UNICAMP

Projeto Computacional 1 - Redes Neurais

Aprendizado de Máquinas: Aspectos Teóricos e Práticos
MS571 / MT571

Amanda Rocha Faga - 212573

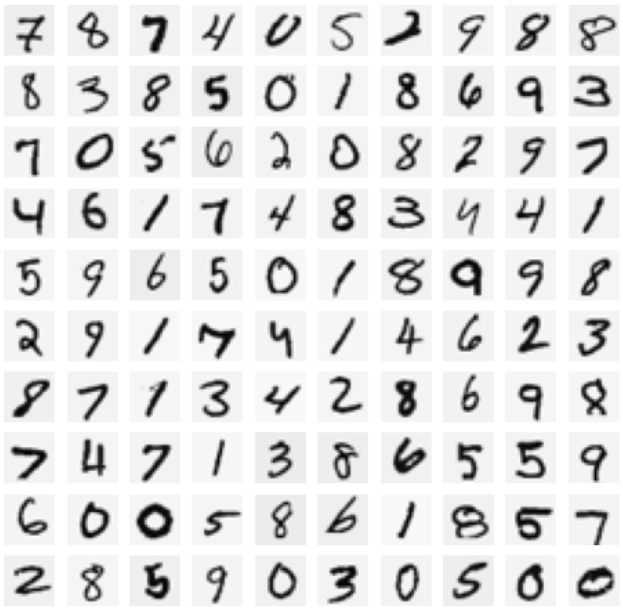
Caique Oliveira Alves da Silva - 168023

Vítor Hugo Miranda Mourão - 137856

2° Sem/2022 - Campinas - SP

Introdução

Nesse projeto, nosso objetivo será resolver os exercícios propostos visando ter uma maior compreensão sobre os tópicos de machine e deep learning. Ao final desta atividade, esperamos classificar, com boa precisão, imagens de dígitos escritos à mão, classificando-as de acordo com seus respectivos números distribuídos entre inteiros de zero e nove. Veja a seguir alguns exemplos de imagens a serem classificadas, através de uma implementação em Python com o auxilio das bibliotecas Pandas, Numpy, Scipy, Copy e Matplotlib.pyplot:



Deste modo, nossa rede neural será implementada e testada em uma base MNIST, possuindo 400 atributos, que por sua vez representarão nossas 10 classes preditas (dígitos de 0 a 9).

Modelagem e Implementação

Nesta seção, entraremos em mais detalhes a respeito da modelagem e da implementação realizada.

Primeiramente, nota-se que este projeto consiste em um problema de classificação, no qual o computador deve identificar em qual das 10 classes - que neste caso são os dígitos entre 0 e 9 - se enquadra determinado exemplo, que neste caso é a imagem de um dígito escrito à mão.

Observa-se também que este é um problema de aprendizado supervisionado, pois os exemplos de treinamento são rotulados, ou seja, o computador é treinado tendo acesso à classe correta de cada exemplo.

Para solucionar o problema, utilizou-se uma das principais técnicas do Aprendizado de Máquinas (Machine Learning): as Redes Neurais (Neural Networks).

A arquitetura da rede neural utilizada foi a seguinte:
- 401 neurônios na camada de entrada (Input Layer).
- 25 neurônios em uma única camada escondida (Hidden Layer).
- 10 neurônios na camada de saída (Output Layer).
- Função de ativação: sigmóide / logística.
- Método de treinamento: gradiente descendente.

Os hiperparâmetros do treinamento foram definidos como:
- Critério de parada: 400 iterações
- Taxa de aprendizado: $\alpha = 0{,}8$
- Termo regularizador: $\lambda = 1$

Com relação aos dados utilizados, os exemplos rotulados foram extraídos do arquivo "*imageMNIST.csv*", e os seus rótulos foram extraídos do arquivo "*labelMNIST.csv*". Estes dados consistem em 5.000 exemplos com 400 atributos, cada um correspondendo a um pixel da imagem 20x20 de um dígito manuscrito entre 0 e 9. Há exatamente 500 exemplos para cada dígito.

