Como implementar sua primeira rede neural em PyTorch

Introdução

Esse tutorial é, em grande parte, uma tradução para o português brasileiro do tutorial intitulado Build Your First Neural Network with PyTorch [3], entretanto algumas adaptações foram realizadas, tanto no texto, quanto no código anexo, em relação à versão original.

Nesse tutorial, você aprenderá como implementar, treinar e utilizar uma Rede Neural *Feed-Foward* simples para uma tarefa de classificação binária.

Para tal, utilizaremos o pacote PyTorch que é, atualmente, uma das principais ferramentas para a implementação de modelos neurais viáveis.

Implementaremos nossa rede neural em *Python 3* e, além do *PyTorch*, você também precisará instalar e importar os pacotes NumPy, Pandas e Scikit-learn.

A tarefa que usaremos para fins de exemplo será a de prever se choverá ou não numa cidade australiana amanhã, utilizando dados meteorológicos mensurados na mesma cidade no dia de hoje. A redução dessa tarefa de previsão a uma classificação binária é, evidentemente, uma grande simplificação do problema real de previsão meteorológica, mas como veremos, ainda pode apresentar resultados interessantes, além do caráter didático.

Entendendo os dados

As informações que utilizaremos para treinar nosso modelo para a tarefa de previsão de chuvas estão contidas num conjunto que reúne dados meteorológicos de diversas cidades australianas. Esse conjunto de dados foi curado e disponibilizado através do Kaggle[1] por Joe Young[2].

Os dados estão no formato `.csv` e, com eles em mãos, o primeiro passo é carregá-los em um *data-frame* [3], usando as ferramentas do pacote *pandas*:

```
1 raw_data = pd.read_csv(data_path)
```

Com os dados carregados, é possível averiguar que eles são constituídos por 142193 entradas, cada uma contando com 24 variáveis distintas. É possível notar, também, que existem entradas para as quais nem todas as variáveis estão instanciadas. Além disso,

[1] https://www.kaggle.com/jsphyg/weather-dataset-rattle-package

[2] https://www.kaggle.com/jsphyg

[3] Para executar o código da maneira como ele está disponibilizado, coloque o arquivo de dados dentro de um diretório chamado `data/`, dentro do diretório `src/`

que nem todos os valores estão nos formatos que gostaríamos que estivessem para serem processados.

Isso é normal. Dados reais são cheios de falhas e problemas, e exigem trabalho e entendimento para serem utilizados da maneira correta. Por isso, é necessário realizar um **pré-processamento** para adequar os dados, antes de os passarmos para o modelo.

O primeiro passo é escolher quais das variáveis meteorológicas nos interessam. No nosso caso, queremos prever se choverá ou não amanhã, então `RainTomorrow` será nossa variável alvo. Para prevê-la usaremos as variáveis `Rainfall`, `Humidity3pm`, `Pressure9am` e `RainToday`, que serão nossas *features*. Podemos isolá-las das demais assim:

```
cols = ['Rainfall', 'Humidity3pm', 'Pressure9am', 'RainToday', 'RainTomorrow']
processed_data = raw_data[cols]
```

Em seguida, iniciamos o pré-processamento, propriamente dito:

As Variáveis `RainToday` e `RainTomorrow` possuem dois valores possíveis, *"Yes"* e *"No"*. Adequamos esses valores, convertendo-os para 1 e 0, respectivamente:

```
processed_data['RainToday'].replace({'No': 0, 'Yes': 1}, inplace=True)
processed_data['RainTomorrow'].replace({'No': 0, 'Yes': 1}, inplace=True)
```

A seguir, removemos todas as entradas que não tenham instanciados os valores de todas as variáveis de interesse, pois essas entradas são inúteis para treinar nosso modelo:

```
processed_data = processed_data.dropna(how='any')
```

Com os dados pré-processados, é possível, agora, plotar as distribuições das variáveis de interesse, para poder entender melhor como essas distribuições funcionam. Esse tipo de trabalho é muito importante na implementação real de redes neurais, conhecer os dados é fundamental para tirar o maior proveito do seu modelo e entender verdadeiramente seus resultados.

Dentre todas as distribuições das variáveis de interesse, a que mais nos concerne é a da variável alvo, `RainTomorrow`, representada na *Figura 1*.

Essa distribuição nos revela um grande desbalanço entre os dois valores possíveis dessa variável, que constituirão as duas classes do nosso problema de classificação. Esse é um dado importante, pois pode influenciar significativamente a capacidade preditiva do modelo treinado.

