INTELIGÊNCIA ARTIFICIAL APLICADA - MACHINE LEARNING

AULA 6

Prof. Antonio Willian Sousa

CONVERSA INICIAL

Olá, seja bem-vindo! Nesta aula, apresentaremos os fundamentos das redes neurais, exemplos de arquiteturas de múltiplas camadas, além das aplicações de redes neurais e o seu potencial de uso.

As redes neurais artificiais (*artificial neural networks,* em inglês – ANN) são algoritmos computacionais inspirados no funcionamento do cérebro humano, sem, no entanto, serem uma simulação dele. Elas tentam, de forma semelhante ao seu equivalente biológico, ativar ou desativar pequenos elementos que se interconectam em uma rede para, dessa forma, obter uma decisão ou valoração para um conjunto de dados de entrada. Para iniciarmos nosso estudo das redes neurais, precisamos entender como os perceptrons, que são pequenos elementos que se interconectam para formar uma rede maior e mais complexa, conseguem tomar decisões e executar operações.

TEMA 1 – O PERCEPTRON

O perceptron é um algoritmo que pode receber múltiplos valores de entrada e fornece como saída os valores 0 ou 1. Ele pode ser considerado a unidade básica formadora das redes neurais. Os valores obtidos pelo perceptron funcionam de maneira semelhante ao que aprendemos com o classificador de regressão logística, como uma forma de selecionar entre duas classes de um problema.

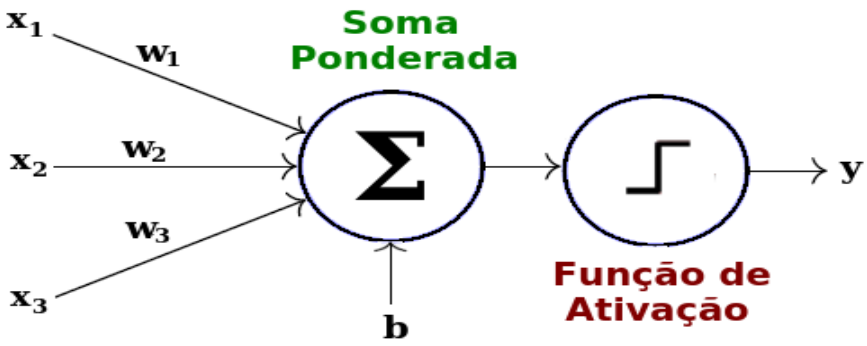
1.1 Funcionamento do perceptron

O perceptron é um algoritmo de classificação binária simples e possui quatro partes importantes:

- valores de entrada ou camada de entrada (x_1, x_2, x_3);
- pesos (w_1, w_2, w_3) e *bias* (b);
- soma ponderada;
- função de ativação.

Para compreender o funcionamento do perceptron, é utilizado uma representação esquemática semelhante à mostrada na Figura 1.

Figura 1 – Representação esquemática do perceptron



Da mesma forma que nos outros algoritmos de classificação que vimos, os valores de entrada são os valores das características e o valor de saída (y) é a nossa classe. Após receber os valores de entrada, cada valor é multiplicado pelo seu peso correspondente, esses resultados são somados e adicionados ao valor do *bias* (b). De maneira que:

$$k = x_1.w_1 + x_2.w_2 + x_3.w_3 \quad \text{(valores de entrada)}$$

$$S = k + b \quad \text{(soma ponderada)}$$

$$f(x) = \begin{matrix} 0\ se\ \ x < 0 \\ 1\ se\ \ x \geq 1 \end{matrix} \quad \text{(função de ativação)}$$

$$y = f(S) \quad \text{(resultado final)}$$

O perceptron pode receber como entrada, não apenas três valores, mas sim quantos forem necessários. Mesmo que a quantidade de valores de entrada mude, todo o processo restante é igual, de maneira que o cálculo da soma ponderada (k) é executado de forma a somar os n produtos dos elementos do vetor de características e os seus respectivos pesos por meio da seguinte fórmula:

$$k = \sum_{i=1}^{n} x_i.w_i$$

O processo de aprendizagem do perceptron ocorre nas seguintes etapas:

1. recebe os valores de entrada e multiplica pelos pesos. Os pesos determinam a importância de cada uma das características fornecidas para o resultado final. Descobrir os valores ideais dos pesos é a principal tarefa executada no processo de aprendizagem das redes neurais;
2. adiciona o valor do *bias*, que permite que a curva de separação obtida pelo perceptron seja deslocada da sua posição original, permitindo um ajuste no modelo obtido;
3. executa a soma ponderada e fornece para a função de ativação para que os valores obtidos com os dados de entrada e do *bias* sejam mapeados em um intervalo de valores como 0 e 1. A característica não linear da função de ativação é o que permite o treinamento de redes neurais complexas;
4. fornece o resultado, que indica a decisão de classificação obtida com base no perceptron;
5. compara o resultado obtido com o valor real, executa um ajuste nos pesos, considerando a diferença entre o valor real e o resultado;
6. inicia uma nova iteração.

Da mesma forma que os demais classificadores, o perceptron pode ser usado para determinar, por exemplo, se um carro se enquadra nas categorias *não popular* e *popular*, como fizemos utilizando um modelo de regressão logística e o kNN. O código a seguir mostra a utilização do perceptron para este problema de classificação.

```
import pandas as pd
from sklearn.linear_model import Perceptron
from sklearn.preprocessing import StandardScaler

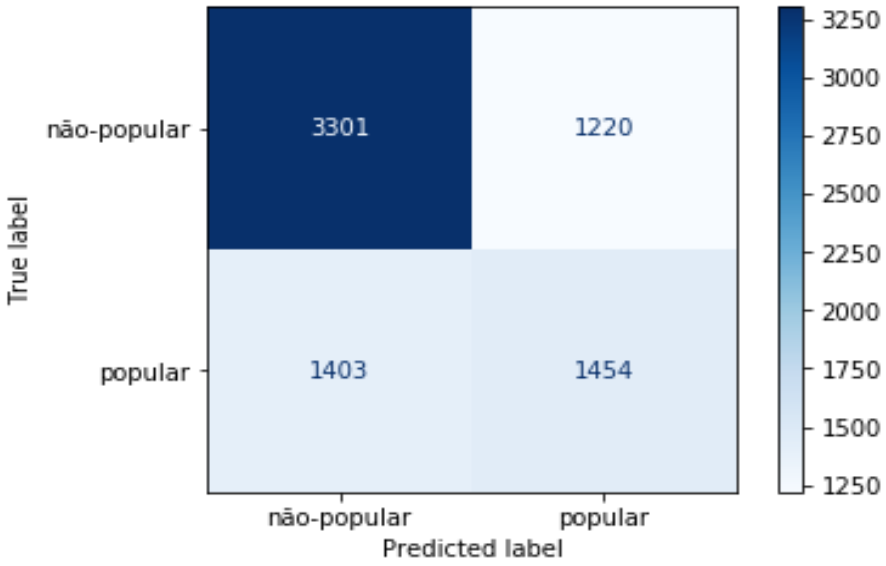
carros = pd.read_csv("carros_modelos_categorical.csv")
X = carros.iloc[:, :-1].to_numpy()
y = carros.iloc[:, -1].to_numpy()

sc = StandardScaler()
sc.fit(X)
```

```
X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
                                    test_size=0.20)
X_treino = sc.transform(X_treino)
X_teste = sc.transform(X_teste)
percpetron_clf = Perceptron(alpha=0.0001, eta0=0.15,
                fit_intercept=True,max_iter=123, shuffle=True,
                warm_start=True, class_weight={1: 0.3})
percpetron_clf.fit(X_treino, y_treino)
y_predicao = percpetron_clf.predict(X_teste)
```

Os resultados obtidos pelo perceptron na classificação das duas classes podem ser vistos na Figura 2 por meio da matriz de confusão.

