Guia: Aprendizagem profunda com PyTorch

Resumo do *notebook* "Introdução à aprendizagem profunda com PyTorch"

Escopo: Este documento é um resumo do *notebook* "Introdução à aprendizagem profunda com PyTorch", do módulo 3 do curso "Deep Learning: Domínio das redes neurais", do MIT xPRO. Sua finalidade é oferecer uma visão geral dos aspectos principais desse *notebook*.

Índice

1. Classificação com PyTorch
   1.a. Treinando o modelo
2. Atributos adicionais
   2.a. Preparação dos dados
   2.b. Treinamento
   2.c. Curvas de treinamento
3. Regularização e *dropout*
4. Modelo de regressão

1. Classificação com PyTorch

Vamos usar o PyTorch para realizar uma tarefa de classificação binária em um conjunto de dados de código aberto. Neste caso, vamos usar um conjunto de dados de abalones e tentar diferenciar abalones jovens e abalones velhos. Após carregar os dados em um *dataframe* (df) usando o Pandas, vamos separar 50% do conjunto de dados para usar no treinamento; os outros 50% serão usados para teste. Preparamos uma nova variável chamada "Old", que indicará se o abalone é velho (positivo, com rótulo 1) ou não (negativo, com rótulo 0). Finalmente, para esta primeira tarefa de classificação, vamos usar apenas as colunas 2 a 5 como nossos atributos (numéricos).

```
column_names = ["Sex", "Length", "Diameter", "Height", "Whole weight",
                "Shucked weight", "Viscera weight", "Shell weights", "Rings"] df =
pd.read_csv('abalone.data', header=None, names=column_names)
df['Old'] = 0 # Por padrão, o abalone é jovem
df.loc[(df['Rings'] >= 10), 'Old'] = 1 # 10 anéis ou mais significa que um Abalone é adulto
class_labels = ['Young', 'Old']  # [0, 1], [N, P]
# Primeiramente, queremos classificar apenas com atributos numéricos, excluindo o sexo e
# considerando apenas o peso total
numerical_feature_columns = column_names[1:5] print(numerical_feature_columns)
label_column = 'Old'
```

Nota: Escolhemos 10 anos como a diferenciação de idade entre abalones jovens e adultos.

Para essa tarefa de classificação, os dados devem ser preparados dividindo o conjunto de dados em três subconjuntos diferentes, conforme descrito no módulo. Em seguida, no *notebook*, procedemos à seleção dos hiperparâmetros (decisões de arquitetura do modelo, tamanho do lote, taxa de aprendizado e número de épocas de treinamento).

1.a. Treinando o modelo

Após definir a arquitetura do modelo e a função de treinamento, podemos prosseguir para treinar a rede de duas camadas ocultas. A rede de três camadas ocultas é treinada de maneira semelhante. Aqui, usamos a função de python "train_model" definida no *notebook*, que utiliza o método de descida do gradiente e o cálculo automático do gradiente no PyTorch para aprender os pesos do modelo. A perda da entropia cruzada é utilizada como função de perda, como é típico em modelos de classificação. Também usamos o Adam, um otimizador altamente popular, com uma taxa de aprendizado inicial especificada. O Adam ajusta (ou altera) a taxa de aprendizado e hiperparâmetros relacionados à medida que o treinamento progride.

```
# Treinamento em duas camadas ocultas
```

```
# Perda e otimizador
criterion = nn.CrossEntropyLoss() # CrossEntropyLoss para classificação
optimizer = torch.optim.Adam(two_layer_model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

# Treinar o modelo. Também armazenaremos os resultados do treinamento para
visualização two_layer_model, training_curves_two_layer = train_model(
                        two_layer_model, dataloaders, dataset_sizes,
                        criterion, optimizer, scheduler,
                        num_epochs=num_epochs
```

Nota: Os resultados mostram que o modelo de três camadas obteve maior precisão do que o modelo de duas camadas. Portanto, a arquitetura de três camadas será usada no *notebook*.

