Machine Learning Assignment 04 - Classification and Regression Neural Network Models using PyTorch

Lucas Herranz Gancedo - M11351802

27th October 2024

1 Classification model

A classification problem is one of the main tasks in Machine Learning, the model's objective is to predict if the data input belongs to a certain category. This problem belongs to supervised learning, meaning that the machine learns by comparing its predictions to a ground-truth set of data. On the following section you will find my approach to building a classification neural network model using the widely known Machine Learning Python library PyTorch.

1.1 Python modules

Below you can find a list of the Pyhton modules and libraries required to run the script that builds, trains and evaluates the classification model.

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Listing 1: Python Modules for the binary classification model

Please also note that in order to optimize training times and use the maximum resources to train the model, I have used Nvidia CUDA platform to train the model on my laptop's GPU, as it can be seen on the next code snippet.

```
# 0) GPU device selection (if avaliable)
device = (
    "cuda"
    if torch.cuda.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

Listing 2: GPU/CPU device selection

If CUDA is not available on your device the script will still run on CPU, it will only probably take longer training time since it cannot make use of the parallelization technology from the GPU.

1.2 Dataset

To build a classification feed forward neural network I have decided to use the Wisconsin Breast Cancer dataset [1]. The dataset consists of 30 input features and two output values (malign or benign tumor), resulting in a binary classification problem. The features are extrapolated from a digitized image of a Fine Needle Aspirate (FNA) [2] of different breast masses. There are ten measurements computed for each cell radius, and the mean, standard error and the largest (mean of the three largest values) of these measurements were computed for each image, resulting in 30 input features.

I created a custom class that loads the dataset, standardizes the input features and transforms the data to PyTorch tensors. When an object of this class is created, the dataset is structured in a list of tuple pairs of features and target.

```
# 1) prepare data for training.
# Create a breast cancer dataset class and scale the data accordingly
class BreastCancerDataset(Dataset):
    def __init__(self, transform=None):
        # data loading
        scaler = StandardScaler()
        bc = datasets.load_breast_cancer()
        X_numpy, y_numpy = bc.data, bc.target
        y_numpy = y_numpy.reshape((len(y_numpy),1))
        X_numpy = scaler.fit_transform(X_numpy)
        self.X = X_numpy.astype(np.float32)
        self.y = y_numpy.astype(np.float32)
        self.n_samples = X_numpy.shape[0]
        self.transform = transform

    def __getitem__(self, index):
        # dataset[index]
        sample = self.X[index], self.y[index]
        if self.transform:
            sample = self.transform(sample)
        return sample

    def __len__(self):
        # len(dataset)
        return self.n_samples


class ToTensor():
    def __call__(self, sample):
        features, targets = sample
        return torch.from_numpy(features), torch.from_numpy(targets)
```

Listing 3: BreastCancerDataset custom class definition

I created a new object of BreastCancerDataset class and also generate a split of the whole dataset into train and test data on a 80-20 proportion. The test data is later used to measure the performance of the model.

```
bc_dataset = BreastCancerDataset(transform=ToTensor())
bc_train, bc_test = train_test_split(bc_dataset, test_size=0.2, random_state=1234)
```

Listing 4: BreastCancerDataset object and train-test split

1.3 Hyper-parameters and Pytorch dataloaders

Next step is to define the hyper-parameters of the model, these are the parameters that will define the training. As mentioned earlier, the input features are 30 and there are two possible output values (cancerous or not - 1 or 0), meaning a single neuron on the output of the network. I trained for 100 epochs, with train data in batches of 35 samples. The learning rate selected was 0.001, usual starting value for the later mentioned Adam optimizer.

