ADNE
Segmentação de Batimentos Cardíacos
Segundo Módulo

João Funenga

May 2022

1 Introdução

Neste trabalho terei como objetivo analisar dados relativos a batimentos cardíacos e segmentá-los, isto é, para um sinal cardíaco, para cada um dos momentos perceber a qual das fases do batimento cardíaco nos encontramos. Isto traduz-se em existirem 4 classes correspondentes a estas 4 fases (S1, Sístole, S2 e Diástole). Em geral, os sinais correspondem a um registo sonoro a 4000Hz, ou seja, existem 4000 medições a cada segundo. São dados 3 *datasets*, de treino, validação e teste. Os modelos serão treinados e será verificada a evolução tanto do erro de treino como o de validação e no final, o modelo treinado será testado nos dados de teste nunca antes visto para termos uma estimativa não enviesada do verdadeiro erro do modelo (o mesmo para a *accuracy*). Cada ficheiro tem tanto os valores do sinal propriamente dito (na primeira linha), como os das verdadeiras classes correspondentes, tomando os valores de 1 a 4 das classes. Isto servirá para podermos fazer a avaliação do modelo visto ser um problema de aprendizagem supervisionada.

2 Objetivos

O propósito deste trabalho será o de fazer a **segmentação dos sinais usando varios tipos de rede** e compará-las para verificarmos qual nos dá um melhor resultado. A performance das várias redes será avaliada recorrendo à precisão das previsões de cada um dos modelos na segmentação de apenas um som individual (ao invés de vários em batch). Isto porque, devido aos sinais terem tamanhos variados e estarmos a fazer um *padded_batch* (preenchendo com zeros), os sinais seriam preenchidos com zeros e caso o modelo prevesse sempre a classe 0, teria uma *accuracy* alta mesmo estando na teoria a fazer "nada".

Caso façamos a avaliação a um batch com apenas 1 sinal (*batch_size = 1)*, este não será preenchido com zeros e assim a avaliação já será precisa visto que o que está a ser analisado é o próprio sinal e não apenas uma parte do sinal e o resto com zeros para preencher. Relativamente a todos os modelos, devido aos sinais poderem ter tamanhos diferentes, comecei por resolver o problema cortando todos os dados pelo tamanho do sinal mais pequeno e, depois de esta abordagem funcionar, fazer a implementação de uma versão mais flexível que permite receber como input sinais com tamanhos diferentes. Isto serve apenas de comentário visto todos os modelos apresentados de seguida serem a versão que recebe sinais de tamanho diferente excepto a feed-forward que precisa de receber como input inputs de tamanho fixo (cuja abordagem será explicada na próxima secção).

3 Aspetos Comúns entre os modelos

Relativamente à preparação dos sinais, este é um aspeto importante a falar. Foi feita uma sequência de funções *lambda* para os ler e "incializar". Primeiramente, precisamos de carregar os ficheiros e estes precisam de ser lidos como *float*. Para realizarmos isto, usaremos uma função criada `read_npy` e tal será aplicada a todos os ficheiros de treino que temos usando a função `map`.

Devido aos dados serem medições discretas dos batimentos cardíacos, usaremos a transformada de *fourier* de curta duração destes de modo a suavizar e fazer as interpolações dos dados. Esta é chamada de curta duração pois funciona aplicando a transformada de *fourier* a intervalos do nosso sinal. Foi escolhido que houvesse sobreposição de metade para poderem ser detados acontecimentos mais facilmente vindo daqui que o tamanho para a o intervalo da transformada é de 256 e o *stride* (o que desliza) de 128 (metade do tamanho do intervalo) (para efeitos de teste, em algumas das redes utilizei 64 de tamanho da janela e 32 de stride que embora seja mais lento, obtém resultados melhores para a *accuracy*). O resultado disto é uma sequência de transformadas de *Fourier* (uma matriz porque como cada transformada de *fourier* é um vetor, a *short time fourier transform* é um sinal de duas dimensões). O principal objetivo do uso desta transformada é pela redução da dimensão

do sinal o que, numa rede *feed forward* com muitos parâmetros em jogo, nota-se uma diferença drástica no tempo de treino destas.

