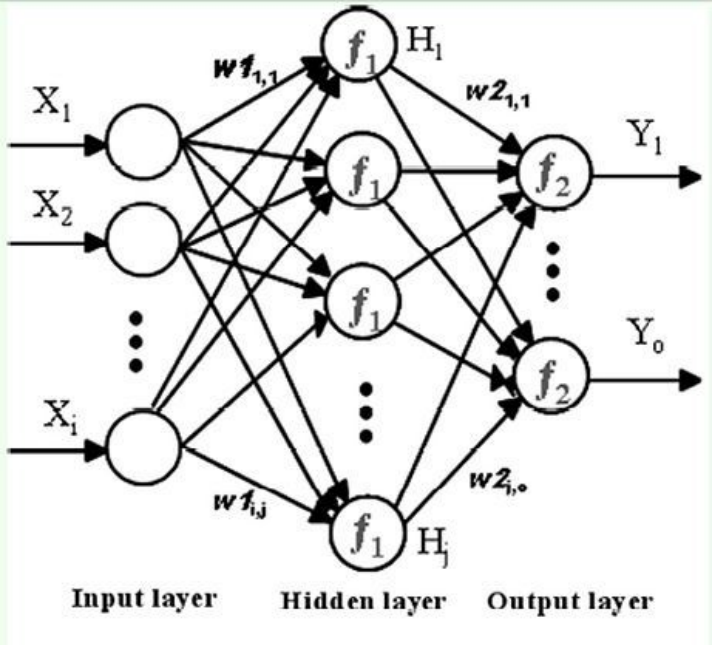
Trabalho Prático 3
Redes Neurais Artificiais

Yuri Diego Santos Niitsuma

Introdução

Neste trabalho propõe implementar um modelo computacional da área de redes neurais artificiais.

Segue um exemplo gráfico de uma rede Perceptron com multi camadas (MLP).



Nela, é composta por várias camadas contendo a camada de entrada (Input Layer), a camada oculta (Hidden Layer) que é composta por uma ou mais camadas e a camada de saída (Output Layer). Além disso, cada ligação/aresta compõe por um peso **wij** que influencia na função de ativação do neurônio. Ao final, a saída é composta pela soma do produto dos pesos com os valores da camada de entrada.

$$Y_j = \sum_i w_{ij} X_i$$

Implementação

Conforme descrito no exemplo da rede, foi utilizado a MLP contendo camadas ocultas de quantidade variáveis tanto de neurônios quanto da própria quantidade de camadas. Através deste detalhe, foi decidido utilizar o método estocástico ADAM para a otimização da rede neural. Nela, possui melhores resultados em redes que possuem várias camadas ocultas. Por ser MLP, cada camada é densa, ou seja, cada neurônio da camada anterior possui uma ligação atribuído com um peso a todo neurônio da camada seguinte,

A biblioteca utilizada foi o **Keras** sobre o **TensorFlow**. Além dela, foi utilizada o **NumPy**, **sklearn** para o balanceamento das classes e **matplotlib**. A versão do Python utilizada foi a Python 3.5.2 em conjunto com **TensorFlow** 1.4.0 e **Keras** 2.1.2.

O particionamento do conjunto de dados para treinamento e testes foi de 2/3 para treinamento e o complemento para testes. (Simple is the best)

O trabalho prático é executado pelo main.py passando os parâmetros na seguinte ordem:

python3 main.py input_data epochs hlayers neuros bsize learnrate

- **input_data**: arquivo contendo os dados para os testes,
- **epochs**: Número de épocas
- **hlayers**: número de camadas ocultas
- **neuros**: Número de neurônios nas camadas ocultas
- **bsize**: tamanho dos batchs nos treinamentos.
- **learnrate:** parâmetro contendo taxa de aprendizado da rede neural.

Como informado na especificação, há um desbalanceamento das classes, disso foi utilizado um normalizador da biblioteca sklean antes de transformar na matriz de classificação e enviado como parâmetro na função de treinamento **model.fit**.