Utilizou-se a biblioteca *pandas* para importar estes dados para o Python. Entretanto, antes desta importação, foi realizada uma formatação no arquivo "*imageMNIST.csv*", alterando o separador decimal de vírgula (padrão brasileiro) para ponto (padrão internacional), e também removendo as aspas do arquivo, para que os atributos fossem lidos como variáveis *float*, e não como *string*.

Com relação ao código implementado, foram definidas diversas funções, sendo as principais:

1) Funções para o funcionamento da rede neural:

- *sigmoid*: define a função de ativação como sendo a sigmóide: $g(z) = \frac{1}{1+e^{-z}}$

- $sigmoid\_gradient$: utiliza a função *sigmoid* para calcular a derivada da função de ativação através da fórmula $g'(z) = g(z) \times (1 - g(z))$, dada por uma propriedade dessa função.

- $activation\_forward$: utiliza a função *sigmoid* para calcular o *forward propagation*, ou seja, a ativação em cada unidade (neurônio), dada pela aplicação da sigmóide na soma ponderada das entradas nesta unidade.

- *prediction*: utiliza a função $activation\_forward$ para calcular a saída da rede em um exemplo qualquer.

2) Funções para o treinamento da rede neural:

- $rand\_init\_weights$: método de Xavier para a inicialização aleatória dos pesos.

- $theta\_init$: utiliza a função $rand\_init\_weight$ para obter a matriz , contendo todos os pesos iniciais.

- $compute\_cost$: utiliza as funções $activation\_forward$ e $sigmoid\_gradient$ para calcular o custo e realizar o *backpropagation*, bem como retornar o custo regularizado e o gradiente regularizado.

- $gradient\_descendent$: utiliza a função $compute_cost$ para treinar a rede utilizando o método do gradiente descendente.

3) Funções para a checagem do gradiente:

- $gradient\_check$: calcula a aproximação da derivada pela secante para a etapa de checagem do gradiente.

- $compute\_cost\_check$: utiliza a função $gradient\_check$ para calcular o custo, o custo regularizado, o gradiente e o gradiente regularizado.

- *vectorize*: os pesos são vetorizados e concatenados em um único vetor para a checagem do gradiente.

Em resumo, essas são as etapas do treinamento da rede neural:

1) Inicialização aleatória dos pesos.

2) Cálculo da imagem da função de hipótese para cada exemplo de treinamento através do *Forward Propagation*.

3) Calcular a função de custo J(Θ).

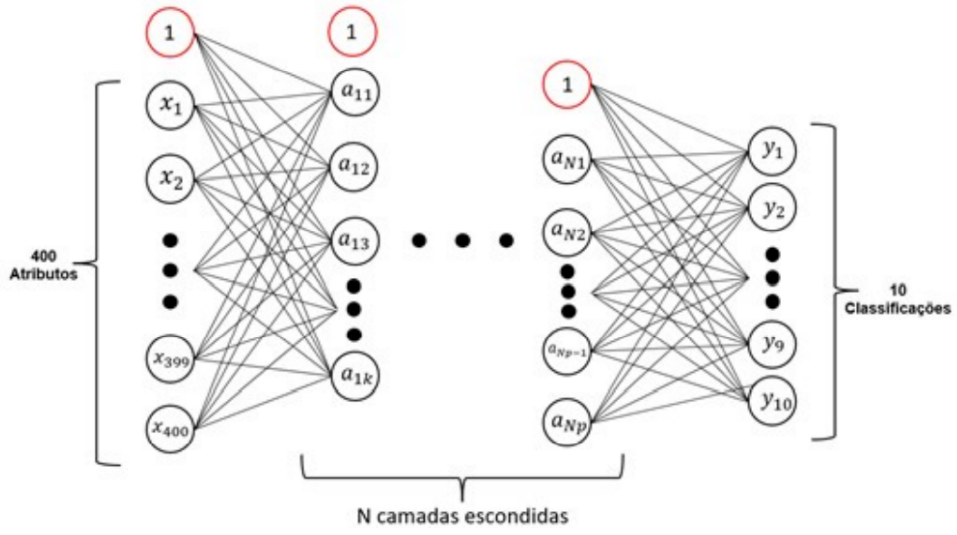
4) Calcular as derivadas parciais da função de custo com relação a cada um dos pesos através do *Backpropagation*.

