Projeto 1: Redes Neurais e Seleção de Modelos

Nicolas Toledo de Camargo RA: 242524
Vinícius Oliveira da Silva RA: 195068
Luana Aparecida Praxedes da Silva RA: 182255

Quando se fala de aprendizado de máquinas, um dos algoritmos mais famosos é o de redes neurais. Redes neurais são poderosas e flexíveis para aprender o que for necessário quando se há muitos dados, uma de suas muitas aplicações é para problemas de classificação. Neste problema temos uma entrada e queremos saber se essa entrada pertence ou não a certa classe de objetos pré estabelecida.

O problema de classificação que trataremos é o de reconhecer digitos manuscritos. Para isso, obtivemos dados de imagens MNIST, que são imagens transformadas em 20x20 pixels. Um arquivo de dados contém 4999 linhas, sendo cada linha uma imagem linearizada para um vetor de tamanho 400, o outro arquivo contém a classificação de cada imagem, sendo dígitos de 1 a 10.

Tivemos como base o algoritmo de rede neural apresentado em sala de aula pelo professor. Neste algoritmo estão definidas as funções sigmóide, gradiente da sigmóide, função custo e gradiente descendente regularizados,e inicialização aleatória dos parâmetros em torno de zero.

A inicialização dos parâmetros gera j linhas e i+1 colunas de parâmetros aleatórios em torno de zero. Sendo i o número de unidades de ativações da camada que os parâmetros vão atuar sobre e j o número de ativações da próxima camada.

No custo computacional, para cada camada são feitas suas matrizes parâmetros, é adicionada às suas ativações mais uma unidade com o valor 1, para ser o bias. Os parâmetros e ativações de cada camada são vetorizados (A_k e θ_k) e, então se fizermos $A_k \; x \; \theta_k^T$ teremos a soma do produto de cada ativação pelo seu parâmetro, e esse valor entrará na função de ativação para gerar um valor de ativação daquela camada para ser usado para a próxima camada, esse é o forward propagation.

É criada uma matriz para ser a saída, em que cada coluna representa a classificação de um dígito, marcando 1 na sua posição e 0 nas outras. Então é definida a função custo da regressão logística e sua regularização. Para o backwards propagation, são calculados os erros de cada unidade e a sua derivada, sendo ela o produto da ativação pelo erro da unidade da próxima camada que corresponde a esse parâmetro.

Na função do gradiente descendente, para K iterações, é calculado o custo e é subtraído dos parâmetros o produto de uma taxa de aprendizado pelo gradiente, com os custos e gradientes gerados pela função do definida anteriormente. Assim, ao final das iterações, temos uma lista com os custos de cada iteração e temos os parâmetros finais treinados.

Por fim, a função predição recebe um valor novo para ser classificado e utiliza o forward propagation da rede com os parâmetros treinados para gerar uma classificação.

A rede neural tratada apresenta 25 unidades em uma camada, com cada unidade realizando regressões logísticas como funções de ativação. Utilizamos as bibliotecas de python: “numpy”,”matplotlib”, “pandas”,”random” e ”scipy”.

Importados os dados, nosso primeiro teste foi treinar esta rede com 1000 iterações, taxa de aprendizado de 0.8 e hiperparâmetro de regularização igual a 1. Obtidos os parâmetros deste backwards propagation, fizemos uma previsão utilizando os próprios dados iniciais. E para ver se estamos convergindo para algum valor, ou seja, se a função custo da regressão logística estava diminuindo, plotamos o gráfico do custo com o decorrer das iterações.

