Roteiro vídeo 2

ASS

Para a MLP, usamos 2 bibliotecas, o Numpy, para as operações com matrizes, e Pandas, para acessar o conteúdo dos arquivos usados para testes e treinamento.

A classe main do código, acessa o diretório e os arquivos de entrada, então usamos o Pandas para ler o arquivo e o Numpy para transformar em arrays as informações de tais arquivos. Também podemos definir os neurônios da camada de saída, tamanho da camada de entrada, número de camadas escondidas e o número de neurônios em cada uma, colocando da forma como o exemplo comentado à direita.

Construtor

O constructor da nossa MLP recebe o número de camada, o número de neurônios nas camadas escondidas, quantidade de camadas de saída e um booleano para controlar o bias.

Os dados fornecidos na criação do MLP são armazenados na estrutura lay, nome das camadas da MLP, estruturada pela junção das camadas de entrada, escondidas e saída.

Agora os pesos, neurônios e bias(se ter) são iniciados, com seus tamanhos em relação ao tamanho das camadas. Os pesos e bias são inicializados com valores aleatórios, os neurônios com valor zero e o bias, dependente se ter ou não ter, pode ser inicializado com true ou false, dependendo se ter bias.

No final do construtor é iniciado a matriz de confusão, com tamanho x por x, onde x é número de camadas de saída.

Treinamento

PESSOA 2 - eu ;)

O treinamento da mlp começa com a chamada da função ‘treinar’, ela recebe três entradas, os dados (que serão usados para treiná-la, no caso será passado os dados do arquivo ‘caracteres-limpos,csv’), a taxa de aprendizado (representamos ela por uma variável chamada alfa) e, por fim, o número de épocas (sendo isto o número de iterações que serão feitas no treinamento da mlp).

Então passamos para a função de treinamento, logo no início temos a criação de um “default_rng()” um objeto que serve para criar números aleatórios sem repetir, entre um intervalo, na linha seguinte vemos ele gerando números entre 0 e (tamanho de dados - 1), a quantidade de números varia de acordo com a quantidade dos dados, no caso, ¼ dos tamanho do conjunto de dados de treinamento e a variável replace, para indicar que não queremos números repetidos.

Então é criado o conjunto “valida” que irá armazenar a parte do conjunto de dados “raw_dados” selecionada pelo “default_rng()” para fazer parte do treinamento, em seguida é criado o conjunto “dados” que irá receber o array de dados bruto retirando a parte que pertence ao conjunto de validação, para isso usamos a função numpy.delete() que recebe o array do qual você deseja excluir, os índices que deseja excluir e uma variável *axis = 0* que sinaliza o eixo em que queremos excluir essas tuplas, no caso 0 representa que vão ser retiradas as linhas selecionadas.

Temos o array index_rest, que tem como finalidade armazenar os índices do conjunto de dados de treinamento, para criarmos ele nós usamos o np.delete() passando um array que armazena números de 0 até o tamanho do conjunto de dados bruto e apagamos dele, os índices gerados pelo default_rng().

Então somamos 1 no index_rest e no random_index para termos a linha exata de cada arquivo e imprimimos na tela para o usuário ter acesso à esses dados.

Então vamos ter um array que armazena os resultados finais da época e algumas variáveis para gerenciar a parada antecipada:

- max_erros: a quantidade máxima de erros seguidos aceita para definir a parada antecipada
- val_erro: o erro da validação na época atual
- ant_erro: o erro da validação na época anterior
- erro_medio: o erro do treinamento na época atual
- ant_erro_medio: o erro do treinamento na epoca anterior
- vai: um boolean para definir quando a condição de parada foi atingida

Vamos então começar o treinamento de fato, todo início de época é inicializado a lista de erros médios, que vai armazenar os erros de cada iteração do treinamento em uma época.

Logo em seguida iremos começar a processar os dados, separamos o conjunto de entrada e o conjunto de saída de cada linha usando a função “separa_dados()” que recebe a linha do conjunto e um boolean pra saber se queremos a resposta ou não (por *default* ele é False), essa função só calcula qual é o conjunto de entrada ou saída baseado no número de neurônios nas camadas de entrada ou saída e retorna um array contendo os dados desejados.

MAHRK

Iniciamos então o processo de *feedforwarding*, que recebe apenas o conjunto de entrada, que já é definido como a primeira camada da nossa rede neural e cria um array contendo os resultados calculados ao final da iteração de cada camada.