```
# 2) Hyper-parameters and data loaders
input_size = 30
hidden_neuron_num = 15
output_size = 1 #binary classification (cancerous or not)
num_epochs = 100
batch_size = 35
learning_rate = 0.001

train_loader = DataLoader(dataset=bc_train, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=bc_test, batch_size=batch_size, shuffle=False)
```

Listing 5: Hyper-parameters and PyTorch Dataloaders for the binary classification model

Please also note the last two lines of code which create two PyTorch DataLoader objects, for the train and test dataset respectively. When training a model, typically we want to pass samples of data in “batches” and re-shuffle the data every epoch to reduce the model overfitting. DataLoader is an iterable object that abstracts this complexity for the user in an easy API.

1.4 Multi-layer neural network binary classification model design

The model features one single hidden layer, 30 neurons on the input (one per input feature), 15 neurons on the first hidden linear layer. The output of each of this neurons is fed through a Rectified Linear Unit (ReLU) function to introduce non-linear relations between the features (or neurons). The output is then fed through

a second linear layer that outputs a single value. Last, I use the sigmoid function at the end of the pipeline to squash the output between 0 a 1. This can be easily understood by remembering that we are performing a binary classification (cancerous or benign - 1 or 0), thus the output data of the model needs to be within this range, indicating the certainty of the prediction towards one or the other target.

```
# 3) Multi-layer Neural Network model definition, activation functions
# [30 inputs] -> [15 hidden neurons] ->  [1 output]
# one hidden layer
class BinaryClassification(nn.Module):
    def __init__(self, n_input_features, n_hidden_neurons, n_output_neurons):
        super(BinaryClassification, self).__init__()
        self.linear1 = nn.Linear(n_input_features, n_hidden_neurons)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(n_hidden_neurons, n_output_neurons)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        x = self.sigmoid(x)
        return x

model = BinaryClassification(n_input_features=input_size, n_hidden_neurons=hidden_neuron_num,
    n_output_neurons=output_size)
model.to(device=device)
```

Listing 6: Binary Classification Neural Network Model

1.5 Loss function and optimizer algorithm

To measure the performance of the model in its classification purpose, I use the Binary Cross-Entropy (BCE) loss function. BCE will measure how different are the ground truth and the model's prediction probability distributions. Additionally, I chose to use Adam optimizer algorithm after it showed better performance than the classic Stochastic Gradient Descent (SGD).

```
# 4) loss and optimizer definition
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Listing 7: Loss funtion and optimizer algorithm for the binary classifcation model

1.6 Training loop

Following this paragraph you can find the training loop. Please observe the use of batch training, for each epoch the training samples are shuffled and divided in batches of 35 samples. The model is trained with this 35 sample batches. The structure of the training loop is the usual one: 1) forward pass (compute prediction), 2) backward pass (compute gradients of the loss function) and 3) update the model's weights. I also requested the script to print the progress of the training every four steps to monitor the status of the training.

```
# 5) training loop (batch training)
history = []
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    loss_per_epoch = []
    for i, (inputs, labels) in enumerate(train_loader):
        inputs = inputs.to(device)
        labels = labels.to(device)

        # forward pass
        output = model(inputs)
        loss = criterion(output, labels)

        loss_per_epoch.append(loss.item())
        # backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i + 1) % 4 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, step {i+1}/{n_total_steps}, loss = {loss.
    item():.4f}')
```

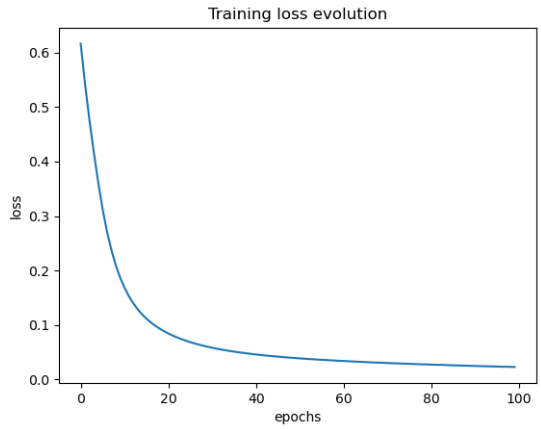

Figure 1: Evolution of the training loss in the binary classification neural network model

```
22
23      history.append(np.mean(loss_per_epoch))
```

Listing 8: Binary classification model training loop

You can also appreciate under these lines part of the command line output of the training, as well as the plotted evolution of the loss for each training epoch on Figure 1. The loss decays alongside the model is trained for more epochs, hence we have the expected behavior during the training for this model.

```
Using cuda device
Epoch 1/100, step 4/13, loss = 0.6227
Epoch 1/100, step 8/13, loss = 0.5724
Epoch 1/100, step 12/13, loss = 0.5799
Epoch 2/100, step 4/13, loss = 0.5361
Epoch 2/100, step 8/13, loss = 0.5395
Epoch 2/100, step 12/13, loss = 0.4673
...
Epoch 99/100, step 4/13, loss = 0.0091
Epoch 99/100, step 8/13, loss = 0.0066
Epoch 99/100, step 12/13, loss = 0.0575
Epoch 100/100, step 4/13, loss = 0.0386
Epoch 100/100, step 8/13, loss = 0.0109
Epoch 100/100, step 12/13, loss = 0.1048
```

