# Relatório

## Perceptron multicamada - Base de dados MNIST

16/0023777 André Filipe Caldas Laranjeira
16/0013615 Luiz Antônio Borges Martins

*Tópicos em Engenharia: Inteligência Artificial, Turma G*
*Departamento de Engenharia Elétrica*
*Universidade de Brasília*
Brasília, Brasil

### I. Teoria estudada

O perceptron multicamada consiste de um conjunto de camadas de neurônios perceptron, em que todas as saídas dos neurônios em uma camada são ligadas às entradas de todos os neurônios na camada seguinte e não há ligação entre os neurônios da mesma camada.

A exceção acontece na primeira e última camada. Na primeira camada, os neurônios tem as entradas conectadas às entradas do problema. E na última camada as saídas indicam, cada uma, a probabilidade de que a entrada do problema pertença à classe correspondente.

Sejam $\mathbf{W_{ij}}$ e $\mathbf{b_{ij}}$ o vetor de pesos e de bias do $j$-ésimo perceptron na camada $i$, calcula-se a saída do perceptron pela equação:

$$y_i^j = f_i(\mathbf{W_{ij}} \cdot \mathbf{y_{i-1}} + \mathbf{b_{ij}})$$

Onde $\mathbf{y_i}$ é o vetor $(y_i^1, \ldots, y_i^{l_i})$, e $l_i$ é o número de neurônios na camada $i$. O caso base é $\mathbf{y_0} = \mathbf{x}$, a entrada da rede.

O algoritmo para treinar a rede realiza mudanças nos pesos de todos os neurônios de modo a minimizar uma função de custo $c = c(\mathbf{y_n}, \mathbf{d})$, onde $\mathbf{d}$ é um vetor com o valor desejado para a saída, e um elemento do vetor é 1 com o resto é 0. Para minimizar a função de custo, usa-se o método de descida do gradiente, onde atualiza-se o peso de acordo com a derivada do custo com respeito ao peso:

$$\Delta\mathbf{W_{ij}}[k] = -\eta\frac{\partial c}{\partial\mathbf{W_{ij}}[k]}$$

Na equação anterior, $\mathbf{W_{ij}}[k]$ é $k$-ésimo elemento de $\mathbf{W_{ij}}$. Atualizamos também o bias pela regra:

$$\Delta\mathbf{b_{ij}}[k] = -\eta\frac{\partial c}{\partial\mathbf{b_{ij}}[k]}$$

Chama-se $\eta$ de constante de aprendizado e $f_i$ de função de ativação da $i$-ésima camada. Costuma-se ter uma função de ativação na camada de saída diferente das camadas escondidas. Uma função de ativação interessante é a softmax.

Para implementar o softmax é preciso que todos os neurônios de saída estejam totalmente interconectados, pois a saída de um neurônio dependerá da saída dos outros pela fórmula:

$$y_n^j = \frac{e^{\mathbf{W_{nj}}\cdot\mathbf{y_{n-1}}+\mathbf{b_{nj}}}}{\sum_k^{l_n} e^{\mathbf{W_{nk}}\cdot\mathbf{y_{n-1}}+\mathbf{b_{nk}}}}$$

A função de custo pode ter várias formas, a mais simples é o erro quadrático:

$$c(\mathbf{y_n}, \mathbf{d}) = \frac{1}{2}\sum_k^{l_n}(\mathbf{d}[k] - \mathbf{y_n}[k])^2$$

Outras funções de custo simplificam a expressão da derivada dependendo da função de ativação, como é o caso da entropia cruzada quando se usa sigmóide na saída:

$$c(\mathbf{y_n}, \mathbf{d}) = -\sum_k^{l_n}[\mathbf{d}[k]\log(\mathbf{y_n}[k]) + (1-\mathbf{d}[k])\log(1-\mathbf{y_n}[k])]$$

O mesmo pode ser feito com saídas softmax usando o *loglikehood*, dado por:

$$c(\mathbf{y_n}, \mathbf{d}) = -\log(\mathbf{y_n}[i(\mathbf{d})])$$

Nessa fórmula $i(\mathbf{d})$ é o índice do elemento de $\mathbf{d}$ que é igual a 1, em outras palavras $\mathbf{y_n}[i(\mathbf{d})]$ é a probabilidade que o neurônio retornou para a classe desejada.

O treinamento pode ser feito em *minibatches*, caso no qual soma-se o custo para cada mini conjunto de exemplos mostrados. Algumas implementações, no entanto, preferem fazer uma média do custo no *minibatch*, o que permite que o custo não dependa do tamanho dele. Neste trabalho optou-se pela utilização da média.