Iteramos então para cada camada, mas para fazer o cálculo do valor da próxima camada podemos fazer uma multiplicação de matrizes (utilizando o numpy.dot) entre uma matriz que representa os valores contidos nos neurônios da camada atual e a matriz de peso entre a camada atual e a seguinte. Depois disso somamos os bias de cada camada e passamos esses resultados para a função de ativação, que no nosso caso é a função sigmóide, depois disso definimos os valores da camada seguinte de neurônios como o resultado de todo esse processamento, e fazemos isso até passar por todas as matrizes de pesos.

Ao voltar para função “treinar” calculamos o erro fazendo uma subtração de matrizes entre a matriz de respostas esperadas e a camada de saída da rede neural, então fazemos o erro da iteração atual pegando esse erro calculado e elevando ao quadrado para filtrar eventuais números negativos e adicionamos à lista de erros médios da época.

Chegamos então ao algoritmo *backpropagation* que recebe o erro calculado pela subtração de matrizes, como o nosso objetivo é ir retrocedendo o resultado da rede devemos começar pelo final, por isso o *for* possui a cláusula *reversed*.

Toda iteração é iniciada já pegando a saída do processamento da camada atual, o equivalente aos resultados armazenados na camada posterior, então derivamos esses resultados passando pela função “derivada_sig()” e fazemos uma multiplicação entre o erro e o resultado da derivação para obtermos o delta, como nós procuramos a derivada de cada peso em relação ao erro, devemos obter ao final uma matriz com as mesmas dimensões da matriz de pesos portanto devemos reajustar as matrizes de forma a conseguirmos fazer uma multiplicação entre elas e adquirirmos uma matriz com as dimensões desejadas, para isso nós usamos a função numpy.reshape() que recebe as dimensões (linha, coluna) da matriz resultante que queremos, shape[0] vai pegar as dimensões da primeira linha da matriz, que será sempre algo do tipo “,n” já que uma linha tem uma dimensão apenas e o “-1” serve para ajustar a matriz de acordo com o outro parâmetro passado ou seja “n”. fazemos isso com o delta e transpomos a matriz depois para que a multiplicação seja possível da maneira que planejamos.

Finalmente então podemos armazenar o resultado dessa multiplicação, no conjunto de “deltas” da nossa rede neural e por fim recalculamos o erro para a próxima iteração fazendo uma multiplicação de matrizes entre o delta e a transposição da matriz de pesos entre a camada posterior e a atual.

Para calcularmos a derivada do bias em relação ao erro, basta recriarmos o mesmo delta do passo anterior, armazená-la no array de deltas dos bias e ir retrocedendo o erro da mesma forma.

Passamos então para a função atualiza_pesos que vai passar por cada peso e somá-lo à seu delta vezes o coeficiente de aprendizado, o processo se repete com o bias.

Validação

Kenzo

Nesta parte iremos realizar a validação de erros, além disso, também vamos armazenar os erros de validação da nossa mlp e encontrar os pesos em uma situação de parada antecipada.

Nós então inserimos em ‘ant_erro’ e ‘ant_erro_medio’ os valores de erro de validação e erro médio da iteração da época anterior (caso seja a primeira iteração usamos os valores definidos anteriormente, no caso 1).

Após isto, o ‘erro_medio’ recebe a média dos erros armazenados em ‘erros_medios’, este valor será escrito no arquivo de “erro.txt” juntamente da época atual.

Logo abaixo, ‘val_erro’ recebe o resultado da função valida(), esta função recebe a variável ‘valida’ criada anteriormente e a usa com o nome de ‘dados’. Primeiramente ela cria uma variável chamada ’val_erros’, essa será usado para armazenar alguns valores em um array, após isto, ela itera por todos os valores armazenados em ‘dados’.

Em cada iteração ela usa a função separa_dados() para armazenar os inputs e respostas em ‘inputs’ e ‘resp’, os inputs então são passados pela função de feedfoward(). Criamos uma variavel chamada de ‘erro’ para armazenar a subtração de resp com a o resultado da camada de saída, a qual passa a ter o resultado do feedfowarding feito, com isso temos o erro de validação dessa iteração, elevamos esse erro a segunda para evitar um número negativo, o resultado dessa potência é então armazenada no val_erros criado previamente.

Ao término das iterações temos agora em val_erros um vetor com os erros de validação de cada um dos dados de validação, com isso vamos então retornar pela função valida() a media dos valores presentes em val_erros.