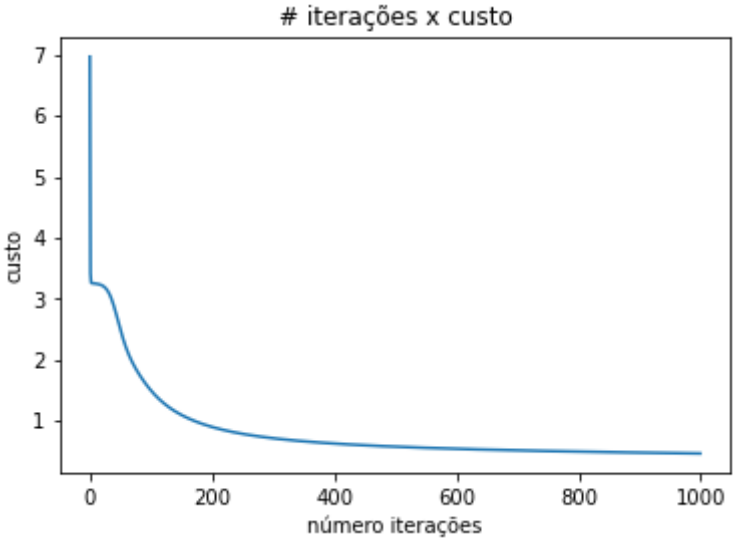

Figura 1: Custo regressão logística da rede conforme iterações.

Note como o custo decai quando mais iterações são realizadas, então a taxa de aprendizado está adequada para nosso teste inicial.

A previsão foi comparada a classificação real dos dados, obtendo uma taxa de acerto de 95.4%, além disso, implementamos um algoritmo para detectar e tabelar quais os dígitos classificados incorretamente e quais seriam os correspondentes certos.

	Errado	Certo
Erro 1	3	10
Erro 2	8	10
Erro 3	8	10
Erro 4	9	10
Erro 5	6	10
...	...	...
Erro 225	8	9
Erro 226	7	9
Erro 227	4	9
Erro 228	10	9
Erro 229	3	9

[229 rows x 2 columns]

Figura 2: Dígitos classificados erroneamente do teste inicial.

Em seguida, é necessário fazer uma comparação de uma aproximação do gradiente da função custo com o gradiente calculado pelo código para ver se está implementado corretamente, como o custo computacional será alto, usaremos uma rede menor. Para isso, sorteamos aleatoriamente entre os exemplos, três deles com suas classificações, e neste exemplo também sorteamos aleatoriamente três atributos para ser a entrada da nossa rede.

Usando 5 unidades na camada escondida, achamos o gradiente pelo algoritmo já definido. Após isso, definimos uma função para calcular a derivada da função custo em relação cada parâmetro aproximada pela secante por um fator $\varepsilon = 10^{-4}$, então:

$\frac{\partial J(\theta)}{\partial \theta_i} = \frac{J(\ldots,\theta_{i-1},\theta_i+\varepsilon,\theta,\theta_{i+1},\ldots.)-J(\ldots,\theta_{i-1},\theta_i-\varepsilon,\theta,\theta_{i+1},\ldots.)}{2\varepsilon}$ é a aproximação da derivada parcial da função custo em relação ao parâmetro i.

Com essas derivadas calculadas, comparamos os resíduos dos gradientes pelo algoritmo e pela aproximação em cada camada. Na primeira camada, esse erro foi de $1,3 * 10^{-13}$ e na segunda de $1,5 * 10^{-7}$, comparados a ε, achamos que estes erros são razoáveis, então concluímos que o cálculo do gradiente pelo algoritmo está correto. Essa parte não é implementada na rotina da rede neural por seu custo computacional alto, ela foi feita apenas para esta checagem.

Adaptamos a função custo e método de calcular o gradiente para utilizarmos na implementação com uso do gradiente conjugado através da função scipy.optimize.minimize do python, mas ao não importava quantas iterações colocamos na função, realizava apenas uma e dava erro de precisão: "Desired error not necessarily achieved due to precision loss". Procuramos solução na internet [2] e tentamos a normalização dos dados, mudar a escala na função e a tolerância de resolução, mas não adiantou. Porém ao alterar o hiperparâmetro de regularizacao $\lambda = [0,\ 0.001,\ 0.003,\ 0.01,\ 0.03,\ 0.1,\ 0.3,\ 1,\ 3,\ 10]$, obtivemos resultados diferentes do valor da função custo para esses parâmetros.