4 Feed-Forward Neural Network

Para a rede *feed forward* usou-se a versão sequencial do keras. Esta rede terá a seguinte arquitetura:

- 1 Camada de *input*
- 2 camadas densas com 64 neurónios
- 1 camada densa com 1000 neurónios (reshaped para (200,5))
- 1 camada de *upsampling* com fator = 10
- 1 Camada de output com ativação *softmax*

Como *input* para treinar a rede, esta receberá o *dataset* constituído por vários sinais. De notar que a rede *feed forward* necessita de inputs de tamanho fixo e igual logo precisamos de cortar os sinais que utilizamos para treinar por um determinado tamanho. No entanto, caso cortemos todos os sinais desde o início até esse determinado tamanho, o nosso modelo só irá conhecer os inicios dos sinais que podem ser todos parecidos e nunca chegar a ver sinais que contenham os batimentos cardíacos noutra fase. Logo, nós queremos treinar a rede olhando para diferentes porções dos sinais para que possa "aprender" mais ao ver mais exemplos diferentes, isto é, se nunca vir algo da classe 3, não irá prever corretamente caso observe um segmento do sinal que pertenca à classe 3. Devido a isto, decidi fazer um corte numa secção aleatória do sinal para que consiga olhar para segmentos em zonas diferentes dos sinais e assim maximizar a diversidade observada.

As duas primeiras camadas densas com 64 neurónios servem para o modelo identificar e definir as relações entre os valores dos dados e em teoria se tivermos mais, introduzimos mais não linearidade e podemos aumentar o poder de classificação mas caso adicionemos demais, o modelo adaptar-se-á demasiado aos dados e terá pouco poder de generalização, daí apenas ter usado 2. Entre estas decidi usar a função de ativação *leaky relu* de forma a evitar o problema de *vanishing gradients* e por ser rápida de calcular (e introduzir não linearidade, o mais importante). Também decidi usar uma camada de *Batch Normalization* antes de passar os dados para a próxima camada de modo a "centrar" os dados e para que a camada seguinte não tenha o trabalho de repetir este trabalho. Isto aumentará a velocidade do treino da rede e estabiliza a média e o desvio padrão para todos os *mini-batches*.

De seguida, por este ser um problema que pode alcançar dimensões muito elevadas relativamente ao número de parâmetros, decidi usar uma camada densa com 1000 neurónios para condensar tudo, e fazer o *reshape* (200, 5) para ficarmos com a forma correta e de seguida fazer um *upsampling* com fator = 10 para aumentarmos o número de parâmetros de novo para 10 vezes mais. Esta alteração acelera o processo porque ajuda a reduzir o número de parâmetros com que estamos a trabalhar [1].

Para a última *layer*, por ser um problema de classificação (multi-class) em que um ponto apenas corresponde a uma classe dos batimentos cardiacos, a função de ativação usada foi a *softmax* por corresponder a uma probabilidade para cada uma das classes existentes e cuja soma destas probabilidades todas dá 1. Relativamente à função de erro a ser utilizada, por ser um problema de *multi-class classification*, a escolha lógica será a *sparse categorical cross entropy*. Relativamente ao *stride* usado ao fazer a transformada de *Fourier*, este foi de 128 e não foi alterado por já apresentar bons resultados mas, em troca de alguma performance, caso este valor fosse reduzido para um quarto, os resultados seriam ligeiramente melhores.

4.1 Análise dos Resultados - Treino e Validação

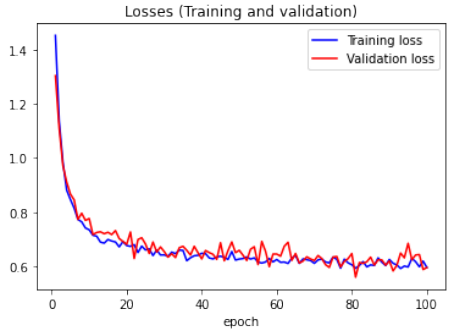
Para verificar agora a qualidade do modelo antes de o aplicar aos dados de teste usarei os dados de treino e validação para ver como evoluem a *loss* e a *accuracy* tanto de treino como de validação. Este foi treinado com o *optimizer*

Adam e com a *Learning Rate default* visto que o SGD demora bastante a convergir. O modelo foi treinado para 100 épocas e para *batches* de 64 exemplos de modo a aumentar também a velocidade de treino ao invés de *batches* mais pequenos.

