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

Глухарев Д.Е.

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
5	cardiotocography	классификация	CLASS/NSP

Код программ:

1)

```
import os
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.preprocessing import StandardScaler,
MinMaxScaler
from sklearn.metrics import f1_score, confusion_matrix,
classification_report
import random

csv_path = "CTG.csv"
```

```
def seed_everything(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
seed_everything(42)

device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")

def load_ctg(csv_path):
    df = pd.read_csv(csv_path)
    target_col = None
    for c in df.columns:
        if "NSP" in c or "CLASS" in c or "class" in
c.lower():
            target_col = c
            break
    if target_col is None:
        raise RuntimeError("Target column (NSP/CLASS) not
found in csv")
    df[target_col] = pd.to_numeric(df[target_col],
errors='coerce')
    df = df.dropna(subset=[target_col])
    X =
df.drop(columns=[target_col]).select_dtypes(include=[np.numbe
r]).copy()
    y = df[target_col].astype(int).values - 1
    return X.values, y

class RBM:
    def __init__(self, n_visible, n_hidden, k=1, lr=1e-3,
use_cuda=False):
        self.nv = n_visible
        self.nh = n_hidden
        self.k = k
        self.lr = lr
        self.device = torch.device("cuda" if (use_cuda and
torch.cuda.is_available()) else "cpu")
        W = torch.randn(n_visible, n_hidden) * 0.01
        self.W = W.to(self.device)
        self.v_bias = torch.zeros(n_visible,
device=self.device)
        self.h_bias = torch.zeros(n_hidden,
device=self.device)

    def sample_h(self, v):
        prob = torch.sigmoid(torch.matmul(v, self.W) +
self.h_bias)
        return prob, torch.bernoulli(prob)

    def sample_v(self, h):
        prob = torch.sigmoid(torch.matmul(h, self.W.t()) +
self.v_bias)
```

```
        return prob, torch.bernoulli(prob)

    def contrastive_divergence(self, v0):
        v = v0.to(self.device)
        ph_prob, ph_sample = self.sample_h(v)
        nv = v
        for _ in range(self.k):
            _, h = self.sample_h(nv)
            nv_prob, nv = self.sample_v(h)
        nh_prob, _ = self.sample_h(nv)
        pos_grad = torch.matmul(v.t(), ph_prob)
        neg_grad = torch.matmul(nv.t(), nh_prob)
        batch_size = v.size(0)
        self.W += self.lr * (pos_grad - neg_grad) /
batch_size
        self.v_bias += self.lr * torch.mean(v - nv, dim=0)
        self.h_bias += self.lr * torch.mean(ph_prob -
nh_prob, dim=0)
        loss = torch.mean((v - nv_prob) ** 2).item()
        return loss

    def transform(self, X):
        X_t = torch.tensor(X, dtype=torch.float32,
device=self.device)
        h_prob = torch.sigmoid(torch.matmul(X_t, self.W) +
self.h_bias)
        return h_prob.cpu().numpy()

class AEEncoder(nn.Module):
    def __init__(self, input_dim, hidden_dims):
        super().__init__()
        layers = []
        dims = [input_dim] + hidden_dims
        for i in range(len(hidden_dims)):
            layers.append(nn.Linear(dims[i], dims[i+1]))
            layers.append(nn.ReLU())
        self.encoder = nn.Sequential(*layers)
    def forward(self, x):
        return self.encoder(x)

class AutoencoderFull(nn.Module):
    def __init__(self, input_dim, hidden_dims):
        super().__init__()
        enc_layers = []
        dims = [input_dim] + hidden_dims
        for i in range(len(hidden_dims)):
