.869	1:25

Figura 2.2: 16 melhores combinações de hiperparâmetros do modelo MLP

Uma curiosidade ao avaliar todos os modelos testados é que os modelos sem *data augmentation*, todos eles, apresentam uma *accuracy* bastante superior aos modelos que aplicavam *data augmentation*. O *learning rate* a 0.001 também se superiorizou ao *learning rate* a 0.01. O modelo com maior *accuracy* que implementa *data augmentation* apresentou uma percentagem de acerto média de 0.79 que são cerca de 8 pontos percentuais abaixo, imagine-se, do modelo mais "fraco"que não implementa *data augmentation*. Portanto, a maior conclusão que se tira destes resultados, é que *data augmentation* não traz vantagens neste *dataset* e nesta *rede neuronal* bem como o *learning rate* a 0.01.

Nota: Os tempos apresentados foram obtidos através da execução do código com o CPU.

2.6.2 Convolutional Neural Network

1	epochs	batch_size	data_aug	activation_function	learning_rate	mean_score	time
2	10	64	False	relu	0.001	0.932	14:30:00
3	10	128	False	relu	0.001	0.930	10:53:00
4	5	64	False	relu	0.001	0.918	08:30:00
5	5	128	False	relu	0.001	0.917	05:54:00
6	10	64	False	sigmoid	0.001	0.882	18:09:00
7	5	128	False	relu	0.01	0.861	05:51:00
8	10	64	False	relu	0.01	0.856	12:26:00
9	5	64	False	relu	0.01	0.846	07:27:00
10	10	128	False	relu	0.01	0.840	10:50:00
11	10	128	True	relu	0.001	0.832	13:52:00
12	10	64	True	relu	0.001	0.822	14:42:00
13	5	64	False	sigmoid	0.001	0.793	08:18:00
14	5	64	True	relu	0.001	0.784	08:42:00
15	5	128	True	relu	0.001	0.775	07:43:00

Figura 2.3: 15 melhores combinações de hiperparâmetros do modelo CNN

Dos três modelos de Deep Learning criados a CNN foi aquela que obteve a melhor accuracy, sendo que com a combinação dos seguintes hiperparâmetros: 10 epochs, batch size de 64, sem data augmentation, com a função relu como função de ativação, com uma learning rate de 0.001 foi obtida uma accuracy média para as 5 iterações do cross-validation de 93.2%.

Tendo em conta que este era o modelo mais complexo, tendo o maior número de camadas, era expectável este resultados. Porém, isso também se ressentiu no seu tempo de execução tendo demorado consideravelmente mais que a CNN mesmo tendo corrido o código no Colab e na GPU.

2.6.3 Long Short-Term Memory

1	epochs	batch_size	data_aug	activation_function	learning_rate	mean_score	time
2	10	128	False	relu	0.001	0.913	11:34
3	10	128	False	sigmoid	0.01	0.907	12:06
4	10	64	False	sigmoid	0.01	0.903	15:51
5	5	64	False	relu	0.001	0.901	10:14
6	5	128	False	relu	0.001	0.898	5:59
7	5	64	False	sigmoid	0.01	0.895	8:06
8	5	128	False	sigmoid	0.01	0.89	6:13
9	10	64	False	sigmoid	0.001	0.885	15:45
10	10	128	False	sigmoid	0.001	0.87	12:05

Figura 2.4: 10 melhores combinações de hiperparâmetros do modelo LSTM

O modelo que obteve melhor accuracy foi o modelo em que utilizamos a função relu como função de ativação, um learning rate de 0.001, um batch size de 128, 10 epochs e sem utilizar data augmentation, tendo obtido uma accuracy média entre as 5 iterações do cross-validation de 91.3%.

Tal como nos modelos de Deep Learning anteriores, os modelos em que foi utilizada data augmentation não obtiveram grandes resultados, sendo que no caso da LSTM nenhum modelo com data augmentation entrou sequer no top 10 dos melhores modelos.

