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

Бусень А.Д.

Проверила:

Андренко К.В.

Брест 2025

Цель работы: научиться осуществлять предобучение нейронных сетей с помощью RBM.
Вариант 1

№	Выборка	Тип задачи	Целевая переменная
1	https://archive.ics.uci.edu/dataset/27/credit+approval	классификация	+/-

Код программы:

```
import os
import numpy as np
import random
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix, classification_report, f1_score, accuracy_score,
precision_score, recall_score, roc_auc_score
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

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

DATAFILE = "crx.data"  # положите crx.data в ту же папку или разрешите скачивание

def load_crx(path=DATAFILE):
    if not os.path.exists(path):
        try:
            url = "https://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/crx.data"
            df = pd.read_csv(url, header=None, na_values='?')
            df.to_csv(path, index=False, header=False)
        except Exception as e:
            raise RuntimeError("Не удалось скачать crx.data автоматически. Поместите crx.data в папку и
запустите снова.")
    df = pd.read_csv(path, header=None, na_values='?')
    ncols = df.shape[1]
    df.columns = [f"A{i+1}" for i in range(ncols)]
    return df

df = load_crx()
print("Данные загружены, shape:", df.shape)
def auto_detect_cols(df):
    num_cols, cat_cols = [], []
    for c in df.columns[:-1]:
        # попробуем привести к числу для части значений
        sample = df[c].dropna().astype(str).head(50)
```

```
        num_count = sum(1 for v in sample if v.replace('.','',1).lstrip('-').isdigit())
        if len(sample)>0 and num_count/len(sample) > 0.6:
            num_cols.append(c)
        else:
            cat_cols.append(c)
    return num_cols, cat_cols

num_cols, cat_cols = auto_detect_cols(df)
print("Числовые:", num_cols)
print("Категориальные:", cat_cols)

X = df.drop(columns=[df.columns[-1]])
y = df[df.columns[-1]].map({'+':1, '-':0}).values

try:
    onehot = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
except TypeError:
    onehot = OneHotEncoder(handle_unknown='ignore', sparse=False)

num_transformer = Pipeline([('imputer', SimpleImputer(strategy='median')), ('scaler', StandardScaler())])
cat_transformer = Pipeline([('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
('onehot', onehot)])

preprocessor = ColumnTransformer([('num', num_transformer, num_cols), ('cat', cat_transformer,
cat_cols)])
X_proc = preprocessor.fit_transform(X)

feature_names = []
if num_cols:
    feature_names += num_cols
if cat_cols:
    cat_names =
preprocessor.named_transformers_['cat'].named_steps['onehot'].get_feature_names_out(cat_cols)
    feature_names += list(cat_names)
print("Итоговое число признаков:", X_proc.shape[1])

X_train, X_test, y_train, y_test = train_test_split(X_proc, y, test_size=0.3, random_state=RND, stratify=y)

def to_loader(X, y, batch_size=32, shuffle=True):
    Xt = torch.tensor(X, dtype=torch.float32)
    yt = torch.tensor(y, dtype=torch.float32).unsqueeze(1)
    ds = TensorDataset(Xt, yt)
    return DataLoader(ds, batch_size=batch_size, shuffle=shuffle)

batch_size = 32
train_loader = to_loader(X_train, y_train, batch_size)
test_loader = to_loader(X_test, y_test, batch_size, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
```

```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dims=[128,64,32,16], dropout=0.2):
        super().__init__()
        layers = []
        prev = input_dim
        for h in hidden_dims:
            layers.append(nn.Linear(prev, h))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout))
            prev = h
        layers.append(nn.Linear(prev, 1))
        layers.append(nn.Sigmoid())
        self.net = nn.Sequential(*layers)