Costuma-se incluir nas funções de custo a chamada regularização. Regularização tem como objetivo penalizar a rede por manter pesos grandes (em módulo) ou muitos pesos não-nulos. O objetivo é criar uma rede que não usa muitos pesos e ter menos chance de *overfitting*. Chama-se regularização L1 a adição no custo da expressão:

$$\lambda\sum_{ijk}|\mathbf{W_{ij}}[k]|$$

Chama-se regularização L2 a adição a função de custo da expressão:

$$\frac{\lambda}{2}\sum_{ijk}\mathbf{W_{ij}}[k]^2$$

Aqui $\lambda$ aparece como mais um hiperparâmetro da rede a ser escolhido. Uma outra técnica de regularização é o *dropout*, em que se escolhe com probabilidade $p$ a cada iteração do treinamento pesos de uma camada para serem anulados. O intuito é fazer com que a rede crie uma representação interna distribuída.

## II. Descrição do problema

O trabalho proposto consiste em implementar um programa de perceptron multicamada e utilizar ele para aprender a base de dados **MNIST**, composta por imagens de dígitos manuscritos. Cada objeto de entrada é uma matriz $28 \times 28$, cujos valores são inteiros de 0 a 255 que representam a cor em escala de cinzas de cada pixel na imagem do dígito escrito a mão. Cada rótulo de saída é um inteiro de 0 a 9 indicando o dígito manuscrito contido na imagem, de forma que, tome $n$ o número de camadas, a última camada deve satisfazer $l_n = 10$.

O programa de perceptron multicamada deve ser capaz de instanciar um perceptron multicamada com camadas escondidas e que possa ser treinado por meio de um algoritmo padrão de backpropagation com os dados de treinamento do MNIST.

O perceptron multicamada instanciado também deve ser capaz de, com base em uma entrada de teste, prever qual o dígito manuscrito que a entrada de teste representa, sem realizar o ajuste de seus pesos. Ou seja, o perceptron multicamada deve receber como entrada uma matriz $28 \times 28$ que representa um dígito manuscrito em escala de cinzas e retornar como saída o dígito manuscrito que ele acredita ser mais provável que esteja na imagem.

Por fim, o programa de perceptron multicamada também deve possuir uma série de funcionalidades extras, os quais incluem:

- Uso de múltiplas camadas escondidas de tamanhos arbitrários.
- Uso de entropia cruzada como custo.
- Uso de regularização (L1, L2 e *dropout*).
- Uso de saída em camada softmax e custo log-likelihood.
- Uso de sequências de camadas RBM com treinamento não-supervisionado.
- Teste com padrões adicionais (obtidos pelos autores).

## III. Descrição do programa implementado

O programa que escrevemos para resolver esse problema implementou as seguinte funcionalidades:

- Leitura e pré-processamento dos arquivos de entrada do MNIST (fez-se mudança de escala dos pixels de $[0, 255]$ para $[-1, 1]$).
- Treinamento de um PMC com uma camada escondida e custo erro quadrático médio.
- Teste da rede com arquivo de teste sem afetar o treinamento.
- Uso de múltiplas camadas escondidas de tamanhos arbitrários.
- Uso de entropia cruzada como custo.
- Uso de regularização L1 e L2.
- Uso de função de ativação softmax e custo log-likelihood.
- Teste com padrões adicionais produzidos para este trabalho.

Os parâmetros utilizados para treinamento do perceptron (número de épocas, tamanhos das camadas escondidadas, função de custo, etc...) são todos configurados por meio da *passagem de argumentos* para o programa. Para maiores detalhes, consulte a documentação do programa rodando o comando **python src/main.py –help**, certifique-se de usar python 3.7.

Infelizmente, devido a limitações de tempo encontradas, não foram implementadas as seguintes funcionalidades:

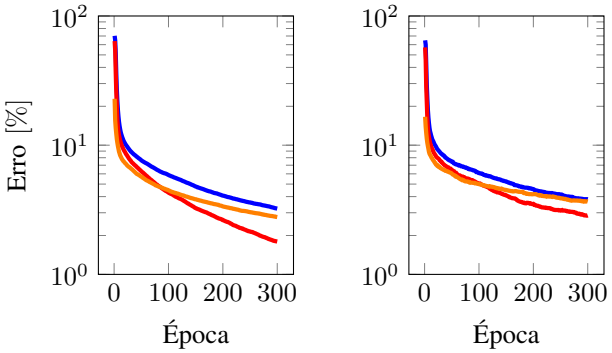
- Uso de RBM
- Uso de regularização por *dropout*

## IV. Resultados observados

Para testar nosso programa de perceptron multicamada e comparar as várias possíveis configurações de parâmetros entre si, nós realizamos um conjunto de testes com várias configurações de perceptron multicamada. Além disso, imagens com o erro da época e do conjunto de testes ao longo do treinamento estão presentes, junto com um comentário discutindo os resultados. Por fim, especificamos que *todos* os teste realizados utilizaram a taxa de aprendizado igual a 0.01 e o treinamento com *mini-batches* de tamanho 20.