            enc_layers.append(nn.Linear(dims[i], dims[i+1]))
            enc_layers.append(nn.ReLU())
        dec_layers = []
        for i in range(len(hidden_dims)-1, -1, -1):
            dec_layers.append(nn.Linear(dims[i+1], dims[i]))
            if i != 0:
                dec_layers.append(nn.ReLU())
        self.encoder = nn.Sequential(*enc_layers)
```

```
        self.decoder = nn.Sequential(*dec_layers)
    def forward(self, x):
        z = self.encoder(x)
        x_rec = self.decoder(z)
        return x_rec

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dims, num_classes):
        super().__init__()
        layers = []
        dims = [input_dim] + hidden_dims
        for i in range(len(hidden_dims)):
            layers.append(nn.Linear(dims[i], dims[i+1]))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(dims[-1], num_classes))
        self.net = nn.Sequential(*layers)
    def forward(self,x):
        return self.net(x)

def train_classifier(model, train_loader, val_loader,
epochs=30, lr=1e-3):
    model = model.to(device)
    opt = optim.Adam(model.parameters(), lr=lr)
    crit = nn.CrossEntropyLoss()
    for ep in range(epochs):
        model.train()
        for Xb, yb in train_loader:
            Xb, yb = Xb.to(device), yb.to(device)
            opt.zero_grad()
            out = model(Xb)
            loss = crit(out, yb)
            loss.backward()
            opt.step()
    model.eval()
    ys, preds = [], []
    with torch.no_grad():
        for Xb, yb in val_loader:
            Xb = Xb.to(device)
            out = model(Xb)
            pred = out.argmax(1).cpu().numpy()
            preds.extend(pred)
            ys.extend(yb.numpy())
    return np.array(ys), np.array(preds), model

def run_experiment(csv_path):
    X, y = load_ctg(csv_path)
    mask = ~np.isnan(X).any(axis=1)
    X = X[mask]; y = y[mask]
    num_classes = len(np.unique(y))
    n = len(y)
    idx = np.arange(n)
    np.random.shuffle(idx)
    train_n = int(0.8 * n)
    tr_idx = idx[:train_n]; te_idx = idx[train_n:]
```

```
    X_train, X_test = X[tr_idx], X[te_idx]
    y_train, y_test = y[tr_idx], y[te_idx]
    scaler_clf = StandardScaler().fit(X_train)
    Xtr_clf = scaler_clf.transform(X_train)
    Xte_clf = scaler_clf.transform(X_test)
    scaler_rbm = MinMaxScaler().fit(X_train)
    Xtr_rbm = scaler_rbm.transform(X_train)
    Xte_rbm = scaler_rbm.transform(X_test)
    Xtr_tensor = torch.tensor(Xtr_clf, dtype=torch.float32)
    Xte_tensor = torch.tensor(Xte_clf, dtype=torch.float32)
    ytr_tensor = torch.tensor(y_train, dtype=torch.long)
    yte_tensor = torch.tensor(y_test, dtype=torch.long)
    train_ds = TensorDataset(Xtr_tensor, ytr_tensor)
    test_ds = TensorDataset(Xte_tensor, yte_tensor)
    train_loader = DataLoader(train_ds, batch_size=32,
shuffle=True)
    test_loader = DataLoader(test_ds, batch_size=64)
    input_dim = X.shape[1]
    hidden_dims = [128, 64]
    mlp_hidden = hidden_dims

    print("\n--- Baseline (no pretraining) ---")
    model_base = MLP(input_dim, mlp_hidden, num_classes)
    y_true_base, y_pred_base, model_base =
train_classifier(model_base, train_loader, test_loader,
epochs=40, lr=1e-3)
    f1_base = f1_score(y_true_base, y_pred_base,
average='macro')
    print("Baseline F1 (macro):", f1_base)
    print(confusion_matrix(y_true_base, y_pred_base))

    print("\n--- Autoencoder stacked pretraining ---")
    ae = AutoencoderFull(input_dim, hidden_dims)