    def forward(self, x):
        return self.net(x)

input_dim = X_proc.shape[1]
hidden_dims = [128,64,32,16]  # 4 слоя (>3)
print("MLP input_dim:", input_dim, "hidden_dims:", hidden_dims)

def train_epoch(model, loader, opt, criterion, device):
    model.train()
    tot = 0.0
    for xb, yb in loader:
        xb, yb = xb.to(device), yb.to(device)
        opt.zero_grad()
        out = model(xb)
        loss = criterion(out, yb)
        loss.backward()
        opt.step()
        tot += loss.item() * xb.size(0)
    return tot / len(loader.dataset)

def eval_model(model, loader, device):
    model.eval()
    ys, preds, probs = [], [], []
    with torch.no_grad():
        for xb, yb in loader:
            xb = xb.to(device)
            out = model(xb).cpu().numpy()
            ys.append(yb.numpy())
            probs.append(out)
            preds.append((out>=0.5).astype(int))
    ys = np.vstack(ys).ravel()
    probs = np.vstack(probs).ravel()
    preds = np.vstack(preds).ravel()
    return ys, preds, probs

def print_metrics(name, ytrue, ypred, probs=None):
    print(f"--- {name} ---")
    print("Accuracy:", accuracy_score(ytrue, ypred))
```

```
    print("Precision:", precision_score(ytrue, ypred))
    print("Recall:", recall_score(ytrue, ypred))
    print("F1:", f1_score(ytrue, ypred))
    if probs is not None:
        try:
            print("AUC:", roc_auc_score(ytrue, probs))
        except:
            pass
    print(classification_report(ytrue, ypred, digits=4))
    print("Confusion matrix:\n", confusion_matrix(ytrue, ypred))

def train_from_scratch():
    model = MLP(input_dim, hidden_dims).to(device)
    crit = nn.BCELoss()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    epochs = 100
    for ep in range(1, epochs+1):
        loss = train_epoch(model, train_loader, opt, crit, device)
        if ep==1 or ep%20==0:
            ytrue, ypred, probs = eval_model(model, test_loader, device)
            f1 = f1_score(ytrue, ypred)
            print(f"[Scratch] epoch {ep}/{epochs} loss={loss:.4f} test_f1={f1:.4f}")
    ytrue, ypred, probs = eval_model(model, test_loader, device)
    return model, ytrue, ypred, probs

class SimpleAE(nn.Module):
    def __init__(self, enc_dims):
        super().__init__()
        layers_e = []
        prev = enc_dims[0]
        for h in enc_dims[1:]:
            layers_e.append(nn.Linear(prev, h)); layers_e.append(nn.ReLU()); prev = h
        self.encoder = nn.Sequential(*layers_e)
        layers_d = []
        rev = list(reversed(enc_dims))
        prev = rev[0]
        for h in rev[1:]:
            layers_d.append(nn.Linear(prev, h))
            if h != enc_dims[0]:
                layers_d.append(nn.ReLU())
            prev = h
        self.decoder = nn.Sequential(*layers_d)
    def forward(self, x):
        z = self.encoder(x)
        xrec = self.decoder(z)
        return xrec

def ae_layerwise_pretrain(X_train_np, hidden_dims, ae_epochs=50, batch=64):
    device_local = device
    encoders = []
    data = torch.tensor(X_train_np, dtype=torch.float32).to(device_local)
```

```python
    current_input = data
    for i, h in enumerate(hidden_dims):
        in_dim = current_input.shape[1]
        enc_dims = [in_dim, h]
        ae = SimpleAE(enc_dims).to(device_local)
        opt = torch.optim.Adam(ae.parameters(), lr=1e-3)
        loss_fn = nn.MSELoss()
        loader = DataLoader(TensorDataset(current_input), batch_size=batch, shuffle=True)
        print(f"AE: training layer {i+1}/{len(hidden_dims)} in_dim={in_dim} out={h}")
        for ep in range(1, ae_epochs+1):
            tot=0.0
            for (xb,) in loader:
                opt.zero_grad()
                xb = xb.to(device_local)
                xr = ae(xb)
                loss = loss_fn(xr, xb)
                loss.backward()
                opt.step()
