Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

Лабораторная работа №4

По дисциплине «Интеллектуальный анализ данных»

Тема: «Предобучение нейронных сетей с использованием RBM»

Выполнил:

Студент 4 курса

Группы ИИ-23

Вышинский А. С.

Проверила:

Андренко К. В.

Брест 2025

Цель: научиться осуществлять предобучение нейронных сетей с помощью RBM

Общее задание

1. Взять за основу нейронную сеть из лабораторной работы №3. Выполнить обучение с предобучением, используя стек ограниченных машин Больцмана (RBM – Restricted Boltzmann Machine), алгоритм которого изложен в лекции. Условие останова (например, по количеству эпох) при обучении отдельных слоев как RBM выбрать самостоятельно.

2. Сравнить результаты, полученные при
- обучении без предобучения (ЛР 3);
- обучении с предобучением, используя автоэнкодерный подход (ЛР3);
- обучении с предобучением, используя RBM.

3. Обучить модели на данных из ЛР 2, сравнить результаты по схеме из пункта 2;

4. Сделать выводы, оформить отчет по выполненной работе, загрузить исходный код и отчет в соответствующий репозиторий на github.

Задание по вариантам

№ в-а	Выборка	Тип задачи	Целевая переменная
11	https://archive.ics.uci.edu/dataset/863/maternal+health+risk	классификация	RiskLevel

Код:

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from ucimlrepo import fetch_ucirepo
import os
```

```
import datetime


torch.manual_seed(42)
np.random.seed(42)

SAVE_RESULTS = True
RESULTS_DIR = "./results/"
os.makedirs(RESULTS_DIR, exist_ok=True)



def load_maternal_health_data():
    data = fetch_ucirepo(id=863)
    X = data.data.features
    y = data.data.targets["RiskLevel"]

    le = LabelEncoder()
    y = le.fit_transform(y)

    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    return X_scaled, y



def load_rice_data():
    data = fetch_ucirepo(id=545)
    X = data.data.features
    y = data.data.targets["Class"]

    le = LabelEncoder()
    y = le.fit_transform(y)

    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    return X_scaled, y



class ClassificationNet(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(ClassificationNet, self).__init__()
        layers = []
        prev_size = input_size
        for h_size in hidden_sizes:
            layers.append(nn.Linear(prev_size, h_size))
            layers.append(nn.ReLU())
            prev_size = h_size
        layers.append(nn.Linear(prev_size, output_size))
        self.network = nn.Sequential(*layers)
```

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



class Autoencoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Linear(input_size, hidden_size)
        self.decoder = nn.Linear(hidden_size, input_size)

    def forward(self, x):
        x = torch.relu(self.encoder(x))
        x = self.decoder(x)
        return x



class RBM(nn.Module):
    def __init__(self, n_visible, n_hidden, k=1):
        super(RBM, self).__init__()
        self.W = nn.Parameter(torch.randn(n_hidden, n_visible) * 0.01)
        self.h_bias = nn.Parameter(torch.zeros(n_hidden))
        self.v_bias = nn.Parameter(torch.zeros(n_visible))
        self.k = k

    def sample_h(self, v):
        prob_h = torch.sigmoid(F.linear(v, self.W, self.h_bias))
        return prob_h, torch.bernoulli(prob_h)

    def sample_v(self, h):
        prob_v = torch.sigmoid(F.linear(h, self.W.t(), self.v_bias))
        return prob_v, torch.bernoulli(prob_v)

    def contrastive_divergence(self, v, lr=0.01):
        prob_h0, h0 = self.sample_h(v)
        v_k = v
        for _ in range(self.k):
            prob_v_k, v_k = self.sample_v(h0)
            prob_h_k, h_k = self.sample_h(v_k)

        self.W.data += lr * ((torch.matmul(prob_h0.t(), v) -
torch.matmul(prob_h_k.t(), v_k)) / v.size(0))
        self.v_bias.data += lr * torch.mean(v - v_k, dim=0)
        self.h_bias.data += lr * torch.mean(prob_h0 - prob_h_k, dim=0)

        loss = torch.mean((v - v_k) ** 2)
        return loss



def pretrain_layers(input_data, hidden_sizes, epochs=50, lr=0.01):
```

