Este documento pretende justificar o programa construído. Foi elaborado durante estudo do tema e serviu de guia pessoal. Primeiramente são apresentadas as equações utilizadas para treinamento da rede, depois algumas características do programa, e ao fim uma lista de referências externas que foram consultadas.

Definição de rede neural

Uma rede neural é composta de NC camadas, cada qual com um número respectivo de entradas e saídas. O objetivo da rede é classificar determinado vetor de entrada como pertencente a alguma das possíveis classes de saída. O número de entradas da primeira camada é fixo como a dimensão do vetor de entrada e, por convenção, o número de saídas da última camada é fixo no número de classificações possíveis.

Nomenclatura de variáveis:

NC , número de camadas

$\boldsymbol{w}_i$, matriz de coeficientes da camada i

$\boldsymbol{b}_i$, vetor linha de bias da camada i

$\boldsymbol{z}_i$, vetor linha ponderação das entradas da camada i pelos respectivos pesos

$\boldsymbol{a}_i$, vetor linha de ativação da camada i

$f_i(\boldsymbol{z})$, função de ativação da camada i

$\boldsymbol{x}$, vetor linha de entrada

$\boldsymbol{y}$, vetor linha de saída correta e previamente conhecida para dada entrada

$\widetilde{\boldsymbol{y}}$, vetor linha de saída aproximada pela rede (ativações da última camada)

η , taxa de aprendizado

λ , coeficiente de normalização

R , classe correta a que uma entrada pertence

As equações básicas de propagação da rede:

$$\boldsymbol{z_i} = (\boldsymbol{w_i} \times \boldsymbol{a_{i-1}}) + \boldsymbol{b_i}$$

$$\boldsymbol{a_i} = f_i(\boldsymbol{z_i})$$

Função de custo e objetivo da regressão

Para mensurar a qualidade de uma classificação, é escolhida uma função de custo. Sendo a função de custo bem definida (de forma que tenda a menores valores para melhores respostas), minimizar o custo de uma rede aproxima-se de classificar corretamente a entrada.

O treinamento consiste em aplicar correções sucessivas aos coeficientes do sistema de forma a minimizar o custo. Para tal, é requerido o calculo da derivada parcial do custo em relação a cada variável ajustável do sistema.

A principal função de custo encontrada nas referências é a função de erro quadrático, mas após alguns testes resolveu-se usar uma função de custo logaritmo, que resultou em aprendizado mais rápido nas épocas iniciais e melhor adaptação à camada *softmax* implementada para última camada:

$$C(\widetilde{\boldsymbol{y}}, \boldsymbol{y}) = -\log(\widetilde{\boldsymbol{y}}_R)$$

$$\frac{dC(\widetilde{\boldsymbol{y}}, \boldsymbol{y})}{d\,\widetilde{\boldsymbol{y}}_R} = -\frac{1}{\widetilde{\boldsymbol{y}}_R}$$

Notar que o erro só depende do componente R do vetor de saída, associado à classe correta. Esta simplificação é justificada pelo fato de que referências do sistema são sempre vetores com todos os termos nulos, exceto pelo correspondente a classe correta. Será mostrado na análise da função *softmax* que todos os termos de $\widetilde{\boldsymbol{y}}$ influenciam em y_R, porém.

Função de ativação

A função de ativação é a conexão entre as ponderações da entrada de uma camada e a saída apresentada por cada neurônio. Requer-se que seja não linear nas camadas intermediárias (mas pode conter intervalos lineares).

Sigmoide

A função de ativação escolhida para as camadas intermediárias é a sigmoide, de acordo com maior parte das referências. Destaca-se também a derivada desta, pois será requerida para calculo das derivadas parciais de *backpropagation*:

$$\sigma(x) = SIG(x) = \frac{1}{1 + e^{-x}}$$

$$NOR(x) = \frac{dSIG(x)}{dx} = \frac{e^{-x}}{(e^{-x} + 1)^2} = SIG(x) \cdot (1 - SIG(x))$$

Softmax

A função de softmax é uma ponderação que objetiva aproximar a saída do sistema de uma distribuição probabilística. É usada na camada final para classificar o resultado. O somatório de todos os termos de um vetor que passa por softmax é um, e mantem-se a propriedade que maiores valores de entrada são mapeados para maiores valores de saída:

$$softmax(x)_j = \frac{e^{x_j}}{\sum_{<k>} e^{x_k}}$$

Por exemplo:

$$softmax(<2{,}4{,}6>) = \begin{cases} \dfrac{e^2}{e^2 + e^4 + e^6} \\ \dfrac{e^4}{e^2 + e^4 + e^6} \\ \dfrac{e^6}{e^2 + e^4 + e^6} \end{cases} = \begin{cases} 0.0158 \ldots \\ 0.1173 \ldots \\ 0.8668 \ldots \end{cases}$$