    ae = ae.to(device)
    ae_opt = optim.Adam(ae.parameters(), lr=1e-3)
    ae_crit = nn.MSELoss()
    Xtr_ae = torch.tensor(Xtr_clf,
dtype=torch.float32).to(device)
    ae_epochs = 30
    for ep in range(ae_epochs):
        ae.train()
        ae_opt.zero_grad()
        rec = ae(Xtr_ae)
        loss = ae_crit(rec, Xtr_ae)
        loss.backward()
        ae_opt.step()
    model_ae = MLP(input_dim, mlp_hidden, num_classes)
    with torch.no_grad():
        enc_layers = [l for l in ae.encoder if isinstance(l,
nn.Linear)]
        mlp_lin = [l for l in model_ae.net if isinstance(l,
nn.Linear)]
        for i in range(len(enc_layers)):
            mlp_lin[i].weight.data =
```

```
enc_layers[i].weight.data.clone()
            mlp_lin[i].bias.data =
enc_layers[i].bias.data.clone()
    y_true_ae, y_pred_ae, model_ae =
train_classifier(model_ae, train_loader, test_loader,
epochs=40, lr=1e-3)
    f1_ae = f1_score(y_true_ae, y_pred_ae, average='macro')
    print("AE-pretrain F1 (macro):", f1_ae)
    print(confusion_matrix(y_true_ae, y_pred_ae))

    print("\n--- RBM stacked pretraining ---")
    rbm1 = RBM(n_visible=input_dim, n_hidden=hidden_dims[0],
k=1, lr=0.01, use_cuda=False)
    epochs_rbm = 20
    batch_size = 64
    Xrbm = Xtr_rbm
    for ep in range(epochs_rbm):
        perm = np.random.permutation(len(Xrbm))
        losses = []
        for i in range(0, len(Xrbm), batch_size):
            batch = torch.tensor(Xrbm[perm[i:i+batch_size]],
dtype=torch.float32, device=rbm1.device)
            loss = rbm1.contrastive_divergence(batch)
            losses.append(loss)
        if (ep+1)%5==0:
            print(f"RBM1 epoch {ep+1},
recon_loss={np.mean(losses):.6f}")
    H1 = rbm1.transform(Xrbm)
    rbm2 = RBM(n_visible=hidden_dims[0],
n_hidden=hidden_dims[1], k=1, lr=0.01, use_cuda=False)
    for ep in range(epochs_rbm):
        perm = np.random.permutation(len(H1))
        losses = []
        for i in range(0, len(H1), batch_size):
            batch = torch.tensor(H1[perm[i:i+batch_size]],
dtype=torch.float32, device=rbm2.device)
            loss = rbm2.contrastive_divergence(batch)
            losses.append(loss)
        if (ep+1)%5==0:
            print(f"RBM2 epoch {ep+1},
recon_loss={np.mean(losses):.6f}")
    model_rbm = MLP(input_dim, mlp_hidden, num_classes)
    with torch.no_grad():
        model_rbm.net[0].weight.data = rbm1.W.t().clone()
        model_rbm.net[0].bias.data = rbm1.h_bias.clone()
        model_rbm.net[2].weight.data = rbm2.W.t().clone()
        model_rbm.net[2].bias.data = rbm2.h_bias.clone()
    y_true_rbm, y_pred_rbm, model_rbm =
train_classifier(model_rbm, train_loader, test_loader,
epochs=40, lr=1e-3)
    f1_rbm = f1_score(y_true_rbm, y_pred_rbm,
average='macro')
    print("RBM-pretrain F1 (macro):", f1_rbm)
    print(confusion_matrix(y_true_rbm, y_pred_rbm))
```

```
    print("\n=== SUMMARY (F1 macro) ===")
    print(f"Baseline: {f1_base:.4f}")
    print(f"AE pretrain: {f1_ae:.4f}")
    print(f"RBM pretrain: {f1_rbm:.4f}")
    print("\nBaseline report:\n",
classification_report(y_true_base, y_pred_base))
    print("\nAE report:\n", classification_report(y_true_ae,
y_pred_ae))
    print("\nRBM report:\n",
classification_report(y_true_rbm, y_pred_rbm))

if __name__ == "__main__":
    csv_path = "CTG.csv"
    if not os.path.exists(csv_path):
        raise FileNotFoundError(f"{csv_path} not found in
working directory.")
    run_experiment(csv_path)
```