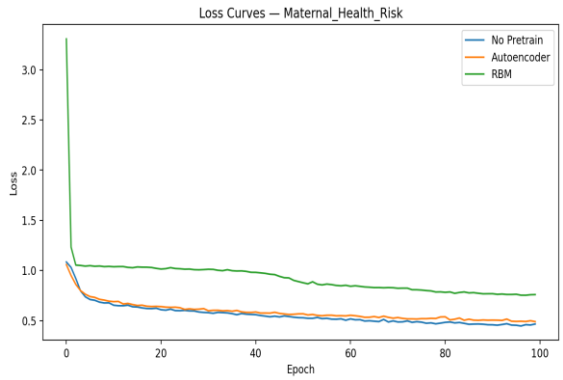
```
    pretrained_weights = []
    current_input = input_data

    for h_size in hidden_sizes:
        ae = Autoencoder(current_input.shape[1], h_size)
        optimizer = optim.Adam(ae.parameters(), lr=lr)
        criterion = nn.MSELoss()
        dataset = TensorDataset(current_input, current_input)
        loader = DataLoader(dataset, batch_size=32, shuffle=True)

        prev_loss = float('inf')
        patience = 5
        wait = 0

        for epoch in range(epochs):
            total_loss = 0
            for data, target in loader:
                optimizer.zero_grad()
                output = ae(data)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()

            avg_loss = total_loss / len(loader)
            if avg_loss > prev_loss - 1e-5:
                wait += 1
                if wait >= patience:
                    break
            else:
                wait = 0
            prev_loss = avg_loss

        with torch.no_grad():
            current_input = torch.relu(ae.encoder(current_input))
        pretrained_weights.append((ae.encoder.weight.data.clone(),
ae.encoder.bias.data.clone()))

    return pretrained_weights



def pretrain_rbm_layers(input_data, hidden_sizes, epochs=30, lr=0.01):
    pretrained_weights = []
    current_input = input_data

    for h_size in hidden_sizes:
        rbm = RBM(current_input.shape[1], h_size)
        dataset = DataLoader(current_input, batch_size=32, shuffle=True)

        for epoch in range(epochs):
            total_loss = 0
            for batch in dataset:
```