Figura 2 – Matriz de confusão do perceptron com acurácia de 62%

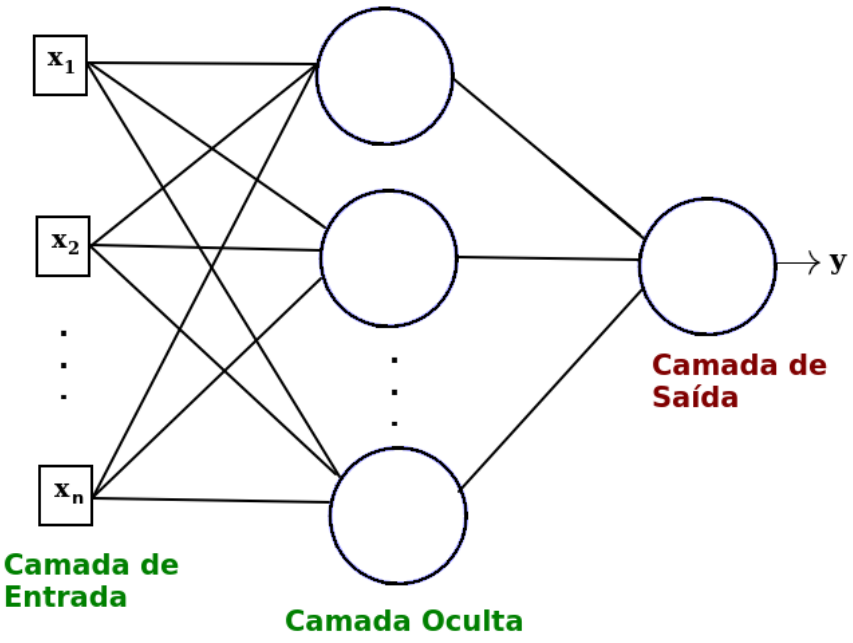


O resultado obtido, comparado com os classificadores que já aprendemos, não é algo de se impressionar. No caso da classificação da popularidade de automóveis, ainda foi necessário definir pesos para as classes como forma de melhorar o desempenho. Isso pode dar a impressão de que o uso de outros classificadores são uma melhor escolha que o perceptron. No entanto, a grande capacidade do perceptron de encontrar as fronteiras de separação entre classes não está no seu uso isolado, mas sim quando várias unidades do algoritmo são combinadas formando uma rede de múltiplas camadas. Essa atuação conjunta pode apresentar resultados surpreendentes quando comparada com o desempenho de uma única unidade e com outros classificadores.

1.2 Perceptron de múltiplas camadas

Um *multilayer perceptron* (MLP) ou perceptron de múltiplas camadas é uma estrutura que agrupa diversos neurônios em camadas para a resolução de problemas de classificação ou de regressão. O perceptron é utilizado como uma unidade básica e cada um deles funciona da mesma forma que o perceptron padrão, em que o conjunto de neurônios em cada nível é chamado de camada. Assim, temos uma camada de entrada, que é a primeira a interagir com os dados de entradas, que deverão ser os valores do nosso vetor de características. Os valores resultantes da camada de entrada são repassados para as próximas camadas – camadas ocultas –, que podem existir em diferentes quantidades e farão a propagação dos dados até chegar à camada de saída, que é responsável pelo resultado final. Na Figura 3 temos a representação esquemática de um MLP.

Figura 3 – Representação esquemática do MLP



Para cada neurônio do MLP mostrado na Figura 3 são determinados diferentes pesos e executados os procedimentos de soma ponderada e ativação, que irão definir se aquele neurônio, naquela camada, irá ter influência sobre o resultado. Um neurônio com um valor constante igual a 1, denominado *neurônio de bias*, é repassado para as camadas e tem a mesma função do

valor de *bias* do perceptron padrão, permitindo um ajuste da função obtida por meio de um deslocamento da curva determinada por ela.

O MLP em sua camada de entrada pode receber quantos valores de características forem necessários e estes valores devem ser repassados para cada um dos perceptrons da sua camada oculta, e o resultado desta, por sua vez, são repassados para a camada de saída. A quantidade de perceptrons (nós) a serem utilizados na camada oculta não está relacionada com o tamanho do vetor de características e deve ser escolhida no momento da criação do modelo, da mesma forma que a escolha do tamanho da vizinhança (k) do kNN.

Por conta dessa característica estrutural de os valores se propagarem para os perceptrons que compõem às camadas à frente, o MLP também é referenciado como *Feed Forward Neural Network* pelo fato. Uma diferença considerável entre o MLP e o perceptron é o fato de que este gera resultados binários, por conta de sua função de ativação degrau, já o MLP pode utilizar diferentes funções de ativação. A utilização de diferentes funções de ativação permite que os valores de saída sejam valores reais comprimidos em intervalos como 0 e 1 ou -1 e 1, gerando como resultados valores de probabilidades ou a classificação em múltiplas classes.

O objetivo principal de um MLP é obter uma aproximação para uma função *f* tal que y = f(x), em que x são os dados de entrada e y é a variável de saída ou a classe. Para isso, a função obtida é definida como y = f(x; θ), restrita a aprender o melhor conjunto de parâmetros θ na obtenção dessa função, de maneira que a cada camada de um perceptron de múltiplas camadas, ocorre uma composição de funções. Assim, se temos um MLP com três camadas, a nossa função final obtida pode ser entendida da seguinte forma:

$$f(x) = f_3\left(f_2\left(f_1(x)\right)\right)$$

Em que:

- f_1 é a função definida pela primeira camada;
- f_2 é a função definida pela segunda camada;
- f_3 é a função definida pela terceira camada;
- x é o vetor de características de entrada.

Para cada unidade de cada camada são realizados os cálculos com distintos valores de pesos e são executadas a funções de ativação. O conjunto de pesos a ser descoberto no caso do MLP vai depender do número de camadas, da quantidade de neurônios (perceptrons) utilizados e do tamanho do vetor de características.

A utilização de uma quantidade de neurônios muito superior ao tamanho do vetor de características é algo bem comum no processo de treinamento e pode gerar bons resultados. A escolha do tamanho das camadas deve ser feita por meio da avaliação dos resultados obtidos e de testes. O código a seguir executa um classificador MLP sobre o mesmo conjunto de dados classificado anteriormente utilizando o perceptron.

```
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler

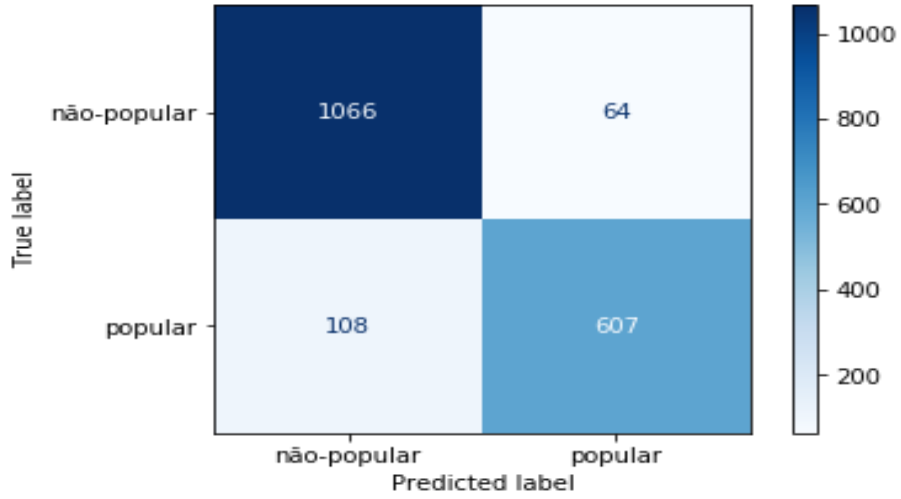

sc = StandardScaler()
sc.fit(X)
mlp_clf = MLPClassifier(warm_start=True, max_iter=500,
            hidden_layer_sizes=(100,))


X_treino_std = sc.transform(X_treino)
X_teste_std = sc.transform(X_teste)


mlp_clf.fit(X_treino_std, y_treino)
y_pred = mlp_clf.predict(X_teste_std)
```

Obtendo o resultado mostrado pela matriz de confusão na Figura 4.

