

0.1 Introdução

O presente documento chamado Projeto II, tem como o objetivo apresentar os resultados obtidos durante a implementação e estudos realizados no tópico de redes neurais que faz parte do conteúdo da disciplina "Aprendizado de Máquinas". A primeira parte do documento faz referência a uma rede neuronal regularizada contendo quantidades distintas de neurônios e camadas escondidas, e todo o que tem a ver com os resultados solicitados e produzidos ao aplicar-o o código de programação. A segunda parte é todo o relacionado à rede da parte anterior mas com seleção de modelo.

O projeto fez uso novamente da base de dígitos manuscritos MNIST. Durante a exposição dos resultados, se irá mencionando conteúdos teóricos das duas regressões, para dizer os fatos mais relevantes ao fazer uso delas.

0.2 Parte I

Redes Neurais

Uma rede neural artificial é um modelo de computação inspirado na estrutura das redes neurais no cérebro. Consiste em um grande número de dispositivos básicos de computação (neurônios) que estão conectados uns aos outros em uma rede de comunicação complexa, por meio da qual o cérebro é capaz de realizar cálculos altamente complexos, é por isto que são chamadas como construções de computação formais modeladas de acordo com esse paradigma de computação.

A rede está composta de uma entrada, saída e uma ou mais camadas ocultas. Cada nó da camada de entrada é conectado a um nó da camada oculta "neurônio" e cada um destes são conectados a um nó da camada de saída. Geralmente, há algum peso associado a cada conexão. A camada de entrada representa a informação bruta que é alimentada na rede.

Cada neurônio faz regressão logística e é chamado de unidade de ativação. Cada

parâmetro θ_j é chamado de peso. As funções de ativação são precisas nas camadas para introduzir não linearidade, pois se a função fosse linear o modelo de regressão ou classificação obtido a partir da rede não seria tão poderosas. A função de ativação sigmoide é geralmente usada para cada camada porque combina comportamentos quase lineares, curvilíneos, quase constantes e deriváveis.

$$g(\theta^t \mathbf{x}) = \frac{1}{1 + e^{-\theta^t \mathbf{x}}}.$$

Agora, a rede utilizada neste projeto trabalha sob aprendizagem supervisionada e, portanto, precisa de um conjunto de treinamento que descreve cada saída e seu valor de saída esperado como segue,

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}.$$

O treinamento de uma rede multicamada se realiza mediante um processo de aprendizagem, para isto, se deve ter definida a topologia da rede isto é: número de neurônios na camada de entrada que depende do número de componentes do vetor de entrada, quantidade de camadas e o número de neurônios em cada uma delas, o número de neurônios na camada de saída que depende da quantidade de classes.

Já que não existe uma técnica para determinar a topologia anteriormente mencionada, esta escolha é feita pela experiência. Deste modo, a rede é regularizada com uma camada escondida que contem 25 neurônios, destacando que o algoritmo funciona também para um número qualquer de exemplos de treinamento, de classes, de camadas escondidas e de neurônios. A notação e os parâmetros aplicados no código são,

- $x^{(i)}$ dados de treinamento i .
- $y^{(i)}$ classe do dado de treinamento i .
- $L = 3$ Número total de camadas da rede.
- $n_l = 25$ Número de unidades (neurônios) na camada l .
- $K = 10$ Número de classes. Pois cada imagem tem escrito à mão um número de 0 até o 9.
- $z_i^{(j)}$ Soma ponderada das entradas na unidade i na camada j .
- $a_i^{(j)} = g(z_i^{(j)})$ ativação da unidade i na camada j .

- $\Theta^{(j)} = g(z_i^{(j)})$ matriz de pesos sobre a saída da camada j , saída que é uma das entradas da camada $j + 1$.
- $\Theta_{ab}^{(j)}$ parâmetro que multiplica a saída da unidade b da camada j para compor a entrada da unidade a na camada $j + 1$.

Dado que vamos usar uma rede neural multiclass, o problema se aborda utilizando a metodologia "um vs. todos". Portanto, agora $y^{(i)}$ é um vetor de longitude igual ao número de classes tal que

$$y_k^{(i)} = \begin{cases} 1 & \text{se } y^{(i)} \text{ pertence à classe } k \\ 0 & \text{outro caso.} \end{cases}$$

Uma amostra dos dados de treinamento são os mostrados na figura 1

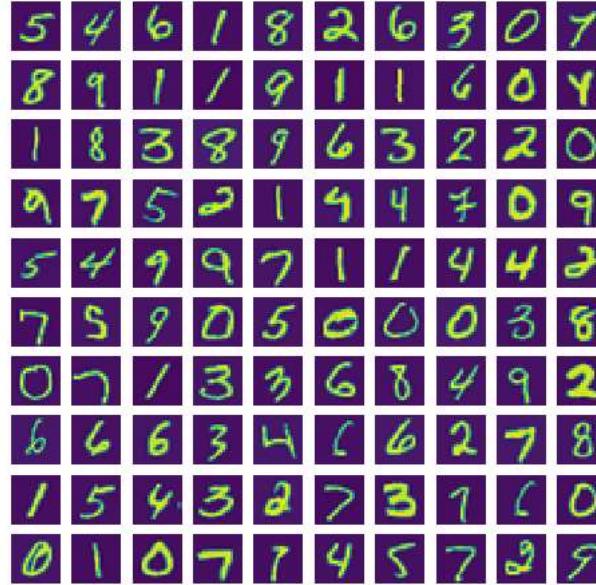


Figura 1: Manuscritos do documento MNIST

Se temos atributos X , parâmetro θ e variáveis de saída $y \in \{0, 1, \dots, 9\}$, então a função de hipóteses é definida como,

$$h_\theta(X) = g(\theta^T X) = \frac{1}{1 + e^{-\theta^T X}}.$$

Visto que cada unidade de saída faz uma regressão logística, logo para calcular o θ ideal fizemos uso da teoria vista na seção anterior e a função de custo para cada classe é

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)})(1 - h_\theta(x^{(i)}))].$$

Logo, a função de custo da rede é a soma dos custos de todas as saídas K ,

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_\theta(x^{(i)})_k) + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k) \right].$$

em que $h_\Theta(x)_k$ é a k -ésima saída da função de hipótese $y_k^{(i)}$ é o k -ésimo componente do vetor de saída y .