Com o fim da função valida() agora iremos armazenar o resultado disso no arquivo “Val_Erro.txt” juntamente da época atual.

Caso o val_counter (inicialmente = 0) for menor ou igual ao max_erros, isso significa que não chegamos à “parada antecipada”, então os mesmos resultados armazenados em “Erro.txt” e “Val_Erro.txt” serão armazenados em "Erro_PA.txt" e "Val_Erro_PA.txt" respectivamente.

Caso o val_counter passe dos erros máximos ou for a última época, exportamos os pesos atuais usando a função exportPesos() para um arquivo chamado “PA_Pesos.txt”, este então armazena os pesos da nossa mlp com parada

antecipada, a função de exportar textos começa inicializando um writer, então ela irá escrever cada linha de cada matriz em uma linha do arquivo especificado (no caso PA_Pesos.txt), para isto iniciamos um for para cada elemento (matriz) dos pesos, dentro desse for ele irá iterar para cada linha presente nessas matrizes e, por fim, ele irá escrever cada número presente nesta linha, após escrever todos elementos desta linha ele pula uma linha.

Após o fim do primeiro for, o que itera as matrizes em pesos, inicializaremos outro ciclo, agora para as bias, primeiro se inicia um for que itera por cada elemento (vamos chamá-lo de linhas) presente em bias, dentro de cada iteração será feito um for para escrever cada número presente nessa linha e após isto será escrito uma quebra de linha. Feito tudo isto a função exportPesos() acaba e retornamos para a sua chamada.

Com isso alteramos o valor de vai para False, indicando que houve a parada antecipada, o que evita que val_counter seja incrementado ou zerado nas linhas a seguir.

Caso o erro de validação atual for maior que o erro de validação anterior, e o erro médio anterior for maior que o erro médio atual, e a época atual for maior que oito centésimos do número de épocas total, aumentamos o val_counter em 1, caso contrário, se a variável vai ainda for True resetamos o val_counter para 0.

Testes

Pisni

Após retornar do treinamento, são exportados os pesos finais para o arquivo “pesosFinais.txt” e, depois de criar um array com todos os arquivos de testes carregados em formato de matrizes, vai se iniciar o processo de testagem de cada um dos arquivos.

Primeiro, os dados são convertidos de DataFrame para array e armazenados na variável dados, então é chamada a função “projeta()” que recebe os dados, e que vai percorrer cada linha dos dados, fazer o *feedforwarding* para cada uma, usar a função “resposta()” com o boolean “caracter = True” para armazenar no array de resultados, essa função formata os resultados de uma iteração para o for mato desejado, podendo estar puro, arredondado para inteiro ou em caracteres, como queremos armazenar de um jeito que seja facilmente entendível, usamos em caracteres.

Para finalmente adicionarmos o resultado na matriz de confusão, passando os resultados adquiridos e os resultados esperados como parâmetros. A função “adicionaConfusao()” começa inicializando duas listas que correspondem às posições dos neurônios que possuem resultado arredondado igual a 1 e à posição do resultado esperado que era igual a 1.

Tendo essas posições salvas, podemos percorrer a lista de índices dos resultados adquiridos e adicionarmos 1 no ponto de coordenadas onde a linha será

o resultado esperado e a coluna será o resultado obtido, isso precisa estar em um loop já que a rede pode chutar 2 caracteres para 1 linha de testes.

Depois de percorrer todas as linhas dos dados, armazenar seus resultados e ajustar a matriz de confusão, a função retornará ao main, o array de resultados obtidos que serão exportados para um arquivo “Resultados_MLP.txt”. Para finalmente printarmos no prompt os resultados obtidos e esperados, a matriz de confusão e uma porcentagem de acerto que é calculada dividindo o número de acertos pelo número de testes vezes 100.

Então são carregados os pesos que foram armazenados ao fim da parada antecipada utilizando a função “loadPesos()” que recebe uma string com o nome do arquivo que será carregado, essa função irá criar uma matriz de pesos temporária, ler cada linha do arquivo e fazer dela uma linha de cada matriz peso tendo como base as dimensões das matrizes de peso atual, adiciona a matriz à uma lista de matrizes peso que no final substitui a lista com as matrizes de peso atual da MLP, o processo é análogo para o bias que acontece logo abaixo.

Voltando ao main, a matriz de confusão é reiniciada, todos os seus valores se tornam 0 de novo, e os testes são rodados com a MLP como se tivesse feito a parada antecipada, como o processo abaixo é o mesmo da MLP normal, não é necessário repetir.