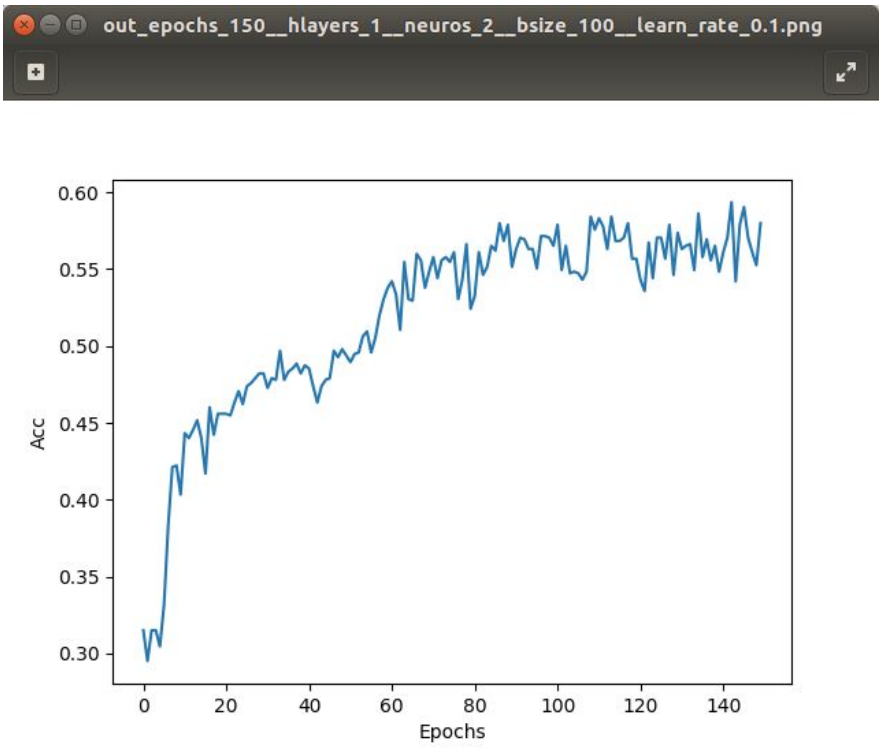
Testes

Os testes foram executados em vários parâmetros variáveis com exceção do epochs, que é fixo em 150. O resto é variado em:

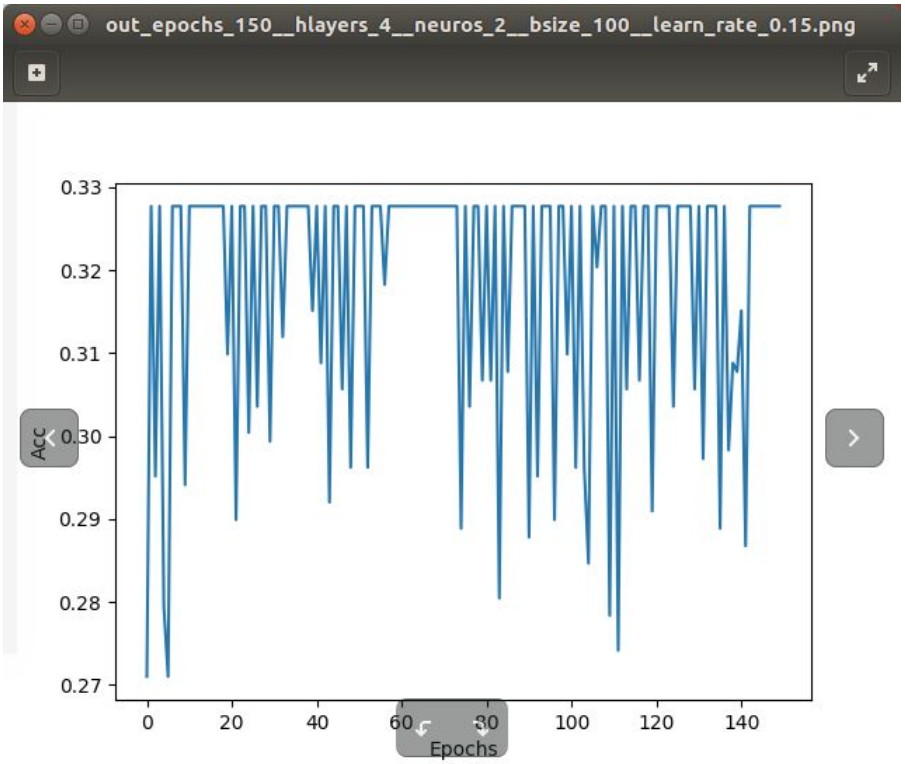
- $1 \leq hlayers \leq 9$
- $1 \leq neurons \leq 9$
- $bsize \in \{100,\ 150\}$
- $learnrate \in \{0.01,\ 0.1,\ 0.15\}$