A Backpropagation é um algoritmo de aprendizagem supervisado, que emprega um ciclo de propagação para trás e adaptação de duas fases. O sinal de saída se compara com a saída desejada e se calcula um sinal de erro para cada uma das saídas. Partindo da camada de saída e indo para todos os neurônios da camada escondida que contribuem diretamente a saída são calculadas as saídas do erro, este processo se repete, camada por camada, até que todos os neurônios da rede tenham recebido um sinal de erro que descreve sua contribuição relativa ao erro total. Depois de isto os pesos de conexão de cada neurônio são atualizados para que a rede convirja ou possa classificar corretamente todas as atribuições de treinamento.

Definimos o erro do nó j na camada l como $\delta_i^{(l)}$ e na unidade de saída como: $\delta_i^{(L)} = a_i^{(L)} - y$. Para as demais camadas o erro é definido como

$$\delta_i^{(l)} = \left(\sum_{j=1}^{n_{l+1}} \Theta_{ji}^{(l)} \delta_j^{(l+1)} \right) a_i^{(l)} \left(1 - a_i^{(l)} \right)$$

Por outro lado, no caso quando os dados são muito grandes é interessante a vetorização, logo,

$$\delta_i^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} * a^{(l)} * (1 - a^{(l)})$$

em que $.*$ é o produto ponto-a-ponto. E, geral o algoritmo é,

Algorithm 1: Backpropagation

Input: X, k, m, y, L

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

for $i = 1$ até m **do**

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

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

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

Calcular $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

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

end

As derivadas parciais são obtidas pelo algoritmo,

Algorithm 2: Derivadas parciais do Algoritmo 1

```

if  $i \neq 0$  then
     $D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ 
else:
     $D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$ 
     $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \Delta_{ij}^{(l)}$ 
end

```

Para correr o modelo, é necessário usar o parâmetro θ aleatório. Empregando os resultados anteriores e por causa do tempo de corrida e complexidade do algoritmo, usamos número de iterações $n_{\text{iter}} = 1000$, valores $\lambda = 0.1, 1, 5, 10$ e 25 neurônios na camada oculta, se obteve uma precisão do conjunto de treinamento na seguinte tabela

λ	Precisão
0.1	95.18%
1	95%
5	94.18%
10	93.08%

Tabela 1: Precisão do algoritmo implementado.

Da tabela 1 concluímos que para maiores valores de λ a precisão diminui. Na figura 4 é evidente que o valor custo diminui para cada valor de λ en quanto o número de iterações incrementar. Sendo menor o custo para $\lambda = 0.1$, mas com precisão quase similar para $\lambda = 1$. Logo, para outras análises de resultados continuaremos usando o valor $\lambda = 1$.

Uma analise relevante para os problemas de classificação é usar a matriz de confusão M em que a posição M_{ij} contem o número de elementos que foram classificados na classe j e são realmente o número i . Para o caso mencionado anteriormente, obtemos a matriz da figura 2.

Foi testado o problema anterior aumentando o número de camadas escondidas, mas a precisão diminui significativamente, isso é pois quando temos maior número de camadas, o modelo está propenso ao overfitting.

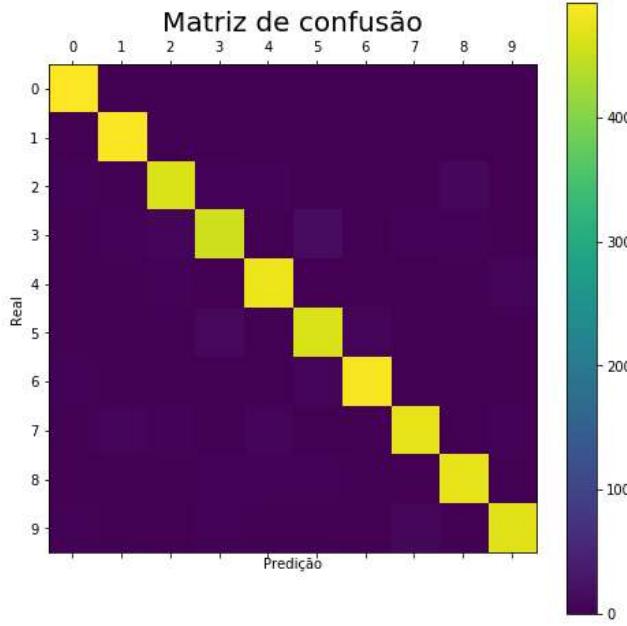


Figura 2: Matriz de confusão.

Na seção 0.4 temos a lista dos dados que não foram classificados corretamente. Na figura 3 temos uma amostra desses dados, onde (a, b) significa que pertence à classe a e foi classificado na classe b . Ao se tratar de números manuscritos, pode-se observar que tem dígitos que podem ser similares aos outros, como no caso do 4 que é bem similar ao 9. Mesmo com os dígitos 1 e 2.