Como podemos ver pelos gráficos, é visível que o *jitter* no treino é baixo o que é natural por ir estabilizando ao longo do tempo, embora ainda exista algum, provavelmente por ter usado o *Adam* como optimizer que é *greedy* ao invés do SGD que não o é mas que demora mais a convergir. Relativamente ao *jitter* presente no conjunto de validação isto é normal por serem dados que o modelo não viu antes e assim acaba por demorar mais a convergir para um valor. Relativamente à *accuracy* vemos que os valores, embora tenham *jitter*, até parecem coincidir sempre com os de treino o que mostra que não existiu *overfit* aos dados de treino com o modelo.

4.2 Previsão - Teste

O objetivo da rede é segmentar o sinal para ver em qual das fases estamos em cada instante para um sinal. No entanto, como explicado acima, a rede *feed-forward* só lida com *inputs* do mesmo tamanho logo é preciso "partirmos" o sinal em pedaços de igual tamanho e o que a rede receberá é um *batch* contendo estes segmentos. No entanto, dependendo do tamanho de



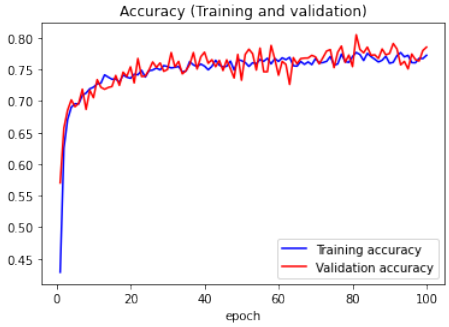


Figure 1: Losses e Accuracy - Feed Forward

cada um dos sinais, esta separação em segmentos de tamanho N (que escolhi 2000) pode causar com que o último segmento não perfaça o tamanho N e neste caso é preciso preencher com zeros (embora a diferença nunca seja muito grande e é apenas no último segmento logo não afetará a capacidade de previsão do modelo). De seguida, como o modelo foi treinado com as transfomadas *fourier* do sinal, é necessário também aplicar esta operação a cada um dos segmentos que temos e, com este resultado, criar um batch com todos estes segmentos de um sinal para o modelo fazer a previsão. O que resulta desta previsão são os resultados para cada segmento logo agora queremos "desenrolar" para podermos comparar com os valores reais e termos uma medida de *accuracy*. Neste passo, iremos remover a diferença dos zeros que adicionámos antes de utilizarmos o modelo com os valores verdadeiros para terem o mesmo comprimento e poderem ser comparados. Tendo tudo isto feito, fiz estas operações explicadas em cima para cada um dos ficheiros de teste e fiz a média das *losses* e das *accuracies* para ter uma estimativa média final para estes valores. Usei a função do *keras* `evaluate` para ter a estimativa da *loss* e *accuracy* para os dados de teste sendo estas respetivamente 0.624 e 0.768. Para exemplificar o uso da função extra que recebe um array `numpy` correspondente ao sinal e devolve um array com a segmentação do mesmo, abaixo comparo o sinal com a segmentação prevista que parece ter sido bem feita.

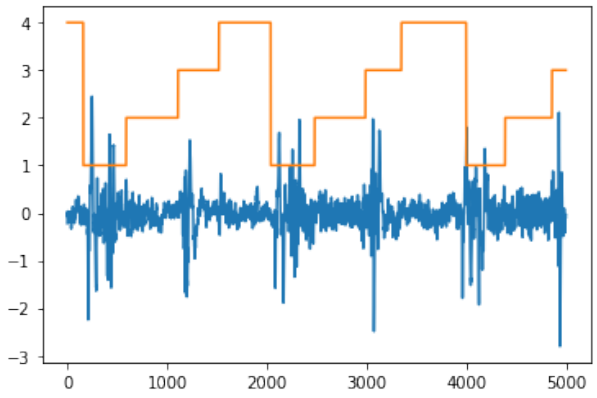


Figure 2: Segmentação de um sinal - Feed Forward

5 Rede Convolucional

Em oposição à rede *feed-forward*, as convolucionais têm a vantagem de ter bons resultados sem ter de estarmos a fazer grande *feature engineering*. Estas *features* são extraídas por cada camada convolucional logo faz sentido termos algumas camadas convolucionais 1D. Assim, a arquitetura da rede foi a seguinte:

- 2 Camadas para o input
- 4 Camadas Convolucionais (64, 64, 64 e 128 neurónios e filtro = 5, com *padding = same*)
- 1 Camada *TimeDistributed* com Densa com 35 neurónios (reshaped para (-1,5))
- 1 Camada *UpSampling* com fator = 78
- 1 Camada de *output* com função de ativação *softmax*

Para esta rede usarei 4 camadas de convolução e, para introduzir não linearidade, usarei a função de ativação *Leaky Relu* pela mesma razão que na rede anterior, ser rápida e fazer o necessário [2]. Inicialmente comecei por verificar se existia *overfit* do modelo por estar a usar várias camadas e caso existisse usaria `SpatialDropout1D` para tentar minimizar isto ao máximo. De facto como podemos ver pelo gráfico abaixo houve algum *overfit* no final mas, ao introduzir o dropout a segmentação do sinal ficava bastante má (sempre a prever 4's, por isso não utilizei).

Depois das camadas de convolução, aplico uma camada *TimeDistributed* com uma *Dense* com 35 neurónios que, por estarmos a tratar de uma sequência temporal (os batimentos ao longo do tempo), esta função permite aplicar uma camada densa passada como parâmetro a cada trecho temporal do *input*, reduzindo assim o número de dados com que estamos a trabalhar. Devido a fazermos esta redução com a camada densa por estarmos a trabalhar com uma rede que se pode tornar muito grande e com muitos parâmetros, é boa ideia aplicarmos agora *UpSampling* aos dados para voltarmos ao tamanho que tinhamos antes de usar a camada densa. Para escolhermos o fator de upsampling, é verificado o número de parâmetros que existem antes e depois de aplicar a camada densa. Dependendo dos neurónios que queremos na camada *TimeDistributed*, dividimos o número de neurónios pelas 5 classes. Multiplicamos o valor resultante pelo número de parâmetros antes da camada *TimeDistributed* e fazemos a divisão entre este valor com o número de parâmetros depois da camada *TimeDistributed*, este será o fator de *upsampling*. Optei por usar *padding* same por obter melhores resultados com padding do que sem *padding*. Para o output da rede, bem como nas outras, usa-se a *softmax* por ser um problema *multi-class*.

Relativamente aos dados usados por esta rede, esta recebe um sinal inteiro ao invés de segmentos de tamanho fixo como a *feed-forward*. Assim, os sinais com que a rede vai treinar terão tamanhos diferentes e é preciso a rede saber lidar com isto. Por isso, ao construir o dataset, juntamente com a transformada de *fourier*, também passo o valor do tamanho do sinal para depois o poder usar no final. É por esta razão que uso dois inputs para a rede, um com o sinal em si e o outro com o tamanho do sinal para no final poder cortar.

5.1 Análise dos Resultados - Treino e Validação

Verificando agora a qualidade da rede criada, irei então verificar a evolução tanto do erro de treino como de validação ao longo das 50 épocas. Optei por utilizar apenas 50 épocas na versão final ao invés de mais por chegar a um ponto em que o modelo não melhorava (começava a haver *overfitting*). Relativamente ao optimizer utilizado, decidi utilizar também o *Adam* visto ter funcionado bem na rede anterior. Relativamente à *accuracy* esta é semelhante à de treino, embora divirga um pouco no final o que demonstra que pode ter havido algum *overfit* do modelo aos dados, mas como explicado acima o dropout não ajudou para a segmentação.

Figure 3: Losses e Accuracy - Convolucional

5.2 Previsão - Teste

Relativamente à previsão da rede para o conjunto de teste, apenas a *feed-forward* dá mais trabalho por necessitar de todo o processo explicado na secção anterior. Agora, a previsão é feita dando um sinal completo. Relembrando o explicado no início do relatório, devido ao fazermos *padded_batch*, os sinais de tamanhos diferentes serão *padded* com zeros e caso o modelo prevesse 0, teria uma accuracy alta, logo usarei *batch_size = 1* e usarei a função *evaluate* do keras. Os resultados deste modelo para a loss e accuracy foram respetivamente 1.418 e 0.383. Embora não tenham sido bons, ao verificar a previsão do modelo para um sinal conseguimos perceber que este não prevê apenas 4's, logo aprendeu alguma coisa.