Nota: Os tempos apresentados foram obtidos através da execução do código com o CPU.[1]

[1] É de salientar que os códigos do tuning dos 3 modelos acima apresentados foram executados em computadores diferentes, não se devendo portanto utilizar os tempos de execução como métrica de comparação precisa.

Capítulo 3

PrediChord

3.1 Contextualização

Na segunda parte do projeto optamos por conceber o *PrediChord* que é um modelo que classifica acordes musicais, dada uma imagem de uma pessoa a tocar guitarra.

Para isso, utilizamos 6 acordes básicos, sendo eles o **Am** (Lá Menor), **G** (Sol), **Em** (Mi Menor), **F** (Fá), **C** (Dó), **D** (Ré).

3.2 Dataset

Neste projeto, uma das maiores dificuldades encontradas foi precisamente a escolha de um bom *dataset*, visto que há poucos relativos a música e que envolvam imagem.

Tomamos portanto a decisão de criar o nosso próprio *dataset*. Para isso, extraímos vídeos da internet em que pessoas estavam a tocar apenas um acorde, partimos em frames e utilizamos o facto de sabermos que apenas estava a ser tocado um acorde para realizar o *labeling* das imagens. Ao todo, temos cerca de **768 imagens** com dimensão 1280x720, divididas entre os 6 acordes selecionados.

Para o dataset de teste, temos 4 imagens por acorde que não pertencem ao dataset. No total são 24 acordes que estamos a tentar prever.

Abaixo seguem alguns exemplos das imagens utilizadas:

(a) (b)

Figura 3.1: Acorde Am

3.3 Processamento dos dados

3.3.1 One Hot Encoding

Como o objetivo deste trabalho era utilizar redes neuronais artificiais e as nossas labels eram strings, optamos por realizar ***one hot encoding*** de forma a facilitar a inserção de dados na rede.

Para isso, utilizamos o seguinte dicionário:

```
one_hot_encoding = {
    'Am' : [1, 0, 0, 0, 0, 0],
    'C'  : [0, 1, 0, 0, 0, 0],
    'D'  : [0, 0, 1, 0, 0, 0],
    'Em' : [0, 0, 0, 1, 0, 0],
    'F'  : [0, 0, 0, 0, 1, 0],
    'G'  : [0, 0, 0, 0, 0, 1],
}
```

3.3.2 Manipulação de imagem

Relativamente à forma como processamos os nossos dados, optamos por algumas abordagens diferentes.

Inicialmente, trabalhamos com as imagens de dimensão 1280x720, porém, ao correr o modelo notamos que as mesmas tinham muito ruído de fundo, isto é, cores e objetos que não eram relevantes para a nossa *predict* do acorde. Por isso, optamos por fazer um *script* que recortava todas as imagens do *dataset* para a dimensão 1280x450. Com isto, conseguimos ganhar 15 minutos de eficiência a correr o modelo.

Para além do recorte das imagens, também testamos a opção das imagens estarem apresentadas apenas com 1 canal de cor, isto é, numa escala de cinzento. Porém, os resultados obtidos foram bastante fracos, em que tivemos 0% de *accuracy* na classificação de 12 imagens distintas e, por isso, apenas consideramos o *dataset* com imagens recortadas.

3.4 Arquitetura da CNN

Para prever qual o acorde dado como input, optamos por construir uma CNN - *Convolutional Neural Network*.

Optamos por utilizar uma CNN com 5 camadas, quatro convolucionais e uma *fully-connected*. Nas quatro camadas convolucionais utilizamos 32, 64, 128 e 256 neurónios enquanto que na *fully-connected* optamos por utilizar 128 neurónios. Como output temos uma camada com 6 neurónios devido ao facto de termos 6 classes possíveis.

Optamos por utilizar ***padding*** em todos os neurónios convolucionais de forma a não perder informação das imagens e a **função de ativação** usada foi a **Relu** de forma a termos não linearidade no flow dos dados.

O tamanho dos filtros utilizados foi 3x3. Porém, na *layer* onde realizamos ***MaxPooling*** optamos por utilizar dimensão 2x2 no ***pool size*** e nas ***strides***.