O Python tem implementado librárias com o gradiente conjugado em que a complexidade do algoritmo é mais ótima, portanto o tempo de corrida é menor ademais, é mais preciso no sentido de que requer menos iterações (50 pelo menos), para obter uma precisão do conjunto de treinamento superior a 90%. Mas o problema é que o tempo de corrida é longo. Razão pela qual na prática é mais usado o gradiente descendente. No nosso algoritmo foi implementada a função do Python chamada "scipy.optimize.minimize" e na tabela 2 podemos observar os resultados onde variamos os $\lambda = 0.1, 1, 5, 10$ e o número de iterações onde observamos que o número de iterações é quase irrelevante em termos da precisão do algoritmo, pois quem influencia na precisão é a eleção do lambda.

Por outro lado, um resultado interessante para analisar é que cada linha de $\Theta^{(1)}$ contém informação sobre como cada camada atua na rede. Na figura 5 temos o

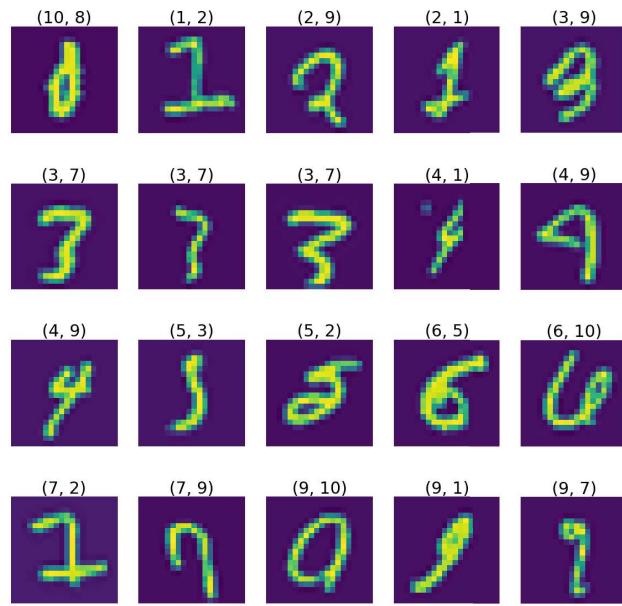


Figura 3: Amostra dos dados não classificados corretamente.

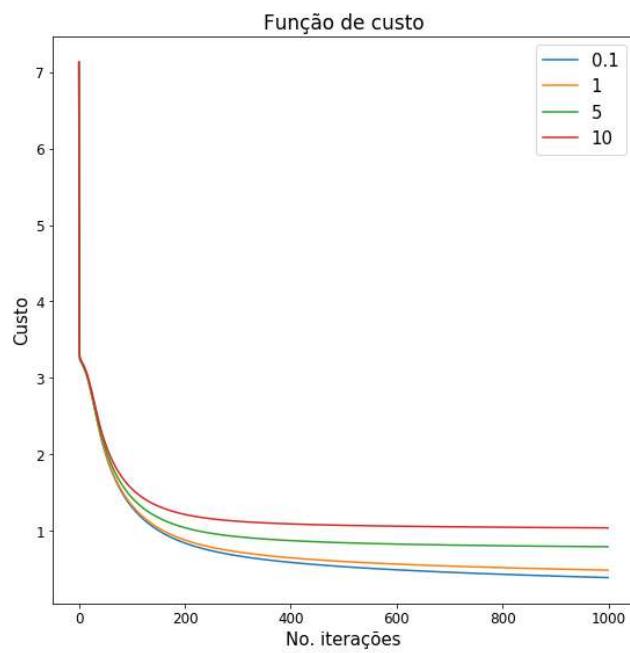


Figura 4: Valor da Função de custo vs No. de iterações.

Número de iterações	λ	Precisão
100	0.1	99.36%
	1	98.28%
	5	95.96%
	10	93.88%
400	0.1	100.0%
	1	99.58%
	5	96.22%
	10	94.36%
500	0.1	100.0%
	1	99.52%
	5	96.18%
	10	94.32%

Tabela 2: Precisão usando libraria do Python.

resultado para os 25 neurônios da camada escondida usando o algoritmo implementado manualmente. Vemos como vai centrando a informação no centro da imagem, onde geralmente está localizado cada dígito manuscrito. Observa-se que cada linha aparenta ser similar, mas os dados numéricos são diferentes. Cada gráfico representa uma porção dos tracos de cada dígito. Por exemplo, pode-se observar que nas primeiras imagens temos traços circulares, como dos dígitos 8, 0, 9, e nas últimas, temos porções de linhas como os números 1, 7, 2.

Tendo em vista que usando a libraria Python foi possível obter 100% de precisão no caso $\lambda = 0.1$, na figura 6 observamos como atua cada camada na rede. Observa-se o mesmo resultado anterior, o qual tem algumas imagens similares, mas os dados numéricos são diferentes. Também tem uma maior dispersão nos dados, para aumentar o rango de precisão no momento de classificar cada exemplo.

De outro modo, mesmo que o custo diminua como foi analisado na gráfica 4, é possível ter um bug. Por isso é importante checar as derivadas do gradiente descendente Backpropagation pela reta secante, isso é, verificamos para ϵ pequeno se

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}.$$

Se temos n parâmetros, então checamos para cada $j \in \{1, 2, \dots, n\}$ se

$$\frac{\partial}{\partial \theta_j} J(\theta) \approx \frac{J(\theta_1, \theta_2, \dots, \theta_j + \epsilon, \dots, \theta_n) - J(\theta_1, \theta_2, \dots, \theta_j - \epsilon, \dots, \theta_n)}{2\epsilon}. \quad (1)$$