Figura 4 – Matriz de confusão do MLP para uma acurácia de 91%



O resultado obtido é muito superior àquele obtido bom o perceptron, pois o modelo utilizado consegue aprender fronteiras de separação não lineares dos dados. Nesse exemplo, utilizou-se uma camada oculta com 100 perceptrons para determinar a fronteira de separação entre as classes. Essa determinação da quantidade de neurônios utilizada em cada e a quantidade de camada são hiperparâmetros do modelo, e a sua determinação é feita por meio de testes e avaliação de resultados durante o processo de treinamento.

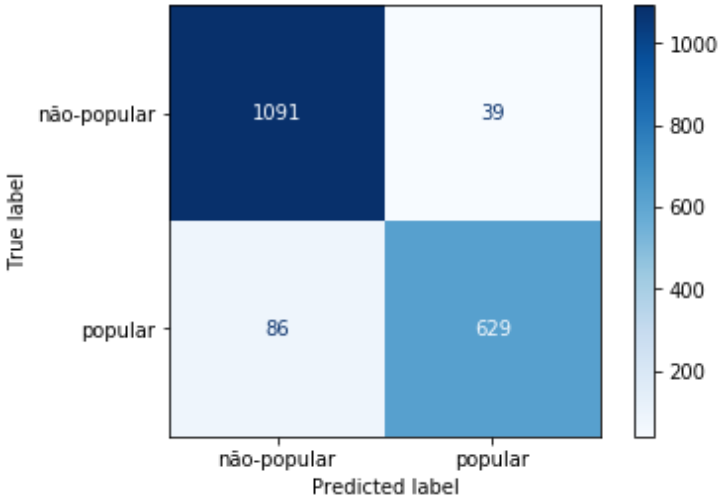
Durante o processo de treinamento de modelos de redes neurais, há diversos hiperparâmetros que podem melhorar o desempenho obtido. Assim, há duas etapas importantes a serem consideradas: a definição da organização e estrutura da rede, por meio dos hiperparâmetros e da otimização do modelo, por meio da análise do desempenho, ajustes e seleção de atributos, bem como outras formas de melhorar as capacidades do modelo.

Para termos uma ideia de como mudanças na estrutura da rede podem provocar mudanças significativas, consideremos a rede que utilizamos anteriormente para classificar as instâncias do nosso *dataset* de carros em *popular* e *não popular*. A rede utilizada possuía apenas uma única camada oculta contendo 100 neurônios. Ao modificarmos a estrutura da rede para que esta passe a ter duas camadas ocultas, contendo a mesma quantidade de neurônios no total, conforme o código a seguir:

```
mlp_clf = MLPClassifier(warm_start=True, max_iter=500,
                        hidden_layer_sizes=(50,50),
                        learning_rate = 'adaptive')
```

Obtemos o resultado mostrado na Figura 5:

Figura 5 – Matriz de confusão do MLP com 93% de acurácia



Perceba que alteramos apenas a estrutura da rede neural, mudando a quantidade de camadas ocultas e a forma de tratar a taxa de aprendizagem da rede, o que permitiu um ganho de acurácia da ordem de 2%, o que é uma melhoria considerável.

Para as situações em que se deseja ganhar tempo do processo de treinamento, pode-se utilizar arquiteturas que já foram testadas em problemas semelhantes e para as quais já se tenha um conjunto de pesos que podem ser utilizados na inicialização. Esse processo permite ganhar tempo e mesmo que a arquitetura utilizada não tenha sido criada para o seu problema, pela definição das redes neurais, a convergência é assegurada.

TEMA 2 – REDES NEURAIS ARTIFICIAIS

Nesta seção apresentaremos algumas definições do funcionamento das redes neurais, bem como a forma como elas podem ser construídas. Além disso, também falaremos sobre o processo de aprendizagem e como otimizar um modelo já definido e que tenha passado pelas etapas iniciais de treinamento.

2.1 Definição do modelo

Para ficar mais clara a separação entre as etapas de preparação do modelo e o seu aprendizado, devemos distinguir os que são hiperparâmetros e os parâmetros do modelo. Os parâmetros do modelo são internos à rede neural definida, como os pesos dos neurônios em cada uma das camadas. Já os hiperparâmetros, são externos ao modelo e definem a maneira como este será estruturado, como a quantidade de neurônios por camada ou a quantidade de camada ocultas.

Dentre os hiperparâmetros que definem a estrutura de uma rede neural, devemos nos atentar para os seguintes:

- **número de camadas ocultas:** adicionando mais camadas ocultas geralmente melhora o desempenho;
- **função de ativação a ser utilizada:** a escolha da função de ativação afeta todos os neurônios da rede e tem impacto na capacidade da rede convergir e no seu tempo de treinamento;
- **inicialização dos pesos:** sabemos que a rede neural essencialmente buscará definir os valores dos pesos, mas no início do processo de treinamento é preciso fornecer valores iniciais para os pesos e essa inicialização pode ter grande impacto no processo de treinamento e convergência;
- **taxa de *dropout*:** essa taxa se refere a uma técnica utilizada durante o treinamento das redes neurais, na qual é definida uma quantidade de neurônios da rede que serão escolhidos aleatoriamente e desativados a cada iteração do processo de aprendizagem.

Além dos hiperparâmetros relacionados à estrutura da rede, temos aqueles que estão relacionados ao treinamento do modelo. Dentre estes hiperparâmetros, alguns do mais importantes são:

- **número de *epochs*:** o termo *epoch* se refere à passagem de todo o conjunto de dados pelo modelo durante o processo de treinamento;
- **tamanho do *batch*:** quando o conjunto de dados de treino é muito grande para ser apresentado ao modelo em uma única *epoch*, ele deve ser dividido em partes menores (*batches*), que serão fornecidas ao modelo;

- **algoritmo de otimização:** durante o processo de aprendizado da rede neural é utilizado um algoritmo de otimização responsável pela determinação dos valores ideais dos pesos;
- **taxa de aprendizagem:** a rede neural recebe os valores iniciais dos pesos a serem utilizados, executa os cálculos e encaminha os valores para as camadas subsequentes, porém estes pesos necessitam ser ajustados em todas as camadas. A taxa de aprendizagem controla a forma como esse ajuste é determinado.

Dentre as operações que podem ser feitas para evitar *overfitting* no modelo de rede neural, estão:

- retreinar a rede com diferentes pesos na inicialização dos neurônios;
- treinar diversos modelos ao mesmo tempo, com diferentes pesos e estruturas para cada modelo;
- monitorar o valor do erro obtido durante os treinamentos;
- aplicar alguma técnica de regularização.