Existem maneiras de se lidar com o desbalanceamento dos dados, mas nesse tutorial utilizaremos os dados dessa forma. Isso significa que o *baseline* para a performance do nosso modelo deve ser 78%, isso porque, se um modelo chutasse que amanhã não irá chover, todas as vezes, ele obteria uma performance dessa ordem e, como esperamos gerar um modelo mais inteligente que isso, esperamos também que a a nossa performance seja superior a essa.

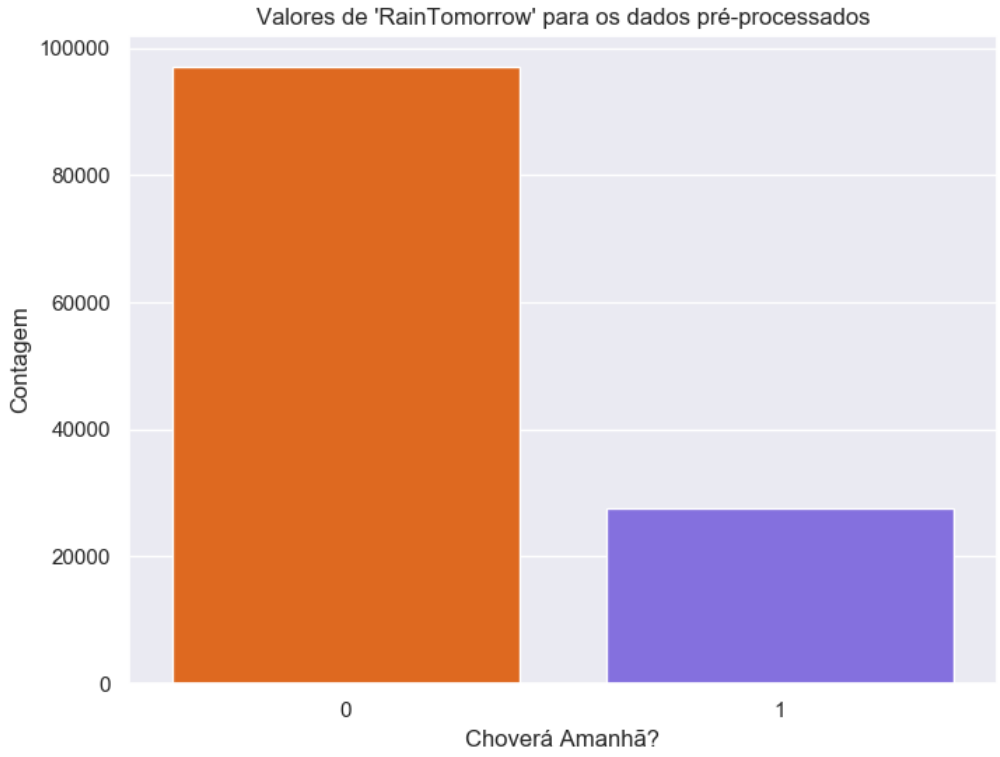

Figura 1: Distribuição de valores para a variável `RainTomorrow`, após o pré-processamento

Todo o código referente a esse pré-processamento está devidamente organizado no arquivo `data.py`, que também contém a função utilizada para a geração do gráfico da *Figura 1*, que, se executada, gera, também, visualizações para as distribuições das outras variáveis de interesse.

Montando a arquitetura

Com os dados pré-processados, podemos implementar o modelo. Nossa intenção é montar um Rede Neural *Feed-Foward* simples e totalmente conectada, com duas camadas ocultas. Tal arquitetura está ilustrada na *Figura 2*.

Podemos construir esse modelo através do seguinte código:

```
1  class FFN_2HLayers(nn.Module):
2
3      def __init__(self, n_features, n_hl1, n_hl2):
4          super(FFN_2HLayers, self).__init__()
5
```

```
        self.hidden_layer_1 = nn.Linear(n_features, n_hl1)
        self.hidden_layer_2 = nn.Linear(n_hl1, n_hl2)
        self.output_layer = nn.Linear(n_hl2, 1)


    def forward(self, x):
        h1 = nn.functional.relu(self.hidden_layer_1(x))
        h2 = nn.functional.relu(self.hidden_layer_2(h1))
        y = sigmoid(self.output_layer(h2))
        return y
```