### A. Teste com diferentes custos

Para testar os custos três treinamentos foram feitos: custo erro quadrático e saída identidade, custo entropia cruzada e saída sigmoide e custo *loglikehood* e saída softmax. Todos sem regularização e uma camada escondida de tamanho 100.




(a) Erro da época (b) Erro do teste

Figura 1. Erro da época e do teste para treino em três redes: de laranja com erro médio quadrático, de vermelho com *loglikehood* e saída softmax e de azul entropia cruzada e saída sigmoide

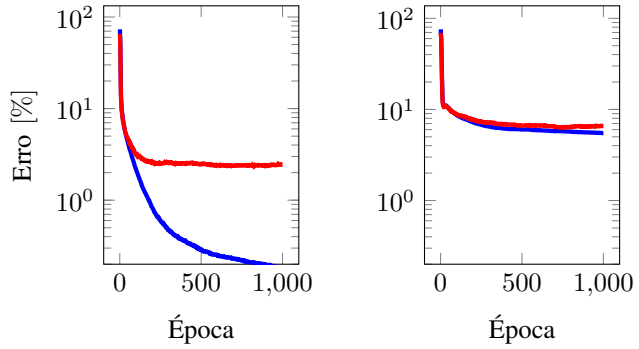
Pela figura (1) vemos que o custo *loglikehood* com saída softmax (de vermelho) apresenta um desempenho melhor para minimizar o erro da época e de teste. Por conta desse fato, os testes seguintes serão feitos usando esse custo.

### B. Teste de overfitting

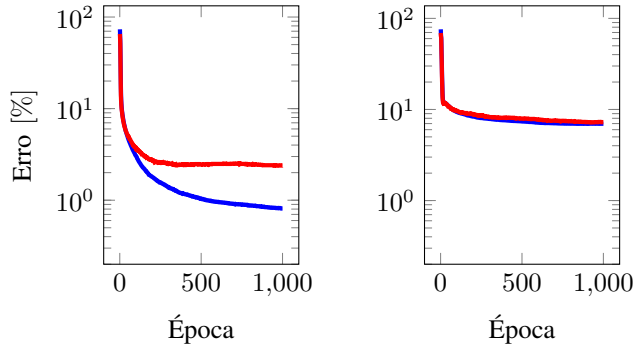
Tentou-se detectar *overfitting* para esse problema com o uso de três camadas escondidas de tamanho 100. Para esse teste, usou-se custo *loglikehood* e saída softmax em quatro treinamentos, um sem regularização, um com regularização L1 de 0.0001, outro com regularização L2 de 0.0001 e outro com ambas. A figura (2) mostra o erro da época e do conjunto de validação ao longo de um treinamento com 1000 épocas.

Nota-se da figura que, sem regularização, o erro da época (conjunto de treinamento) cai mais que com regularização, em contrapartida o erro no teste estabiliza. Nesse cenário a rede está se especializando no conjunto de treinamento. Ao colocar regularização L1 a rede demora bem mais para aprender, mas não vemos uma divergência entre os erros de teste e da época.

A regularização L2 apresentou um resultado intermediário, o erro no teste não divergiu tanto comparado ao da época e a rede aprendeu mais rápido do que com L1. Quando ambas as regularizações são usadas temos que a L1 domina e o resultado é semelhante ao da figura (b).




(a) Sem regularização (b) Com regularização L1



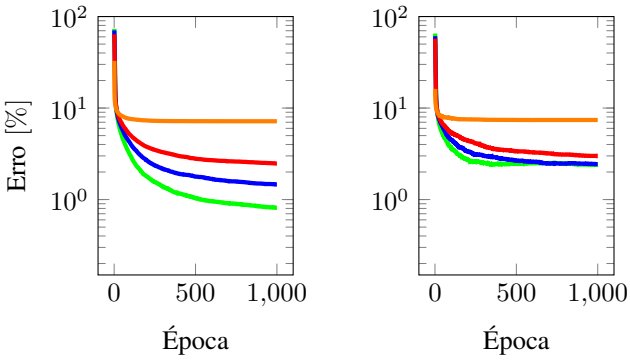

(c) Com regularização L2 (d) Com regularização L1 e L2

Figura 2. Erro percentual para o teste de *overfitting*, de azul o erro da época e de vermelho o erro do teste