2)

```
import os
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.preprocessing import StandardScaler,
MinMaxScaler
from sklearn.metrics import f1_score, confusion_matrix,
classification_report
import random

def seed_everything(seed=42):
    random.seed(seed); np.random.seed(seed);
torch.manual_seed(seed)
seed_everything(42)

device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")

class RBM:
    def __init__(self, n_visible, n_hidden, k=1, lr=1e-3,
use_cuda=False):
        self.nv = n_visible; self.nh = n_hidden; self.k = k;
self.lr = lr
        self.device = torch.device("cuda" if (use_cuda and
torch.cuda.is_available()) else "cpu")
        W = torch.randn(n_visible, n_hidden) * 0.01
        self.W = W.to(self.device)
```

```
        self.v_bias = torch.zeros(n_visible,
device=self.device)
        self.h_bias = torch.zeros(n_hidden,
device=self.device)
    def sample_h(self, v):
        prob = torch.sigmoid(torch.matmul(v, self.W) +
self.h_bias)
        return prob, torch.bernoulli(prob)
    def sample_v(self, h):
        prob = torch.sigmoid(torch.matmul(h, self.W.t()) +
self.v_bias)
        return prob, torch.bernoulli(prob)
    def contrastive_divergence(self, v0):
        v = v0.to(self.device)
        ph_prob, ph_sample = self.sample_h(v)
        nv = v
        for _ in range(self.k):
            _, h = self.sample_h(nv)
            nv_prob, nv = self.sample_v(h)
        nh_prob, _ = self.sample_h(nv)
        pos_grad = torch.matmul(v.t(), ph_prob)
        neg_grad = torch.matmul(nv.t(), nh_prob)
        batch_size = v.size(0)
        self.W += self.lr * (pos_grad - neg_grad) /
batch_size
        self.v_bias += self.lr * torch.mean(v - nv, dim=0)
        self.h_bias += self.lr * torch.mean(ph_prob -
nh_prob, dim=0)
        loss = torch.mean((v - nv_prob) ** 2).item()
        return loss
    def transform(self, X):
        X_t = torch.tensor(X, dtype=torch.float32,
device=self.device)
        h_prob = torch.sigmoid(torch.matmul(X_t, self.W) +
self.h_bias)
        return h_prob.cpu().numpy()

class AutoencoderFull(nn.Module):
    def __init__(self, input_dim, hidden_dims):
        super().__init__()
        enc = []
        dims = [input_dim] + hidden_dims
        for i in range(len(hidden_dims)):
            enc.append(nn.Linear(dims[i], dims[i+1]));
enc.append(nn.ReLU())
        dec = []
        for i in range(len(hidden_dims)-1, -1, -1):
            dec.append(nn.Linear(dims[i+1], dims[i]));
            if i!=0: dec.append(nn.ReLU())
        self.encoder = nn.Sequential(*enc)
        self.decoder = nn.Sequential(*dec)
    def forward(self,x):
        z = self.encoder(x); return self.decoder(z)
```

```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dims, num_classes):
        super().__init__()
        layers = []
        dims = [input_dim] + hidden_dims
        for i in range(len(hidden_dims)):
            layers.append(nn.Linear(dims[i], dims[i+1]));
layers.append(nn.ReLU())
        layers.append(nn.Linear(dims[-1], num_classes))
        self.net = nn.Sequential(*layers)
    def forward(self,x): return self.net(x)

def load_wholesale(csv_path="wholesale.csv"):
    df = pd.read_csv(csv_path)
    if "Channel" in df.columns:
        target = "Channel"
    elif "Region" in df.columns:
        target = "Region"
    else:
        df["TargetBin"] =
(df.select_dtypes(include=[np.number]).sum(axis=1) >
df.select_dtypes(include=[np.number]).sum(axis=1).median()).a
stype(int)
        target = "TargetBin"
    df = df.dropna(subset=[target])
    X =
df.drop(columns=[target]).select_dtypes(include=[np.number]).
values
    y = df[target].astype(int).values - 1
    return X, y

def train_classifier(model, Xtr, ytr, Xte, yte, epochs=40,
batch_size=32, lr=1e-3):
    model = model.to(device)
    opt = optim.Adam(model.parameters(), lr=lr)
    crit = nn.CrossEntropyLoss()
    ds_tr =
TensorDataset(torch.tensor(Xtr,dtype=torch.float32),
torch.tensor(ytr,dtype=torch.long))
    dl_tr = DataLoader(ds_tr, batch_size=batch_size,
shuffle=True)
    for ep in range(epochs):
        model.train()
        for Xb, yb in dl_tr:
            Xb, yb = Xb.to(device), yb.to(device)
            opt.zero_grad()
            out = model(Xb)
            loss = crit(out, yb)
            loss.backward(); opt.step()
    model.eval()
    preds=[]; truths=[]
    ds_te =
TensorDataset(torch.tensor(Xte,dtype=torch.float32),
torch.tensor(yte,dtype=torch.long))
```

```
    dl_te = DataLoader(ds_te, batch_size=128)
    with torch.no_grad():
        for Xb,yb in dl_te:
            Xb = Xb.to(device)
            out = model(Xb)
            preds.extend(out.argmax(1).cpu().numpy())
            truths.extend(yb.numpy())
    return np.array(truths), np.array(preds), model

def run(csv_path="wholesale.csv"):
    X,y = load_wholesale(csv_path)
    mask = ~np.isnan(X).any(axis=1)
    X = X[mask]; y = y[mask]
    num_classes = len(np.unique(y))
    n = len(y)
    idx = np.arange(n); np.random.shuffle(idx)
    tr = int(0.8*n)
    train_idx, test_idx = idx[:tr], idx[tr:]
    Xtr, Xte = X[train_idx], X[test_idx]
    ytr, yte = y[train_idx], y[test_idx]
    print("Classes:", np.unique(y, return_counts=True))
    scaler_clf = StandardScaler().fit(Xtr)
    Xtr_clf = scaler_clf.transform(Xtr); Xte_clf =
scaler_clf.transform(Xte)
    scaler_rbm = MinMaxScaler().fit(Xtr)
    Xtr_rbm = scaler_rbm.transform(Xtr); Xte_rbm =
scaler_rbm.transform(Xte)
    input_dim = Xtr.shape[1]
    hidden_dims = [128, 64]
    mlp_hidden = hidden_dims
    print("\n--- Baseline ---")
    model_base = MLP(input_dim, mlp_hidden, num_classes)
    y_true_b, y_pred_b, model_base =
train_classifier(model_base, Xtr_clf, ytr, Xte_clf, yte,
epochs=40, lr=1e-3)
    f1_b = f1_score(y_true_b, y_pred_b, average='macro')
    print("Baseline F1:", f1_b);
print(confusion_matrix(y_true_b, y_pred_b))
    print("\n--- AE pretrain ---")
    ae = AutoencoderFull(input_dim, hidden_dims).to(device)
    ae_opt = optim.Adam(ae.parameters(), lr=1e-3); ae_crit =
nn.MSELoss()
    Xtr_tensor = torch.tensor(Xtr_clf,
dtype=torch.float32).to(device)
    for ep in range(30):
        ae.train()
        ae_opt.zero_grad()
        rec = ae(Xtr_tensor)
        loss = ae_crit(rec, Xtr_tensor)
        loss.backward(); ae_opt.step()
    model_ae = MLP(input_dim, mlp_hidden, num_classes)
    with torch.no_grad():
        enc_layers = [l for l in ae.encoder if isinstance(l,
nn.Linear)]
```

```
        mlp_lin = [l for l in model_ae.net if isinstance(l,
nn.Linear)]
        for i in range(len(enc_layers)):
            mlp_lin[i].weight.data =
enc_layers[i].weight.data.clone()
            mlp_lin[i].bias.data =
enc_layers[i].bias.data.clone()
    y_true_ae, y_pred_ae, model_ae =
train_classifier(model_ae, Xtr_clf, ytr, Xte_clf, yte,
epochs=40)
    f1_ae = f1_score(y_true_ae, y_pred_ae, average='macro')
    print("AE F1:", f1_ae); print(confusion_matrix(y_true_ae,
y_pred_ae))
    print("\n--- RBM pretrain ---")
    rbm1 = RBM(n_visible=input_dim, n_hidden=hidden_dims[0],
k=1, lr=0.01)
    epochs_rbm = 20; batch_size = 64
    for ep in range(epochs_rbm):
        perm = np.random.permutation(len(Xtr_rbm))
        losses=[]
        for i in range(0,len(Xtr_rbm),batch_size):
            batch =
torch.tensor(Xtr_rbm[perm[i:i+batch_size]],
dtype=torch.float32, device=rbm1.device)
            l = rbm1.contrastive_divergence(batch);
losses.append(l)
        if (ep+1)%5==0:
            print(f"RBM1 ep {ep+1}, loss
{np.mean(losses):.6f}")
    H1 = rbm1.transform(Xtr_rbm)
    rbm2 = RBM(n_visible=hidden_dims[0],
n_hidden=hidden_dims[1], k=1, lr=0.01)
    for ep in range(epochs_rbm):
        perm = np.random.permutation(len(H1))
        losses=[]
        for i in range(0,len(H1),batch_size):
            batch = torch.tensor(H1[perm[i:i+batch_size]],
dtype=torch.float32, device=rbm2.device)
            l = rbm2.contrastive_divergence(batch);
losses.append(l)
        if (ep+1)%5==0:
            print(f"RBM2 ep {ep+1}, loss
{np.mean(losses):.6f}")
    model_rbm = MLP(input_dim, mlp_hidden, num_classes)
    with torch.no_grad():
        model_rbm.net[0].weight.data = rbm1.W.t().clone()
        model_rbm.net[0].bias.data = rbm1.h_bias.clone()
        model_rbm.net[2].weight.data = rbm2.W.t().clone()
        model_rbm.net[2].bias.data = rbm2.h_bias.clone()
    y_true_r, y_pred_r, model_rbm =
train_classifier(model_rbm, Xtr_clf, ytr, Xte_clf, yte,
epochs=40)
    f1_r = f1_score(y_true_r, y_pred_r, average='macro')
    print("RBM F1:", f1_r); print(confusion_matrix(y_true_r,
```

```
y_pred_r))
    print("\n=== SUMMARY ===")
    print(f"Baseline F1: {f1_b:.4f}")
    print(f"AE F1:       {f1_ae:.4f}")
    print(f"RBM F1:      {f1_r:.4f}")
    print("\nBaseline report:\n",
classification_report(y_true_b, y_pred_b))
    print("\nAE report:\n", classification_report(y_true_ae,
y_pred_ae))
    print("\nRBM report:\n", classification_report(y_true_r,
y_pred_r))

if __name__ == "__main__":
    if not os.path.exists("wholesale.csv"):
        raise FileNotFoundError("wholesale.csv not found in
working dir")
    run("wholesale.csv")
```