5) A partir das taxas de aprendizado e das derivadas parciais, atualizar cada um dos pesos através do método do Gradiente Descendente.

6) Checar temporariamente o gradiente, para se certificar de que o método do gradiente descendente está obtendo resultados próximos ao método numérico, a partir da aproximação da derivada pela secante.

Parte I - Redes Neurais

Com o intuito de classificar cada uma dessas imagens escritas à mão, construiremos uma Rede Neural com arquitetura generalizada. Assim, podemos adicionar quantas camadas escondidas desejarmos ao processo, sendo que, cada camada poderá conter quantidades diferentes de neurônios. Como o nosso dataset dispõe de 400 atributos, e desejamos reconhecer 10 inteiros, de zero a nove, definimos a primeira camada com uma largura de 401 neurônios, na qual um deles é destinado ao bias. Além disso, definimos a última camada com uma largura de 10 neurônios, e N camadas escondidas entre a primeira e a última, com k neurônios cada. Observe:



Na estrutura de nosso código, o número de neurônios de entrada e saída também foram generalizados, entretanto para o exercício esses valores foram fixados em 401 e 10, respectivamente. Primeiramente, utilizaremos o algoritmo de vetorização do *forward propagation*,

tal que:

$$z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$$
$$a^{(j)} = g(z^{(j)}) = \frac{1}{1+e^{-z^{(j)}}}$$

Assim, executamos o código 6 vezes com os seguintes parâmetros, obtendo acurácias entre 92.18% e 92.86%:

número de interações = 400

$\alpha = 0.8$

$\lambda = 1$

número de camadas escondidas = 1

número de neurônios por camada escondida = 25

Rodada	Acurácia(%)
1	92.18
2	92.40
3	92.76
4	92.38
5	92.62
6	92.86

Em comparação, com 50 neurônios na camada escondida, obtemos acurácias que variam de 92.70% a 92.96%:

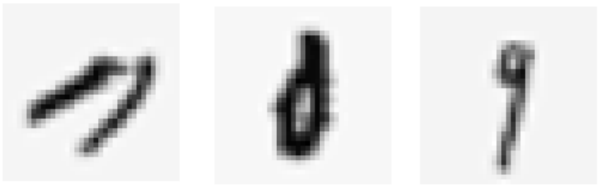
Rodada	Acurácia(%)
1	92.80
2	92.90
3	92.70
4	92.96
5	92.90
6	92.90

Podemos notar uma melhora significativa das acurácias, devido ao aumento no número de neurônios na camada escondida. Como, ao iniciar o método, os parâmetros θ são escolhidos aleatoriamente, o aumento no número de interações também leva a uma convergência do método e portanto, a uma menor variação dos valores de acurácia.

Agora, devemos observar onde o método não obteve sucesso em classificar os dígitos no caso de uma única camada escondida de 25 neurônios. Vejamos a seguir alguns exemplos aleatórios de imagens escritas à mão, seguidas de suas classificações incorretas encontradas pelo algoritmo:

7	2	7	9	9	4	0	1	6	4
3	4	2	3	7	4	7	9	7	4
6	1	0	0	6	5	2	5	3	8
2	6	1	4	1	0	6	1	9	6
2	0	1	3	6	3	1	6	6	5
3	3	2	9	6	0	3	0	8	0
7	0	7	6	8	4	6	8	5	0
3	8	2	9	8	5	1	9	2	7
2	3	0	2	2	6	3	2	8	7
4	6	3	2	7	6	1	9	2	6

Observe a seguir alguns casos em que até mesmo um ser humano não teria 100% de convicção ao classificar um dígito, aumentando também chance do algoritmo fracassar:



Veja a seguir o algoritmo do Backpropagation:

Treinamento: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

$\delta_j^{(l)}$: "erro" do nó j na camada l.