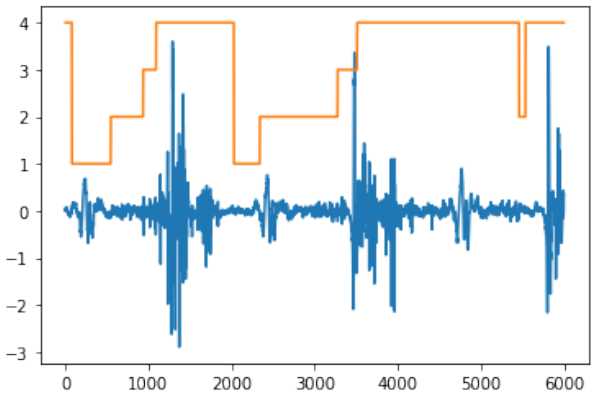


Figure 4: Segmentação de um sinal - Convolucional

6 Rede Recorrente

Agora, criaremos uma rede recorrente. Esta foca-se em suportar e utilizar informação antiga a cada previsão. A cada passo, a rede recorrente faz uma previsão mas, como indica o nome, existem ligações recorrentes entre os *timesteps* consecutivos de modo a transportar assim informação antiga para a previsão atual. Para introduzirmos este conceito na rede, serão usadas camadas LSTM. Assim, ao invés dos clássicos neurónios, usam-se estes blocos de memória que têm *gates* que fazem esta tarefa. Relativamente ao uso de funções de ativação entre as layers, decidi não utilizar porque estas tendem a "suavizar" a volatilidade e variações em *timeseries*. Isto pode resultar em melhores métricas (não é garantido) mas como desvantagem, significa que a rede pode não detetar variações importantes nos dados (para identificar a fase em que estamos) e por esta razão decidi não utilizar [3].

A arquitetura para esta rede foi a seguinte:

- 2 Camadas para o *input*
- 3 Camadas convolucionais (32, 64, 128 com kernel= 3/5 e *padding=same*)
- 1 Camadas LSTM (com 128 unidades)
- 1 Camada *TimeDistributed* com densa com 35 neurónios (reshaped para (-1,5))
- 1 Camada *UpSampling* com fator = 40
- 1 Camada de *output* com função de ativação *softmax*

Para esta rede, decidiu usar-se 3 camadas convolucionais, 1 camadas LSTM, 1 camada *TimeDistributed* com uma densa e 1 de *upsampling*. Devido às LSTMs já introduzirem não linearidade, decidi não utilizar nenhuma função de ativação adicional visto ser este o objetivo destas. Relativamente ao *optimizer*, utilizei o *Adam* por ser rápido e este tipo de redes já demorar muito tempo. Igualmente às outras redes, para a função de ativação no *output* usa-se a *softmax* e relativamente à *loss function*, esta é a *Sparse Categorical Crossentropy*.

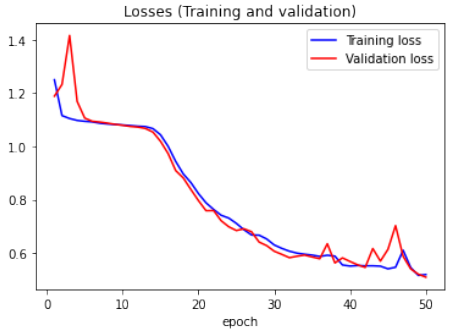
6.1 Análise dos Resultados

Verificando os gráficos abaixo, consegue-se perceber que tanto a *loss* como a *accuracy* não aparecem muito *jittered* logo o *Adam* funcionou relativamente bem para este caso.