                tot += loss.item()*xb.size(0)
            if ep==1 or ep%20==0:
                print(f" AE layer {i+1} ep {ep}/{ae_epochs} loss {tot/len(current_input):.6f}")
        encoders.append(ae.encoder)
        with torch.no_grad():
            current_input = ae.encoder(current_input).detach()
    return encoders

def build_model_from_encoders(encoders):
    model = MLP(input_dim, hidden_dims).to(device)
    mlp_linears = [m for m in model.net if isinstance(m, nn.Linear)]
    enc_linears = []

    for enc in encoders:
        if isinstance(enc, nn.Sequential):
            for sub in enc:
                if isinstance(sub, nn.Linear):
                    enc_linears.append(sub)
        elif isinstance(enc, nn.Module):
            lin = nn.Linear(enc.W.shape[1], enc.W.shape[0])
            lin.weight.data = enc.W.data.clone()
            lin.bias.data = enc.h_bias.data.clone()
            enc_linears.append(lin)

    n_copy = min(len(enc_linears), len(mlp_linears))
    for i in range(n_copy):
        if mlp_linears[i].weight.data.shape == enc_linears[i].weight.data.shape:
            mlp_linears[i].weight.data.copy_(enc_linears[i].weight.data)
            mlp_linears[i].bias.data.copy_(enc_linears[i].bias.data)
    return model

class RBM(nn.Module):
    def __init__(self, n_vis, n_hid):
```

```
        super().__init__()
        self.W = nn.Parameter(torch.randn(n_hid, n_vis) * 0.01)  # weight matrix (hidden x visible)
        self.v_bias = nn.Parameter(torch.zeros(n_vis))
        self.h_bias = nn.Parameter(torch.zeros(n_hid))

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

    def sample_v(self, h):
        prob_v = torch.sigmoid(torch.matmul(h, self.W) + self.v_bias)
        return prob_v, torch.bernoulli(prob_v)

    def gibbs_hvh(self, h0):
        v_prob, v_sample = self.sample_v(h0)
        h_prob, h_sample = self.sample_h(v_sample)
        return v_prob, v_sample, h_prob, h_sample

    def forward(self, v):
        prob_h = torch.sigmoid(torch.matmul(v, self.W.t()) + self.h_bias)
        return prob_h

def train_rbm(rbm, data_tensor, epochs=50, batch=64, k=1, lr=1e-3):
    opt = torch.optim.SGD(rbm.parameters(), lr=lr)
    loader = DataLoader(TensorDataset(data_tensor), batch_size=batch, shuffle=True)
    loss_fn = nn.MSELoss()
    for ep in range(1, epochs+1):
        tot = 0.0
        for (vb,) in loader:
            vb = vb[0].to(device) if isinstance(vb, tuple) else vb.to(device)
            v0 = vb
            # positive phase
            ph0 = torch.sigmoid(torch.matmul(v0, rbm.W.t()) + rbm.h_bias)
            # Gibbs sampling k steps (CD-k)
            vk = v0
            for _ in range(k):
                hk_prob = torch.sigmoid(torch.matmul(vk, rbm.W.t()) + rbm.h_bias)
                hk = torch.bernoulli(hk_prob)
                vk_prob = torch.sigmoid(torch.matmul(hk, rbm.W) + rbm.v_bias)
                vk = torch.bernoulli(vk_prob)
            phk = torch.sigmoid(torch.matmul(vk, rbm.W.t()) + rbm.h_bias)
            # weight update (CD)
            dW = torch.matmul(ph0.t(), v0) - torch.matmul(phk.t(), vk)
            # but shapes: ph0 (batch, nh), v0 (batch, nv) — so dW should be (nh, nv)
            # Using manual grad-less update:
            grad_W = - (dW / v0.size(0))
            # simple SGD step:
            with torch.no_grad():
                rbm.W += lr * (grad_W)
                rbm.h_bias += lr * (ph0.mean(0) - phk.mean(0))
                rbm.v_bias += lr * (v0 - vk).mean(0)
```

```python
            # optional reconstruction loss for monitoring
            recon = vk_prob
            tot += loss_fn(recon, v0).item() * v0.size(0)
        if ep==1 or ep%20==0:
            print(f" RBM epoch {ep}/{epochs} recon_loss {tot/len(data_tensor):.6f}")
    return rbm