$\Delta_{ij}^{(l)} := 0 \qquad \forall l, i, j$

Para $i = 1$ até m

$a^{(1)} := x^{(i)}$

Calcular $a^{(l)}$ para $l = 2, 3, \cdots, L$ (*forward*)

$\delta^{(L)} := a^{(L)} - y^{(i)}$

Calcular $\delta^{(L-1)}, \delta^{(L-2)}, \cdots, \delta^{(2)}$: $\delta_i^{(l)} = \left(\sum_{j=1}^{n_{l+1}} \Theta_{ji}^{(l)} \delta_j^{(l+1)}\right) a_i^{(l)}(1 - a_i^{(l)})$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

---

derivadas parciais:

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}$ se $j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$ se $j = 0$

$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Apesar desse algoritmo funcionar com boa eficiência em diversas situações, devido ao seu grande número de passos, ele ainda pode produzir algumas falhas difíceis de serem percebidas. Por isso, é necessário adicionar uma etapa de checagem ao cálculo dos gradientes. Durante a checagem, estes são aproximados tanto pelo Backpropagation, quanto pela secante da função de custo e, logo depois, comparados. A aproximação do gradiente de checagem é calculado da seguinte forma:

$$\begin{aligned}
\frac{\partial}{\partial\theta_1} J(\theta) &\approx \frac{J(\theta_1+\epsilon,\theta_2,\theta_3,\cdots,\theta_n) - J(\theta_1-\epsilon,\theta_2,\theta_3,\cdots,\theta_n)}{2\epsilon} \\
\frac{\partial}{\partial\theta_2} J(\theta) &\approx \frac{J(\theta_1,\theta_2+\epsilon,\theta_3,\cdots,\theta_n) - J(\theta_1,\theta_2-\epsilon,\theta_3,\cdots,\theta_n)}{2\epsilon} \\
&\vdots \\
\frac{\partial}{\partial\theta_n} J(\theta) &\approx \frac{J(\theta_1,\theta_2,\theta_3,\cdots,\theta_n+\epsilon) - J(\theta_1,\theta_2,\theta_3,\cdots,\theta_n-\epsilon)}{2\epsilon}
\end{aligned}$$

Sendo um cálculo que pode gerar certa lentidão no código, adaptamos a função de custo *computeCost_checagem* para que seja rodada uma única vez e que, além do Backpropagation itera a função *checagem_gradiente* para calcular a aproximação. Comparando os

vetores dos gradientes calculados das duas formas, é possível ver a seguir que, os resultados ficaram bem próximos, como era esperado:

	0	1	2	3	4	5
0	-0.00123964	1.97407e-05	-1.85826e-05	1.59113e-05	-8.76038e-06	-1.10451e-05
1	-0.00125585	2.19185e-05	-7.88402e-06	5.6851e-06	-2.48354e-05	1.64532e-05
2	4.60562e-05	1.93457e-05	-8.50817e-06	-6.72798e-06	2.75537e-05	2.19696e-05
3	0.000549057	1.52201e-06	2.23878e-05	1.56208e-05	2.1308e-05	5.97432e-06
4	6.38773e-05	1.03341e-05	1.80805e-05	-2.8291e-06	1.46715e-05	2.59422e-05

Gradiente Backpropagation

	0	1	2	3	4	5
0	1.16867e-10	1.97408e-05	-1.85825e-05	1.59169e-05	-8.82359e-06	-1.11713e-05
1	1.16867e-10	2.19186e-05	-7.88391e-06	5.68021e-06	-2.47652e-05	1.60964e-05
2	1.16867e-10	1.93458e-05	-8.50805e-06	-6.7253e-06	2.75489e-05	2.1742e-05
3	1.16867e-10	1.52212e-06	2.23879e-05	1.56277e-05	2.12451e-05	5.8923e-06
4	1.16867e-10	1.03342e-05	1.80806e-05	-2.83038e-06	1.47221e-05	2.55469e-05