A derivada da softmax é mais trabalhosa que a da sigmoide. Começa-se notando que os termos são interdependentes, então a existe uma derivada parcial em relação a cada termo de entrada para cada termo da saída. Depois, nota-se que observando a função em relação a um termo fixo da entrada e o mesmo termo na saída:

$$softmax(x)_j = \frac{e^{x_j}}{A + e^{x_j}}$$

Onde A é uma constante, soma de todos outros termos exponenciados.

Já quando a derivada é observada relacionando um termo da entrada com um termo diferente na saída:

$$softmax(x)_j = \frac{B}{C + e^{x_j}}$$

Onde B é uma constante que representa o termo da saída observado, e C é uma constante que representa todos os outros termos da entrada exponenciados. As derivadas em cada caso:

$$\frac{d\left(\frac{e^x}{A + e^x}\right)}{dx} = \frac{A \cdot e^x}{(A + e^x)^2}$$

$$\frac{d\left(\frac{B}{C + e^x}\right)}{dx} = -\frac{B \cdot e^x}{(C + e^x)^2}$$

Notar que a derivada no caso do próprio termo é sempre positiva (ou seja, aumentar um dos termos de entrada sempre aumenta a respectiva saída ao aplicar *softmax*) e a derivada em relação aos outros termos é sempre negativa (ou seja, aumentar qualquer termo da entrada diminui todas as *outras* saídas).

Algoritmo de *backpropagation*

Para que sirvam como referência mais próxima da forma como os algoritmos foram programados, os elementos das variáveis serão indexados por colchetes nas equações a seguir. O primeiro índice refere-se à camada do sistema, o segundo índice a linha dos vetores/matrizes, e o terceiro índice refere-se à coluna da matriz.

O equacionamento da rede, assim indexado:

$$\boldsymbol{z}[c][j] = \left(\sum_{<k>} \boldsymbol{w}[c][j][k] \cdot \boldsymbol{a}[c-1][k] \right) + \boldsymbol{b}[c][j]$$

$$\boldsymbol{a}[c][j] = SIG(\boldsymbol{z}[c][j]), \text{ camadas intermediárias}$$

$$\boldsymbol{a}[c][j] = softmax(\boldsymbol{z}[c][j]), \text{ camada final}$$

A *regra da cadeia* do cálculo é usada intensivamente para justificar o algoritmo. A seguinte substituição será usada com frequência:

$$\frac{df\big(g(x)\big)}{dx} = \frac{df\big(g(x)\big)}{dg(x)} \cdot \frac{dg(x)}{dx}$$

A derivada do custo é inicialmente calculada em relação a cada neurônio de saída final (da última camada), e depois propagada para cada camada anterior. Para cada camada, é salva a quantia $\delta = dC/dz$. Com esta quantia, o erro é propagado para cada elemento de W e b pelas suas relações lineares com z.

Definindo-se *uC* como índice da última camada do sistema:

$$\widetilde{\boldsymbol{y}} = \boldsymbol{a}[uC]$$

Logo,

$$\frac{\partial C(\tilde{y})}{\partial \boldsymbol{a}[uC][j]} = \frac{\partial C(\widetilde{\boldsymbol{y}})}{\partial \widetilde{\boldsymbol{y}}} = -\frac{1}{\boldsymbol{a}[uC][R]}$$

A última camada é softmax, então a propagação do custo tem a forma (regra da cadeia):

$$\frac{\partial C(\tilde{y})}{\partial \boldsymbol{z}[uC][j]} = \frac{\partial C(\tilde{y})}{\partial \boldsymbol{a}[uC][j]} \cdot \frac{\partial \boldsymbol{a}[uC][j]}{\partial \boldsymbol{z}[uC][j]}$$

$$\frac{\partial C(\tilde{y})}{\partial \boldsymbol{z}[uC][j]} = -\frac{1}{\boldsymbol{a}[uC][R]} \cdot \frac{\partial softmax(\boldsymbol{z}[uC])_R}{\partial \boldsymbol{z}[uC][j]}$$