Note que o uso da regularização L1 e L2 diminui a discrepância entre a taxa de erro da época e a taxa de erro de teste, melhorando, assim, a generalização do perceptron multicamada. Entretanto, essa generalização causou um impacto negativo não apenas na taxa de erro da época (o que era de se esperar pela teoria), mas também na taxa de erro de teste. Isso nos leva a crer que a taxa de erro de teste dos modelos treinados com regularização L1 e/ou L2 precisará de mais épocas de treinamento para alcançar os níveis de performance obtidos sem o uso de regularização, pois o uso de regularização torna o treinamento do perceptron multicamada mais complexo.

### C. Teste com múltiplas camadas escondidas

Um outro teste feito foi treinar quatro redes: uma sem camada escondida, outra com uma camada de tamanho 100 e outra com duas camadas de tamanho 100 e mais uma com três camadas de tamanho 100. Para esses testes foi usado *loglikehood* e regularização L2 de 0.0001. Podemos notar pela figura 3 que o desempenho da rede melhora com o aumento das camadas.




(a) Erro da época (b) Erro do teste

Figura 3. Erro da época e do teste para treino em quatro redes: de laranja sem camadas escondidas, de vermelho com uma camada de tamanho 100, de azul com duas camadas de tamanho 100 e de verde três camadas de tamanho 100 .

Assim, percebemos que a adição de camadas escondidas pode levar a resultados melhores. Entretanto, não devemos simplesmente adicionar camadas escondidas à rede esperando um ganho de performance na taxa de erro de teste, pois cada camada nova na rede aumenta o custo computacional do perceptron multicamada e o ganho de performance na taxa de erro de teste passa a ser desprezível após certa quantidade de épocas de treinamento, como observado na figura 3 para os modelos treinados com 2 e 3 camadas escondidas após 1000 épocas.

### D. Teste com padrões adicionais

Foram produzidos 10 imagens com o GIMP de tamanho $28 \times 28$ para serem usadas na rede, veja a figura (4). O melhor resultado obtido foi com a rede de duas camadas escondidas de tamanho 100 e entropia cruzada, onde ela errou dois desses padrões: o 7 e o 0. O segundo melhor resultado foi com três camadas escondidas de tamanho 100 com *loglikehood*, em que ela errou três padrões: 4, 3 e 0.

Figura 4. Padrões adicionais feitos pelos autores

Note que uma boa taxa de acerto em exemplos feitos por nós corrobora a corretude do treinamento do perceptron multicamada e demonstra a possibilidade da aplicarmos o perceptron multicamada em aplicações práticas nos mais diversos campos.

## V. Conclusão

Com base nos resultados observados, podemos concluir que a utilização de um perceptron multicamada nos fornece uma excelente ferramenta de aprendizado de máquinas devido à baixa taxa de erro apresentada para o conjunto de teste após o treinamento e à grande quantidade de parâmetros contidos no modelo, os quais facilitam o ajuste do modelo treinado às necessidades do projetista.

A performance apresentada pelo modelo variou muito com os parâmetros utilizados para sua construção (número de camadas escondidas, tamanho das camadas escondidas, número de épocas treinadas, entre outros). Em nosso melhor modelo, nos aproximamos de uma acurácia de $97.5\%$ nos exemplos de teste, um nível de performance não apenas próximo daquele esperado de um ser humano, mas também alto o suficiente para ser utilizado em algumas aplicações práticas, como evidenciado pelos testes utilizando padrões feitos pelos autores do artigo.

Entretanto, esses benefícios não advém sem apresentar novas dificuldades. Embora a utilização do perceptron multicamada apresente excelentes taxas de acerto, seu treinamento requer mais tempo e recursos computacionais que outros modelos mais simples, em especial se almejamos alcançar taxas de acerto boas o suficiente para aplicações práticas. Além disso, o treinamento de modelos que sejam capazes de atingir taxas de acerto mais elevadas requer a utilização de parâmetros corretos, os quais requerem tempo para serem encontrados devido à grande quantidade de parâmetros do modelo e ao tempo necessário para se treinar um modelo e avaliar seus resultados.

## Referências


[1] Multilayer Perceptron. DeepLearning 0.1 Documentation. http://deeplearning.net/tutorial/mlp.html

[2] Slides de aula.