Alterações na estrutura também podem ser aplicadas quando se necessita evitar *underfitting* dos modelos. Dentre as alterações, podemos destacar:

- aumento do número de neurônios;
- aumento da quantidade de parâmetros de entrada;
- acrescentar instâncias de treino ou melhorar a qualidade destas;
- executar um procedimento de *droupout*.

Normalmente, os *frameworks* e bibliotecas que implementam os algoritmos de redes neurais já possuem métodos que permitem executar alterações na estrutura e nos processos de treinamento. O valor dos neurônios de *bias*, por exemplo, não precisa de definição, pois a maioria das bibliotecas executa o seu treinamento e busca do melhor valor.

Caso aconteça, durante o treinamento da rede neural, de você perceber que o modelo está sofrendo de *underfitting* ou *overfitting*, é possível executar procedimentos de ajuste. O importante em relação a estes métodos e essas alterações estruturais é a compreensão de como elas podem afetar o modelo e o que deve ser aplicado para cada situação. Além, é claro, de buscar testar

diferentes cenários e valores para agregar experiência ao conhecimento teórico de como executar um treinamento eficiente.

2.2 Treinamento do modelo

O treinamento de um modelo de rede neural, após definição da sua estrutura, é feito nas seguintes etapas:

1. inicialização dos pesos de todos os neurônios;
2. propagação para frente (*forward*);
3. cálculo do erro;
4. propagação reversa (*backpropagation*);
5. atualização dos pesos;
6. realiza novas iterações até obter convergência.

Durante a propagação para frente, cada um dos neurônios recebe os valores iniciais dos pesos e realiza a adição do valor do neurônio de *bias* e propaga os valores obtidos para as próximas camadas. Após passar por todas as camadas, obtém-se o valor final de predição. O valor de predição obtido é então comparado com os valores reais das classes de cada instância por meio de uma função de cálculo do custo de obter a predição correta (função de *loss*). O valor obtido pela função de *loss* será propagado de forma reversa pela rede, realizando ajustes. Desta forma, os pesos da rede são ajustados e uma nova *epoch* pode ser realizada, para uma nova iteração do processo de aprendizagem.

Como o processo de aprendizagem é supervisionado, cada vetor de características de cada uma das instâncias do nosso conjunto de treinamento é associado com o valor de uma classe, cujo valor é conhecido. Assim, a função de *loss* baseado nesse valor de referência (*ground truth*) determinará o desempenho da rede neural, restringindo o problema de treinar a encontrar o conjunto de pesos da rede que minimiza a função de *loss*, ou seja, reduzindo o erro da predição ao mínimo.

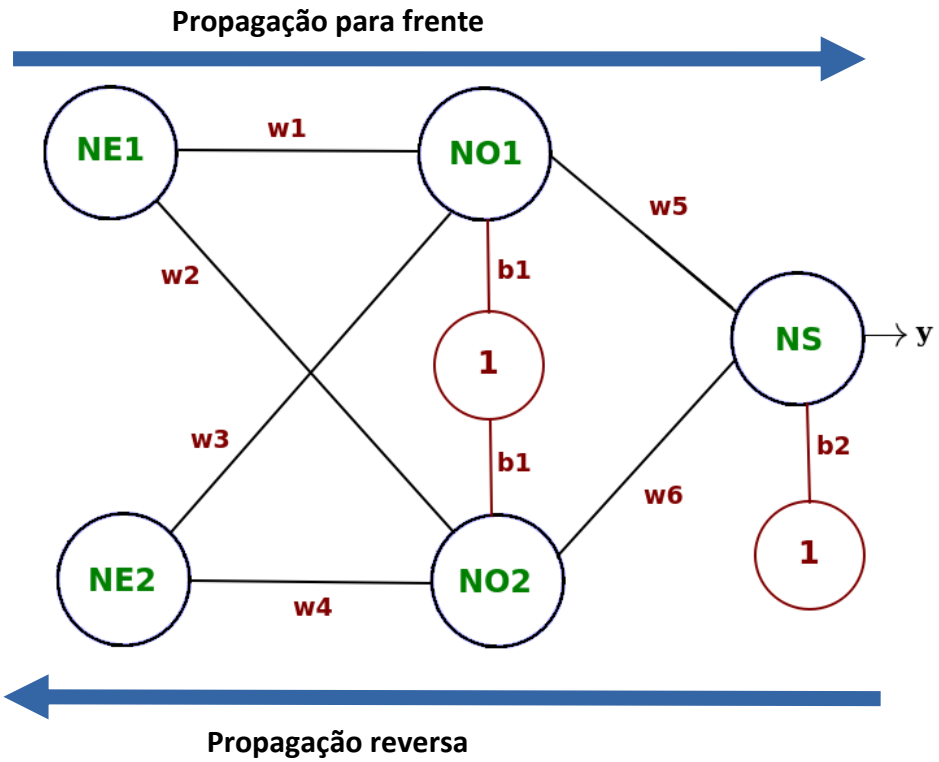
A busca pelo valor mínimo da função de *loss* é feita de modo iterativo, pelo algoritmo de otimização, utilizando-se de uma direção de movimentação na busca dos melhores valores dos pesos. Essa direção de movimentação é o gradiente, e a taxa de aprendizagem define o quão grande é o tamanho desse deslocamento. De maneira simplista, podemos dizer que o gradiente determina

um deslocamento e uma direção, enquanto a taxa de aprendizagem determina o quanto se deve incrementar esse deslocamento à medida que novos resultados são obtidos.

A quantidade de pesos a serem treinados varia de acordo com quantidade de neurônios e devemos considerar que, para cada camada oculta e, também, para a camada de saída, teremos um peso de *bias* a ser calculado. Na Figura 6 é possível ver a representação esquemática de um MLP com uma camada de entrada e uma camada oculta, em que temos os seguintes neurônios e pesos:

- neurônios da camada de entrada: **NE1, NE2**;
- neurônios da camada oculta: **NO1, NO2**;
- neurônio de saída: **NS**;
- pesos dos neurônios: **w1, w2, w3, w4, w5, w6**;
- pesos dos *bias*: **b1, b2**.

Figura 6 – Propagação para frente e reversa no MLP



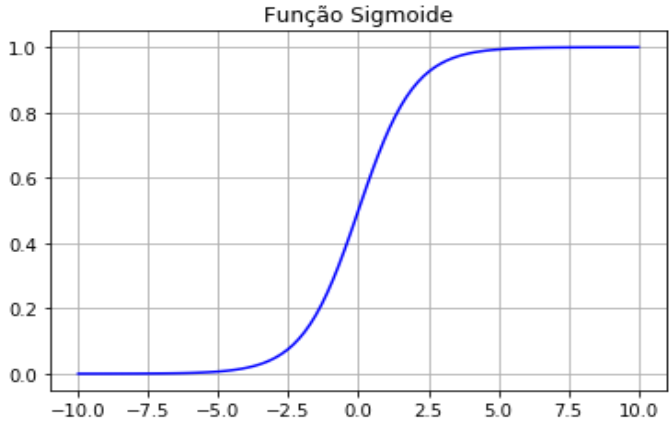
Como já citamos anteriormente, cada neurônio possui uma função de ativação. Esta função determinará se o neurônio terá influência ou não no

resultado final da saída da rede neural. Uma função comumente utilizada como função de ativação é a função sigmoide, definida da seguinte forma:

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

Sua utilização como função de ativação está relacionada com o fato dela retornar para qualquer valor real, uma saída entre 0 e 1, como mostrado no seu gráfico apresentado na Figura 7.

