Guide to Deep Learning with PyTorch

PyTorch Introduction to Deep Learning with PyTorch Notebook Summary

Scope: This document is a summary of the “Module 3: Introduction to Deep Learning with PyTorch” notebook from the course “Deep Learning: Mastering Neural Networks” by MIT xPRO. This summary offers a quick overview of the main aspects included in the notebook.

Table of Contents

1. Classification with PyTorch

We will use PyTorch to perform a binary classification task on an open-source dataset. In this case, we will use an abalone dataset and attempt to differentiate young and old abalones. After loading the data into a dataframe (df) using Pandas, we sample 50% of the data set to use for training; the other 50% will be used for testing. We prepare a new variable “Old” that will indicate if the abalone is old (positive, with label 1) or not (negative, with label 0). Finally, for this first classification task, we’ll just use the 2nd through 5th columns as our (numerical) features.

```
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  # By default, abalone is Young
df.loc[(df['Rings'] >= 10), 'Old'] = 1 # 10 rings or more means an Abalone is old
class_labels = ['Young', 'Old']   # [0, 1], [N, P]
# We first only want to classify with numerical features excluding sex and
# only the whole weight
numerical_feature_columns = column_names[1:5]
print(numerical_feature_columns)
label_column = 'Old'
```

! **Note:** we have chosen 10 years as the age differentiation between old and young abalones.

For this classification task, the data must be prepared by splitting the dataset into three different subsets as described in the module. Then, in the notebook we proceed to the selection of hyperparameters (model architecture decisions, batch size, learning rate, and number of training epochs).

1.a. Training the Model

After defining the model architecture and the training function, we can proceed to train the two-hidden-layer network. The three-hidden-layer network is trained in a similar way. Here, we use the “train_model” python function defined in the notebook, which uses gradient descent (and automatic gradient calculation in PyTorch) to learn the model weights. Cross entropy loss is used as the loss function, as is typical with classification models. We also use the highly popular Adam optimizer with an initial specified learning rate; Adam schedules (or changes) the learning rate and related learning hyperparameters, as the training progresses.

```
# Two-hidden-Layer Training
```

```
# loss and optimizer
criterion = nn.CrossEntropyLoss() # CrossEntropyLoss for classification!
optimizer = torch.optim.Adam(two_layer_model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

# Train the model. We also will store the results of training to visualize
two_layer_model, training_curves_two_layer = train_model(
                        two_layer_model, dataloaders, dataset_sizes,
                        criterion, optimizer, scheduler, num_epochs=num_epochs
                        )
```

! **Note:** results show that the three-layer model achieved higher accuracy than the two-layer model. Therefore, the three-layer architecture is used moving forward in the notebook.

2. Additional Features

By only choosing numerical features, critical information about the abalone is not included. To solve this, the notebook extends the dataframe so as to include abalone sex in the model. Since this is a categorical variable, we use one-hot encoding to create three category features, 'M", 'F', and 'I', where the data value will be 1 if that this the data sample's sex, or 0 if not:

```
encoded_df = df.copy(True)
encoded_df.insert(1, 'M', 0)
encoded_df.insert(1, 'F', 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. Data Preparation

Once again, a Train-Validation-Test split must be performed on the dataset. We make sure that standardization is not performed on the new one-hot encoding features.

```
# Set up pytorch Datasets and DataLoaders
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),
                 'test': len(encoded_test_dataset)}
```

2.b. Training

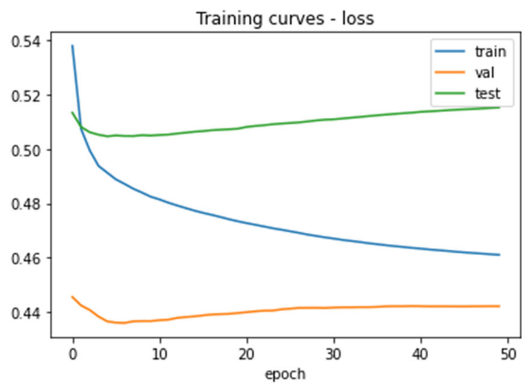
Then we set up and perform the training, similar to before but now with the three layer model, and new one-hot encoded data:

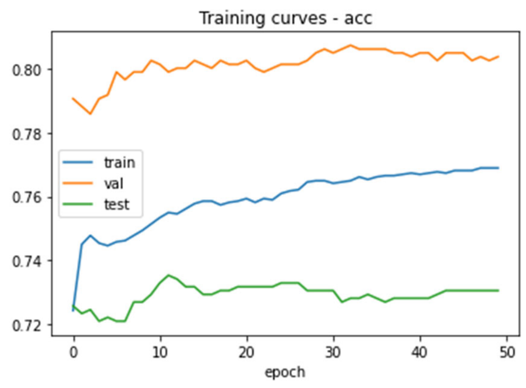
```
# loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(three_layer_encoded_model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
three_layer_encoded_model, training_curves_three_layer_encoded = train_model(
  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

As a result, the notebook shows that accuracy improves.

