Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Computação Natural

Trabalho Prático 3
Redes Neurais para Problemas de Classificação

Ingrid Rosselis Sant'Ana da Cunha

Novembro 2017

1 Introdução

O último trabalho da disciplina Computação Natural propõe a construção de uma rede neural para resolver um problema de classificação onde deseja-se identificar em qual parte da célula uma proteína pode ser encontrada. O objetivo deste trabalho é voltado para a parametrização da rede, sendo possível utilizar bibliotecas com recursos avançados para facilitar a construção da mesma. A ideia do uso da rede neural é, dado um novo exemplo sem classe, que a rede seja capaz de definir a mesma da maneira mais precisa possível.

Para verificar a funcionalidade da rede foi fornecida uma base de dados composta por 1429 exemplos, onde cada um possui 8 valores que o caracteriza, mais a sua respectiva classe. O problema também conta com 7 classes distintas nas quais um exemplo pode ser classificado.

2 Recursos Utilizados

Como a ideia do trabalho é voltada apenas para a parametrização, foi decidido utilizar o **TensorFlow 1** com auxílio do **Keras**. `TensorFlow` é uma biblioteca `open source` para aprendizado de máquina aplicável que pode criar e treinar redes neurais. Já o `Keras` é uma API de redes neurais que trabalha em conjunto com o `TensorFlow` para facilitar ainda mais a construção das redes, provendo uma variedade de ferramentas de personalização, mas ainda sim mantendo toda a implementação do modelo em poucas linhas de código.

Este trabalho foi feito em `Python 3` e utiliza as seguintes bibliotecas que necessitam de prévia instalação:

- TensorFlow;
- Keras;
- Pandas;
- Numpy.

3 Modelo

Para solucionar o problema dado foi definida a utilização de uma rede *Multi-Layer Perceptron* (MLP) pois estas redes são de fácil implementação e conseguem solucionar eficientemente o problema proposto. As MLPs são redes supervisionadas definidas por não possuírem ligação entre os nós internos de cada camada e entre duas camadas se estas não são consecutivas, caso contrário elas serão totalmente conectadas.

3.1 Transformação nos Dados de Entrada

As classificações dos dados de entrada são originalmente do tipo *string* e precisaram ser passadas para inteiro e depois categorizadas no formato *one-hot enconding*, que é exigido pela função de custo escolhida. O formato *one-hot enconding* transforma um vetor de números inteiros numa matriz binária onde as linhas são os números do vetor e as colunas são as classes existentes.

Em termos práticos, o vetor composto pelas classes [CYT, EXC, ME1, ME2, ME3, MIT, NUC] seria transformado em [0, 1, 2, 3, 4, 5, 6] e codificado para o formato *one-hot* da seguinte forma:

```
CYT  [[1 0 0 0 0 0 0]
EXC   [0 1 0 0 0 0 0]
ME1   [0 0 1 0 0 0 0]
ME2   [0 0 0 1 0 0 0]
ME3   [0 0 0 0 1 0 0]
MIT   [0 0 0 0 0 1 0]
NUC   [0 0 0 0 0 0 1]]
```

3.2 Arquitetura da Rede Neural

A rede neural desenhada é composta por, no mínimo, três camadas: a de entrada, uma camada escondida e a de saída. As camadas de entrada e saída possuem cada uma 7 nós, que são referentes ao número de classes existentes no problema. Enquanto essas duas camadas foram bem definidas para não serem variadas, o número de camadas escondidas e quantos neurônios existem nelas podem ser variados para teste.

Depois de vários testes feitos, concluiu-se que a inicialização padrão do Keras (*glorot_uniform*) é a melhor inicialização se comparada com o uso de 0, 1 ou valores uniformes aleatórios. Dentre as taxas de aprendizagem 0, 0.001, 0.005 e 0.1, 0.005 se mostrou mais efetiva e evitou dois problemas reparados: com 0 a rede não aprendeu nada, mas com 0.1 e valores maiores a rede aprendeu demais, convergiu extremamente rápido e não generalizou muito bem. O aprendizado da rede é feito por *back-propagation*, e este é automaticamente feito pelo Keras.