Podemos ver abaixo o resultado de executar o comando ***model.summary()***.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 450, 1280, 32)     896
 conv2d_9 (Conv2D)           (None, 450, 1280, 32)     9248
 max_pooling2d_4 (MaxPoolin  (None, 225, 640, 32)      0         g2D)
 dropout_5 (Dropout)         (None, 225, 640, 32)      0
 conv2d_10 (Conv2D)          (None, 225, 640, 64)      18496
 conv2d_11 (Conv2D)          (None, 225, 640, 64)      36928
 max_pooling2d_5 (MaxPoolin  (None, 112, 320, 64)      0         g2D)
 dropout_6 (Dropout)         (None, 112, 320, 64)      0
 conv2d_12 (Conv2D)          (None, 112, 320, 128)     73856
 conv2d_13 (Conv2D)          (None, 112, 320, 128)     147584
 max_pooling2d_6 (MaxPoolin  (None, 56, 160, 128)      0         g2D)
 dropout_7 (Dropout)         (None, 56, 160, 128)      0
 conv2d_14 (Conv2D)          (None, 56, 160, 256)      295168
 conv2d_15 (Conv2D)          (None, 56, 160, 256)      590080
 max_pooling2d_7 (MaxPoolin  (None, 28, 80, 256)       0         g2D)
 dropout_8 (Dropout)         (None, 28, 80, 256)       0
 flatten_1 (Flatten)         (None, 573440)            0
 dense_2 (Dense)             (None, 128)               73400448
 dropout_9 (Dropout)         (None, 128)               0
 dense_3 (Dense)             (None, 6)                 774
=================================================================
Total params: 74573478 (284.48 MB)
Trainable params: 74573478 (284.48 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

3.5 Treino do modelo

Para treinar o modelo optamos por utilizar algumas estratégias e obtemos resultados significativamente diferentes em algumas delas.

O código correspondente ao treino do modelo é o seguinte:

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001),
    loss=tf.keras.losses.categorical_crossentropy,
    metrics=['accuracy']
)
validation_split = 0.05
split_index = int(len(x_train) * (1 - validation_split))
x_val = x_train[split_index:]
y_val = y_train[split_index:]
x_train = x_train[:split_index]
y_train = y_train[:split_index]
if not apply_data_augmentation:
    print('No data augmentation')
    history = model.fit(
        x=x_train,
        y=y_train,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(x_val, y_val),
        shuffle=True,
        callbacks=[callbacks]
    )
else:
    print('Using data augmentation')
    datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rotation_range=90,
        zoom_range=0.,
        horizontal_flip=False,
        vertical_flip=True,
        rescale=None,
        preprocessing_function=None
    )
    datagen.fit(x_train)
    history = model.fit(
        datagen.flow(
            x=x_train,
            y=y_train,
            batch_size=batch_size
        ),
        epochs=epochs,
        validation_data=(x_val, y_val),
        workers=1,
        callbacks=[callbacks]
    )
return model, history
```

Como podemos ver, o **otimizador** escolhido foi o **Adam**. Optamos também por duas estratégias no treino que é a presença ou ausência de ***Data Augmentation***. Para a função de *loss* optamos pela ***Categorical Crossentropy*** e **validamos** os nossos dados com 5% do *dataset*.

3.6 Tuning do modelo

3.6.1 Sem Data Augmentation e Batch Size = 64

Inicialmente, treinamos o nosso modelo **sem utilizar *Data Augmentation***, utilizamos um ***Batch Size*** igual a **64** e as imagens foram processadas a cores durante **10 épocas**. Os resultados obtidos foram os seguintes:

```
1/1 [==============================] - 2s 2s/step - loss: 2.7294 - accuracy: 0.2500
Evaluation Loss:  2.7294304370880127
Evaluation Accuracy:  0.25
```

Figura 3.2: Avaliação do modelo