Figura 7 – Gráfico da função sigmoide



Ao iniciar o treinamento de uma rede neural, os pesos são inicializados com valores que dependem do método de inicialização utilizado e assim ocorre a propagação para frente, nas seguintes etapas:

1. recebe os valores de entrada e multiplica pelos pesos iniciais:

   (**ne1** * **w1**) + (**ne2** * **w2**) e (**ne1** * **w3**) + (**ne2** * **w4**);

2. adiciona-se o valor do *bias*:

   x_{NE1} = (**ne1** * **w1**) + (**ne2** * **w2**) + **b1**
   x_{NE2} = (**ne1** * **w3**) + (**ne2** * **w4**) + **b1**;

3. aplicação da função de ativação sobre os valores obtidos:

   NO1 = 1 / (1 + exp(-1 * $\mathbf{x_{NE1}}$) e **NO2** = 1 / (1 + exp(-1 * $\mathbf{x_{NE2}}$);

4. o resultado dos neurônios ocultos NO1 e NO2 é multiplicado por novos pesos:

   NO1 * **w5** + **NO2** * **w6**;

5. adiciona-se o valor do *bias*:

   $_{NO1,2}$ = **NO1** * **w5** + **NO2** * **w6** + **b2**;

6. aplicação da função de ativação sobre os valores obtidos:

   NS = 1 / (1 + exp(-1 * $x_{NO1,2}$);

7. gera-se um valor de saída ŷ que será comparado com o valor real (y) para o caso da aprendizagem supervisionada;
8. terminada a propagação para frente, calcula-se o erro, ou seja, o quão distante está a predição do valor correto;
9. a partir do valor do erro, inicia-se a propagação reversa (*backpropagation*) e os ajustes dos pesos são calculados. Após os ajustes dos pesos, repete-se todo o ciclo até que se atinja algum critério de parada baseado em alguma métrica, por exemplo, a acurácia.

   Os pesos podem ser ajustados de diferentes formas, como:

- atualizar a cada instância no conjunto de treinamento: realiza-se uma propagação para a frente para cada instância, calculando os pesos e atualizando;
- atualização em *batches*: divide-se o conjunto de treino em *batches* e os pesos são atualizados após a propagação para a frente de todas as instâncias do *batch*;
- mini*batches* aleatórios: escolhe *batches* pequenos e aleatórios, atualizando os pesos a cada iteração, porém os tamanhos dos *batches* podem mudar e isso ajuda a eliminar tendências no processo de seleção das instâncias.

Ao executar o treinamento de um MPL, por exemplo, podemos acompanhar o processo por meio do valor da função de *loss* (custo), que, quanto mais próximo de zero, indica que a predição do nosso modelo está mais próxima do valor real no conjunto de dados de treino. Assim, o treinamento exibirá a saída conforme mostrado a seguir:

Iteration 1, *loss* = 0.66513861
Iteration 2, *loss* = 0.60650190
Iteration 3, *loss* = 0.58015350
Iteration 4, *loss* = 0.55984362
Iteration 5, *loss* = 0.54107713

...

Iteration 297, *loss* = 0.12573948

Iteration 298, *loss* = 0.12244368

Iteration 299, *loss* = 0.12327690

Podemos perceber que o valor de *loss* diminui a cada iteração. Observar o comportamento desse valor é importante para o processo de treinamento, bem como avaliar nos dados de validação se é necessário retreinar o modelo para obter um melhor ajuste.

2.3 Ajustes dos hiperparâmetros

Os ajustes dos hiperparâmetros dos modelos podem ter grande impacto na eficiência destes, mas a gama de valores e as possíveis combinações entre os valores destes parâmetros é tão grande que encontrar valores ótimos é uma tarefa de difícil execução.

A forma mais simples de obter hiperparâmetros adequados ao nosso modelo é por meio de tentativa e erro, fornecendo diferentes valores e combinações. No entanto, esse tipo de busca pode ser confuso e impraticável se a quantidade de combinações for muito grande. Ainda assim, a busca manual por valores, quando feita por alguém com experiência, pode ser efetiva. No entanto, não há como saber se os valores utilizados são os valores ótimos.

Outra maneira de buscar valores dos hiperparâmetros, de uma forma mais sofisticada que a busca manual, é o método chamado de *grid search.* Por meio desse método é feita uma busca sistemática dos valores de cada um dos hiperparâmetros do modelo e retreinando o modelo, selecionando os melhores valores um a um. Uma das vantagens dessa abordagem é reduzir o espaço de possibilidades ao reduzir o escopo de forma parcial ao abordar um parâmetro por vez.

A quantidade de parâmetros e o alto custo de treinamento podem representar barreiras para o método de *grid search*, tornando o processo todo muito lento para ser utilizado. Em situações como essa, pode ser mais vantajoso a utilização de uma escolha aleatória de valores, que apesar de parecer contraintuitivo, demonstrou na prática obter melhores resultados que as outras abordagens, pois permite uma cobertura maior do espaço de possibilidades ao invés de se manter confinada em regiões promissoras, que

podem não ser ótimas e, também, algumas vezes manter a busca em regiões que apenas parecem promissoras e conduzem a resultados ruins.

A maioria dos pacotes e bibliotecas que trabalham com aprendizagem de máquina oferecem alguma ferramenta para obter modelos por meio de busca aleatória e de busca em grid. O código a seguir mostra como realizar uma busca aleatória de um modelo de MLP com base em um conjunto de faixas de diferentes hiperparâmetros definidos, utilizando o objeto RandomSearchCV da biblioteca Scikit-Learn:

```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
import scipy.stats as stats
from sklearn.utils.fixes import loguniform
from sklearn.model_selection import RandomizedSearchCV

# Preparação dos dados
X_treino, y_treino, X_teste, y_teste = load_dataset()

# Declaração de um MLP
mlp_clf = MLPClassifier(warm_start=True, max_iter=100)

# Faixas de busca dos hiperparâmetros
opcoes_parametros = {
   'hidden_layer_sizes': [(30,30,30),(20,80),(25,50,5)],
   'activation': ['tanh', 'relu', 'logistic'],
   'solver': ['sgd', 'adam', 'lbfgs'],
   'alpha': stats.uniform(0.0001, 0.9),
   'learning_rate': ['constant','adaptive']}

# Busca aleatória
n_iteracoes = 100
random_mlp = RandomizedSearchCV(mlp_clf,
                                param_distributions=opcoes_parametros, n_iter=n_iteracoes)
random_mlp.fit(X_treino_std, y_treino)
```

A execução do código de busca aleatória necessita que seja definida a quantidade de interações e dentro dessa quantidade serão escolhidos diferentes valores dos hiperparâmetros. Já na busca em grid, não há necessidade de uma quantidade predefinida de tentativas, mas todas as

combinações possíveis de serem feitas com as faixas de valores são testadas, o que pode levar a uma quantidade maior de testes e, consequentemente, um tempo maior na obtenção do melhor modelo candidato. O código a seguir pode ser utilizado para executar uma busca em grid:

```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
import scipy.stats as stats
from sklearn.utils.fixes import loguniform
from sklearn.model_selection import GridSearchCV

# Preparação dos dados
X_treino, y_treino, X_teste, y_teste = load_dataset()
# Declaração de um MLP
mlp_clf = MLPClassifier(warm_start=True, max_iter=100)
# Faixas de busca dos hiperparâmetros
opcoes_parametros = {
   'hidden_layer_sizes': [(30,30,30),(20,80),(25,50,5)],
   'activation': ['tanh', 'relu', 'logistic'],
   'solver': ['sgd', 'adam', 'lbfgs'],
   'alpha': [0.0001, 0.09],
   'learning_rate': ['constant','adaptive']}
# Busca em grid
grid_mlp = GridSearchCV(mlp_clf, param_grid=opcoes_parametros)
grid_mlp.fit(X_treino_std, y_treino)
```

Ao final da execução dos métodos de busca, teremos os valores dos parâmetros com melhor desempenho. Os valores podem ser utilizados diretamente para definir um modelo, uma vez que os objetos utilizados para realizar as buscas (GridSearchCV e RandomSearchCV) armazenam os parâmetros de todos os candidatos testados. Assim, é possível escolher uma lista de modelos candidatos promissores.