Listing 9: Command line output of the binary classifier training loop

1.7 Testing the model

Last but not least, I test the model by measuring the accuracy of its classification ability on the test dataset. To do so, I feed batches of test data through the model, then compute the number of correct predictions for each of the test batches. At last the accuracy is computed by counting the amount of correct predictions versus the total samples.

```
# test
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for inputs, labels in test_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)

        predictions = torch.round(outputs)
        n_samples += labels.size(0)
        n_correct += (predictions == labels).sum().item()

    acc = 100.0 * n_correct /n_samples
    print("\nTest metrics: ")
    print(f'accuracy = {acc:.4f} %')
```

Listing 10: Code snippet for testing the binary classification model

Please check the command line output of the testing metrics, you can see how the model performs very well in this classification task with an accuracy score of almost 96.5%.

```
Test metrics:
accuracy = 96.4912 %
```

Listing 11: Command line output of the test metrics for the binary classification model

2 Linear Regression model

Linear regression is also one of the main problems that Machine Learning can tackle, the model's objective in this case is to learn from the input data to the most optimized linear functions that can be used for prediction with new datasets. Again, this is a supervised learning problem, meaning the model makes use of labeled data to compute loss values and update its weights, also PyTorch is used in this case.

2.1 Python modules

The Python modules and libraries used for this model are very similar to the previous one besides the data scaler object and Python's in-built "copy" module. I also enabled the Nvidia CUDA platform to run the training on GPU, I will not attach the code snippet to avoid repetition.

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import copy
import matplotlib.pyplot as plt
```

Listing 12: Python modules for the linear regression model

2.2 Dataset

For this model, I used the diabetes dataset from Efron et al. paper [3], which features 10 baseline variables: age, sex, body mass index, average blood pressure, and six blood serum measurements. These were obtained for each of the 442 patients, the size of the dataset. The target value is the response of interest, a quantitative measure of the disease's progression one year after the baseline. The approach is very similar to the previous model, the dataset is defined in its own class as tuples of input features and target, when the object is declared the "*ToTensor()*" transform is also passed to output the dataset samples as PyTorch tensors. A train-test dataset split in an 80-20 proportion is also done.

```
# 1) prepare data for training.
# Create a diabetes dataset class and scale the data accordingly
class DiabetesDataset(Dataset):
    def __init__(self, transform=None):
        # data loading
        diabetes = datasets.load_diabetes()
        sc = MinMaxScaler()
        X_numpy, y_numpy = diabetes.data, diabetes.target
        y_numpy = y_numpy.reshape(-1,1)
        y_numpy = sc.fit_transform(y_numpy)
        self.X = X_numpy.astype(np.float32)
        self.y = y_numpy.astype(np.float32)
        self.n_samples = X_numpy.shape[0]
        self.transform = transform

    def __getitem__(self, index):
        # dataset[index]
        sample = self.X[index], self.y[index]
        if self.transform:
            sample = self.transform(sample)
        return sample

    def __len__(self):
        # len(dataset)
        return self.n_samples


```

```
class ToTensor():
    def __call__(self, sample):
        features, targets = sample
        return torch.from_numpy(features), torch.from_numpy(targets)

diabetes_dataset = DiabetesDataset(transform=ToTensor())
diabetes_train, diabetes_test = train_test_split(diabetes_dataset, test_size=0.2, random_state
    =1234)
```

Listing 13: DiabetesDataset custom dataset class

Note that the data scaling here is only applied to the target data, this is because the dataset is loaded using the in-built "*load_dataset()*" method from the "*sklearn.datasets*" module, where the input features are already standardized but not the output target values. Thus, I only normalize the output target using the Min-Max scaler which constrains the values between 0 and 1. Normalization of the output target showed in this case better performance than standardization.

2.3 Hyper-parameters and PyTorch DataLoaders

For this model the selection of hyperp-arameters is listed under this sentences, please note that as mentioned earlier there are 10 input features in the dataset and a single output target. Differently from the previous model, this time only one PyTorch DataLoader object is created to divide the data in batches for future training. The testing or evaluation of the model will be performed with the whole test dataset at the end of each training epoch, thus no requirement for a DataLoader.