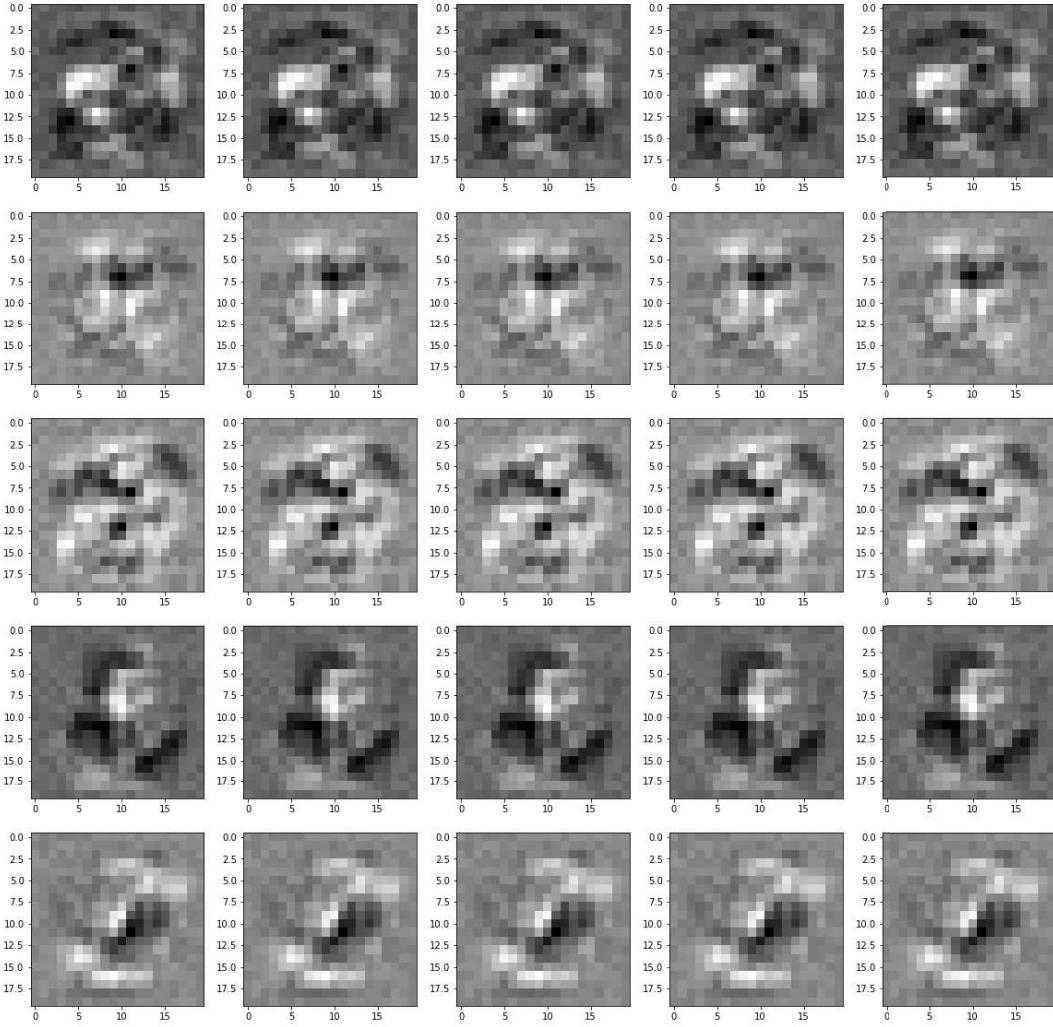


Figura 5: Influencia de cada camada para 1000 iterações e $\lambda = 1$.

Dado que a rede implementada para o problema dos manuscritos é grande de mais, então o tempo de checagem é longo. Portanto, foi necessário fazer uma rede pequena para checar as derivadas do gradiente descendente, usando o algoritmo implementado para a função custo. Para isso, foi selecionada uma rede com 3 unidades na entrada, 5 na camada escondida e 3 na saída, para 3 exemplos do treinamento.

Na figura 0.2 temos a topologia da rede, onde o nó $+1$ é o bias da camada 1 e a_0 é o bias da camada escondida. Cada arista do grafo contem o peso $\theta_{ij}^{(l)}$ e a derivada parcial $\Delta_{ij}^{(l)}$ e cada função de ativação a_j contem o valor sigmoidal para a soma ponderada dos pesos e cada valor x_j . Cada y_i são as unidades de saída e x_i as de entrada.