def rbm_stack_pretrain(X_train_np, hidden_dims, rbm_epochs=50, batch=64, k=1):
    encoders = []
    current = torch.tensor(X_train_np, dtype=torch.float32).to(device)
    for i, h in enumerate(hidden_dims):
        n_vis = current.shape[1]
        n_hid = h
        print(f"RBM: training layer {i+1}/{len(hidden_dims)} vis={n_vis} hid={n_hid}")
        rbm = RBM(n_vis, n_hid).to(device)
        rbm = train_rbm(rbm, current, epochs=rbm_epochs, batch=batch, k=k, lr=1e-3)
        # after training, define encoder: sigmoid(v W^T + h_bias)
        class RBMEncoder(nn.Module):
            def __init__(self, W, h_bias):
                super().__init__()
                self.W = W
                self.h_bias = h_bias
            def forward(self, v):
                return torch.sigmoid(torch.matmul(v, self.W.t()) + self.h_bias)
        enc = RBMEncoder(rbm.W.detach().clone(), rbm.h_bias.detach().clone())
        encoders.append(enc)
        with torch.no_grad():
            current = enc(current).detach()
    return encoders

print("\n\nTraining from scratch...")
model_scratch, y_true_s, y_pred_s, probs_s = train_from_scratch()
print_metrics("Result (scratch)", y_true_s, y_pred_s, probs_s)

print("\n\nAE layerwise pretraining...")
ae_epochs = 50
ae_encoders = ae_layerwise_pretrain(X_train, hidden_dims, ae_epochs=ae_epochs, batch=32)
model_ae = build_model_from_encoders(ae_encoders).to(device)
print("Finetuning AE-initialized model...")
crit = nn.BCELoss()
opt = torch.optim.Adam(model_ae.parameters(), lr=1e-4)
finetune_epochs = 100
for ep in range(1, finetune_epochs+1):
    loss = train_epoch(model_ae, train_loader, opt, crit, device)
    if ep==1 or ep%20==0:
        ytrue, ypred, probs = eval_model(model_ae, test_loader, device)
        print(f"[AE-finetune] ep {ep}/{finetune_epochs} loss {loss:.4f} test_f1 {f1_score(ytrue, ypred):.4f}")
ytrue_ae, ypred_ae, probs_ae = eval_model(model_ae, test_loader, device)
print_metrics("Result (AE pretraining)", ytrue_ae, ypred_ae, probs_ae)

print("\n\nRBM stack pretraining...")
```

```
rbm_epochs = 50
rbm_encoders = rbm_stack_pretrain(X_train, hidden_dims, rbm_epochs=rbm_epochs, batch=32, k=1)
model_rbm = build_model_from_encoders(rbm_encoders).to(device)
print("Finetuning RBM-initialized model...")
opt = torch.optim.Adam(model_rbm.parameters(), lr=1e-4)
for ep in range(1, finetune_epochs+1):
    loss = train_epoch(model_rbm, train_loader, opt, crit, device)
    if ep==1 or ep%20==0:
        ytrue, ypred, probs = eval_model(model_rbm, test_loader, device)
        print(f"[RBM-finetune] ep {ep}/{finetune_epochs} loss {loss:.4f} test_f1 {f1_score(ytrue,
ypred):.4f}")
ytrue_rbm, ypred_rbm, probs_rbm = eval_model(model_rbm, test_loader, device)
print_metrics("Result (RBM pretraining)", ytrue_rbm, ypred_rbm, probs_rbm)

import pandas as pd
rows = []
for name, yt, yp, pr in [
    ("scratch", y_true_s, y_pred_s, probs_s),
    ("ae", ytrue_ae, ypred_ae, probs_ae),
    ("rbm", ytrue_rbm, ypred_rbm, probs_rbm)
]:
    rows.append({
        "method": name,
        "accuracy": accuracy_score(yt, yp),
        "precision": precision_score(yt, yp),
        "recall": recall_score(yt, yp),
        "f1": f1_score(yt, yp),
        "auc": roc_auc_score(yt, pr) if pr is not None else None
    })
df_results = pd.DataFrame(rows).set_index('method')
print("\nSummary:\n", df_results)

fig, axes = plt.subplots(1,3, figsize=(15,4))
for ax, (name, yt, yp) in zip(axes, [("scratch", y_true_s, y_pred_s), ("AE", ytrue_ae, ypred_ae), ("RBM",
ytrue_rbm, ypred_rbm)]):
    cm = confusion_matrix(yt, yp)
    ax.imshow(cm, interpolation='nearest')
    ax.set_title(name)
    ax.set_xlabel('pred')
    ax.set_ylabel('true')
    for (i,j), val in np.ndenumerate(cm):
        ax.text(j, i, str(val), ha='center', va='center', color='white')
plt.show()
```