Gradiente aprox. pela secante

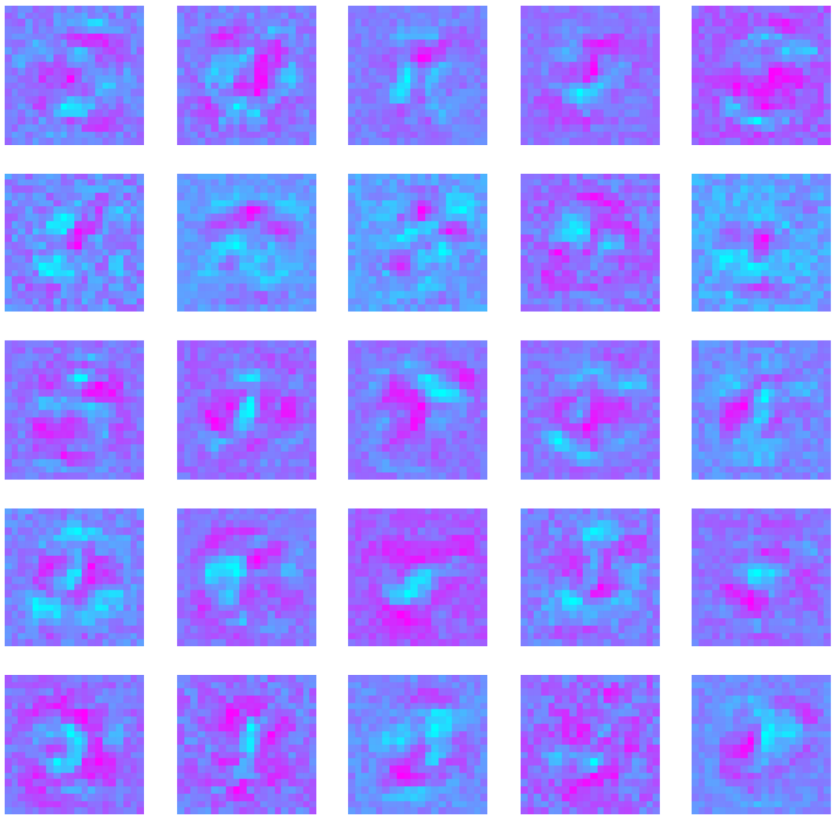
Visando minimizar a função de custo através gradiente conjugado, utilizamos a Lib Optimization do Scipy, e a função $Minimize(method =' CG')$. Também foi preciso atualizar o formato dos thetas, que até agora se encontravam como uma lista de matrizes. Para isso, vetorizamos theta, redefinido-o dentro da função de custo por meio das funções $Vetorize()$ e $Matrix()$. Assim, variando λ, o número de interações e utilizando 1% da base total (devido ao tempo do algorítimo), obtivemos os resultados:

Parâmetros		Custo	Interações reais
λ=1	1 interação	4.069082	1
λ=1	400 interações	1.823668	253
λ = 0.2	300 interações	0.717207	171

Disso, podemos concluir que a obtenção de uma função de custo mínima desejada está totalmente atrelada à escolha correta do parâmetro λ e do seu número de iterações. Perceba que, ao comparar os dois primeiros testes, como nos dois casos $\lambda = 1$, o número de interações se mostra inversamente proporcional ao custo, tornando o algoritmo mais ágil ao aumentar o número de interações. Em contrapartida, o último teste, com 300 iterações e $\lambda = 0.2$ obteve custo satisfatoriamente mais baixo em relação aos dois anteriores, já que possui alto número de interações e baixo λ. Sendo assim, também percebemos que λ é diretamente

proporcional ao custo do problema, sendo que um λ menor pode tornar o algoritmo mais eficiente.