```
1/1 [==============================] - 2s 2s/step
Predictions shape:  (24, 6)
Predicted a G Real value is D
Predicted a D Real value is D
Predicted a Am Real value is D
Predicted a G Real value is D
Predicted a G Real value is C
Predicted a Am Real value is C
Predicted a G Real value is C
Predicted a Am Real value is C
Predicted a D Real value is Em
Predicted a D Real value is Em
Predicted a G Real value is Em
Predicted a Am Real value is Em
Predicted a Em Real value is G
Predicted a G Real value is G
Predicted a G Real value is G
Predicted a G Real value is G
Predicted a Am Real value is Am
Predicted a G Real value is Am
Predicted a G Real value is Am
Predicted a Am Real value is Am
Predicted a D Real value is F
Predicted a G Real value is F
Predicted a D Real value is F
Predicted a G Real value is F
```

Figura 3.3: Previsões geradas pelo modelo

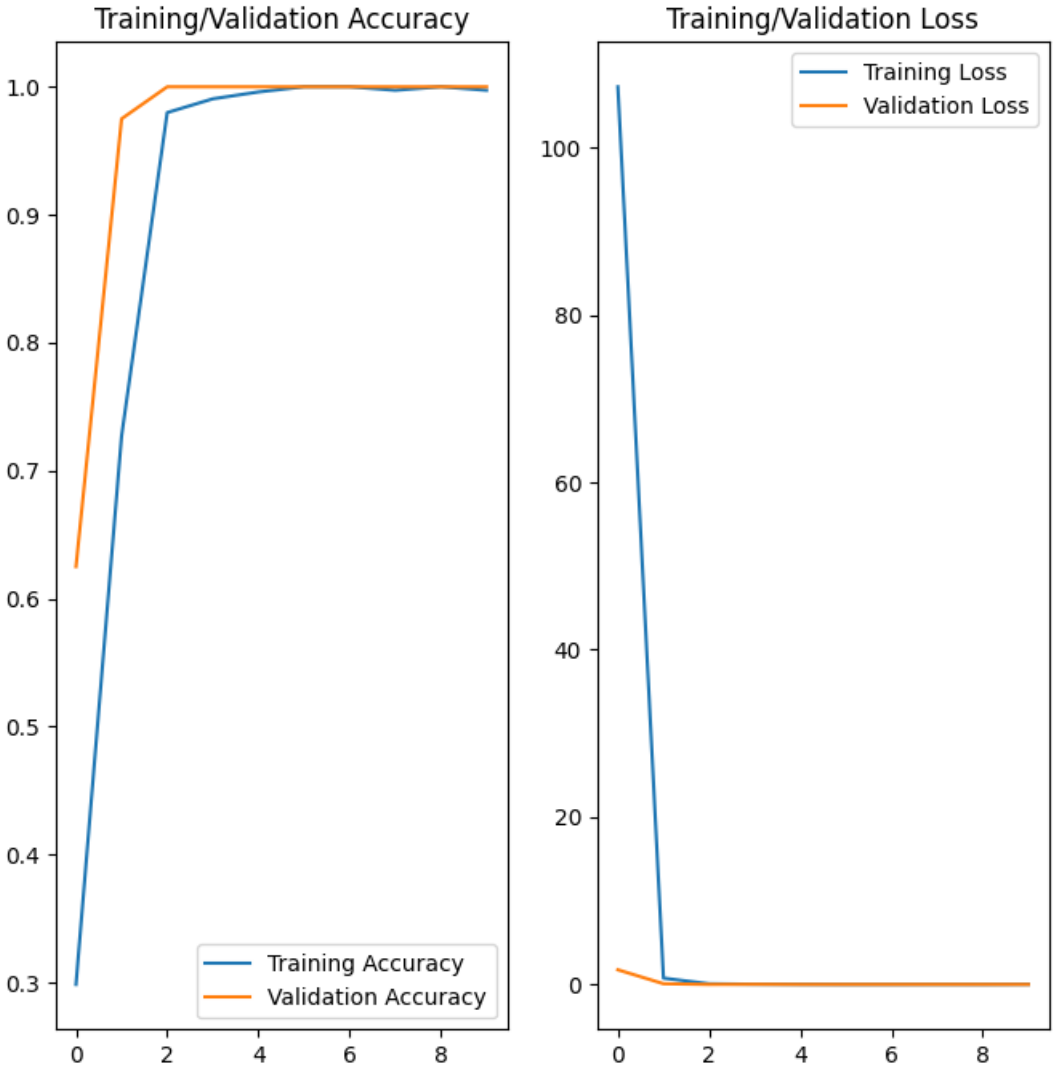

Figura 3.4: Evolução do treino da rede

Como se pode observar pelas figuras, os resultados não foram bons. A *accuracy* do modelo para os dados de teste foi de 25%, o que significa que em 24 imagens, apenas acertou em 6.

Para além disso, nota-se também que o mesmo está extremamente *overfitted* visto que no treino apontava para um valor muito próximo dos 100% de *accuracy* mas isso não se reflete nos valores de teste.

3.6.2 Com Data Augmentation e Batch Size = 64

Após verificarmos que o modelo estava *overfitted*, optamos por aplicar *Data Augmentation*, com os mesmos parâmetros do modelo anterior. Os resultados obtidos foram os seguintes:

```
1/1 [==============================] - 2s 2s/step - loss: 1.8226 - accuracy: 0.1250
Evaluation Loss:  1.8226102590560913
Evaluation Accuracy:  0.125
```

Figura 3.5: Avaliação do modelo

```
1/1 [==============================] - 2s 2s/step
Predictions shape:  (24, 6)
Predicted a Em Real value is G
Predicted a F Real value is G
Predicted a Em Real value is G
Predicted a F Real value is G
Predicted a Am Real value is C
Predicted a F Real value is C
Predicted a F Real value is C
Predicted a Em Real value is C
Predicted a Em Real value is Am
Predicted a Em Real value is Am
Predicted a F Real value is Am
Predicted a Em Real value is Am
Predicted a F Real value is F
Predicted a G Real value is F
Predicted a F Real value is F
Predicted a G Real value is F
Predicted a Em Real value is Em
Predicted a G Real value is Em
Predicted a Am Real value is Em
Predicted a G Real value is Em
Predicted a F Real value is D
Predicted a Em Real value is D
Predicted a Am Real value is D
Predicted a G Real value is D
```

Figura 3.6: Previsões geradas pelo modelo

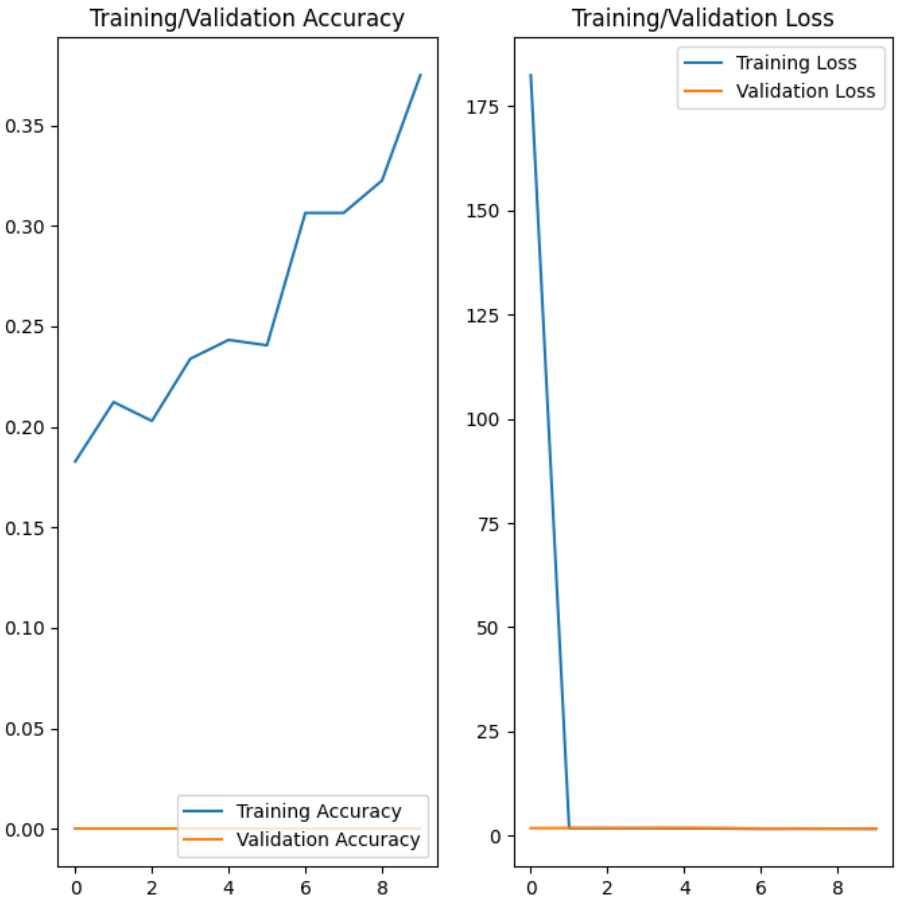

Figura 3.7: Evolução do treino da rede

Após correr este modelo verificamos que o overfitting tinha diminuído bastante porém a performance do modelo foi bastante má. Notamos que época após época a *accuracy* subia, porém o mesmo não acontece com a *accuracy* relativa aos dados de validação que se manteve constantemente em 0.

Capítulo 4

Conclusão

No que diz respeito ao dataset de Fashion-MNIST aplicamos 3 modelos de Deep Learning diferentes, os quais, do nosso ponto de vista, obtiveram todos bons resultados, sendo que o obteve melhor accuracy foi a CNN.

Os modelos em que utilizamos data augmentation foram aqueles que obtiveram os piores resultados, levando-nos a crer que esta técnica não é apropriada para este dataset.

Relativamente ao dataset de acordes musicais tiramos algumas considerações a ter em conta.