```
# 2) Hyper-parameters and dataloaders
input_size = 10
hidden_neuron_num = 4
output_size = 1
num_epochs = 100
batch_size = 10
learning_rate = 1e-3

train_loader = DataLoader(dataset=diabetes_train, batch_size=batch_size, shuffle=True)
```

Listing 14: Hyper-parameters and DataLoaders for the Linear Regression model

2.4 Multi-layer neural network linear regression model design

The model architecture in this case also features one hidden layer, please note that in this case a ReLU activation function is used in the output of the hidden layer, however, no activation function is used for the neural network output. This is done in linear regression problems because the target is a numerical value (not a class like in the previous case). The neural network architecture is also depicted on figure 2, note that ReLU activation function is applied to the output of the hidden layer.

```
# 3) Multi-layer Neural Network model definition, activation functions
# [10 inputs] -> [4 hidden neurons] -> [1 output]
# a single hidden layer
class LinearRegression(nn.Module):
    def __init__(self, n_input_features, n_output_neurons, n_hidden_neurons):
        super(LinearRegression, self).__init__()
        self.linear1 = nn.Linear(n_input_features, n_hidden_neurons)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(n_hidden_neurons, n_output_neurons)

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x

model = LinearRegression(n_input_features=input_size, n_output_neurons=output_size,
    n_hidden_neurons=hidden_neuron_num)
model.to(device=device)
```

Listing 15: Linear Regression model custom class

2.5 Loss function and optimizer algorithm

For this model, the used loss function is the Mean Squared Error (MSE), which computes the average of the squared error. Being the error the difference between the estimated and actual values. As an optimizer algorithm, again I opted to use Adam since it had the best performance.

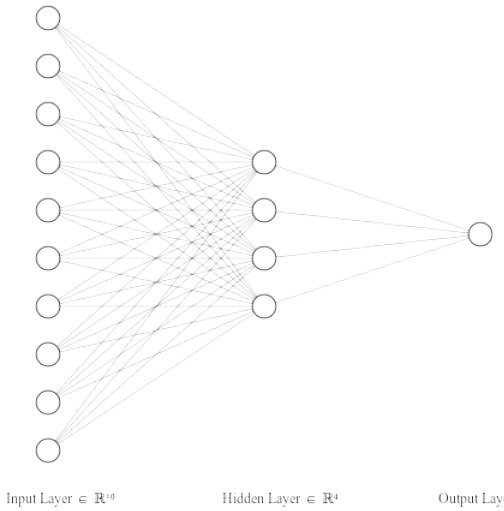

Figure 2: Linear Regression Model Neural Network architecture

```
# 4) loss and optimizer definition
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Listing 16: Loss function and optimizer algorithm for the Linear Regression Model

2.6 Training and validation loop

After all the declarations done prior, it is time for the sauce and thus I will start describing the training loop for this model. First, I define some auxiliary variables that I will make use of during the loop to keep the best model (the one with less MSE loss) and also some lists for plotting later. Secondly, I start looping through each epoch where I divide the train dataset in batches of 10 samples in this case. I go through the usual forward pass, backward pass and weight update. Stats on the training are also printed every six batches. Differently from the previous binary classification model, this time I evaluate the model on the test dataset after each epoch. MSE is computed for the whole dataset, if it is the lowest MSE value so far it is stored and so are the weights for the model with the best loss value.

```
# Hold the best model
best_mse = np.inf   #init to infinity
best_weights = None
history = []

# 5) training loop (batch training)
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (inputs, targets) in enumerate(train_loader):
        inputs = inputs.to(device)
        targets = targets.to(device)

        # forward pass
        output = model(inputs)
        loss = criterion(output, targets)

        # backward pass
        optimizer.zero_grad()
        loss.backward()

        optimizer.step()

        if (i + 1) % 6 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, step {i+1}/{n_total_steps}, loss = {loss.item():.4f}')

    # evaluate accuracy at end of each epoch
    test_in, test_targets = zip(*[diabetes_test[j] for j in range(len(diabetes_test))])
    test_in = torch.stack(test_in).to(device)
```

```
    test_targets = torch.stack(test_targets).to(device)
    y_pred = model(test_in)
    mse = criterion(y_pred, test_targets)
    mse = float(mse)
    history.append(mse)

    if mse < best_mse:
        best_mse = mse
        best_weights = copy.deepcopy(model.state_dict())

# Print the best evaluation metrics
print("\nEvaluation metrics: ")
print(f"\tMSE: {best_mse:.4f}")
print(f"\tRMSE: {np.sqrt(best_mse):.4f}")
print()
```