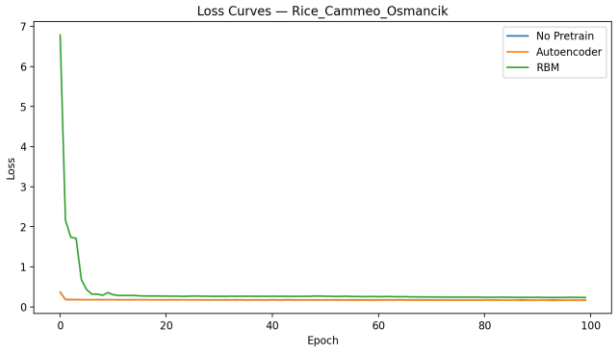
```
                batch = batch[0] if isinstance(batch, (list, tuple)) else
batch
                loss = rbm.contrastive_divergence(batch, lr=lr)
                total_loss += loss.item()
            print(f"RBM layer {h_size} — epoch {epoch+1}, loss={total_loss
/ len(dataset):.6f}")

        with torch.no_grad():
            prob_h, _ = rbm.sample_h(current_input)
            current_input = prob_h
            pretrained_weights.append((rbm.W.data.clone(),
rbm.h_bias.data.clone()))

    return pretrained_weights



def init_with_pretrain(net, pretrained_weights):
    i = 0
    for layer in net.network:
        if isinstance(layer, nn.Linear) and i < len(pretrained_weights):
            w, b = pretrained_weights[i]
            layer.weight.data = w
            layer.bias.data = b
            i += 1



def train_model(net, X_train, y_train, X_test, y_test, epochs=100,
lr=0.001, batch_size=32):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(net.parameters(), lr=lr)
    train_dataset = TensorDataset(X_train, y_train)
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

    losses = []
    for epoch in range(epochs):
        net.train()
        total_loss = 0
        for data, target in train_loader:
            optimizer.zero_grad()
            output = net(data)
            loss = criterion(output, target.long())
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        losses.append(total_loss / len(train_loader))

    net.eval()
    with torch.no_grad():
        y_pred = torch.argmax(net(X_test), dim=1).cpu().numpy()
        f1 = f1_score(y_test.cpu().numpy(), y_pred, average='weighted')
```

```
        cm = confusion_matrix(y_test.cpu().numpy(), y_pred)
    return f1, cm, losses



def process_dataset(name, loader_func):
    print(f"\n===== {name} Dataset =====")
    X, y = loader_func()

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    X_train = torch.tensor(X_train, dtype=torch.float32)
    X_test = torch.tensor(X_test, dtype=torch.float32)
    y_train = torch.tensor(y_train, dtype=torch.long)
    y_test = torch.tensor(y_test, dtype=torch.long)

    input_size = X_train.shape[1]
    hidden_sizes = [64, 32, 16]
    output_size = len(np.unique(y))


    net_no_pre = ClassificationNet(input_size, hidden_sizes, output_size)
    f1_no, cm_no, losses_no = train_model(net_no_pre, X_train, y_train,
X_test, y_test)
    print("\nWithout pretraining:")
    print(f"F1-score: {f1_no:.4f}")


    pretrained = pretrain_layers(X_train, hidden_sizes)
    net_pre = ClassificationNet(input_size, hidden_sizes, output_size)
    init_with_pretrain(net_pre, pretrained)
    f1_pre, cm_pre, losses_pre = train_model(net_pre, X_train, y_train,
X_test, y_test)
    print("\nWith Autoencoder pretraining:")
    print(f"F1-score: {f1_pre:.4f}")


    pretrained_rbm = pretrain_rbm_layers(X_train, hidden_sizes)
    net_rbm = ClassificationNet(input_size, hidden_sizes, output_size)
    init_with_pretrain(net_rbm, pretrained_rbm)
    f1_rbm, cm_rbm, losses_rbm = train_model(net_rbm, X_train, y_train,
X_test, y_test)
    print("\nWith RBM pretraining:")
    print(f"F1-score: {f1_rbm:.4f}")


    if SAVE_RESULTS:
        plt.figure(figsize=(10, 5))
        plt.plot(losses_no, label="No Pretrain")
        plt.plot(losses_pre, label="Autoencoder")
        plt.plot(losses_rbm, label="RBM")
```

```
        plt.title(f"Loss Curves — {name}")
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.legend()
        plt.savefig(os.path.join(RESULTS_DIR, f"{name}_loss.png"), dpi=200,
bbox_inches='tight')
        plt.close()

    print(f"\n===== Summary for {name} =====")
    print(f"No pretrain:  {f1_no:.4f}")
    print(f"Autoencoder:  {f1_pre:.4f}")
    print(f"RBM:          {f1_rbm:.4f}")

    return f1_no, f1_pre, f1_rbm


if __name__ == "__main__":
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    summary_path = os.path.join(RESULTS_DIR, "summary.txt")

    f1_no_mh, f1_pre_mh, f1_rbm_mh =
process_dataset("Maternal_Health_Risk", load_maternal_health_data)
    f1_no_rice, f1_pre_rice, f1_rbm_rice =
process_dataset("Rice_Cammeo_Osmancik", load_rice_data)

    print("\n===== Final Comparison =====")
    print(f"Maternal Health — No: {f1_no_mh:.4f}, Autoenc: {f1_pre_mh:.4f},
RBM: {f1_rbm_mh:.4f}")
    print(f"Rice — No: {f1_no_rice:.4f}, Autoenc: {f1_pre_rice:.4f}, RBM:
{f1_rbm_rice:.4f}")

    if SAVE_RESULTS:
        with open(summary_path, "a", encoding="utf-8") as f:
            f.write(f"\n===== Run at {timestamp} =====\n")
            f.write(f"Maternal Health — No: {f1_no_mh:.4f}, Autoenc:
{f1_pre_mh:.4f}, RBM: {f1_rbm_mh:.4f}\n")
            f.write(f"Rice — No: {f1_no_rice:.4f}, Autoenc:
{f1_pre_rice:.4f}, RBM: {f1_rbm_rice:.4f}\n")
            f.write("=" * 40 + "\n")
```