Os comandos contendo todos os testes estão no **test.sh**. Todos os resultados estão na pasta **tests**, contendo a saída em txt o gráfico do resultado da eficácia da rede. Os nomes dos arquivos também contém os valores dos parâmetros utilizados no teste.

A maior parte dos testes convergiam em torno dos 60% de acerto, como no exemplo abaixo.



Mas possui muita variação conforme aumentamos a taxa de aprendizagem da rede.



Resultados melhores foram obtidos com apenas uma camada oculta e taxa em torno dos 0.1. Com mais camadas dificilmente a eficácia ultrapassava os 50%. Concluindo que a quantidade de camadas ocultas possui um impacto ainda maior nos dados pela quantidade pequena de dimensões dos dados.

Referências

But what *is* a Neural Network? | Chapter 1, deep learning
https://www.youtube.com/watch?v=aircAruvnKk

Udacity - Deep learning: do conceito à execução
https://www.youtube.com/watch?v=KIvB5LFbA0w

Adam: A Method for Stochastic Optimization
https://arxiv.org/abs/1412.6980v8

Keras: The Python Deep Learning library
https://keras.io/

Introdução

O objetivo do trabalho é gerar uma população de expressões matemáticas e utilizando técnicas da programação genética para encontrar um conjunto de soluções que mais se aproximam de um conjunto de dados fornecidos.

Implementação

A linguagem utilizada foi o Python 3.6.2.

A estrutura das expressões matemáticas foi implementado utilizando árvore binárias contendo nós de vários tipos

O arquivo **tree.py** tem o papél de modelar os tipos descritos e manipular a árvore:

Terminais

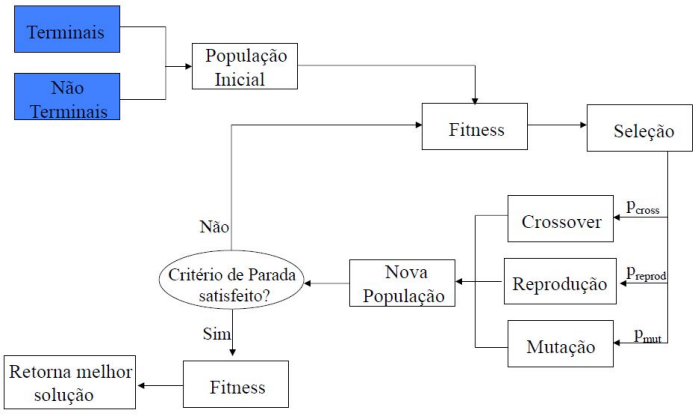
- Variable: $X_i \mid i \in \{1, ..., n\}$
  - A variável em si.
- Constant:
  - Valor constante, na figura é representado pelo *Number*.

Não terminais

- Operator: Composto pelos operadores básicos {+, -, *, /}.
  - Na divisão por zero ficou definido que tem como resultado 1.0.
- Function: Utilizado para agrupar as seguintes funções:
  - $\sin(x) \mid x \in Real$
  - $\cos(x) \mid x \in Real$
  - $\ln(x)$
    - se $x > 0 \Rightarrow \ln(x)$
    - se $x \leq 0 \Rightarrow 0$
  - $\exp(x) \mid x \in Real$

O arquivo **indivudal.py** modela as propriedades do indíviduo: árvore, fitness, crossover e mutação.