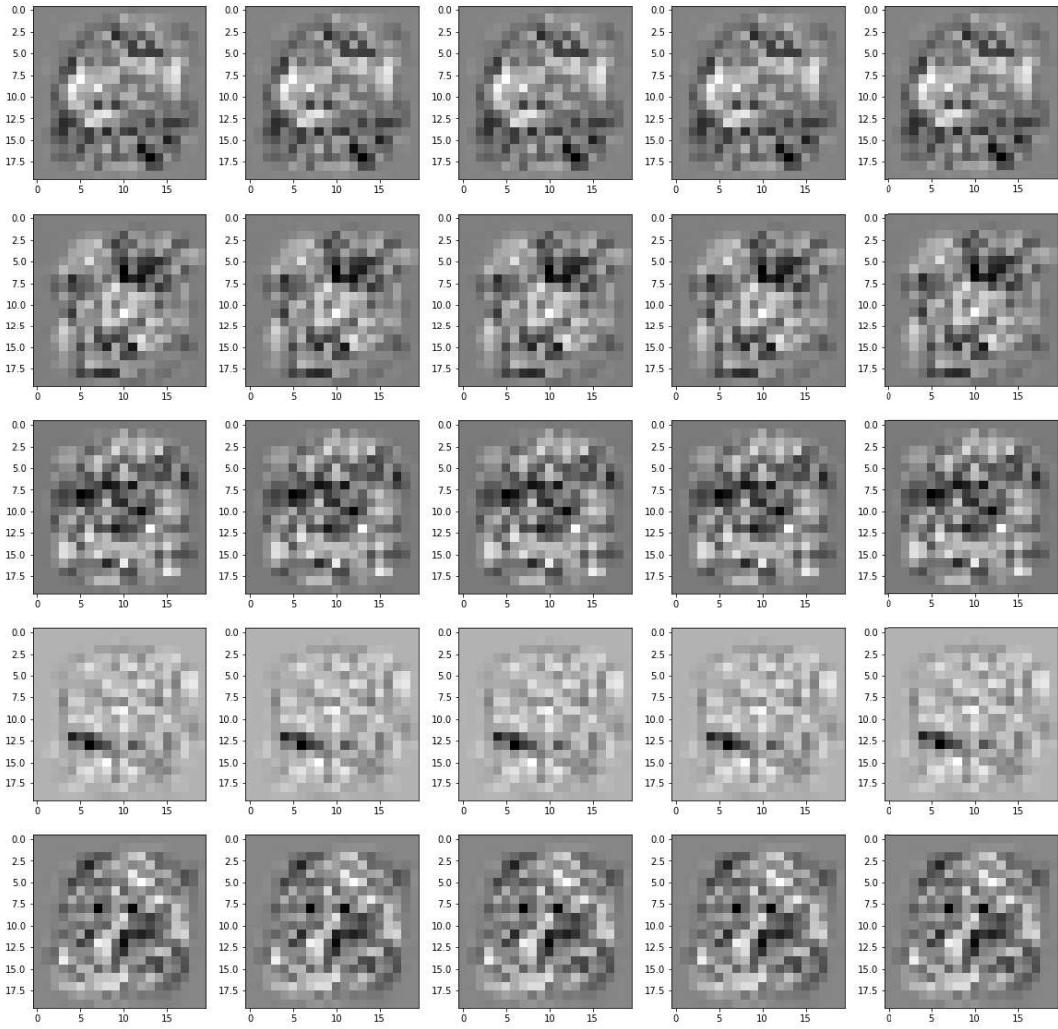


Figura 6: Influencia de cada camada usando libraria do Python, 500 iterações e $\lambda = 0.1$.

Uma vez definida a topologia da rede procedemos a checar, para isso foi definido $\epsilon = 10^{-4}$, uma tolerância da ordem 10^{-9} e valores dos pesos θ aleatórios. Para cada j foi calculado a diferença de (1) e o vetor final for testado pela norma 2. Finalmente concluímos que a função custo e o gradiente descendente não tem bug.

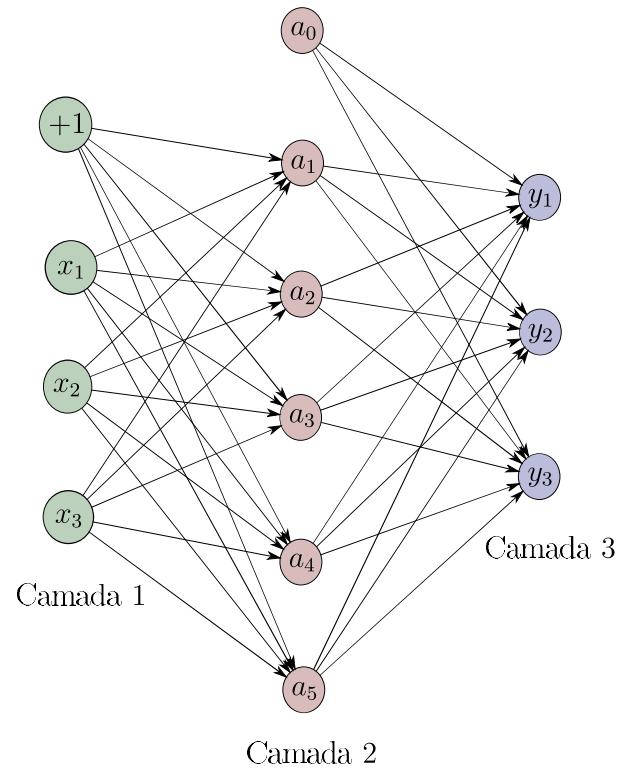


Figura 7: Rede neural com 3 unidades na entrada, 3 classes e 5 neurônios na camada escondida.

0.3 Parte II

Seleção e avaliação de modelos

Ao momento de criar modelos de regressão ou classificação é possível testar ou avaliar o modelo fazendo predição dos mesmos exemplos. Mas o modelo usará dados que são conhecidos. Portanto, uma outra forma de medir a generalização do modelo é usando a metodologia Holdout, que consiste em dividir aleatoriamente todos os m exemplos rotulados em exemplos de treinamento e teste. Assim, usamos a seguinte notação para cada grupo de dados, em que m é o número de exemplos de treinamento e m_{teste} o número de dados para o teste

$$\begin{aligned} & \left\{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)}) \right\}, \\ & \left\{ (x_{\text{teste}}^{(1)}, y_{\text{teste}}^{(1)}), (x_{\text{teste}}^{(2)}, y_{\text{teste}}^{(2)}), \dots, (x_{\text{teste}}^{(m_{\text{teste}})}, y_{\text{teste}}^{(m_{\text{teste}})}) \right\}. \end{aligned}$$

Ao momento de dividir os conjuntos é importante manter a proporção de amostras por classe. Nossa caso, temos 500 exemplos por cada classe, portanto, cada grupo terá a mesma proporção de exemplos de cada digito. Para isso foi criado um algoritmo que faz as amostras, reordenando cada classe e checando não ter termos repetidos. É de enfatizar que dependendo do algoritmo de busca, pode ter uma complexidade da ordem $O(\log m)$ ou $O(m^2)$, o qual poderia ser custoso computacionalmente quando ter número de exemplos por classe maiores.