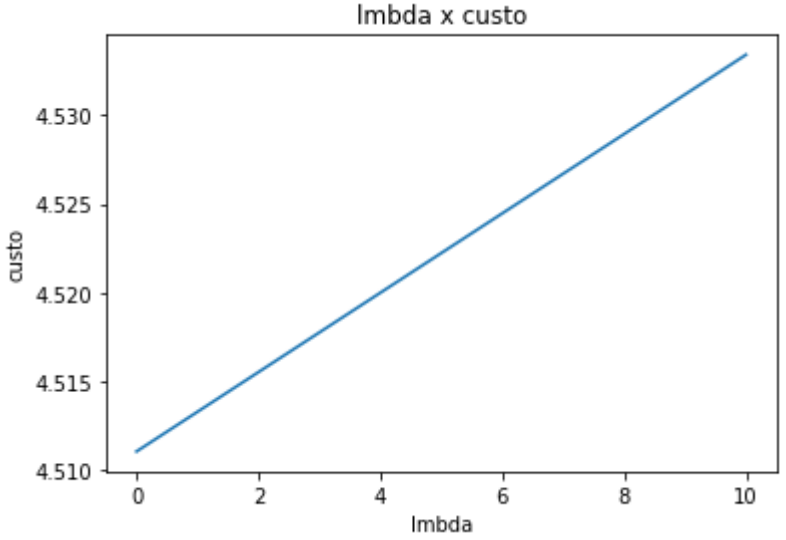

Figura 3: custo ao variar o hiperparâmetro na solução pelo gradiente conjugado.

Para o método do gradiente conjugado, parece que não há muito overfitting, então com valores de λ menor, o valor da função custo é menor. Mas de novo, como estávamos com dificuldade em realizar mais iterações, esse resultado pode não refletir na realidade se fossem realizadas mais iterações.

Note que os pesos da primeira camada obtidos estão guardados em uma matriz $25\ x\ 401$, cada linha representa os parâmetros que atuam em cada unidade de camada escondida com relação ao atributo da camada anterior representado pela coluna. Se descartar a primeira coluna, que é o bias, pode-se rearranjar essa linha para uma imagem $20\ x\ 20$, que representa os parâmetros dos atributos de entrada em relação àquela unidade

escondida, e com isso temos uma representação visual de como cada unidade escondida afeta o treinamento da rede.

Então, separamos essas linhas da matriz dos parâmetros da primeira camada, transformamos nas imagens 20 x 20, normalizamos os valores e fizemos gráficos por curvas de nível para cada imagem para tentar visualizar essa contribuição de cada unidade.