Para propagar a derivada do custo em relação a cada incógnita livre das camadas anteriores (que usam ativação sigmoide):

$$\frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{z}[c]} = \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{z}[c+1]} \cdot \frac{\partial \boldsymbol{z}[c+1]}{\partial \boldsymbol{z}[c]}$$

- O primeiro termo é equivalente a:
$$\frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{z}[c+1]} = \boldsymbol{\delta}[c+1]$$
- E o segundo termo é obtido pela aplicação da regra da cadeia:
$$\frac{\partial \boldsymbol{z}[c+1]}{\partial \boldsymbol{z}[c]} = \left(\frac{\partial \boldsymbol{z}[c+1]}{\partial \boldsymbol{a}[c]} \cdot \frac{\partial \boldsymbol{a}[c]}{\partial \boldsymbol{z}[c]}\right)$$
  - Onde
$$\frac{\partial \boldsymbol{z}[c+1][\#]}{\partial \boldsymbol{a}[c][*]} = w[c+1][\#][*]$$
(notar que a indexação cruzada equivale a matriz $w[l+1]$ transposta)
  - E
$$\frac{\partial \boldsymbol{a}[c][j]}{\partial \boldsymbol{z}[c][j]} = NOR(\boldsymbol{z}[c][j])$$

Em notação matricial, a soma das contribuições pode ser obtida por:

$$\boldsymbol{\delta}[c] = \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{z}[c]} = (\boldsymbol{w}[c+1]\{transposta\} \times \boldsymbol{\delta}[c+1]) \odot NOR(\boldsymbol{z}[c])$$

De posse da derivada parcial do custo em relação à ponderação de cada camada, $\delta[c]$, pode ser calculada a derivada parcial do custo em relação a cada incógnita ajustável do sistema:

$$\frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{b}[c][j]} = \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{z}[c][j]} \cdot \frac{\partial \boldsymbol{z}[c][j]}{\partial \boldsymbol{b}[c][j]} = \boldsymbol{\delta}[c][j] \cdot 1$$

$$\frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{w}[c][j][k]} = \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{z}[c][j]} \cdot \frac{\partial \boldsymbol{z}[c][j]}{\partial \boldsymbol{w}[c][j][k]} = \boldsymbol{\delta}[c][j] \cdot \boldsymbol{a}[c-1][k]$$

Para dada entrada, o conjunto dessas derivadas parciais forma o gradiente que pode ser subtraído do sistema para reduzir o custo. Como este gradiente é relativo à entrada específica, o processo é repetido para diversas amostras de entrada, na esperança que o sistema encontre um mínimo local de baixo custo para a maioria das amostras. A taxa η define a proporção entre o gradiente calculado e a correção:

$$w[c][j][k] \to w[c][j][k] - \eta \cdot \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{w}[c][j][k]}$$

$$b[c][j] \to b[c][j] - \eta \cdot \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{b}[c][j]}$$

Normalização L2

Com os valores de entrada e saída limitados ao intervalo ± 1, pode-se intuir que o sistema seja mais sensível quando os pesos das camadas são menores que a unidade, pois qualquer elemento que multiplique uma entrada por valores *muito altos* tem de ser compensado por outros fatores também *muito altos* para evitar saturação dos neurônios. Além disso, como existem vários mínimos locais no sistema, pode-se arbitrariamente optar por mínimos próximos da região onde os pesos de $\boldsymbol{w}$ e $\boldsymbol{b}$ são *baixos*.

A normalização L2 é uma maneira de punir variáveis de valor elevado supondo adição ao custo quadrático do sistema o quadrado de cada uma delas, escalado por um fator λ:

$$C_{L2}(\widetilde{\boldsymbol{y}}, \boldsymbol{y}) = C(\widetilde{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \cdot \left(\frac{1}{2} \sum_{<J,K>} \left(w_{j,k}\right)^2 + \frac{1}{2} \sum_{<J>} \left(b_j\right)^2 \right)$$

Assim, no cálculo da derivada parcial do custo em relação a cada variável, têm-se:

$$\frac{\partial C_{L2}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{b}[c][j]} = \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{b}[c][j]} + \lambda \cdot \boldsymbol{b}[c][j]$$

$$\frac{\partial C_{L2}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{w}[c][j][k]} = \frac{\partial \boldsymbol{C}(\widetilde{\boldsymbol{y}})}{\partial \boldsymbol{w}[c][j][k]} + \lambda \cdot \boldsymbol{w}[c][j][k]$$

Na prática, a normalização L2 gera tendência de cada peso à zero, que deve ser contrabalanceada por uma contribuição para diminuição do custo para que o peso em questão equilibre em valor não nulo.