2. Atributos adicionais

Ao escolher apenas atributos numéricos, informações essenciais sobre os abalones não são incluídas. Para resolver isso, o *notebook* amplia o *dataframe* para incluir o sexo dos abalones no modelo. Como essa é uma variável categórica, usamos a codificação *one-hot* para criar três categorias desses atributos (“M”, “F” e “I”), em que o valor dos dados será 1 se esse for o sexo da amostra de dados, ou 0, caso contrário:

```
encoded_df = df.copy(True)
encoded_df.insert(1, 'M', 0)
encoded_df.insert(1, 'M', 0)
encoded_df.insert(1, 'I', 0)
encoded_df.loc[(df['Sex'] == 'M'), 'M'] = 1
encoded_df.loc[(df['Sex'] == 'F'), 'F'] = 1
encoded_df.loc[(df['Sex'] == 'I'), 'I'] = 1
encoded_column_names = column_names[:]
encoded_column_names.insert(1,"M")
encoded_column_names.insert(1,"F")
encoded_column_names.insert(1,"I")
encoded_df.head()

encoded_feature_columns = encoded_column_names[1:8]
print(encoded_feature_columns)
label_column = 'Old'
```

2.a. Preparação dos dados

Mais uma vez, uma divisão de treinamento-validação-teste deve ser realizada no conjunto de dados. Devemos nos certificar de que a padronização não é aplicada aos novos atributos da codificação *one-hot*.

```
# Configurar conjuntos de dados e DataLoaders do PyTorch
dataloaders = {'train': DataLoader(encoded_train_dataset,
               batch_size=batch_size), 'val': DataLoader(encoded_val_dataset,
               batch_size=batch_size), 'test': DataLoader(encoded_test_dataset,
               batch_size=batch_size)}
dataset_sizes = {'train': len(encoded_train_dataset),
                 'val': len(encoded_val_dataset),
```

2.b. Treinamento

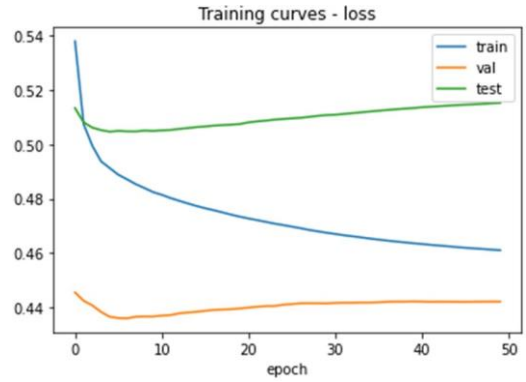
Em seguida, configuramos e realizamos o treinamento, de forma semelhante ao anterior, mas agora com o modelo de três camadas e novos dados codificados com *one-hot*:

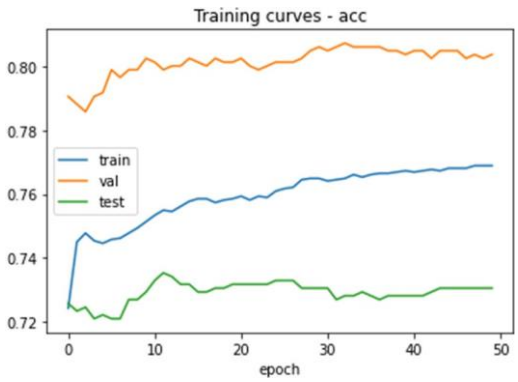
```
# Perda e otimizador
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(three_layer_encoded_model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
three_layer_encoded_model, training_curves_three_layer_encoded = train_model(
                  three_layer_encoded_model, dataloaders, dataset_sizes,
                  criterion, optimizer, scheduler, num_epochs=num_epochs
                  )
```

Como resultado, o *notebook* mostra que a precisão melhora.

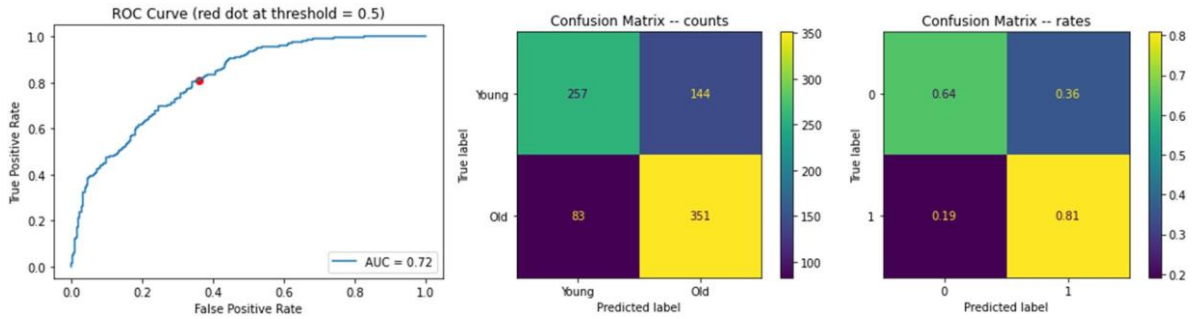
2.c. Curvas de treinamento

Analisamos as curvas de treinamento para perda e precisão para verificar como o modelo está se saindo conforme cada época de treinamento avança. Abaixo, apresentamos um exemplo de curvas (suas curvas podem ser diferentes devido à amostragem aleatória) e encontramos evidências de que o modelo está se sobreajustando no conjunto de dados. A perda de teste parece aumentar após aproximadamente 5 épocas, indicando que o modelo treinado está sobreajustado e generaliza mal os dados de teste.