Isto pode ser explicado pelo *Adam* conseguir lidar melhor com a diferente dinâmica de treino de uma rede recorrente comparativamente ao tradicional SGD [4]. Esta rede foi treinada com 50 épocas igualmente e com a *learning rate default*. Relativamente à generalização do modelo para os dados de teste, esta parecia prometer ser boa visto que até entre o erro de treino e o de validação, o de validação foi mais baixo. O mesmo aconteceu com a *accuracy*, mais alta no conjunto de validação.

6.2 Previsão - Teste

Como se esperava para as métricas no conjunto de teste, devido ao bom desempenho no de validação, os valores de ambas a loss e a accuracy para o conjunto de teste de respetivamente, 0.661 e 0.762. Como se pode ver no gráfico abaixo relativamente à segmentação do sinal, esta parece ter sido muito bem conseguida, sendo visíveis todas as fases do batimento cardíaco.



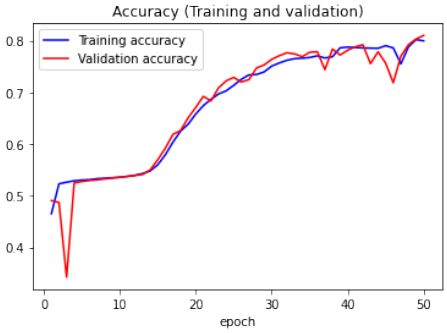


Figure 5: Losses e Accuracy - Recorrente

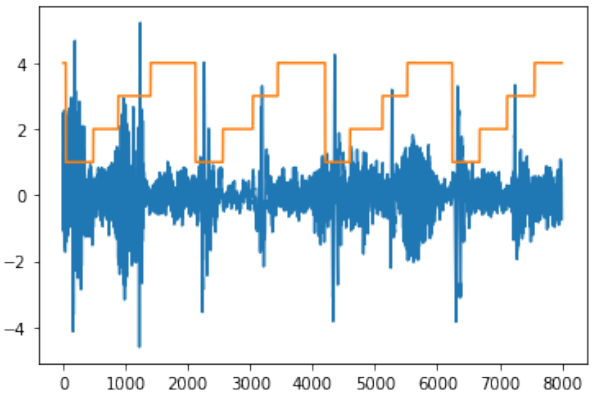


Figure 6: Segmentação de um sinal - Recorrente

7 Rede Transformer

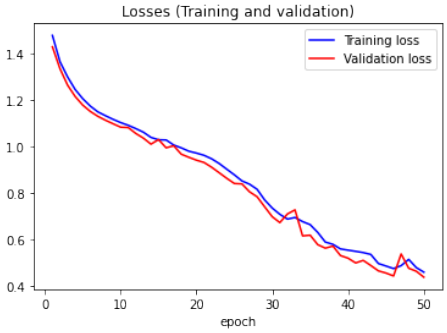
Por último, aplicar-se-á uma rede transformer. A arquitetura foi a seguinte:

- 1 Camada de *input*
- 3 Camadas convolucionais (32, 64 e 128 neurónios, filter = 3/5 com padding = same)
- 1 Camada LSTM (128 unidades)
- 2 *Transformer Blocks* (*encoders*)
- 1 Camada *TimeDistributed* com Densa com 35 neurónios (reshaped para (-1,5))
- 1 Camada *UpSampling* com fator = 40
- 1 Camada de *output* com função de ativação *softmax*

Esta introduz um novo conceito, a atenção que resumidamente, em vez de gerar todo o *output* da sequência a partir de um único vetor de *encoder*, cada passo da sequência de *output* usa um diferente vetor (é o contexto) juntamente com o passo anterior. O *transformer* processará as *features* de acordo com este contexto e de outras *features* e retornará *features* que passarão para o próximo *encoder*. Usei apenas *encoders* e não *decoders* porque, ao usar a transformada de *Fourier* esta retorna uma sequência e isto pode já ser visto como o *embedding* inicial e assim, o *transformer* ao usar estes *embeddings* como contexto, podem calcular os *contextual embeddings*.

7.1 Análise dos Resultados

Verificando os gráficos abaixo, consegue-se perceber que os valores da *accuracy* sobem continuamente (com o da validação sempre a acompanhar o de treino) e o análogo acontece com a *loss* mas a decrescer. Isto representa que não houve overfit. Relativamente ao *optimizer* usado, este foi o Adam com *learning rate default* de modo a não demorar muito tempo visto que o treino desta rede já é lento (caso se usasse o SGD). Como podemos ver pelo gráfico abaixo parece que caso existissem mais épocas que os valores melhorariam. Devido a isto, retreinei o modelo com 100 épocas mas o resultado não melhorou muito após a 50º época.