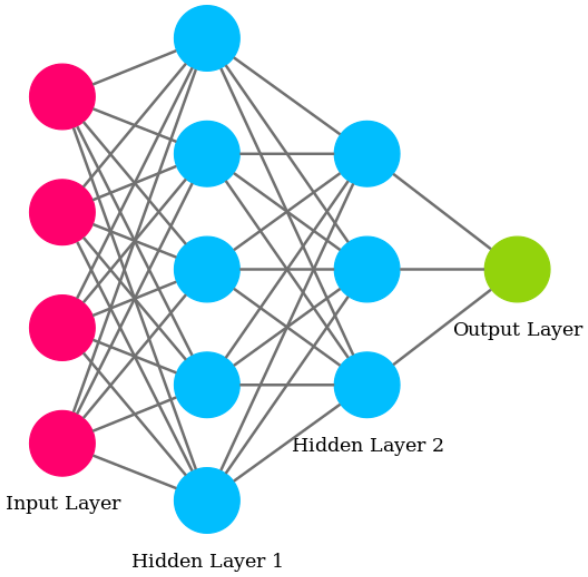

Figura 2: Ilustração da arquitetura neural do nosso modelo preditivo para `n_hl1=5` e `n_hl2=3`, extraída de [3]

Nesse último bloco de código, nosso modelo é representado pela classe `FFN_2HLayers`. O *Pytorch* já oferece um modulo básico de redes neurais, o `nn.Module`, que contém todos os aspectos necessários de implementação de uma rede neural, com exceção das peculiaridades de cada modelo. Ao declarar a classe do nosso modelo, fazemos ela herdar o `nn.Module`, para que nossa rede possa contar com toda essa estrutura pré-implementada.

O construtor instancia as transformações lineares que geram as camadas da rede neural. A primeira transformação recebe os dados de entrada, ou seja, vetores com o número de componentes igual ao número de *features* escolhido, e os transforma em vetores de tamanho correspondente à primeira camada oculta, a segunda transformação leva esses vetores para um espaço de dimensão correspondente a segunda camada oculta e a transformação de

saída transforma esses vetores em vetores de uma única componente (valores escalares), que serão utilizados para realizar a previsão de cada caso.

Com as transformações necessárias definidas, é possível definir o método *foward*, ainda dentro da classe do modelo, que explicita como os dados de entrada fluirão através do arquitetura usando essas transformações.

Você pode notar que o processamento dos dados através da rede neural não se constitui apenas em executar as transformações lineares sobre eles. Os resultados dessas transformações também são passados por outras funções, a *ReLU* e a *Sigmoide*. Essas funções são chamadas de **funções de ativação**, ou funções de não-linearidade, porque são elas que introduzem na rede a capacidade de modelar transformações não-lineares, expandindo vertiginosamente sua habilidade preditiva [1].

A sigla *ReLU*, que significa *Rectified Linear Unit*, define a função dada pela *Equação 1*. Ela não é, entretanto, a única função de não linearidade disponível. A literatura está recheada de várias possibilidades distintas, com suas respectivas vantagens e desvantagens. Escolher as funções de ativação que gerem os melhores resultados empíricos faz parte do processo de desenho da arquitetura de um modelo neural.

$$ReLU(x) = max(0, x) \tag{1}$$

Já a função sigmoide é como definida pela *Equação 2*. Essa função restringe o valor de saída, colocando-o entre 0 e 1, e é especialmente útil como função de saída para um modelo de classificação binária, como o nosso. Isso porque, através dela, podemos interpretar a resposta da rede como a probabilidade da classe positiva, ou seja, a probabilidade de chover amanhã. Existem várias funções que poderiam restringir a imagem do nosso modelo dessa maneira, mas a grande diferença da sigmoide é sua fácil derivabilidade, que permite ao modelo computar de forma eficiente os gradientes necessários para o treinamento.

$$Sigmoide(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

O Código de definição do modelo, aqui apresentado, está disponível no arquivo `model.py`. Com isso estabelecido, podemos nos concentrar no treinamento da rede.

Treinando a Rede

Para começar, nós precisamos definir um **critério de otimização**, também chamado de **custo**, e um **algoritmo de otimização**. O nosso algoritmo de otimização será responsável por alterar progressivamente os parâmetros do modelo, de maneira a minimizar o custo definido. Nós esperamos que a qualidade da nossa previsão aumente conforme o custo é minimizado.