Características do programa, leitor do banco

Cada elemento do banco de dados é representado por uma estrutura denominada dig_ref_t (**t**ipo de **ref**erência de **díg**ito), contendo a imagem e etiqueta:

dig_ref_t
imagem, 28 × 28 bytes etiqueta, 1 byte

Como os bancos possuem milhares de amostras, a imagem foi mantida em bytes para minimizar consumo de memória (este é o formato original, então não há perda de informação). Durante o treino, as imagens e etiquetas são convertidas para vetores de ponto flutuante, conforme necessário, para montar *pacotes* de amostras. A expansão para ponto flutuante escala o requerimento de memória por cerca de quatro ou oito vezes (forem usados *floats* ou *doubles*, respectivamente).

Um vetor de referência é denominado **`dig_refs_t`**.

Foi criada uma função que carrega um banco e associa cada imagem a uma etiqueta:

```
par< dig_refs_t, string> EMNIST_carrega_referencias
    ( string arq_imgs, string arq_lbls, bool transpoe )
```

Esta função está declarada no arquivo *emnist_leitor.hpp*. O primeiro e segundo argumentos são o caminho para os arquivos de imagens e etiquetas, respectivamente, e o terceiro argumento é um booleano que, se verdadeiro, gira cada imagem 90 graus (percebeu-se que no EMNIST elas são fornecidas rotacionadas em relação ao MNIST), deixando-as *de pé*, por conveniência.

O par retornado contém um vetor de dígitos de referência e uma *string* informativa. O vetor retornado pode ser usado para construir um objeto do tipo *classes_separadas_t*, que separa as amostras de acordo com as etiquetas e gera o vetor $\boldsymbol{y}$ correspondente a cada etiqueta única.

Características do programa, classes e funções numéricas

O arquivo *sisneu.hpp* define os seguintes tipos de dados auxiliares:

- **NUM_T** : define tipo de ponto flutuante utilizado. As outas classes são parametrizadas por esta definição. Testou-se o treinamento com *doubles* e *floats*, e não houveram mudanças significativas nos resultados, então o tipo deixado padrão é float, por ser menor e processado mais rapidamente;
- **vec_f** : vetor de ponto flutuante, typedef de std::vector<NUM_T>;
- **vec_i** : vetor de inteiros, typedef de std::vector<int>.

E declara as seguintes estruturas:

- **Matriz_t :** representa uma matriz através de um vetor de vetores. Cada vetor é uma linha, e cada elemento deste uma coluna;
- **camada_neural_t :** contém uma matriz representando os pesos $\boldsymbol{w}$, e um vetor representado o bias $\boldsymbol{b}$;
- **gradiente_camada_neural_t :** contém matriz para dC/dw e vetor para dC/db;
- **gradientes_t :** o gradiente de várias camadas, usado para representar resultado completo de *backpropagation*;
- **sistema_neural_t :** agrega camadas neurais, fornece métodos para *backpropagation* e classificação de entrada. Pode ser salvo/carregado de arquivo;
- **classes_separadas_t :** organiza amostras em classes e gera $\boldsymbol{y}$ referência de cada classe para treinamento/teste;

Características do programa, treino de sistema

Ao ser criado, um sistema deve ser dimensionado e uma estimativa inicial fornecida às variáveis. Foram criadas as funções *dimensiona* e *sorteia_coefs* para estas tarefas.

A função *dimensiona* tem como argumentos o número de entradas do sistema, o número de camadas e um vetor com o número de neurônios por camada intermediária.

Para o treino, as funções principais são *back_prop_inc* e *deriva_por*. A primeira calcula o gradiente do custo do sistema para dada entrada e saída, e a segunda aplica gradiente ao sistema.

Existe um detalhe específico da implementação, porém. Vide assinatura da função de *backpropagation*:

```
void back_prop_inc ( vec_f & x, vec_f & y, gradientes_t & dest);
```

x e y são a entrada e saída referência, respectivamente, mas o último argumento, *dest*, é uma referência ao local onde os gradientes calculados devem ser incrementados. Essa decisão de design foi tomada ao perceber-se que, durante os treinos do sistema, grande parte do tempo de execução era destinado à alocação de memória. Este tempo foi reduzido optando-se por manter memória reservada para os gradientes e reutilizá-la para cálculo de cada pacote de treino (isto aumentou o código, mas o ganho em desempenho foi confortante).