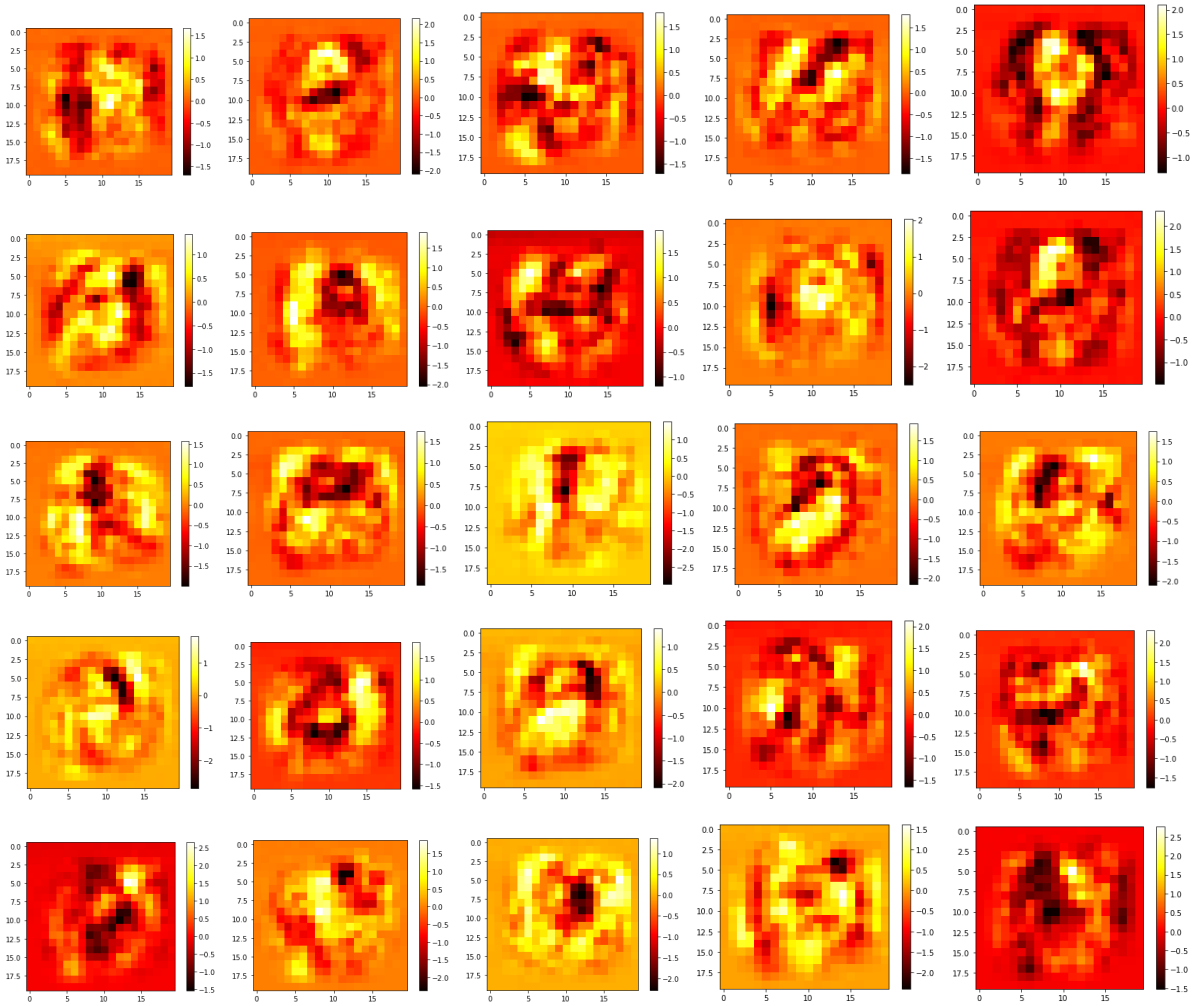

Figura 4: Representação visual da contribuição de cada unidade escondida.

Com o algoritmo geral funcionando, partimos para o próximo passo, que é a seleção de modelo. A primeira etapa da seleção é definir uma função para fazer o holdout dos dados, ou seja separá-los em conjuntos de treino, validação e teste.

Para essa função, fornecemos o conjunto de dados e as porcentagens que queremos separar para treino e validação. São escolhidos aleatoriamente exemplos para o treino, junto com suas classificações; depois é calculada a proporção de cada classificação de dígito para separar os conjuntos com a mesma proporção de cada label.

Definido o conjunto de treino, reduzimos o conjunto de dados, e do resto, o mesmo procedimento é feito para o conjunto de validação, tomando em conta preservar a proporção adquirida para o treino. Por fim, os dados que sobram são para o teste.

Com o holdout definido, definimos uma função para treinar o conjunto de treino com diversos hiperparâmetros de regularização e com cada treino, calcular o erro no conjunto de

validação. Plotamos o gráfico do erro de validação conforme os hiperparâmetros testados $\lambda = [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10]$ para acharmos qual o mais adequado, que é o com menor erro de validação. O teste foi feito com taxa de aprendizado igual a 0.8, 60% de dados para o treino, 20% para validação e 20% para teste, com 1000 iterações e com 10 holdouts para cada parâmetro iteração, então o erro de validação é somado em cada holdout e depois dividido pelo número de holdouts, ou seja, a média.

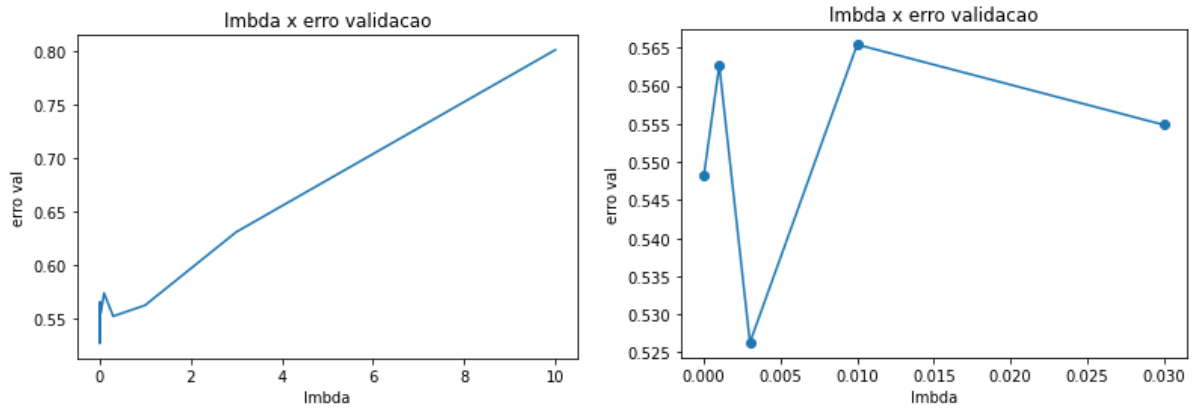

Figura 5: Erro de validação em função do hiperparâmetro. (Todos λ e zoom)