Результат работы программы:

Training from scratch...

[Scratch] epoch 1/100 loss=0.6828 test_f1=0.0000

[Scratch] epoch 20/100 loss=0.2741 test_f1=0.8571

[Scratch] epoch 40/100 loss=0.1632 test_f1=0.8462

[Scratch] epoch 60/100 loss=0.0979 test_f1=0.8691

[Scratch] epoch 80/100 loss=0.0528 test_f1=0.8478

[Scratch] epoch 100/100 loss=0.0359 test_f1=0.8177

--- Result (scratch) ---

Accuracy: 0.8405797101449275

Precision: 0.8314606741573034

Recall: 0.8043478260869565

F1: 0.8176795580110497

AUC: 0.9182419659735349

	precision	recall	f1-score	support
0.0	0.8475	0.8696	0.8584	115
1.0	0.8315	0.8043	0.8177	92
accuracy			0.8406	207
macro avg	0.8395	0.8370	0.8380	207
weighted avg	0.8403	0.8406	0.8403	207

Confusion matrix:

[[100 15]

[18 74]]

AE layerwise pretraining...

AE: training layer 1/4 in_dim=51 out=128

AE layer 1 ep 1/50 loss 0.252894

AE layer 1 ep 20/50 loss 0.002486

AE layer 1 ep 40/50 loss 0.000488

AE: training layer 2/4 in_dim=128 out=64

AE layer 2 ep 1/50 loss 0.328650

AE layer 2 ep 20/50 loss 0.019796

AE layer 2 ep 40/50 loss 0.008080

```
AE: training layer 3/4 in_dim=64 out=32
 AE layer 3 ep 1/50 loss 0.678278
 AE layer 3 ep 20/50 loss 0.080714
 AE layer 3 ep 40/50 loss 0.039859
AE: training layer 4/4 in_dim=32 out=16
 AE layer 4 ep 1/50 loss 1.764016
 AE layer 4 ep 20/50 loss 0.153516
 AE layer 4 ep 40/50 loss 0.116413
Finetuning AE-initialized model...
[AE-finetune] ep 1/100 loss 0.9587 test_f1 0.6154
[AE-finetune] ep 20/100 loss 0.6540 test_f1 0.6154
[AE-finetune] ep 40/100 loss 0.5346 test_f1 0.8276
[AE-finetune] ep 60/100 loss 0.4297 test_f1 0.8276
[AE-finetune] ep 80/100 loss 0.3984 test_f1 0.8457
[AE-finetune] ep 100/100 loss 0.3801 test_f1 0.8539
--- Result (AE pretraining) ---
Accuracy: 0.8743961352657005
Precision: 0.8837209302325582
Recall: 0.8260869565217391
F1: 0.8539325842696629
AUC: 0.9620982986767486
              precision    recall  f1-score   support

         0.0     0.8678    0.9130    0.8898       115
         1.0     0.8837    0.8261    0.8539        92

    accuracy                         0.8744       207
   macro avg     0.8757    0.8696    0.8719       207
weighted avg     0.8749    0.8744    0.8739       207

Confusion matrix:
 [[105  10]
```

[16 76]]

RBM stack pretraining...