Essa decisão é transparente, porém, já que o treino do sistema se dá pela função auxiliar `treina_sistema_epoca`, que aplica os treinos e reserva memória internamente:

```
treina_sistema_epoca (

        sistema_neural_t & S,

        classes_separadas_t const & treino,

        int const N_por_pacote,

        NUM_T const eta,

        NUM_T const lambda);
```

Essa função treina o sistema S, usando as amostras de $treino$, em pacotes de tamanho N_por_pacote, com $\eta = eta$ e $\lambda = lambda$. São criados quantos pacotes quanto necessários para exaurir as amostras fornecidas, e os resultados de cada pacote aplicados imediatamente ao sistema fornecido.

Exemplo do treinamento do sistema é apresentado nos programas em *treino_letras.cpp* e *treino_numeros.cpp*, na pasta *treino*.

Características do programa, teste de sistema

Para testar um sistema, foi criada a função auxiliar *testa_sistema*, que recebe um sistema, um objeto de classes separadas, e retorna uma estrutura com os resultados gerais e por classe. A função aplica a entrada relativa a cada amostra e conta o número de acertos.

Exemplo do teste do sistema é apresentado nos programas em *teste_letras.cpp* e *teste_numeros.cpp*, na pasta *treino*.

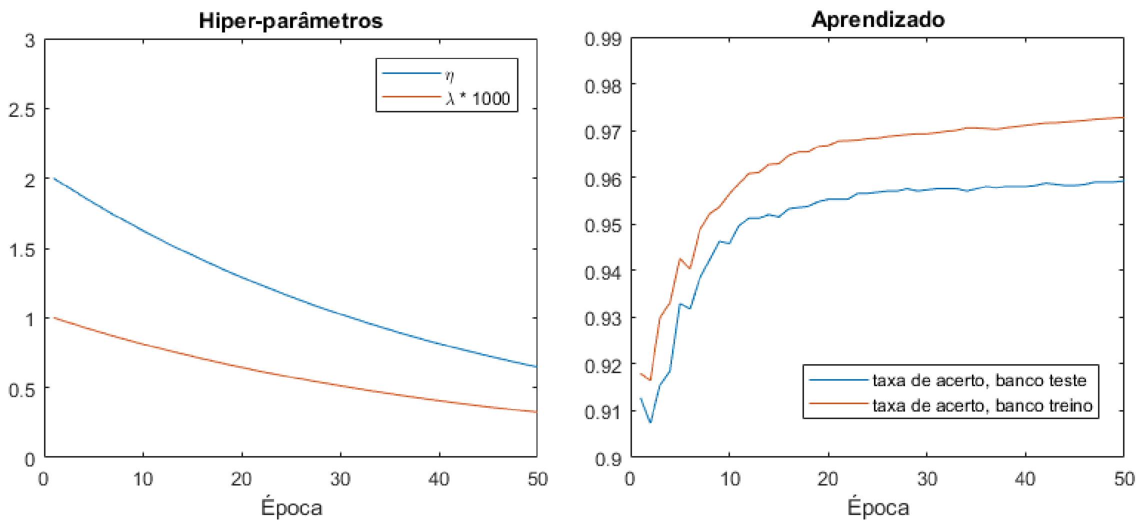
Características do programa, programa exemplo de uso de redes treinadas

Para demonstrar como o código elaborado aplica-se ao desafio, foi gerado um programa exemplo, no arquivo *exemplo.cpp*, na pasta *exemplo*.

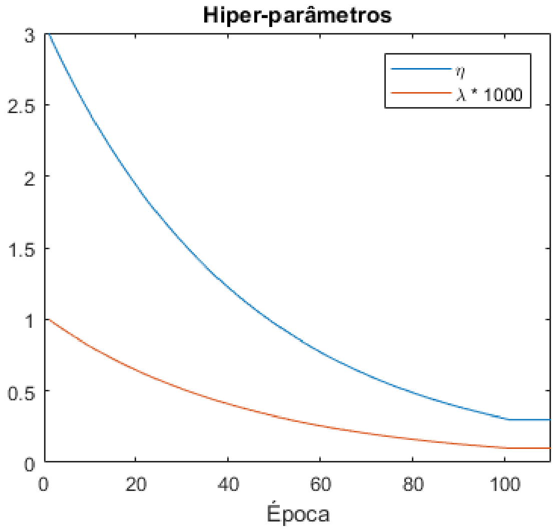
Este programa carrega duas redes neurais previamente treinadas, uma para letras e outra para números, e as aplica em imagens fornecidas como desafio.