Вывод:





C:\Users\arcio\PycharmProjects\IAD\lab4\.venv\Scripts\python.exe
C:\Users\arcio\PycharmProjects\IAD\lab4\main.py

===== Maternal_Health_Risk Dataset =====

Without pretraining:

F1-score: 0.7261

With Autoencoder pretraining:

F1-score: 0.6964

RBM layer 64 — epoch 1, loss=1.314515

RBM layer 64 — epoch 2, loss=1.122724

RBM layer 64 — epoch 3, loss=1.071383

RBM layer 64 — epoch 4, loss=1.034560

RBM layer 64 — epoch 5, loss=1.032994

RBM layer 64 — epoch 6, loss=1.025710

RBM layer 64 — epoch 7, loss=1.014828

RBM layer 64 — epoch 8, loss=1.024843

RBM layer 64 — epoch 9, loss=1.019328

RBM layer 64 — epoch 10, loss=1.012403

RBM layer 64 — epoch 11, loss=1.021523

RBM layer 64 — epoch 12, loss=1.013909

RBM layer 64 — epoch 13, loss=1.017214

RBM layer 64 — epoch 14, loss=1.006342

RBM layer 64 — epoch 15, loss=1.014154

RBM layer 64 — epoch 16, loss=1.037812

RBM layer 64 — epoch 17, loss=1.011657

RBM layer 64 — epoch 18, loss=1.021466

RBM layer 64 — epoch 19, loss=1.011969

RBM layer 64 — epoch 20, loss=0.998158

RBM layer 64 — epoch 21, loss=0.995420

RBM layer 64 — epoch 22, loss=1.003693

RBM layer 64 — epoch 23, loss=0.986326

RBM layer 64 — epoch 24, loss=1.000995

RBM layer 64 — epoch 25, loss=0.986520

RBM layer 64 — epoch 26, loss=0.992968

RBM layer 64 — epoch 27, loss=0.999132

RBM layer 64 — epoch 28, loss=1.001902

RBM layer 64 — epoch 29, loss=0.981851

RBM layer 64 — epoch 30, loss=0.975300

RBM layer 32 — epoch 1, loss=0.418606

RBM layer 32 — epoch 2, loss=0.405753

RBM layer 32 — epoch 3, loss=0.395743

RBM layer 32 — epoch 4, loss=0.394262

RBM layer 32 — epoch 5, loss=0.391402

RBM layer 32 — epoch 6, loss=0.385999

RBM layer 32 — epoch 7, loss=0.385226

RBM layer 32 — epoch 8, loss=0.389980

RBM layer 32 — epoch 9, loss=0.384697

RBM layer 32 — epoch 10, loss=0.375414

RBM layer 32 — epoch 11, loss=0.372071

RBM layer 32 — epoch 12, loss=0.371366

RBM layer 32 — epoch 13, loss=0.360333

RBM layer 32 — epoch 14, loss=0.361216

RBM layer 32 — epoch 15, loss=0.356062

RBM layer 32 — epoch 16, loss=0.358288

RBM layer 32 — epoch 17, loss=0.351592

RBM layer 32 — epoch 18, loss=0.347181

RBM layer 32 — epoch 19, loss=0.343276

RBM layer 32 — epoch 20, loss=0.340207

RBM layer 32 — epoch 21, loss=0.341483

RBM layer 32 — epoch 22, loss=0.333781

RBM layer 32 — epoch 23, loss=0.331115

RBM layer 32 — epoch 24, loss=0.330754

RBM layer 32 — epoch 25, loss=0.326344

RBM layer 32 — epoch 26, loss=0.324086

RBM layer 32 — epoch 27, loss=0.317248

RBM layer 32 — epoch 28, loss=0.317786

RBM layer 32 — epoch 29, loss=0.313775

RBM layer 32 — epoch 30, loss=0.312552

RBM layer 16 — epoch 1, loss=0.345090

RBM layer 16 — epoch 2, loss=0.291598

RBM layer 16 — epoch 3, loss=0.255983

RBM layer 16 — epoch 4, loss=0.235110

RBM layer 16 — epoch 5, loss=0.227809

RBM layer 16 — epoch 6, loss=0.219997

