Dataset 2 - Classification task using MLP

In [1]:
```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,accuracy_score,classification_report
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import pandas
```

Firstly, the dataset is loaded and the features (X) and target (y) are extracted as arrays. Next, the data is split into 80% training and 20% testing to evaluate the model performance, specifying a random state so that the data division will be equal at each run. Features are then scaled to have mean 0 and a standard deviation equal to 1, which helps models to converge faster and perform better.

In [2]:
```
# load dataset
diabetes = fetch_openml("diabetes",version=1,as_frame=True)
X = diabetes.data.values
y = diabetes.target.values
X.shape
y = (y=='tested_positive').astype(np.int64)

# train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size, random_state=42)

# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

In this part of the code, the multilayer perceptron (MLP) is defined. It consists of an input layer, four fully connected hidden layers with 64 neurons each and a final output layer. After each hidden layer, a dropout operation is applied, with a default probability of 0.5. So, during training, half of the neurons are randomly deactivated to reduce overfitting and improve the network's ability to generalize. The hidden layers of this network use the ReLU (Rectified Linear Unit) activation function, which introduces non-linearity and enables the network to capture complex relationships in the data. On the other hand, the output layer has a single neuron and no activation function, leaving its interpretation and any task-specific processing to the choice of loss function.

In [3]:
```
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 64)
        self.out = nn.Linear(64, output_size)

        self.dropout = nn.Dropout(p=dropout_prob)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)
```

```
        x = F.relu(self.fc3(x))
        x = self.dropout(x)

        x = F.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.out(x)
        return x
```

The next section specifies the hyperparameters of this MLP model.

- epochs: The epochs are the number of times the training algorithm will iterate over the entire training dataset. Choosing a number too large might cause overfitting, while too small can result in underfitting. In this case, 100 epochs were used;
- learning rate: This hyperparameter controls how much the model weights are updated in response to the computed gradient during training. In this case, larger numbers can accelerate training but may cause divergence, while smaller values may slow down convergence. A learning rate of 0.0005 is used to ensure small, stable weight updates;
- dropout: a value of 0.1 was used, meaning 10% of neurons are ignored at each training step;
- batch size: the batch size is the number of samples the model looks at before updating its weights. Smaller batch sizes can provide noisier gradient estimates, which may help the model generalize better. When a larger number is considered, it can provide more stable gradients, but requires more memory. In this case, a batch size of 64 was considered.

In [4]:
```
num_epochs=100
lr=0.0005
dropout=0.1
batch_size=64
```

Before training the model, the input features (Xtr, Xte) and targets (ytr, yte) are converted to PyTorch tensors. (Since PyTorch requires tensors for its computations and gradient tracking during training). Then, the training data is wrapped into a TensorDataset, which pairs each input sample with its corresponding target. This allows the data to be fed to the model in a structured way. Finally, a DataLoader is created from the TensorDataset. The DataLoader handles batching and shuffling: batching groups the training samples into batches of the specified batch_size (64) so that the model updates its weights after processing each batch, while shuffling randomly rearranges the data each epoch to improve generalization and prevent model from learning the order of samples.

In [5]:
```
Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32)

# Wrap Xtr and ytr into a dataset
train_dataset = TensorDataset(Xtr, ytr)

# Create DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

In this next code snippet, the model is created and moved to the appropriate device (GPU if available, otherwise CPU). The MLP is initialized with the number of input features and the specified dropout probability.

In this case, the loss function (criterion) is set to BCEWithLogitsLoss for binary classification, which combines a sigmoid activation and binary cross-entropy in a single step. The optimizer is Adam, which updates the model's weights during training using the gradients, with the specified learing rate (lr) controlling the size of these updates.

In [6]:
```
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout).to(device)
```

```
criterion = nn.BCEWithLogitsLoss()  # for binary classification
optimizer = optim.Adam(model.parameters(), lr=lr)
```

The model is trained for the specified number of epochs. At the beginning of each epoch, the model is set to training mode. For each batch from the DataLoader, the input features and targets are moved to the appropriate device. The model then performs a forward pass to compute the predictions (logits) and the loss is calulated using the chosen loss function. Before backpropagation, the optimizer's gradients are reset to zero. The loss is then backpropagated through the network (loss.backward()) and the optimizer updates the model's weights (optimizer.step()). The loss for each batch is accumulated and the average loss for the epoch is printed to monitor training progress.

In [7]:
```
# Training loop
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0.0

    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        logits = model(batch_x)
        loss = criterion(logits, batch_y.view(-1, 1))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    avg_loss = epoch_loss / len(train_dataloader)
    #print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}") commented for the PDF file
```

Finally, after training, the model is evaluated on the test set. The test features (Xte) are passed through the trained model to obtain predictions (y_pred).

Since the model outputs raw values, a threshold of 0.5 is applied to convert them into binary predictions. The predicted labels are then compared with the true targets (yte) to calculate the accuracy, which measures the proportion of the correct predictions.

The model achieved an accuracy of 74.7%

In [8]:
```
y_pred=model(Xte)
#performance metric for classification
print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}') #classification
```

```
ACC:0.7467532467532467
```