O número de épocas foi definido dentre 100, 150 e 300. 150 épocas apresentou o melhor desempenho, aprendendo o suficiente e mantendo os valores entre treino, validação e teste mais aproximados e realistas. Foi definido que a rede terminará a validação e o teste se depois de 30 épocas o valor do retorno da função de custo não sofrer variação.

Foi decidido que o treinamento utilizaria a abordagem de *mini-batches* pois ela é mais rápida e apresentou resultados satisfatórios. Em especial, foram testados os tamanhos 32, 80 e 100. O que melhor apresentou resultado foi 80 *mini-batches*.

3.3 Funções Utilizadas

Para o funcionamento do algoritmo, foram utilizadas 3 tipos de funções: de ativação, de custo e de optimização.

A **função de ativação** serve para propagar o sinal na rede, para manter os valores transformados a partir da entrada em um intervalo aceitável e para trazer não-linearidade à rede – o que permite uma aproximação boa para uma gama de diferentes funções. Ela é utilizada por todas as camadas, exceto a camada de entrada.

Diferentes funções de ativação foram testadas, sendo elas: *Sigmoid*, *Tanh*, *Softmax* e *ReLU*. Após muita leitura e opiniões divididas, as recomendações eram que as camadas escondidas utilizassem *Sigmoid* ou *ReLU*, pois são mais rápidas e melhores. A decisão final veio com resultados práticos, sendo *ReLU* a mais eficiente nas camadas escondidas. Para a camada de saída, as opiniões eram entre *Sigmoid* e *Softmax* (pois está última é tipicamente usada em camadas de saída). Como o problema trata de várias classes, foi recomendada a utilização *Softmax*, o que foi confirmado nos testes.

A **função de custo** é uma medida de erro entre a saída da rede neural ao final de uma época e a saída esperada para um conjunto de exemplos, o quão boa é a rede para generalizar. Ela é utilizada pelo *back-propagation* para reavaliar o conjunto de pesos na rede. Como o problema tratado é de classificação não-binária, a função escolhida foi a *categorical_crossentropy* pois ela foi bem recomendada, possui suporte do Keras e se mostrou efetiva nos testes feitos.

A **função de otimização** é utilizada para minimizar o retorno da função de custo. Isto é, no problema de classificação, a rede recebe o input para classificar e a saída esperada. Depois ela calcula a diferença entre o esperado e a saída obtida. A função do otimizador é modificar os pesos do classificador para que esta diferença diminua nas próximas épocas.

3.4 Treino, Validação e Teste

A base de dados fornecida foi divida em três partes da seguinte maneira: primeiro, a base foi divida em 93% para treino e 7% para teste. Os valores parecem estranhos, à princípio, mas esta divisão foi testada e apresentou melhores resultados, possivelmente porque o número de exemplos passados não era suficiente para um bom treino. Então, seguiu-se uma série de tentativas até que o valor da divisão permitisse que os gráficos de treino e teste não ficassem muito díspares, mesmo que isso acarretasse uma perda na precisão. Por fim, dos 93% reservados a treino, 20% foram utilizados para fazer a validação.

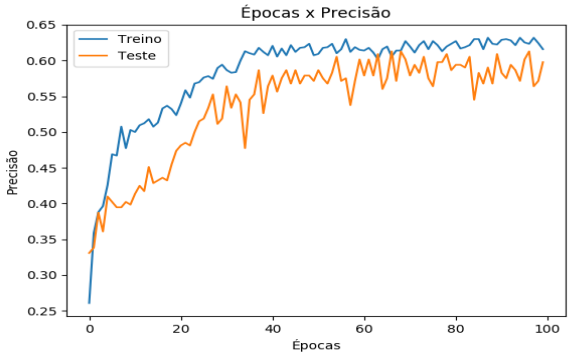
4 Análise Experimental

Diversos testes foram feitos para a parametrização da rede, como foi visto acima, e aqui serão apresentados outros para verificar o que ocorre com a rede quando certos parâmetros são modificados. Para fazer a análise experimental da rede, foi utilizado os seguintes parâmetros base que resultaram nos melhores resultados:

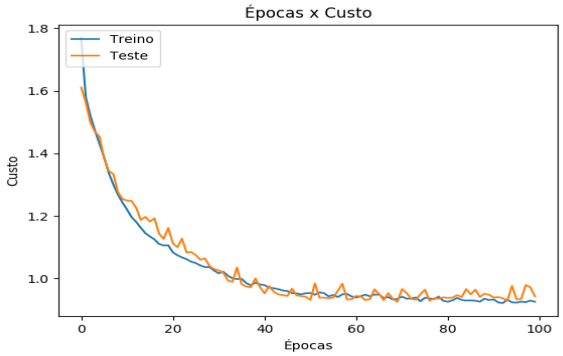
- Número de neurônios na camada de entrada: 7;
- Número de neurônios na camada de saída: 7;
- Número de neurônios nas camadas escondidas: 15;
- Número de camadas escondidas extras: 0;
- Função de ativação da camada de entrada: nenhuma (identidade);
- Função de ativação da camada de saída: *Softmax*;
- Função de ativação das camadas escondidas: $ReLU$;
- Taxa de aprendizagem: 0.005;
- Número de épocas: 150;
- Função de custo: *categorical_crossentropy*;
- Número de *mini-batches*: 80;
- Porcentagem para treino: 93%;

A métrica utilizada para avaliar a classificação foi a *categorical_accuracy*, que avalia as saídas dos neurônios da última camada e seleciona aquele com a maior saída (que será a classe escolhida).

Os gráficos base da precisão e da função de custo em relação às épocas seguem abaixo:



(a) Precisão: 62.73%

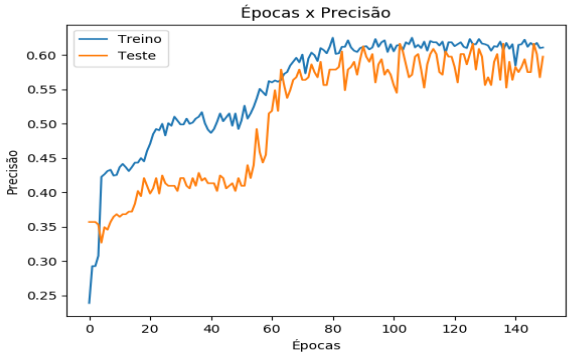


(b) Custo: 92.03%

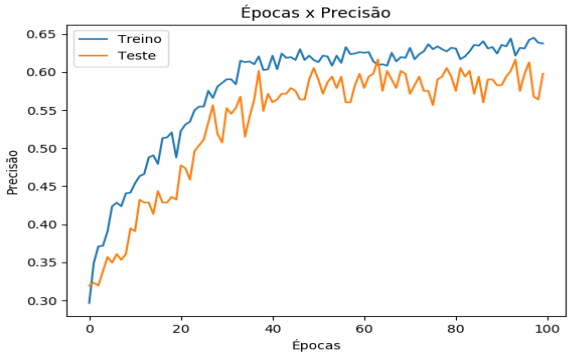
Estes gráficos serão utilizados para as comparações na análise.

4.1 Neurônios na Camada Escondida x Épocas

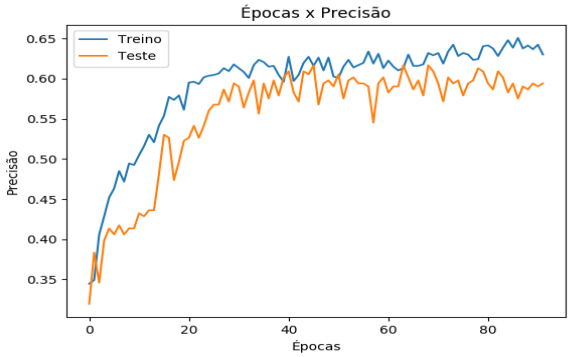
Para verificar o que acontece quando o número de neurônios nas camadas escondidas é variado, utilizou-se os valores 5, 30, 50 e 100, sendo 15 o valor base.