Plotando os valores dos parâmetros da combinação linear das unidades de ativação de entrada para a primeira camada escondida, obtemos o seguinte resultado:



Parte II - Seleção do Modelo

Realizou-se a divisão dos 5.000 exemplos rotulados conforme sugerido nas instruções do projeto. Ou seja, foram selecionados:
- 3.000 exemplos rotulados para o conjunto de treinamento (60%).
- 1.000 exemplos rotulados para o conjunto de validação (20%).
- 1.000 exemplos rotulados para o conjunto de teste (20%).

Não houve viés na seleção dos dados, pois os mesmos foram embaralhados aleatoriamente antes da divisão. Pelo fato dos conjuntos serem grandes e possuírem o mesmo número de exemplos em cada classe, não é tão crítica a utilização de técnicas específicas de estratificação, onde a divisão seria feita com cada conjunto possuindo o mesmo número de exemplos de cada classe. O conjunto de validação tem como objetivo a identificação do valor ideal para o hiperparâmetro λ (termo regularizador), visando minimizar o *overfitting*,

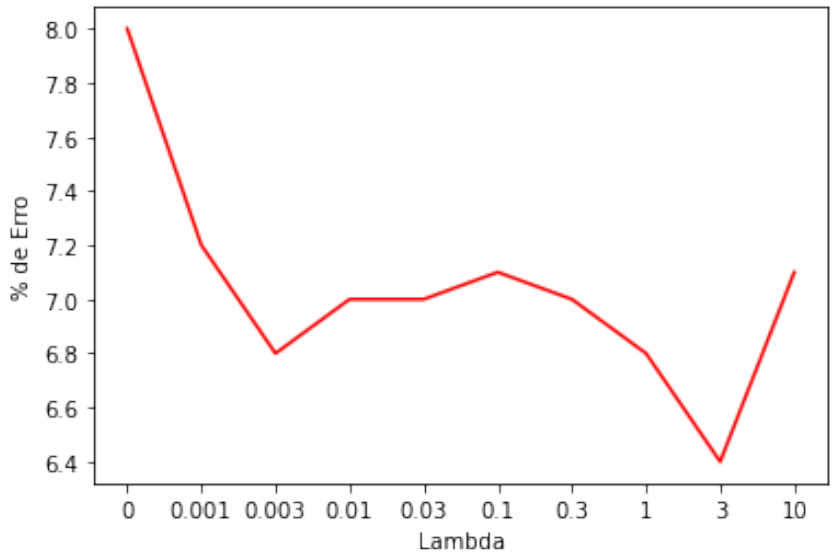
um problema no qual a função de hipótese possui uma alta variância, como se tivesse somente "memorizado" as classificações de treinamento, não aprendendo de fato a identificar os padrões.

Após a divisão dos dados, adotou-se a seguinte metodologia:
1) Treinou-se a rede neural com os exemplos do conjunto de treinamento para obter os seus pesos e *bias*.
2) Verificou-se, para diversos valores do termo regularizador λ (0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10) qual deles apresentava o menor erro nos exemplos do conjunto de validação. Assim, determinou-se o erro de validação.
3) Treinou-se novamente a rede neural, dessa vez com os exemplos dos conjuntos de treinamento e de validação, para obter os novos valores dos seus pesos e *bias*. Assim, determinou-se o erro de treinamento.
4) Calculou-se o erro nos exemplos do conjunto de testes. Assim, determinou-se o erro de teste.

Após este procedimento, não houve necessidade de retreinar a rede neural com todos os 5.000 exemplos dos três conjuntos, pois isso já havia sido efetuado na Parte I.

Na figura abaixo, observa-se o gráfico do erro de validação em função dos valores testados para o hiperparâmetro no passo 2 da metodologia.