Para o critério de otimização, fazemos:

```
1 criterion = torch.nn.BCELoss()
```

O critério escolhido é o `BCELoss`, que é uma função que mede a diferença entre dois vetores binários. No nosso caso, calcularemos a diferença entre o vetor correspondente à previsão realizada pelo nosso modelo e o vetor correspondente a previsão correta, para cada caso. Minimizar esse valor significa que nossas previsões estão ficando cada vez mais próximas das respostas corretas dos exemplos fornecidos à rede.

Para definir o algoritmo de otimização, fazemos:

```
optimizer = torch.optim.Adam(network.parameters(), lr=0.001)
```

O *Adam* é um algoritmo de otimização muito usado para redes neurais, proposto em [2]. Ele recebe, além dos parâmetros da rede neural, que serão atualizados, também o **híper-parâmetro** `lr`, que representa a **taxa de aprendizado** do modelo. A taxa de aprendizado define a sensibilidade do modelo em relação a novos exemplos, quanto maior essa taxa, maior a amplitude da modificação realizada nos parâmetros da rede em função de um único exemplo.

Contudo, se o modelo aprende com base em exemplos, ele pode acabar se adaptando excessivamente aos exemplos fornecidos no treinamento, perdendo a capacidade de generalizar para situações diferentes daquelas apresentadas no conjunto de treinamento. Esse fenômeno é chamado de **hiperaprendizado**, **overfitting** no inglês, e é para evitá-lo, que normalmente se treina o modelo com um conjunto de dados distinto daquele que se usa para avaliá-lo. Tais conjuntos são chamados **Conjunto de treinamento** e **Conjunto de Validação**.

Para separar nossos dados nesses dois conjuntos, fazemos:

```
x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=0.2,
                                    random_state=RANDOM_SEED)
```

Aqui, fizemos uma separação aleatória das entradas, construindo um conjunto de validação que corresponde a 20% do nosso total de dados.

Precisamos, também, converter esses conjuntos em conjuntos de tensores do *Pytorch*, que é o formato necessário para sua utilização no treinamento:

```
x_train = torch.from_numpy(x_train.to_numpy()).float()
y_train = torch.squeeze(torch.from_numpy(y_train.to_numpy()).float())
x_val = torch.from_numpy(x_val.to_numpy()).float()
y_val = torch.squeeze(torch.from_numpy(y_val.to_numpy()).float())
```

Antes de iniciar o treinamento, nós podemos ainda, caso possível, optar por realizá-lo numa placa gráfica (*GPU*). *GPU*s têm uma arquitetura de hardware especialmente eficiente para o treinamento de modelos neurais. Para realizar a seção de treinamento na *GPU* é preciso que todos os objetos necessários, incluindo os conjuntos de treinamento e de

validação, sejam transferidos para a placa gráfica, por isso, a seguir está uma função que faz essa transferência:

```
def transfer_to_device(data, network, criterion):
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    (x_train, y_train, x_val, y_val) = data

    x_train = x_train.to(device)
    y_train = y_train.to(device)

    x_val = x_val.to(device)
    y_val = y_val.to(device)

    network = network.to(device)
    criterion = criterion.to(device)

    trasfered_data = (x_train, y_train, x_val, y_val)
    return (trasfered_data, network, criterion)
```

E, por fim, precisamos definir duas funções auxiliares:

```
def calculate_accuracy(y_true, y_pred):
    predicted = y_pred.ge(.5).view(-1)  # Binariza a previsao
    return (y_true == predicted).sum().float() / len(y_true)


def round_tensor(t, decimal_places=3):
    return round(t.item(), decimal_places)
```

A primeira é responsável por calcular a acurácia de um modelo, ou seja a taxa de acertos do modelo em relação ao número de previsões realizadas, que será nossa medida de performance.

Já a segunda é apenas uma função auxiliar que arredonda os valores para fácil exibição.