Uma vez divididos os exemplos, o parâmetro θ é aprendido a partir dos exemplos de treinamento. Uma vez obtido, procede-se a calcular o erro de treinamento

$$J_{\text{teste}}(\theta) = \frac{1}{m_{\text{teste}}} \sum_{i=1}^{m_{\text{teste}}} \text{err}(h_{\theta}(x_{\text{teste}}^{(i)}, y_{\text{teste}}^{(i)})), \quad (2)$$

em que

$$\text{err}(h_{\theta}(x), y) = \begin{cases} 1 & \text{se foi classificado incorretamente} \\ 0 & \text{outro caso.} \end{cases}$$

Dependendo de cada modelo, podemos concluir a partir do erro se a rede adapta-se ao problema real e quantifica a habilidade de generalização do modelo.

Outro aspecto importante é a seleção dos hiperparâmetros, isso é, os melhores valores de λ ou número de neurônios da camada escondida. Para isso usamos a metodologia de validação cruzada dividindo os exemplos rotulados em 3, treinamento, validação

e teste. Em que os dados de treinamento são usados para a seleção do melhor parâmetro θ , os dados de validação para obter o melhor hiperparâmetro λ e o teste para avaliar o modelo.

Para m número de exemplos de treinamento, m_{cv} exemplos de validação cruzada e m_{teste} exemplos de teste, definimos a seguinte notação

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\},$$

$$\{(x_{cv}^{(1)}, y_{cv}^{(1)}), (x_{cv}^{(2)}, y_{cv}^{(2)}), \dots, (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})\}.$$

$$\left\{ \left(x_{\text{teste}}^{(1)}, y_{\text{teste}}^{(1)} \right), \left(x_{\text{teste}}^{(2)}, y_{\text{teste}}^{(2)} \right), \dots, \left(x_{\text{teste}}^{(m_{\text{teste}})}, y_{\text{teste}}^{(m_{\text{teste}})} \right) \right\}.$$

O conjunto é treinado usando os exemplos de treinamento de acordo com a metodologia da Parte I; posteriormente é testado no conjunto de validação. Daí, o parâmetro θ obtido no treino é usado para procurar o melhor λ , mudando os valores e definindo para o modelo final como aquele que tem o menor erro de validação. Finalmente, o modelo dado pelo θ do treinamento e o λ da validação é checado no conjunto de teste. Cada erro é calculado por (2) de acordo com cada tamanho das amostras e exemplos ([2]).

Dado que a proporção das amostras é aleatória, foi decidido mudar o tamanho da amostra de treinamento e testar cada erro (validação e treino) por meio das curvas de aprendizado como na figura 8, onde observa-se que o erro de validação é maior que o erro de treino. Ademais, quando o tamanho da amostra de treino aumentar, o erro de avaliação aumenta. O exercício foi feito para três valores de $\lambda = 0.1, 1, 3$ e percentagem do tamanho 55%, 60%, 65%, ..., 90%. Devido ao tempo de corrida, foram usadas 500 iterações.

Na figura 8 e 9 temos a comparação dos erros para $\lambda = 1$ e $\lambda = 0.1$ respetivamente, onde se obtém o resultado esperado. Mas no caso da figura 10 temos uma figura que é uma característica dos problemas com alta viés, pois quando o λ aumentar, a complexidade do algoritmo aumenta e tende ao underfitting.

Dos resultados anteriores conclui-se que a escolha do λ é importante, daí, uma análise relevante é comparar diferentes valores do λ e checar o erro de validação, sendo o

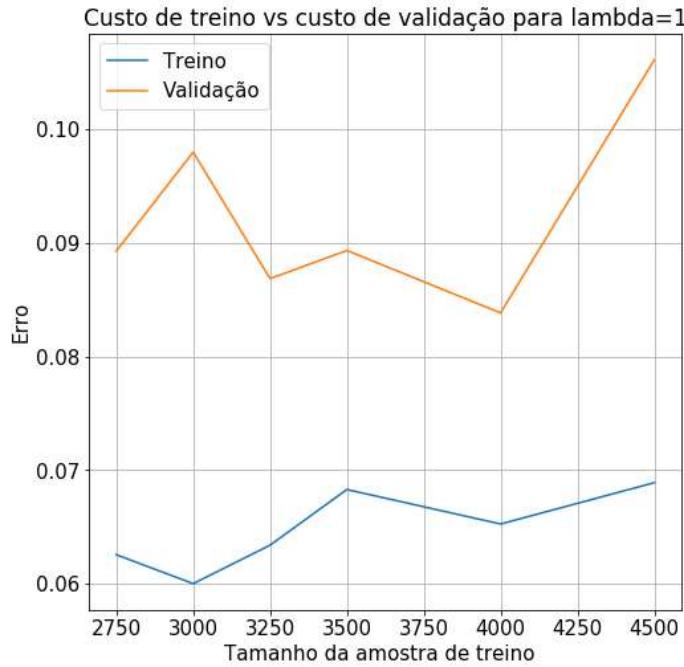


Figura 8: Custo de treino e validação em função do tamanho de treinamento para $\lambda = 1$.

λ ideal aquele onde o erro atinge o mínimo, partindo do θ obtido no treinamento em que o custo foi minimizado. Para percentagem de treino 70% foram testadas $\lambda = 0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10$ e obtemos na figura 11 que o melhor $\lambda = 1$, para obter menor erro. Observa-se que de modo geral, o erro aumenta en quanto o λ aumentar, pois estamos aumentando a complexidade do modelo e tem tendencia ao underfitting.

Independentemente do λ ideal, com o objetivo de checar o comportamento dos erros, partindo de diferentes valores de λ e uma proporção de 60/20/20, os resultados obtidos com 500 iterações temos os erros para o problema de classificação de dígitos apresentados na tabela 3. Observa-se que os valores J_{cv} e J_{teste} são maiores que o J_{treino} , que é o resultado esperado, pois trata-se de dados novos para o modelo. Ademais, na tabela 3, temos que o erro de treino aumenta en quanto o λ aumentar, pois para λ maior, o problema tende ao underfitting.