RBM: training layer 1/4 vis=51 hid=128

RBM epoch 1/50 recon_loss 0.394137

RBM epoch 20/50 recon_loss 0.982443

RBM epoch 40/50 recon_loss 0.982757

RBM: training layer 2/4 vis=128 hid=64

RBM epoch 1/50 recon_loss 0.125922

RBM epoch 20/50 recon_loss 0.129697

RBM epoch 40/50 recon_loss 0.114650

RBM: training layer 3/4 vis=64 hid=32

RBM epoch 1/50 recon_loss 0.102568

RBM epoch 20/50 recon_loss 0.624525

RBM epoch 40/50 recon_loss 0.660149

RBM: training layer 4/4 vis=32 hid=16

RBM epoch 1/50 recon_loss 0.248729

RBM epoch 20/50 recon_loss 0.241185

RBM epoch 40/50 recon_loss 0.203080

Finetuning RBM-initialized model...

[RBM-finetune] ep 1/100 loss 0.6938 test_f1 0.6154

[RBM-finetune] ep 20/100 loss 0.6931 test_f1 0.0000

[RBM-finetune] ep 40/100 loss 0.6924 test_f1 0.0000

[RBM-finetune] ep 60/100 loss 0.6918 test_f1 0.0000

[RBM-finetune] ep 80/100 loss 0.6913 test_f1 0.0000

[RBM-finetune] ep 100/100 loss 0.6907 test_f1 0.0000

--- Result (RBM pretraining) ---

Accuracy: 0.5555555555555556

Precision: 0.0

Recall: 0.0

F1: 0.0

AUC: 0.5

	precision	recall	f1-score	support
0.0	0.5556	1.0000	0.7143	115
1.0	0.0000	0.0000	0.0000	92
accuracy			0.5556	207
macro avg	0.2778	0.5000	0.3571	207
weighted avg	0.3086	0.5556	0.3968	207

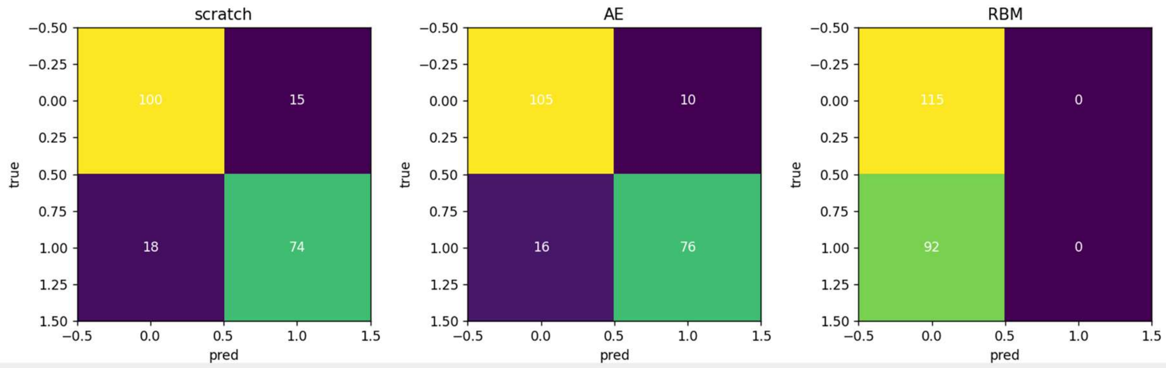
Confusion matrix:

[[115 0]

[92 0]]

Summary:

method	accuracy	precision	recall	f1	auc
scratch	0.840580	0.831461	0.804348	0.817680	0.918242
ae	0.874396	0.883721	0.826087	0.853933	0.962098
rbm	0.555556	0.000000	0.000000	0.000000	0.500000



Вывод: научился применять метод предобучения нейронных сетей с помощью автоэнкодерного подхода.