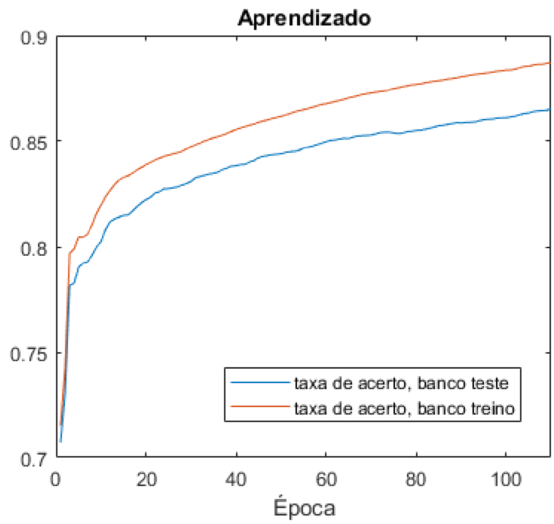
As redes para o exemplo foram treinadas durante diversas épocas, percebeu-se que reduzir gradativamente η fornecia maior taxa de acerto, e introduzir λ para normalização diminuía overfitting do sistema (taxa de acerto em banco de teste mais próxima da taxa de acerto no banco de treino). Perceberam-se também melhores resultados tornando a média da entrada zero, e este processamento foi mantido. O aprendizado de cada sistema fornecido como exemplo é ilustrado a seguir:

Sistema para reconhecimento de números



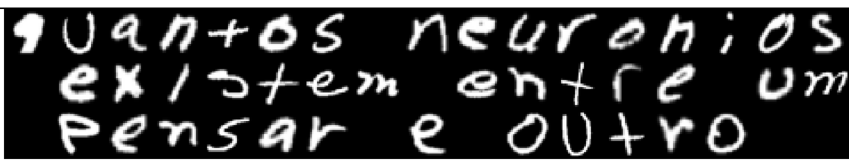
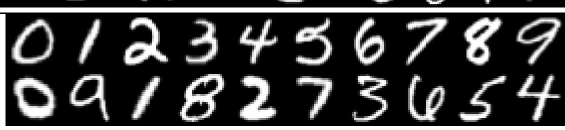
Sistema para reconhecimento de Letras





O sistema para números, mais simples, aprende mais rápido e melhor, atingindo taxa de 96% em cinquenta épocas, já o sistema para reconhecimento de letras passa a ter aprendizado muito lento depois de cerca de cem épocas e foi treinado até cerca de 87% de taxa de acerto (essas taxas são contra o banco de teste, com amostras independentes da do banco de treino).

Foram sorteados alguns elementos do banco para gerar um bitmap representando texto escrito à mão (essas imagens são carregadas, divididas em áreas de 28 por 28 pixels, e alimentadas ao sistema). O resultado do programa:

Entrada	Saída
	quqntos neuronjos existem qntre um pensar e outro
	0123456789 2918273654

Referências:

Vídeo, **But what *is* a Neural Network? | Deep learning, chapter 1**
https://www.youtube.com/watch?v=aircAruvnKk
(essa série de vídeos é muito, muito, **muito** boa)

Vídeo, **Gradient descent, how neural networks learn | Deep learning, chapter 2**
https://www.youtube.com/watch?v=IHZwWFHWa-w

Vídeo, **What is backpropagation really doing? | Deep learning, chapter 3**
https://www.youtube.com/watch?v=Ilg3gGewQ5U

Vídeo, **Backpropagation calculus | Deep learning, chapter 4**
https://www.youtube.com/watch?v=tIeHLnjs5U8

e-book **Neural Networks and Deep Learning**
http://neuralnetworksanddeeplearning.com/
(o capítulo 1 é uma boa visão geral do processo de aprendizado, os capítulos 2 e 6 demonstram a maioria das equações utilizadas)

e-course **CS231n: Convolutional Neural Networks for Visual Recognition**
http://cs231n.stanford.edu/
(pelo que entendi é uma cadeira de um curso de Stanford, mas as notas de aula estão abertas para acesso público)