Partindo do fato que o hiperparâmetro ideal é $\lambda = 1$ (ver figura 11) e usando proporção 60/20/20 ($m_{treino} = 3000, m_{teste} = 1000, m_{cv} = 1000$), na tabela 4 temos que o erro de teste é menor que o erro de validação, o que é comum e bom sinal pois

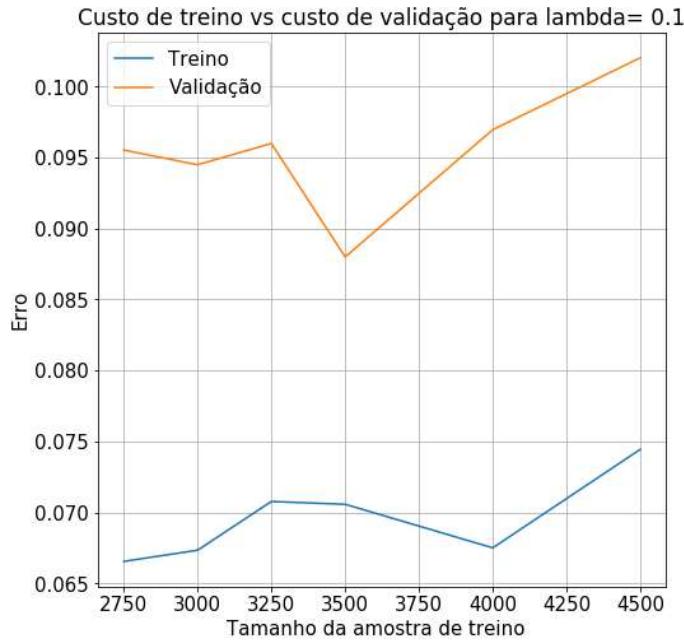


Figura 9: Custo de treino e validação em função do tamanho de treinamento para $\lambda = 0.1$.

λ	J_{treino}	J_{cv}	J_{teste}
0.1	2.96%	9.99%	8.59%
5	5.2%	10.7%	8.4%
10	6.49%	11.7%	9.29%

Tabela 3: Erros de treinamento, validação e teste.

quer dizer que é um bom modelo para generalizar. Além disso, podemos concluir que a variância é alta ou temos overfitting pois temos que os erros de validação e treino são o duplo do erro de treino. É de destacar que ao tratar-se de amostras aleatórias, os valores são diferentes para cada corrida, logo o λ ideal foi diferente cada vez. Portanto, uma solução poderia ser usar a metodologia k -fold que vai mudando os grupos de treino, avaliação e teste por cada corrida e pondera no final.

λ	J_{treino}	J_{cv}	J_{teste}
1	3.4%	9.9%	8.09%

Tabela 4: Erros de treinamento, validação e teste.

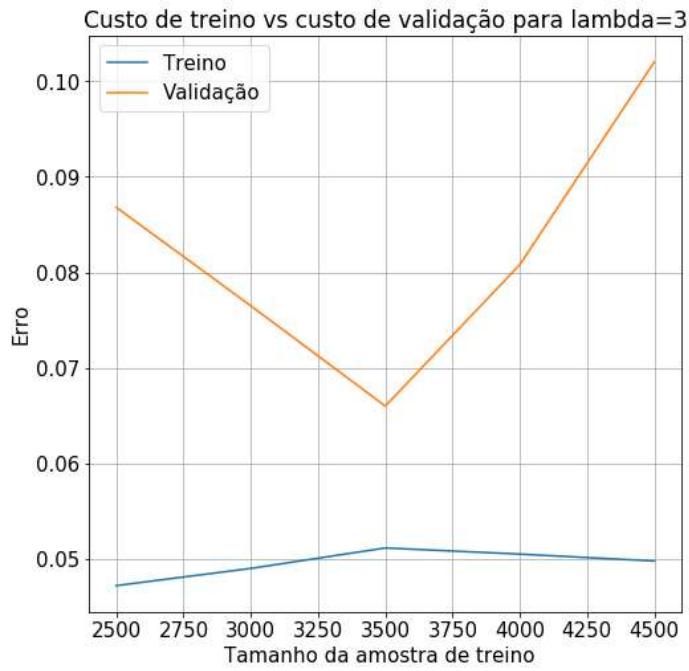


Figura 10: Custo de treino e validação em função do tamanho de treinamento para $\lambda = 0.1$.

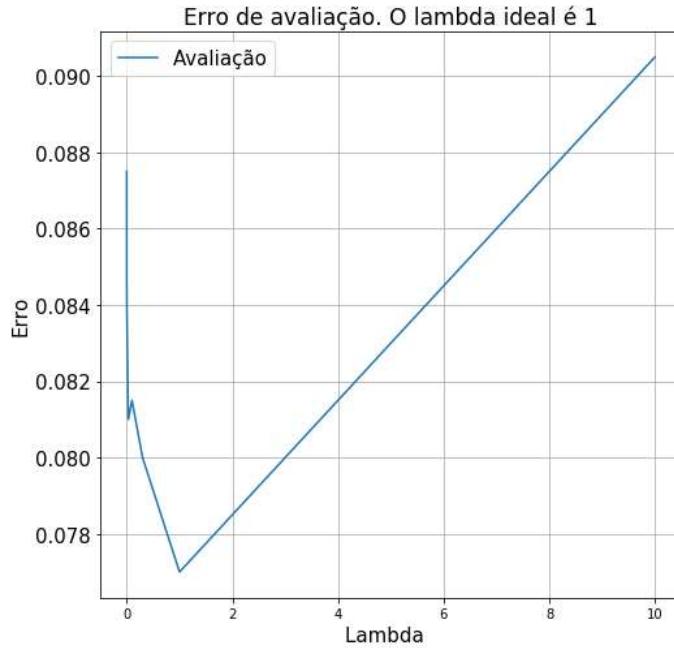


Figura 11: Erro de validação em função do λ .