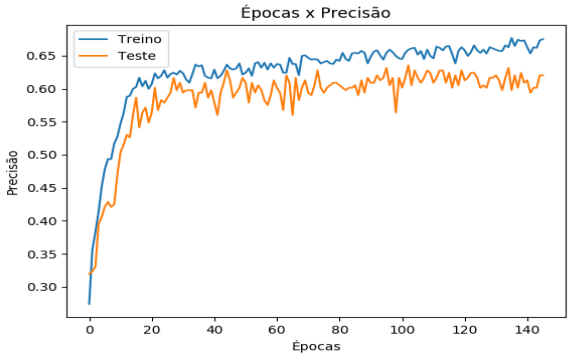
(a) 5 neurônios, precisão: 61.37%



(b) 30 neurônios, precisão: 63.18%



(c) 50 neurônios, precisão: 63.70%

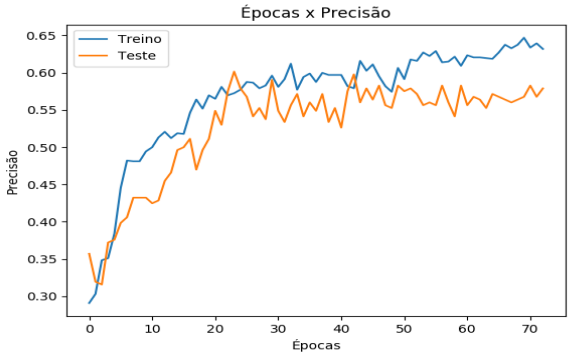


(d) 100 neurônios, precisão: 66.11%

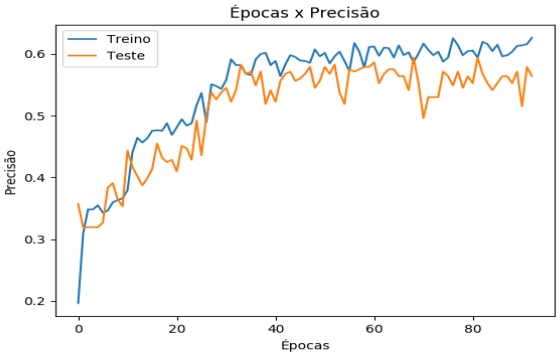
Como pode ser verificado, quando mais neurônios na camada de saída, melhor a rede se sai nos treinos. É importante destacar, no entanto, que a distância entre o resultado do treino e do teste aumenta conforme a rede chega aos 60%, verificando que ela não generaliza tanto a partir desse ponto. Em relação ao número de épocas, há uma variação baseado no quanto pode ser aprendido, com poucos neurônios a rede precisa usar todas as épocas para conseguir um valor bom de aprendizado. Com o aumento do número de neurônios, a rede aproveita mais as épocas para tentar aprender mais, mas com isso perde em generalização.

4.2 Camadas Escondidas x Erro

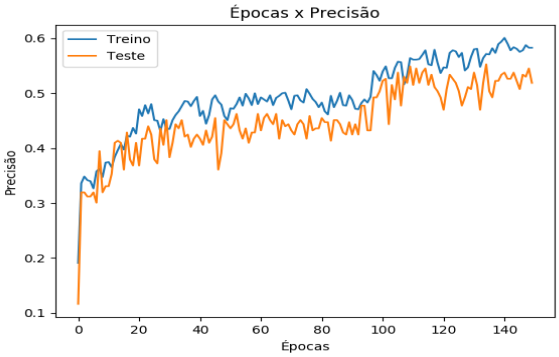
Os testes de variação do número de camadas em relação ao erro serão feitos com os valos 1, 4, 8, 12. Vale lembrar que a rede possui, por definição, uma camada escondida.