Concluímos que o hiperparâmetro com menor erro de validação é $\lambda = 0.003$, com erro de 0.53. Com esse λ ótimo, então calculamos o erro no conjunto de teste, obtendo um erro de 0.56. Como esse lambda não é grande, podemos inferir que nosso modelo não está inicialmente induzido ao overfitting, foi necessária apenas uma pequena regularização para otimizá-lo.

Nossa análise final será a curva de aprendizado da rede regularizada, que representa o erro de treino e validação conforme a proporção de dados selecionada para esses conjuntos no holdout.

Para isso, variamos a proporção de treino de (10 a 90) % variando em 5%, a validação fica com metade do resto. Neste teste, preservamos as condições do teste anterior, mas usamos agora o hiperparâmetro ótimo já encontrado.

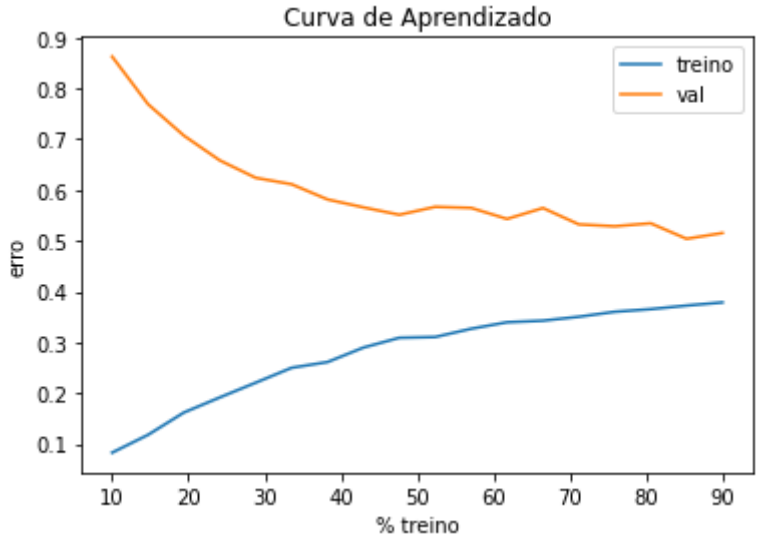

Figura 6: Curva de aprendizado.

As curvas de aprendizado apresentaram o comportamento em que quanto maior o conjunto usado para o treino, maior seria seu erro e menor o erro de validação, e vice-versa. Além disso, as curvas não ficaram nem muito próximas rapidamente nem ficaram muito distantes, nos levando a crer que o modelo não compactua com alto viés ou alta variância. Mas ainda assim indicando um grau variância, dado que conforme o número de dados aumenta, as curvas parecem estar se aproximando, ou seja, para algum número de dados suficientemente grande, terão erros praticamente equivalentes.

Concluindo, com este projeto aprendemos a implementar uma rede neural, checar sua implementação e fazer a seleção de seu modelo, procurando as condições ótimas para os testes realizados. Para nossos testes envolvendo 1000 iterações e 10 holdouts, podemos concluir o melhor hiperparâmetro de de regularização é 0.003 com erro de teste de 0.56. Não selecionamos a taxa de aprendizado, mas o comportamento do custo conforme as iterações indica que 0.8 é uma taxa funcional. Além disso, na seleção do modelo concluímos que nosso modelo está bom, sem alta variância ou viés, mas obtenção de mais dados geraria reduziria um pouco mais a variância presente, gerando um modelo melhor.

Referências:

[1] Slides e Apostila de Machine Learning - Prof. João Batista Florindo. (https://github.com/jbflorindo/MS-MT571)

[2]scipy is not optimizing and returns "Desired error not necessarily achieved due to precision loss" - (https://stackoverflow.com/questions/24767191/scipy-is-not-optimizing-and-returns-desired-error-not-necessarily-achieved-due)