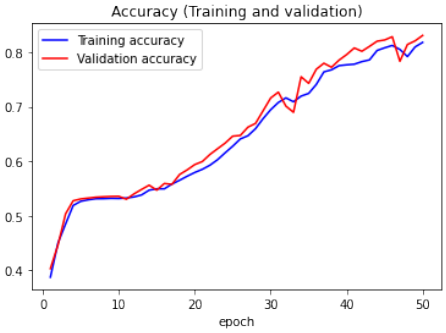
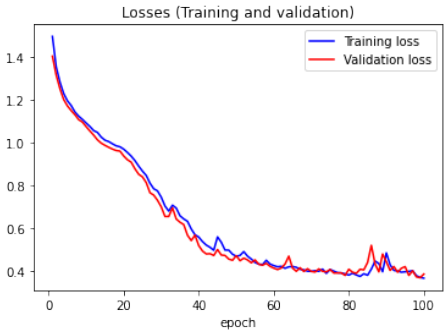


Figure 7: Losses e Accuracy - *Transformer* (50 épocas)



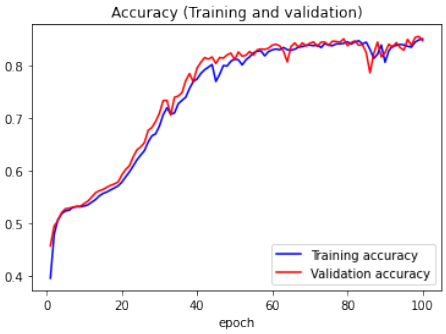


Figure 8: Losses e Accuracy - *Transformer* (100 épocas)

7.2 Previsão - Teste

Relativamente a este modelo, aconteceu o mesmo que nas recorrentes, uma boa segmentação dos sinais. Assim, a *loss* e a *accuracy* estimadas de teste foram respetivamente 0.665 e 0.758. Podemos ver o resultado da segmentação de um sinal em baixo.

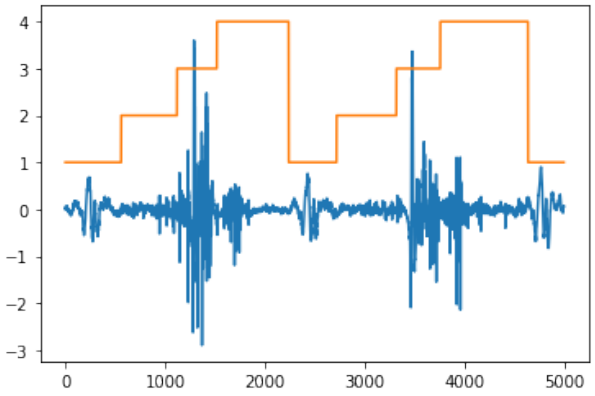


Figure 9: Segmentação de um sinal com transformer

References

[1] Rodrigues R. *Upsampling: a useful tool for Heart Sounds Segmentation*
Acedido a 07/06/2022, em
https://moodle.fct.unl.pt/pluginfile.php/571685/mod_resource/content/5/upSamplig.pdf

[2] Ahsan A. *Convolutional Neural Network and Regularization Techniques with TensorFlow and Keras*
Acedido a 07/06/2022, em
https://medium.com/intelligentmachines/convolutional-neural-network-and-regularization-techniques-with-tensorflow-and-keras-5a09e6e65dc7

[3] Grogan M. *Activation function between LSTM layers*
Acedido a 07/06/2022, em https://stats.stackexchange.com/questions/444923/activation-function-between-lstm-layers

[4] Hafner D. *Tips for Training Recurrent Neural Networks*
Acedido a 07/06/2022, em
https://danijar.com/tips-for-training-recurrent-neural-networks/

[5] MiloMinderbinder. *Ordering of batch normalization and dropout?*
Acedido a 07/06/2022, em
https://stackoverflow.com/questions/39691902/ordering-of-batch-normalization-and-dropout