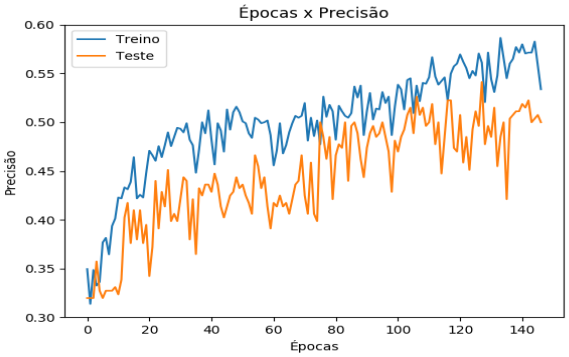
(a) 1 camada, precisão: 63.25%



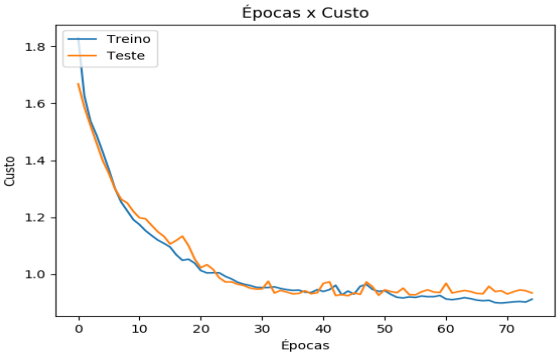
(b) 4 camadas, precisão: 60.39%



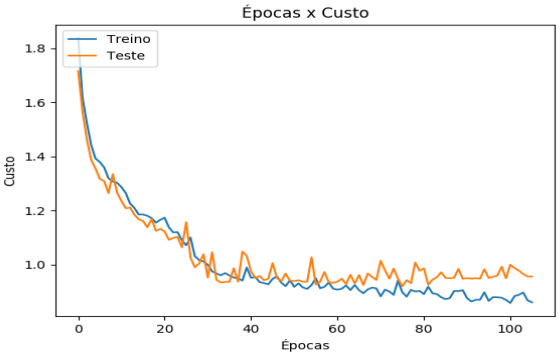
(c) 8 camadas, precisão: 58.06%



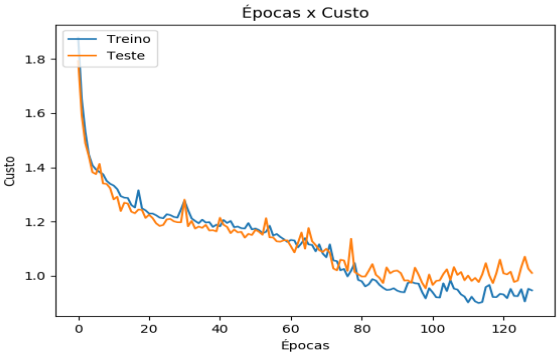
(d) 12 camadas, precisão: 57.15%



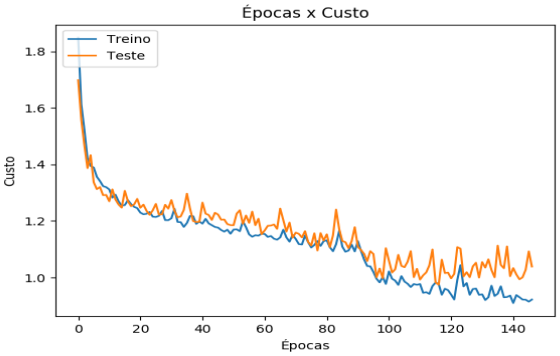
(a) 1 camada, perda: 89.61%



(b) 4 camadas, perda: 85.74%



(c) 8 camadas, perda: 93.29%



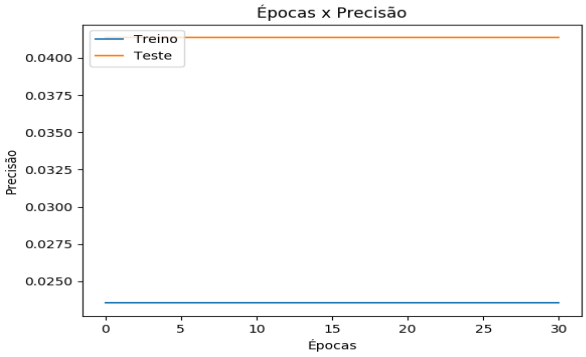
(d) 12 camadas, perda: 95.15%

Apesar da distância entre a linha do teste e a do treino ter aumentado ao final das épocas, a adição de uma camada escondida extra ainda se mostra favorável. Contudo, mais camadas aumentam a complexidade e