TEMA 3 – REDES NEURAIS CONVOLUCIONAIS

Anteriormente, estudamos a estrutura, funcionamento e treinamento de uma rede neural de múltiplas camadas, o Perceptron de Múltiplas Camadas (MLP). A partir da estrutura do MLP, diversas outras arquiteturas de redes neurais foram propostas como solução de diferentes problemas que não eram

possíveis de resolver utilizando a estrutura básica de redes neurais de múltiplas camadas.

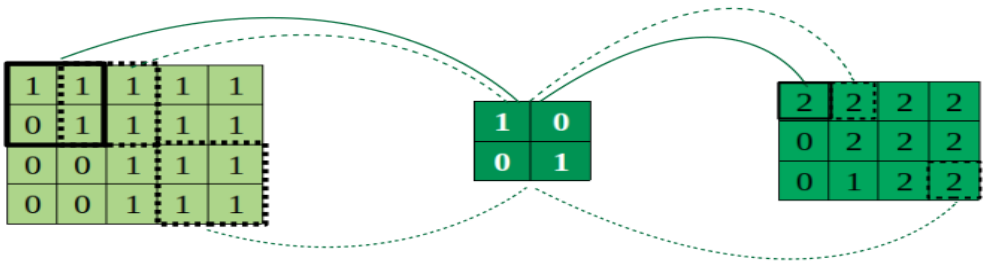
3.1 *Convolutional Neural Networks* (CNN)

As redes neurais convolucionais (CNN) se referem a uma arquitetura de rede neural desenvolvida para lidar com dados armazenados de forma matricial ou em grade, especialmente no campo de visão computacional para o tratamento de dados de imagens digitais.

A arquitetura das CNN foi desenhada de maneira que a rede, antes de fornecer os dados aos neurônios, sofre diferentes etapas de processamento com a execução de procedimentos de convoluções (*convolution*) e reduções de dimensionalidade (*pooling*). Estas operações são realizadas diversas vezes sobre uma imagem de entrada, antes que os valores obtidos possam servir de entrada para a rede neural.

A convolução é uma operação matemática sobre duas funções que produz uma terceira função. No caso das imagens, é aplicada uma operação matricial sobre uma pequena porção da imagem de entrada, utilizando-se uma matriz de tamanho pequeno (filtro), obtendo-se uma nova matriz de dados. A Figura 8 ilustra como ocorre o processo de convolução para uma matriz 4x5 com a aplicação de um filtro 2x2.

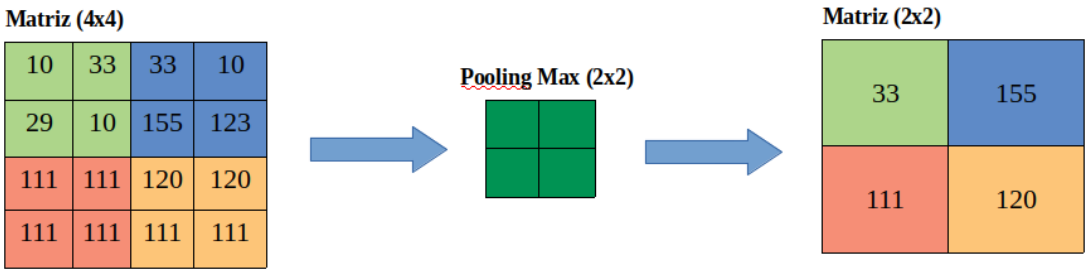
Figura 8 – Convolução sobre uma matriz com um filtro 2x2



Por sua vez, a operação de redução de dimensionalidade (*pooling*) é aplicada sobre uma pequena porção da imagem, gerando um único valor com base nos elementos que definem a matriz desta pequena porção. Dessa forma, uma porção que contenha quatro valores, quando aplicado uma redução de 2x2, resultará em um único valor. A aplicação de uma redução 2x2, por

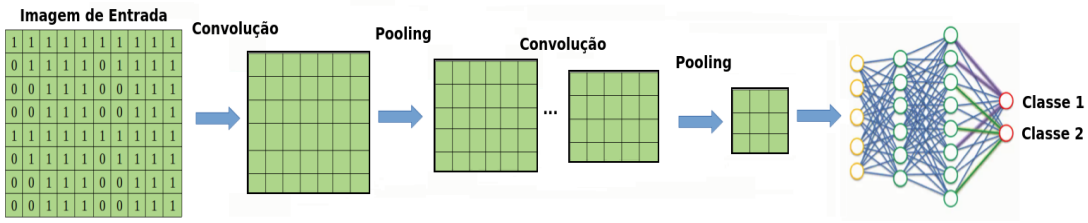
exemplo, se realizada sobre toda a imagem, resultará em uma matriz com 25% do tamanho original. A Figura 9 ilustra a aplicação de uma redução de dimensionalidade 2x2 sobre uma matriz 4x4, em que a cada quatro elementos se escolhe o maior.

Figura 9 – Redução de dimensionalidade 2x2 do tipo Max



Após a execução dos processos de convolução e de redução de dimensionalidade, os resultados obtidos são repassados para as camadas de neurônios da CNN, que apresentam a peculiaridade de terem todos os seus neurônios conectados. Ou seja, cada neurônio repassa a sua saída para todos os outros da próxima camada e, ao final, é obtida uma categoria ou classe para a imagem de entrada. É fácil notarmos que esta é a mesma construção do perceptron de múltiplas camadas que vimos anteriormente. A arquitetura e o fluxo dos dados na CNN são representados de forma simplificada na Figura 10.

Figura 10 – Representação simplificada de uma rede neural do tipo CNN



3.2 Problemas resolvidos com CNN

As redes do tipo CNN podem se apresentar em diferentes arquiteturas e podem ser aplicadas a diferentes tarefas, com cada uma dessas arquiteturas sendo especializadas para tarefas específicas. Dentre as tarefas mais comuns, podemos destacar:

- **classificação de imagens:** consiste em, dada uma imagem, determinar a qual classe ela pertence. O processo de treinamento é feito de forma supervisionada, em que são fornecidos diversos exemplares das diferentes categorias e são fornecidas as informações das categorias destas imagens;
- **segmentação de objetos:** consiste em, dada uma imagem, separar ou recortar determinado objeto do restante ela;
- **detecção de objetos:** consiste em, dada uma imagem, detectar se nela há determinado objeto e gerar uma marcação na imagem que permita localizá-lo dentro dela;
- **detecção de faces:** consiste em, dada uma imagem, localizar e segmentar as faces que forem detectadas nela. É uma aplicação de detecção e localização de objetos, porém é uma tarefa muito recorrente, exigindo uma certa especialização;
- **reconhecimento facial:** consiste em, dada uma imagem contendo uma face, identificar em um conjunto de faces armazenadas em uma base, o nível de semelhança da face fornecida com as demais.