Erro de validação em função do termo regularizador:



Por se tratar de um problema de classificação, para o cálculo das porcentagens de acurácia A e erro E nos três casos (treinamento, validação e teste), foram utilizadas as fórmulas a seguir:

1) E = (nº de exemplos classificados corretamente / nº de exemplos verificados)×100

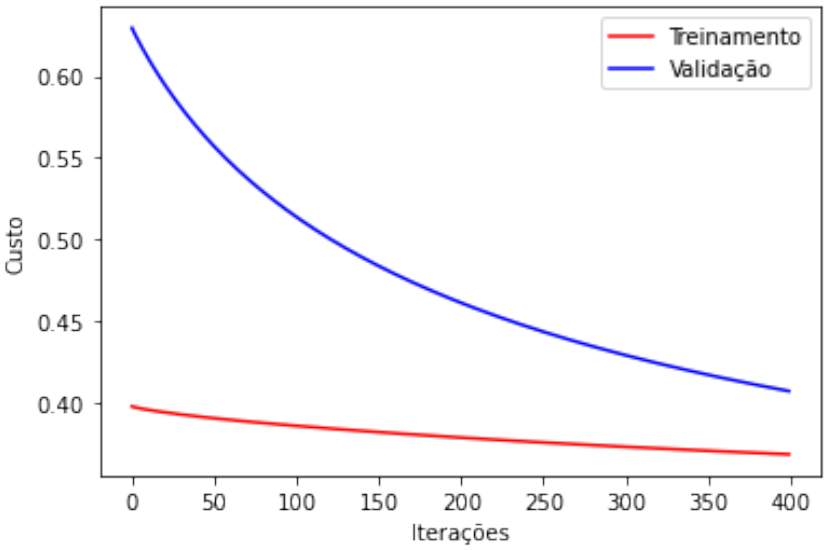
2) A = 100% - E

No conjunto de validação, a melhor acurácia foi de 93,60%, para $\lambda = 3$. Já no treino realizado com 4.000 exemplos rotulados (conjuntos de treinamento + validação), a acurácia foi de 95,88%. Por fim, no conjunto de teste, a acurácia foi de 92,20%. A partir dos valores de acurácia, pode-se determinar o erro em cada um dos casos. Em resumo, os resultados obtidos foram:

- Erro de treinamento: 4,12%
- Erro de validação ($\lambda = 3$): 6,40%
- Erro de teste: 7,80%

Na Figura 4.2 é possível observar a curva de aprendizado da rede neural, ou seja, o custo em função do número de iterações.

Curvas de aprendizado:



Conclusão

Ao analisar os resultados, observa-se que o erro de treinamento foi o menor de todos, o que já era esperado, visto que ele é calculado com relação aos exemplos que foram utilizados para treinar a rede neural. A acurácia de teste representa a acurácia final do modelo, que é de 92,20%. Ou seja, a cada 1.000 novas imagens que a rede neural analisar, espera-se que ela classifique corretamente 922. Para efeitos de comparação, se essas imagens fossem classificadas aleatoriamente, como há 10 classes, seria esperado que somente 100 imagens seriam classificadas corretamente.

O problema abordado neste projeto é um problema clássico de Machine Learning. Embora existam algoritmos mais avançados que conseguem valores maiores de acurácia, foi possível concluir que até mesmo modelos mais simples, como uma rede neural padrão, são o suficiente para se atingir níveis de acurácia satisfatórios, acima de 90%.

Por fim, observa-se que o resultado obtido é ainda mais positivo ao considerarmos o tempo necessário para executar o treinamento da rede neural, que foi de apenas poucos segundos nos computadores utilizados.

Referências

[1] Materiais (slides, notas de aula, exemplos de códigos) disponibilizados pelo professor João Batista Florindo.

[2] João Batista Florindo. **Machine Learning**. YouTube. Disponível em: <https://www.youtube.com/playlist?list=PLGwGFVrptiyRmFoDWxruNGgTu2cSPnTLX>. Acesso em: 04 de outubro de 2022.

[3] 3Blue1Brown. **Neural Networks**. YouTube. Disponível em: <https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R167000DxZCJB-3pi>. Acesso em: 04 de outubro de 2022.