0.4 Anexo

A seguinte tabela contém os dados mal classificados para o caso $\lambda = 1$ e 1000 iterações. O índice é o correspondente número na base de dados. (Número 10 se refere ao dígito 0).

Indice	Número real	Número predito
112	10	3
133	10	5
142	10	8
261	10	6
265	10	4
316	10	3
351	10	7
392	10	6
505	1	9
521	1	4
531	1	5
561	1	5
637	1	4
659	1	7
675	1	8
765	1	2
844	1	4
952	1	5
1003	2	7
1006	2	6
1025	2	10
1026	2	7
1041	2	3
1045	2	1
1057	2	5
1062	2	1
1087	2	9
1097	2	8
1112	2	8
1125	2	1
1153	2	9
1163	2	8

Indice	Número real	Número predito
1167	2	3
1169	2	6
1189	2	8
1207	2	8
1213	2	10
1231	2	8
1241	2	6
1267	2	3
1311	2	8
1362	2	8
1363	2	4
1368	2	10
1373	2	10
1383	2	8
1399	2	8
1408	2	4
1410	2	1
1412	2	4
1413	2	4
1421	2	8
1423	2	8
1440	2	1
1453	2	6
1476	2	7
1482	2	3
1484	2	10
1488	2	10
1494	2	10
1524	3	5
1527	3	7
1550	3	7
1563	3	2
1564	3	2
1570	3	5
1588	3	6
1599	3	2
1602	3	5
1607	3	5

Indice	Número real	Número predito
1611	3	8
1617	3	9
1618	3	5
1626	3	2
1629	3	2
1630	3	5
1677	3	8
1700	3	5
1701	3	7
1707	3	1
1757	3	5
1764	3	7
1766	3	5
1776	3	5
1791	3	8
1798	3	5
1801	3	2
1876	3	2
1905	3	5
1923	3	9
1945	3	8
1955	3	5
1959	3	1
1963	3	5
1970	3	7
1972	3	8
1976	3	5
1979	3	7
1981	3	2
1999	3	5
2078	4	9
2087	4	6
2104	4	9
2108	4	2
2112	4	1
2166	4	1
2171	4	9
2187	4	9

Indice	Número real	Número predito
2195	4	2
2201	4	9
2217	4	2
2238	4	9
2263	4	2
2277	4	8
2282	4	9
2352	4	10
2361	4	6
2366	4	1
2384	4	9
2393	4	9
2467	4	6
2479	4	9
2498	4	9
2506	5	3
2520	5	2
2556	5	2
2570	5	4
2574	5	3
2580	5	4
2583	5	6
2593	5	3
2601	5	6
2606	5	3
2611	5	6
2616	5	8
2620	5	10
2637	5	4
2681	5	6
2693	5	3
2697	5	6
2698	5	6
2704	5	3
2718	5	3
2739	5	2
2752	5	3
2788	5	2

Indice	Número real	Número predito
2790	5	10
2796	5	1
2833	5	8
2850	5	3
2857	5	9
2907	5	10
2908	5	6
2920	5	10
2925	5	1
2943	5	4
2946	5	4
2952	5	4
2958	5	2
2995	5	3
3043	6	10
3078	6	8
3081	6	1
3174	6	5
3204	6	1
3236	6	5
3328	6	8
3341	6	8
3351	6	5
3382	6	10
3400	6	5
3466	6	8
3480	6	5
3515	7	9
3521	7	1
3537	7	1
3570	7	1
3595	7	2
3603	7	4
3616	7	4
3619	7	4
3627	7	4
3629	7	4
3634	7	10
3660	7	9

Indice	Número real	Número predito
3664	7	4
3668	7	9
3708	7	1
3725	7	10
3732	7	2
3795	7	2
3823	7	9
3830	7	1
3838	7	9
3853	7	9
3895	7	1
3909	7	9
3955	7	9
3972	7	9
4032	8	4
4049	8	2
4051	8	6
4057	8	1
4069	8	2
4100	8	1
4110	8	9
4125	8	6
4140	8	3
4168	8	9
4183	8	1
4197	8	9
4202	8	6
4231	8	10
4248	8	9
4251	8	5
4256	8	5
4265	8	4
4293	8	5
4294	8	2
4319	8	3
4343	8	3
4387	8	4
4393	8	6
4395	8	6

Indice	Número real	Número predito
4428	8	4
4455	8	2
4464	8	3
4477	8	3
4503	9	7
4506	9	5
4509	9	10
4519	9	7
4531	9	7
4537	9	3
4568	9	4
4576	9	7
4582	9	4
4590	9	7
4593	9	10
4612	9	7
4627	9	8
4636	9	3
4639	9	3
4656	9	7
4672	9	10
4689	9	4
4703	9	4
4708	9	7
4774	9	10
4781	9	10
4783	9	7
4790	9	2
4793	9	7
4833	9	3
4844	9	1
4860	9	1
4863	9	7
4865	9	8
4902	9	7
4917	9	4
4936	9	10
4990	9	3

Bibliografia

- [1] Víctor Manuel Gutiérrez Corzo, Darío González Galindo, Miguel Ángel Ruiz Pinto, Francisco Ronay López Estrada, and Joaquín Eduardo Domínguez Zen-
teno. Red neuronal artificial backpropagation aplicada al reconocimiento de
dígitos hexadecimales.
- [2] Cyril Goutte. Note on free lunches and cross-validation. *Neural Computation*,
9, 04 2001.