RBM layer 16 — epoch 7, loss=0.221279

RBM layer 16 — epoch 8, loss=0.221523

RBM layer 16 — epoch 9, loss=0.218353

RBM layer 16 — epoch 10, loss=0.214631

RBM layer 16 — epoch 11, loss=0.215857

RBM layer 16 — epoch 12, loss=0.212388

RBM layer 16 — epoch 13, loss=0.215559

RBM layer 16 — epoch 14, loss=0.217019

RBM layer 16 — epoch 15, loss=0.213845

RBM layer 16 — epoch 16, loss=0.216345

RBM layer 16 — epoch 17, loss=0.212592

RBM layer 16 — epoch 18, loss=0.215326

RBM layer 16 — epoch 19, loss=0.215208

RBM layer 16 — epoch 20, loss=0.214152

RBM layer 16 — epoch 21, loss=0.210177

RBM layer 16 — epoch 22, loss=0.213049

RBM layer 16 — epoch 23, loss=0.212865

RBM layer 16 — epoch 24, loss=0.211043

RBM layer 16 — epoch 25, loss=0.208568

RBM layer 16 — epoch 26, loss=0.208837

RBM layer 16 — epoch 27, loss=0.208381

RBM layer 16 — epoch 28, loss=0.205565

RBM layer 16 — epoch 29, loss=0.209194

RBM layer 16 — epoch 30, loss=0.208790

With RBM pretraining:

F1-score: 0.6118

===== Summary for Maternal_Health_Risk =====

No pretrain: 0.7261

Autoencoder: 0.6964

RBM: 0.611

===== Rice_Cammeo_Osmancik Dataset =====

Without pretraining:

F1-score: 0.9174

Maternal Health — No: 0.7261, Autoenc: 0.6964, RBM: 0.6118
Rice — No: 0.9174, Autoenc: 0.9147, RBM: 0.8910

Maternal Health Risk Dataset:

Вариант обучения	F1-score
Без предобучения	0.7261
С автоэнкодером	0.6964
С RBM-предобучением	0.6118

Наилучший результат показала модель без предобучения. Предобучение с помощью автоэнкодера немного снизило точность (−0.03), но модель всё ещё сохраняла адекватное разделение классов. Предобучение на основе RBM показало заметное ухудшение качества (−0.11). Это может быть связано с тем, что данные относительно просты, и RBM не смог эффективно сформировать скрытые представления — при этом добавленный уровень вероятностной стохастичности мог ухудшить устойчивость весов. Потери на этапе обучения RBM постепенно снижались (что видно по логу обучения), но финальные представления оказались менее информативными для классификатора.

Rice Cammeo vs Osmancik Dataset

Вариант обучения	F1-score
Без предобучения	0.9174
С автоэнкодером	0.9147
С RBM-предобучением	0.8910

Все модели продемонстрировали высокое качество классификации (>0.89).

В отличие от Maternal Health, падение F1-score при использовании RBM составило менее 3%, что указывает на меньшую чувствительность модели к типу предобучения.

Это объясняется тем, что признаки в наборе Rice имеют чёткие границы между классами, и простая полносвязная сеть без предобучения уже достаточно хорошо решает задачу.

Во всех случаях предобучение не дало улучшения, а в некоторых — привело к ухудшению F1-score.
Это может связано с несколькими факторами: относительно небольшим размером выборок, переизбыточностью признаков.

Вывод: научился осуществлять предобучение нейронных сетей с помощью RBM