Com isso, temos todas ferramentas necessárias para realizar o treinamento:

```
def train_network(data, network, optimizer, criterion,
                  MODEL_PATH):

    (data, network, criterion) = transfer_to_device(data, network, criterion)
    (x_train, y_train, x_val, y_val) = data

    # O treinamento consiste na repeticao  do cilo foward-backward atraves
```

```
    # do conjunto de treinamento, repetidas vezes (epocas):
    for epoch in range(1000):
        y_pred = network(x_train)  # faz-se a previsao
        y_pred = torch.squeeze(y_pred)
        train_loss = criterion(y_pred, y_train)  # calcula-se o custo
        # Imprime resutados parciais:
        if epoch % 100 == 0:
            train_acc = calculate_accuracy(y_train, y_pred)
            y_val_pred = network(x_val)
            y_val_pred = torch.squeeze(y_val_pred)
            val_loss = criterion(y_val_pred, y_val)
            val_acc = calculate_accuracy(y_val, y_val_pred)

            tr_lss = round_tensor(train_loss)
            tr_acc = round_tensor(train_acc)
            vl_lss = round_tensor(val_loss)
            vl_acc = round_tensor(val_acc)

            print('EPOCH {0}:'.format(epoch))
            print('Train Set --- loss:{0}; acc:{1}'.format(tr_lss, tr_acc))
            print('Validation Set  --- loss:{0}; acc:{1}'.format(vl_lss, vl_acc))

        optimizer.zero_grad()  # zera os gradientes do otimizador
        train_loss.backward()  # faz a passagem de volta
        optimizer.step()  # atualiza os parametros

    # Salva o modelo em disco:
    torch.save(network, MODEL_PATH)
```

O treinamento consiste na apresentação repetitiva do conjunto de dados à rede. Em cada iteração, apresentamos uma entrada ou conjunto de entradas ao modelo, calculamos a resposta da rede àquela entrada, calculamos o custo daquela resposta, comparando-a com a resposta correta, resetamos o valor dos gradientes, propagamos o custo pela rede, recalculando os gradientes, e atualizamos o valor dos parâmetros do modelo em função desses gradientes.

Cada um dos ciclos em que apresentamos todos os dados do conjunto de treinamento à rede é chamado de **época**. Normalmente, várias épocas são necessárias para maximizar a performance.

De tempos em tempos, calculamos o valor do índice de performance, no nosso caso a acurácia do modelo, para os conjuntos de treinamento e validação. A acurácia de treinamento pode continuar crescendo mesmo depois que a acurácia de validação tenha saturado ou começado a decrescer. Isso está normalmente relacionado ao hiperaprendizado e, por isso, é normal parar o treinamento quando alcançamos a acurácia de validação máxima,

utilizando o modelo como estava neste momento do treinamento.

Quando estamos satisfeitos com a performance, salvamos os parâmetros, que constituem o modelo, em disco. Isso é tudo que precisaremos para instanciar o modelo treinado e usá-lo para realizar previsões.

Todo o código referente ao treinamento, apresentado nessa seção, está implementado no arquivo `train.py`

Usando a Rede

Para usar um modelo treinado, basta carregá-lo a partir de um arquivo em disco:

```
network = torch.load(model_path)
```

Fornecendo, em seguida, as *features*, devidamente convertidas para tensores do *PyTorch*, e binarizar o *output* recebido. Interpretamos, então, a resposta 0 como negativa e 1 como a resposta positiva em relação a questão de se vai ou não chover amanhã.

```
def will_it_rain(network, device, rainfall, humidity, rain_today, pressure):
    entry = torch.as_tensor([rainfall, humidity, rain_today, pressure]) \
        .float() \
        .to(device)
    output = network(entry)
    return output.ge(0.5).item()  # Binariza o output da rede
```

Essa função, bem como funções de uso amigável ao usuário e iterativo, estão definidas no arquivo `use.py`

Conclusões

Se você rodar o arquivo principal `main.py`, ele executará todo o fluxo de trabalho da rede, apresentado até aqui, pré-processando os dados, treinando o modelo com eles e abrindo um menu para seu uso.

Olhando as estatísticas do treinamento, você pode ver que o modelo alcança uma acurácia de validação da ordem de 83.6%. Ela está significativamente acima do nosso *baseline*, o que é um bom indício, mas uma análise mais profunda de sua eficácia pode ser reveladora. Apesar de ela estar fora do escopo desse Tutorial, você pode ler um pouco mais em [3], se você quiser.

Esperamos que esse tutorial tenha introduzido você à implementação de Redes Neurais usando o *PyTorch*. Esse é um exemplo simples, tanto em modelagem quanto em aplicação, mas engloba vários dos aspectos do uso real de redes neurais.

Referências

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.

[2] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. pages 1–15, 2014.

[3] Venelin Valkov. Build your first neural network with pytorch. https://www.curiousily.com/posts/build-your-first-neural-network-with-pytorch/, 2020. Accessado: 27/05/2020.