tornam a rede menos generalizável, o que diminui sua precisão. Vale destacar que o número de épocas necessárias para convergência também diminui com a adição de uma camada escondida, mas aumenta quando o número de camadas cresce.

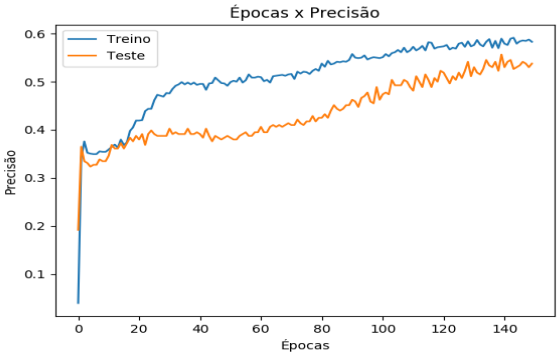
Em relação ao erro, pode-se perceber que com a adição de mais camadas o erro tende a aumentar e a curva do teste tende a se afastar da de treino, o que indica que não generaliza tão bem mais, e que a adição de uma camada extra ainda se mostra vantajosa.

4.3 Taxa de aprendizagem x Convergência

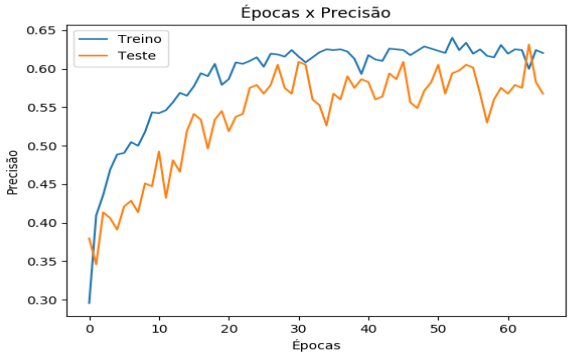
Para este experimento, a taxa de variação foi difinida para 0, 0.001, 0.01 e 0.1, além da base de 0.005.