Além disso, a curva ROC e a matriz de confusão fornecem um resumo do desempenho do modelo:



A curva ROC tem uma área sob a curva (AUC) de 0,72. Isso indica uma capacidade preditiva razoável, mas com alguns erros (como a taxa de falsos positivos de cerca de 0,36 no ponto vermelho, correspondente ao limiar de decisão padrão de 0,5).

Na matriz de confusão, vemos que cerca de 36% dos abalones jovens são classificados erroneamente como abalones adultos, cerca de 19% dos abalones adultos são classificados erroneamente como jovens. Esse valor de 19% corresponde a 1 menos a taxa de verdadeiros positivos (abalones adultos classificados corretamente como adultos), ou um valor de 0,81, que está no ponto vermelho em relação ao eixo vertical da curva ROC.

3. Regularização e *dropout*

Regularização e *dropout* são dois métodos usados para evitar o sobreajuste. Ambos estão incorporados na estrutura do PyTorch. Primeiramente, vamos aplicar a regularização para tentar evitar o sobreajuste. Aqui, fazemos a regularização L2 usando decaimento de peso no otimizador e, em seguida, retreinamos:

```
# Reiniciar o modelo
three_layer_encoded_l2_model = SimpleClassifier3Layer(input_size, hidden_size1,
hidden_size2,
                                                      hidden_size3, num_classes).to(device)
# Perda e otimizador
criterion = nn.CrossEntropyLoss()

# Ao adicionar um termo de weight_decay (decaimento de peso) ao otimizador, estamos
incluindo a Regularização L2
optimizer = torch.optim.Adam(three_layer_encoded_l2_model.parameters(),
                             lr=learning_rate, weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
three_layer_encoded_l2_model, training_curves_three_layer_encoded_l2 = train_model(
          three_layer_encoded_l2_model, dataloaders, dataset_sizes,
```

Nota: A regularização parece ter ajudado com o problema de sobreajuste. No entanto, a perda nos testes ainda está abaixo do esperado de acordo com as novas curvas de treinamento. No *notebook*, em seguida, tentaremos resolver isso introduzindo o *dropout* após cada camada, como mostrado abaixo:

```
# Modelo de classificação simples com três camadas ocultas e dropout
class SimpleClassifier3LayerDropout(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, hidden_size3,
                 num_classes, dropout):
        super(SimpleClassifier3LayerDropout, self).__init__()
        self.dropout = nn.Dropout(dropout) # Taxa de dropout
        self.layers = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU(),
            self.dropout, # ADICIONADO
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU(),
            self.dropout, # ADICIONADO
            nn.Linear(hidden_size2, hidden_size3),
            nn.ReLU(),
            self.dropout, # ADICIONADO
            nn.Linear(hidden_size3, num_classes),
        )

    def forward(self, x):
        return self.layers(x)
```

Importante: Neste exemplo, o *notebook* mostra que o *dropout* não foi a melhor escolha e resultou em um desempenho pior do modelo. Isso é comum em modelos profundos relativamente simples.

5. Modelo de regressão

Em seguida, construiremos um modelo de regressão para prever o número de anéis que os diferentes abalones possuem, com base em seus outros atributos. O *notebook* fornece detalhes sobre duas mudanças importantes entre a tarefa de classificação e a de regressão.

Primeiramente, definimos a rede neural "regression_model", cuja saída será o número de anéis previstos, em vez de um rótulo de classe. Em segundo lugar, treinaremos o modelo com uma função de perda diferente. Neste caso, usamos o erro quadrático médio (EQM, ou MSE em inglês), geralmente apropriado para problemas de regressão.

Após aplicar os procedimentos usuais de preparação, prosseguimos para treinar o modelo de regressão.

```
# Perda e otimizador
criterion = nn.MSELoss() # Erro Quadrático Médio (MSELoss) em vez de entropia cruzada
(CrossEntropy)
optimizer = torch.optim.Adam(regression_model.parameters(), lr=learning_rate,
                             weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

regression_model, training_curves_regression =
                train_regression_model( regression_model,
                dataloaders, dataset_sizes, criterion, optimizer,
```

Observando os resultados do nosso modelo de regressão no *notebook*, podemos ver como a classificação e a regressão são semelhantes e onde estão as diferenças importantes.