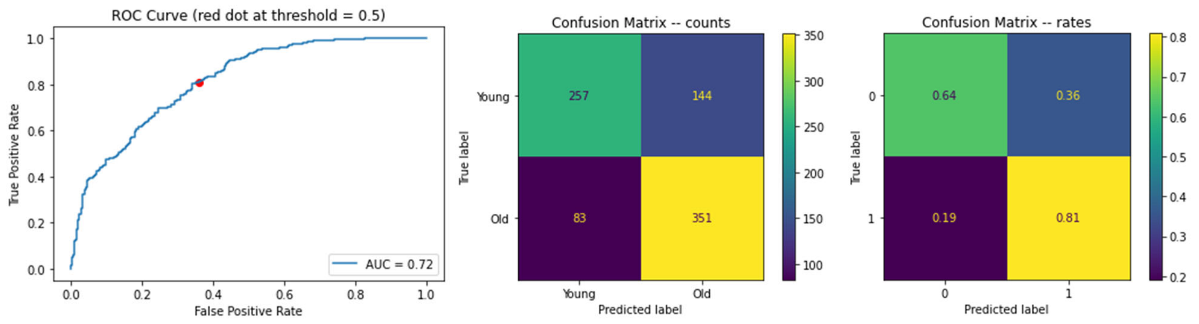
2.c. Training Curves

We look at the training curves for loss and accuracy to see how the model is performing as each epoch of training proceeds. An example set of curves is shown below (your curves may differ due to random sampling). Below, we find evidence that the model is overfitting to the dataset: the test loss appears to increase after about 5 epochs, indicating that the trained model is overfitting and generalizes poorly to the test data.





Additionally, the ROC curve and confusion matrix provide a summary of the model performance:



The ROC curve has an area under curve (AUC) of 0.72, which provides some reasonable predictive capability, but with some errors (like the False Positive Rate of about 0.36 at the red dot corresponding to the default 0.5 decision threshold).

In the confusion matrix, we see that about 36% of Young abalones are misclassified as Old, and about 19% of Old abalones are misclassified as Young. This 19% corresponds to 1 minus the True Positive Rate (correctly classifying Old abalones as Old) or value 0.81, on the vertical axis in the ROC at the red dot.

3. Regularization and Dropout

Regularization and dropout are two methods used to avoid overfitting. Both of them are built into the PyTorch framework. Let's first apply regularization to try to avoid overfitting. Here we do L2 regularization using weight decay in the optimizer, and then retrain:

```
# Reset the model
three_layer_encoded_l2_model = SimpleClassifier3Layer(input_size, hidden_size1, hidden_size2,
                                                      hidden_size3, num_classes).to(device)
# loss and optimizer
criterion = nn.CrossEntropyLoss()

# By adding a weight_decay term to the optimizer, we are including L2 Regularization!
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,
           criterion, optimizer, scheduler, num_epochs=num_epochs
           )
```

!Note: Regularization seems to have helped with the overfitting problem. However, the test loss is still underperforming according to the new training curves. In the notebook, we next attempt to solve this by introducing dropout after each layer, as below:

```
# Simple three-hidden-layer classification model with 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) # dropout rate
        self.layers = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU(),
            self.dropout, #ADDED
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU(),
            self.dropout, #ADDED
            nn.Linear(hidden_size2, hidden_size3),
            nn.ReLU(),
            self.dropout, #ADDED
            nn.Linear(hidden_size3, num_classes),
        )

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

!Important: In this example, the notebook shows that dropout was not the best choice and resulted in worse model performance. This is a common occurrence with relatively simple deep models.

5. Regression Model

We next build a regression model to predict the number of rings that the different abalones have, based on their other features. The notebook provides details on two important changes from the classification task, to the regressions task.

First, we define a “regression_model” neural network, where the output will be the predicted number of rings, rather than a class label. Second, we will train the model with a different loss function; in the case, we use mean square error (MSE), as typically appropriate for regression problems.

After applying the usual preparation procedures, we proceed to train the regression model.

```
# loss and optimizer
criterion = nn.MSELoss() # MSE Loss instead of 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, scheduler, num_epochs=num_epochs
                )
```

According to the results of our regression model in the notebook, we see how classification and regression are similar, and where the important differences are.