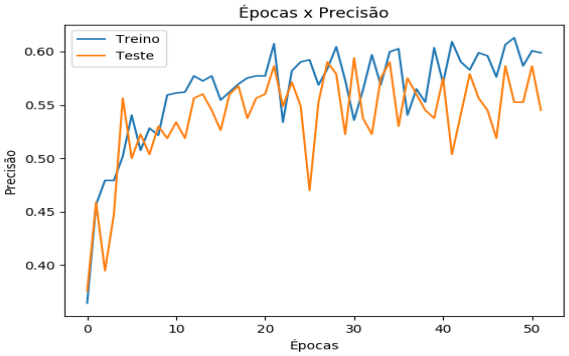
(a) Taxa: 0, precisão: 2.71%



(b) Taxa: 0.001, precisão: 57.76%



(c) Taxa: 0.01, precisão: 61.37%

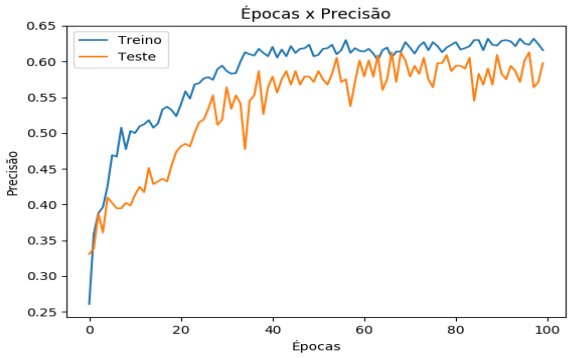


(d) Taxa: 0.1, precisão: 60.77%

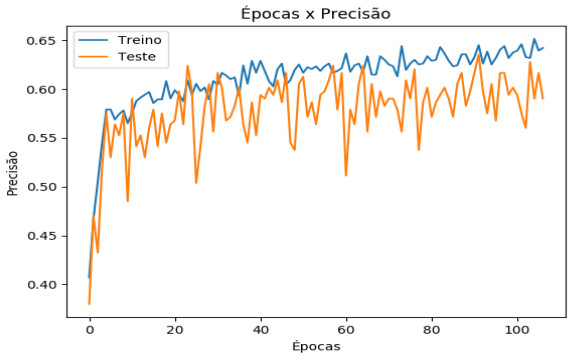
A rede não aprende com o valor da taxa de aprendizagem definido em 0 e não irá generalizar com o valor em 1. Dentre esse intervalo, conforme a taxa é aumentada, a rede começa a melhorar, mas existe um intervalo onde a precisão passa a crescer e depois volta a decrescer, sendo este entre 0.001 e 0.01. Baseado nesse teste que o valor de 0.005 foi definido previamente.

4.4 Stocastic Gradient Descent x Mini-Batch

O melhor valor de *mini-batches* encontrado foi 80, dentre os testes feitos. Nesta parte será verificado a diferença entre esse valor e o *Stocastic Grandient Descent* (quando o *mini-batch* tem tamanho 1).



(a) 80 *mini-batches*, precisão: 62.73%



(b) 1 *mini-batch*, precisão: 63.86%

Como pode ser visto, a diferença entre os dois testes não é tão grande e existe mais variação no *SGD*, além do mais o tempo de execução do *SGD* foi significativamente maior. Considerando todos esses fatores, o procedimentos de *mini-batches* deve ser utilizado.

4.5 Erro na validação x Erro no Teste

Pelos gráficos base apresentados no início dessa seção, pode-se verificar que a distância entre o treino e o teste não é tão grande, ou seja, a rede generaliza bem e os valores que foram avaliados correspondem à realidade. As funções de custo em treino e teste também ficaram bastante próximas pois já era de se esperar uma variação em alguns pontos no teste.

Alguns testes feitos nesta seção apresentaram maior divergência entre os dois gráficos de precisão, mostrando que houve um pequeno *overfitting* com a variação de alguns parâmetros como o número de neurônios na camada escondida, o número de camadas escondidas e a taxa de aprendizagem.

Para evitar o *overfitting*, pode-se, por exemplo, tomar cuidado com o número de épocas (para não treinas demais), possuir uma quantidade de exemplos que torne viável o aprendizado da rede ou definir uma taxa de aprendizado pequena o suficiente para que não ocorra *overfitting* (mas grande o bastante para que a rede aprenda o necessário).

5 Conclusão

Por fim, os parâmetros base apresentados no início da seção de Análise Experimental foram escolhidos como principais, apesar de outros terem obtido melhores resultados na precisão. Os parâmetros base mantiveram um bom custo benefício entre a acurácia e a distância de valores entre treino e teste, além de garatir menos variação nos testes, evitarem *overfitting* e lidarem bem com o número de exemplos dados.

Este trabalho teve o objetivo de parametrizar uma rede neural para um problema de classificação. Os resultados vistos aqui foram satisfatórios pois a taxa de acerto ficou por volta dos 60%, como foi descrito na especificação do trabalho prático. Além disso, a variação nos parâmetros proporcionou uma visão mais prática de suas importâncias, apesar dos resultados já serem teoricamente conhecidos.

6 Bibliografia

- Keras
- Develop Your First Neural Network in Python With Keras Step-By-Step
- How To Build Multi-Layer Perceptron Neural Network Models with Keras
- Multi-Class Classification Tutorial with the Keras Deep Learning Library
- Introduction to Python Deep Learning with Keras
- Display Deep Learning Model Training History in Keras
- How to Evaluate the Skill of Deep Learning Models
- numpy convert categorical string arrays to an integer array (stackoverflow)
- Adam - A Method for Stochastic Optimization