Listing 17: Training and validation loop for the Linear Regression Model

The best evaluation metrics, MSE and RMSE (which is the square-root of the MSE), of the whole training loop are printed alongside the training milestones, as you can see on the command line output shown below.

```
Using cuda device
Epoch 1/100, step 6/36, loss = 0.5187
Epoch 1/100, step 12/36, loss = 0.5845
Epoch 1/100, step 18/36, loss = 0.6039
Epoch 1/100, step 24/36, loss = 0.6313
Epoch 1/100, step 30/36, loss = 0.4906
Epoch 1/100, step 36/36, loss = 0.1718
Epoch 2/100, step 6/36, loss = 0.4577
Epoch 2/100, step 12/36, loss = 0.5419
Epoch 2/100, step 18/36, loss = 0.3903
Epoch 2/100, step 24/36, loss = 0.5629
Epoch 2/100, step 30/36, loss = 0.5070
Epoch 2/100, step 36/36, loss = 0.3819
...
Epoch 99/100, step 6/36, loss = 0.0317
Epoch 99/100, step 12/36, loss = 0.0318
Epoch 99/100, step 18/36, loss = 0.0219
Epoch 99/100, step 24/36, loss = 0.0328
Epoch 99/100, step 30/36, loss = 0.0165
Epoch 99/100, step 36/36, loss = 0.0759
Epoch 100/100, step 6/36, loss = 0.0430
Epoch 100/100, step 12/36, loss = 0.0286
Epoch 100/100, step 18/36, loss = 0.0284
Epoch 100/100, step 24/36, loss = 0.0459
Epoch 100/100, step 30/36, loss = 0.0183
Epoch 100/100, step 36/36, loss = 0.0275

Evaluation metrics:
        MSE: 0.0279
        RMSE: 0.1672
```

Listing 18: Command line output of the linear regressor training loop

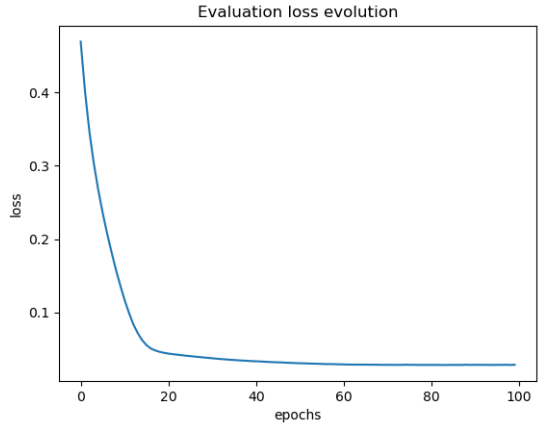

Figure 3: Evolution of the evaluation loss (MSE) in the linear regression neural network model

As earlier mentioned the targets are normalized, thus if we consider their range we achieved an acceptable RMSE value, however the model should be further fine-tuned to lower more this loss, ideally to achieve RMSE scores under 0.1. However, the evaluation MSE loss decays satisfactorily during the duration of the training, as seen in figure 3.

References

[1] W. Wolberg, O. Mangasarian, N. Street, and W. Street, "Breast Cancer Wisconsin (Diagnostic)," UCI Machine Learning Repository, 1993, DOI: https://doi.org/10.24432/C5DW2B.

[2] V. C. U. Health, "Fine needle aspiration," https://www.vcuhealth.org/services/breast-imaging/imaging-guided-procedures/fine-needle-aspiration, accessed: 2024-09-30.

[3] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *The Annals of Statistics*, vol. 32, no. 2, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1214/009053604000000067