Вывод программы:

1)

C:\Users\Asus\AppData\Local\Programs\Python\Python39\python.exe "C:\Users\Asus\PycharmProjects\ИАД\ЛАБА 4 1.py"

```
--- Baseline (no pretraining) ---
Baseline F1 (macro): 1.0
[[ 64   0   0   0   0   0   0   0   0   0]
 [  0 123   0   0   0   0   0   0   0   0]
 [  0   0  14   0   0   0   0   0   0   0]
 [  0   0   0   9   0   0   0   0   0   0]
 [  0   0   0   0  10   0   0   0   0   0]
 [  0   0   0   0   0  74   0   0   0   0]
 [  0   0   0   0   0   0  61   0   0   0]
 [  0   0   0   0   0   0   0  24   0   0]
 [  0   0   0   0   0   0   0   0  14   0]
 [  0   0   0   0   0   0   0   0   0  33]]
```

```
--- Autoencoder stacked pretraining ---
AE-pretrain F1 (macro): 1.0
[[ 64   0   0   0   0   0   0   0   0   0]
 [  0 123   0   0   0   0   0   0   0   0]
 [  0   0  14   0   0   0   0   0   0   0]
 [  0   0   0   9   0   0   0   0   0   0]
 [  0   0   0   0  10   0   0   0   0   0]
 [  0   0   0   0   0  74   0   0   0   0]
 [  0   0   0   0   0   0  61   0   0   0]
 [  0   0   0   0   0   0   0  24   0   0]
 [  0   0   0   0   0   0   0   0  14   0]
 [  0   0   0   0   0   0   0   0   0  33]]

--- RBM stacked pretraining ---
RBM1 epoch 5, recon_loss=0.048008
RBM1 epoch 10, recon_loss=0.047800
RBM1 epoch 15, recon_loss=0.047841
RBM1 epoch 20, recon_loss=0.047810
RBM2 epoch 5, recon_loss=0.000243
RBM2 epoch 10, recon_loss=0.000230
RBM2 epoch 15, recon_loss=0.000213
RBM2 epoch 20, recon_loss=0.000204
RBM-pretrain F1 (macro): 1.0
[[ 64   0   0   0   0   0   0   0   0   0]
 [  0 123   0   0   0   0   0   0   0   0]
 [  0   0  14   0   0   0   0   0   0   0]
 [  0   0   0   9   0   0   0   0   0   0]
```

```
 [  0   0   0   0  10   0   0   0   0   0]
 [  0   0   0   0   0  74   0   0   0   0]
 [  0   0   0   0   0   0  61   0   0   0]
 [  0   0   0   0   0   0   0  24   0   0]
 [  0   0   0   0   0   0   0   0  14   0]
 [  0   0   0   0   0   0   0   0   0  33]]

=== SUMMARY (F1 macro) ===
Baseline: 1.0000
AE pretrain: 1.0000
RBM pretrain: 1.0000

Baseline report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        64
           1       1.00      1.00      1.00       123
           2       1.00      1.00      1.00        14
           3       1.00      1.00      1.00         9
           4       1.00      1.00      1.00        10
           5       1.00      1.00      1.00        74
           6       1.00      1.00      1.00        61
           7       1.00      1.00      1.00        24
           8       1.00      1.00      1.00        14
           9       1.00      1.00      1.00        33

    accuracy                           1.00       426
```