Diversas outras arquiteturas do tipo CNN podem ser utilizadas para outras tarefas, por meio do treinamento da rede em uma base de treinamento, que deve ser mantida à medida que o modelo gerado seja colocado em produção e instâncias inéditas sejam apresentadas ao modelo e o seu desempenho se degrade.

TEMA 4 – REDES NEURAIS SEQUENCIAIS

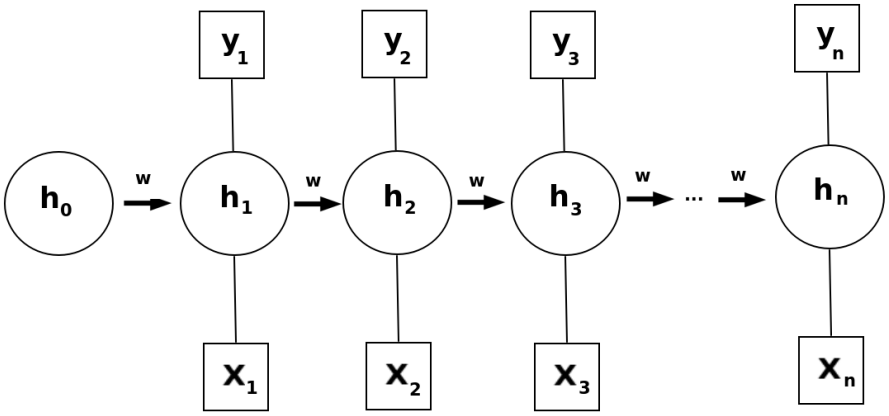
Até agora vimos modelos de aprendizagem de máquina tradicionais e modelos de redes neurais que lidam com dados de tamanho fixo, mas há situações e dados de entrada que podem apresentar tamanho variável. Esse tipo de dado não pode ser fornecido como entrada para uma arquitetura de rede neural com o perceptron de múltiplas camadas, nem às redes neurais convolucionais. A razão para isso reside no fato dessas arquiteturas exigirem receber dados de tamanho fixo e que não apresentem dependência entre as diferentes instâncias. No entanto, dados como uma sequência de palavras em um texto podem ter tamanho variado e dependem das instâncias que os

precedem. Para lidar com problemas desses tipos foram criadas as redes neurais recorrentes (RNN) (*recurrent neural networks*).

4.1 *Recurrent neural networks*

As redes do tipo RNN possuem uma peculiaridade de reter informações das entradas anteriores enquanto geram as saídas, e não apenas durante a fase de treinamento, como ocorre em outras arquiteturas de redes neurais, de maneira que as redes recorrentes podem receber um ou mais valores de entradas e produzir a mesma quantidade na saída e os valores resultantes são influenciados pelas entradas e saídas anteriores. A arquitetura de uma rede neural recorrente é apresentada na Figura 11.

Figura 11 – Representação expandida de rede neural recorrente



4.2 Evolução das RNN

As redes do tipo RNN, apesar de permitirem lidar com entradas de tamanho variável e conseguirem reter informação das entradas e saídas anteriores, permitindo modelar o contexto de informações que apresentem dependência sequencial e temporal, também apresentam dificuldades no seu processo de treinamento por conta da forma como a informação é propagada.

Durante o processo de propagação dos pesos das camadas ocultas anteriores, o cálculo do valor do gradiente pode gerar valores muito elevados ou muito pequenos, levando a duas situações conhecidas, respectivamente, como explosão e desaparecimento do gradiente. Essas situações ocorrem por

conta do acúmulo de valores muito grandes ou muitos pequenos, que são propagados por meio das camadas e multiplicados sucessivamente.

Diversas soluções e diferentes arquiteturas foram propostas como forma de resolver os problemas que impediam um uso efetivo das RNN. Dentre as propostas, em uma arquitetura definida como *Long Short-Term Memory* (LSTM), conseguiram mitigar o problema de explosão e desaparecimento do gradiente, bem como obtiveram uma melhoria na arquitetura que permitia reter informações de curto prazo (*short-term memory*) e, também, a influência dos pesos para longas sequências de entrada (*long-term memory*).

4.3 Problemas resolvidos com RNN e LSTM

As redes recorrentes como RNN e LSTM podem ser aplicadas a problemas de diferentes áreas. Dentre as principais aplicações, podemos destacar:

- **modelagem de linguagem:** consiste em obter um modelo que, dada uma palavra, consegue predizer a próxima palavra;
- **tradução de máquina:** tradução automática de uma sequência de texto de uma língua para outra;
- **reconhecimento de voz:** consiste em receber como entrada um áudio e convertê-lo para texto;
- **sumarização de texto:** consiste em receber um texto e gerar uma versão resumida dele.

Vários outros problemas podem ser resolvidos utilizando redes deste tipo, sendo uma importante ferramenta para situações em que arquiteturas de redes neurais profundas e redes convolucionais não podem ser utilizadas.

TEMA 5 – *FRAMEWORKS* PARA REDES NEURAIS

Até o momento, todos os algoritmos com os quais trabalhamos puderam ser criados e manipulados por meio da biblioteca *scikit-learn*, cujos processos matemáticos estão todos baseados em duas outras bibliotecas do Python, o Numpy e o SciPy. Todavia, existem outras bibliotecas e *frameworks* que podem executar os mesmos tipos de procedimentos e complementam o *scikit-learn*, provendo maneiras de criar diferentes arquiteturas de redes neurais por conta das ferramentas que possuem, que são especializadas para trabalhar com a

criação e treinamento destes tipos de rede. Dentre os *frameworks* utilizados, podemos destacar o Keras, que será utilizado para demonstrar como construir uma arquitetura de rede neural profunda e uma rede do tipo CNN.

5.1 Criando uma rede neural em Keras

O Keras pode ser utilizado para criar uma estrutura de rede neural semelhante àquela que utilizamos em seções anteriores. É importante observar que os processos de preparação e carga dos dados de treinamento e de teste permanecem os mesmos, bem como os procedimentos para avaliar o desempenho e acompanhar o treinamento do modelo.

A código a seguir executa a criação de uma rede neural (um MLP), que recebe como entrada um vetor de tamanho 10, possui duas camadas ocultas com 30 neurônios cada e uma camada final com apenas dois neurônios:

```
import keras
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(30, input_dim=10,  activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(2, activation='softmax'))
```

Uma vez que a estrutura da rede neural esteja criada, o modelo pode ser compilado, para em seguida ser treinado. As chamadas para treinamento e predição são semelhantes às utilizadas no *scikit-learn* (método *fit* e *predict*), e os métodos de avaliar o desempenho dos modelos que aprendemos anteriormente também podem ser utilizados. O código a seguir mostra a compilação e o treinamento do modelo definido anteriormente:

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(X_treino, y_treino, epochs=100, batch_size=64)
```

Analisando o processo de definição e treinamento do modelo, podemos ver os mesmos elementos que foram utilizados para definir o nosso modelo de MLP. A função de *loss*, o otimizador e a métrica de avaliação durante o treinamento do modelo são definidos na chamada de compilação, enquanto os

dados de treino, a quantidade de *epochs* e o tamanho do *batch* a ser utilizado no treinamento são todos passados na chamada do método *fit*.

O processo de treinamento gera uma saída que permite acompanhar sua evolução. Em geral, o treinamento dos modelos gera uma saída, como a mostrada abaixo:

Epoch 1/100
116/116 [==============================] - 0s 1ms/step - _loss_: 0.2962 - accuracy: 0.8695 - auc_1: 0.9480
Epoch 2/100
116/116 [==============================] - 0s 2ms/step - _loss_: 0.2901 - accuracy: 0.8729 - auc_1: 0.9505

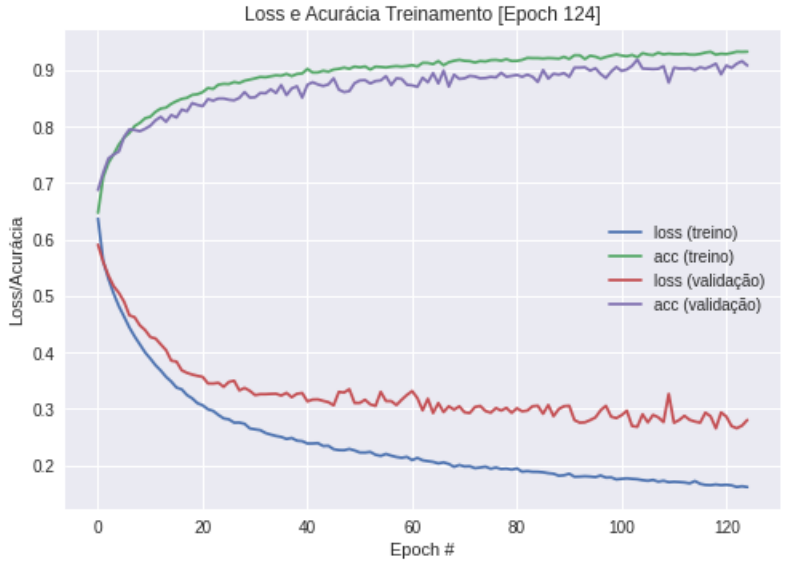
...

Epoch 99/100
116/116 [==============================] - 0s 1ms/step - _loss_: 0.1736 - accuracy: 0.9234 - auc_1: 0.9824
Epoch 100/100
116/116 [==============================] - 0s 1ms/step - _loss_: 0.1837 - accuracy: 0.9176 - auc_1: 0.9797

Os valores da métrica escolhida para avaliar o desempenho – neste caso a acurácia – e o valor da função de *loss* são exibidos a cada *epoch*. Assim, é possível avaliar os resultados que o modelo está obtendo no treinamento. Contudo, esse tipo de abordagem pode ser difícil, pois a quantidade de *epochs* pode ser muito alta ou o processo muito demorado.

Os *frameworks* especializados em redes neurais possuem métodos que facilitam a observação da evolução do modelo à medida que aumenta o número de *epochs* durante o treinamento. Permitindo, inclusive, definir funções, que são chamadas em momentos específicos do treinamento. Assim, é uma prática comum, ao invés de observamos a evolução apenas pela saída textual, gerar um gráfico que permite ter uma noção melhor da evolução do modelo. A Figura 12 mostra o gráfico de um processo de treinamento de um modelo de rede neural.

Figura 12 – Treinamento e validação



Outra possibilidade oferecida por *frameworks* com o Keras é a de salvar os pesos da rede em intervalos ou condições definidas. Assim, é possível restaurar o modelo a um estado anterior que tenha obtido um resultado que interesse. Um gráfico contendo dados do treinamento pode indicar onde o ponto de interesse está localizado.

5.2 Criando uma CNN em Keras

Assim como o Keras permite a criação de uma rede neural tradicional, seus objetos e métodos permitem criar redes de arquiteturas diferentes como uma CNN. O processo de definição é semelhante, porém, como vimos anteriormente, a CNN possui características próprias como as camadas de convolução e de redução de dimensionalidade.

A quantidade, o tipo e a forma como as camadas de convolução e *pooling*, bem como as camadas que as precedem se relacionam, são alguns dos principais componentes que definem a arquitetura de uma rede neural. No caso específico das CNNs, existem arquiteturas que obtiveram bons resultados em tarefas de classificação ou detecção de imagens, sendo bastante conhecidas e disponibilizadas por padrão em *frameworks* especializados.

Uma CNN, diferentemente de uma rede neural comum, necessita receber dados de duas ou três dimensões, bem como, as suas camadas de convolução necessitam da definição do tamanho dos filtros a serem aplicados. A definição de uma CNN para lidar com imagens de tamanho 80x80x3 (3

dimensões de profundidade), com duas camadas de convolução com filtros 3x3 e pooling 2x2 aplicados, seguidos por uma camada densa contendo 64 neurônios e uma última camada com um único neurônio, pode ser vista no código a seguir:

```
model = Sequential()
# Adiciona uma camada de convolução com 64 unidades de filtros 3x3
model.add(Conv2D(64,(3,3),
                 activation = 'relu',
                 input_shape = (80,80,3)))
# Adiciona uma camada de pooling
model.add(MaxPooling2D(pool_size = (2,2)))
# Adiciona mais uma camada de convolução e pooling
model.add(Conv2D(64,(3,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
# Retifica os valores
model.add(Flatten())
model.add(Dense(64, activation='relu'))
# Adciona uma camada de softmax
model.add(Dense(1, activation='sigmoid'))
```

Da mesma maneira que ocorre com as redes neurais comuns definidas em Keras, os demais processos são semelhantes. Inclusive o acompanhamento do processo de carga de dados e treinamento, sendo possível também carregar um modelo de uma arquitetura pré-definida, bem como carregar os valores dos pesos obtidos no treinamento de uma base de imagens e executar um novo treinamento para as classes do problema que se deseja resolver. De fato, essa é a prática mais comum em classificação de imagens, pois o treinamento completo de uma rede neural profunda desde o início é um processo custoso do ponto de vista computacional.

FINALIZANDO

Nesta aula apresentamos os principais conceitos relacionados às redes neurais, desde o seu componente mais básico, o perceptron, até as definições de arquiteturas profundas especializadas em tarefas relacionadas à visão computacional e processamento de dados sequenciados.

Vale lembrar que os conceitos aprendidos no início do nosso curso sobre o estudo do problema e uma avaliação criteriosa e operações para obter

dados de qualidade a serem usados no treinamento são válidos e de grande importância para qualquer processo de aprendizagem de máquina. Devemos sempre atentar para o fato de que, a escolha de um vetor de características que permita obter a melhor representação das instâncias dos dados pode poupar tempo no processo de treinamento e conduzir a modelos mais robustos.

É de grande importância que o profissional de aprendizagem de máquina saiba que existem *frameworks* e bibliotecas especializados que permitem executar os processos de aprendizagem e até mesmo criar estruturas de alta complexidade, como redes neurais de grande profundidade. Contudo, compreender, testar e avaliar o funcionamento dos diferentes algoritmos e modelos de rede neural é algo que deve ser feito com base na experiência e na percepção do profissional, sendo necessário ter técnica, criatividade e um pouco de arte.

REFERÊNCIAS

CUESTA, H. **Practical data analysis**. [s.l.]: Packt Publishing Ltd., 2013.

GÉRON, A. **Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow:** concepts, tools, and techniques to build intelligent systems. [s.l.]: O'Reilly Media, 2019.

MCKINNEY, W. **Python para análise de dados:** tratamento de dados com Pandas, NumPy e IPython. São Paulo: Novatec Editora, 2019.

RICHERT, W. **Building machine learning systems with Python**. [s.l.]: Packt Publishing Ltd., 2013.

RUSSEL, S. J.; NORVIG, P. **Inteligência Artificial:** uma abordagem moderna. 3. ed./Tradução Regina Célia Simille. Rio de Janeiro, Brasil: Editora Elsevier, 2013.