O **gp.py** fica a cargo de seguir o fluxo da programação genética.



A árvore é gerada aleatoriamente com profundidade aproximada de 3. Um esforço de eliminar alguns "*intros*" é somando duas constantes entre um operador.

Experimentos

Os logs do resultado dos experimentos estão no diretório **output**.

keijzer-7

Segundo o artigo *Genetic Programming Needs Better Benchmarks*, a função objetivo é $\ln(x)$.

As instâncias que obtiveram os melhores resultados são:

- 12.txt
  (ln((X1 * 1.133836227858187)) + ln(cos(exp(7.859096223899583))))
  RMSE: 0.16038689504251544
- 14.txt
  (cos(ln((X1 / X1))) + ln((X1 / 2.900185344196262)))
  RMSE: 1.1896973429403954
- 18.txt
  (cos(ln(-2.2314588880200366)) - ln((2.4938123996942174 / X1)))
  RMSE: 1.3173450836810257
- 23.txt
  (ln((X1 + X1)) - cos(cos(0.5341914244103876)))
  RMSE: 0.7702321063637263
- 29.txt
  ln(((X1 + X1) / 1.9446354701999558))
  RMSE: 0.2793192126171043

Observe que todos contém logaritmo com algumas variações de constantes e funções trigonométricas, já que elas apenas influenciam em algumas flutuações se estiver no numerador.

Com um aumento da probabilidade de crossover para a margem de 50% foi obtido melhores resultados.

keijzer-10

No mesmo artigo a função objetivo é x^y contendo duas variáveis.

Com exceção do 1.txt primeiro, todos obtiveram erro abaixo de 1. Talvez pelo fato que a curva da função é mais comportada. Por exemplo a função constante 1.56 contém um erro RMSE aproximadamente de 0.89.

house

Estes dados possuem 10 variáveis e a função objetivo é desconhecida. Obtive muitos erros numéricos e grandes (na casa de 48 milhões). Talvez com tempo maior de execução ou refinamento no processo de mutação das árvores possa ajustar para menor.

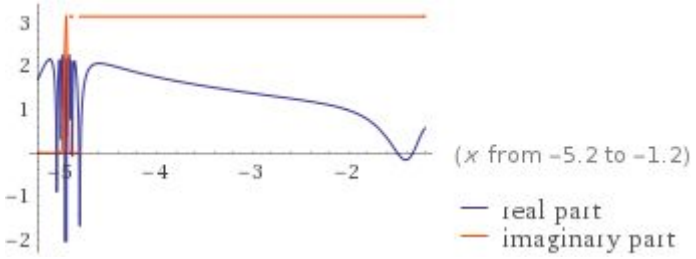
Conclusões

O código consiste de alguns problemas. Como a utilização de exponenciais e funções no denominador devem ser bem controladas ou eliminadas. Pois muitas funções compostas, principalmente no denominador, torna a função bem complexa, o que

Por exemplo a função

ln(((cos(exp((-2.627176889768328 / x))) - ln(exp((x / -0.5995299135427423)))) * cos(((-4.575619132309248 - x) / (x + 4.938891699281122)))))

Tedo uma frequência alta na vizinhança do -5.



Isto ocorre pois torna as suas derivadas sucessivas não limitadas formando funções “mal comportadas”, gerando alguns overfittings em que os indivíduos perdem o ranqueamento no cálculo da fitness sobre os dados de testes.

Outro ponto a destacar é a repetição de indivíduos gerados pelo processo de elitismo e seleção. Não encontrei uma forma de eliminar isto pois descartando alguns indivíduos acabam divergindo ainda mais da solução na iteração.

Referência

McDermott, James, Kenneth De Jong, Una-May O’Reilly, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, et al. 2012. “Genetic Programming Needs Better Benchmarks.” In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference - GECCO ’12*. doi:10.1145/2330163.2330273.