macro avg	1.00	1.00	1.00	426
weighted avg	1.00	1.00	1.00	426

AE report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	64
1	1.00	1.00	1.00	123
2	1.00	1.00	1.00	14
3	1.00	1.00	1.00	9
4	1.00	1.00	1.00	10
5	1.00	1.00	1.00	74
6	1.00	1.00	1.00	61
7	1.00	1.00	1.00	24
8	1.00	1.00	1.00	14
9	1.00	1.00	1.00	33
accuracy			1.00	426
macro avg	1.00	1.00	1.00	426
weighted avg	1.00	1.00	1.00	426

RBM report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	64

```
           1       1.00      1.00      1.00       123
           2       1.00      1.00      1.00        14
           3       1.00      1.00      1.00         9
           4       1.00      1.00      1.00        10
           5       1.00      1.00      1.00        74
           6       1.00      1.00      1.00        61
           7       1.00      1.00      1.00        24
           8       1.00      1.00      1.00        14
           9       1.00      1.00      1.00        33

    accuracy                           1.00       426
   macro avg       1.00      1.00      1.00       426
weighted avg       1.00      1.00      1.00       426


Process finished with exit code 0
```

2)

```
C:\Users\Asus\AppData\Local\Programs\Python\Python39\pyt
hon.exe "C:\Users\Asus\PycharmProjects\ИАД\ЛАБА 4 2.py"
Classes: (array([0, 1]), array([298, 142]))

--- Baseline ---
Baseline F1: 0.9485680888369374
[[57  3]
 [ 1 27]]
```

```
--- AE pretrain ---
AE F1: 0.9338842975206612
[[58  2]
 [ 3 25]]

--- RBM pretrain ---
RBM1 ep 5, loss 0.038065
RBM1 ep 10, loss 0.031750
RBM1 ep 15, loss 0.030350
RBM1 ep 20, loss 0.030833
RBM2 ep 5, loss 0.000203
RBM2 ep 10, loss 0.000190
RBM2 ep 15, loss 0.000230
RBM2 ep 20, loss 0.000208
RBM F1: 0.9338842975206612
[[58  2]
 [ 3 25]]

=== SUMMARY ===
Baseline F1: 0.9486
AE F1:       0.9339
RBM F1:      0.9339

Baseline report:
              precision    recall  f1-score   support

           0       0.98      0.95      0.97        60
```

	precision	recall	f1-score	support
1	0.90	0.96	0.93	28
accuracy			0.95	88
macro avg	0.94	0.96	0.95	88
weighted avg	0.96	0.95	0.95	88

AE report:

	precision	recall	f1-score	support
0	0.95	0.97	0.96	60
1	0.93	0.89	0.91	28
accuracy			0.94	88
macro avg	0.94	0.93	0.93	88
weighted avg	0.94	0.94	0.94	88

RBM report:

	precision	recall	f1-score	support
0	0.95	0.97	0.96	60
1	0.93	0.89	0.91	28
accuracy			0.94	88
macro avg	0.94	0.93	0.93	88
weighted avg	0.94	0.94	0.94	88

```
Process finished with exit code 0
```

Вывод: научился осуществлять предобучение нейронных сетей с помощью RBM