Primeiramente, concluímos que o grande problema deste projeto está no dataset utilizado. O facto das imagens conterem bastante ruído de fundo podem estar a interferir com o bom desempenho do modelo. Para além disso, uma grande dificuldade que tivemos foi o facto das imagens serem todas de alta dimensão, o que levou a que o tempo de treino fosse demasiado elevado (1 hora de treino para 10 épocas).

Pensamos que a abordagem de classificação direta de imagem pode não ter sido o mais acertado, uma vez que apenas uma pequena fração da imagem é que nos interessava para a classificação da mesma.

Numa nota final, acreditamos que este trabalho nos proporcionou muitas aprendizagens e novas competências, nomeadamente no campo de Computer Vision e Deep Learning.

Bibliografia

[1] Rede Neuronal Convolucional, https://www.tensorflow.org/tutorials/images/cnn?hl=pt-br

[2] Aumento de dados, https://www.tensorflow.org/tutorials/images/data_augmentation?hl=pt-br

[3] Working with RNNs, https://www.tensorflow.org/guide/keras/working_with_rnns

[4] A. Pimenta, D. Carneiro, J. Neves, and P. Novais. "A Neural Network to Classify Fatigue from Human-Computer Interaction."Neurocomputing. 172 (2016), pp. 413–426.

[5] LSTM Tensorflow, https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM

[6] B. Fernandes, F. Silva, H. Alaiz-Moretón, P. Novais, J. Neves, and C. Analide. "Long Short-Term Memory Networks for Traffic Flow Forecasting: Exploring Input Variables, Time Frames and Multi-Step Approaches."INFORMATICA. 31.4 (2020), pp. 723–749.

[7] Hyperparameters tuning, https://www.geeksforgeeks.org/hyperparameter-tuning/

[8] N. Majumder, S. Poria, A. Gelbukh, and E. Cambria. "Deep Learning-Based Document Modeling for Personality Detection from Text."IEEE Intelligent Systems. 32.2 (2017), pp. 74–79.

[9] Cross-Validation, https://scikit-learn.org/stable/modules/cross_validation.html