`

Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

Отчет по лабораторной работе 4

Специальность ИИ-23

Выполнил:

Тутина Е.Д.

Студент группы ИИ-23

Проверил:

Андренко К. В.
Преподаватель-стажёр
Кафедры ИИТ,

«___» ___________ 2025

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.

Вариант: 11

11	https://archive.ics.uci.edu/dataset/27/credit+approval	классификация	+/-

В ЛР 2 : Выборка : Wisconsin Diagnostic Breast Cancer (WDBC), класс: 2-й признак.

Код программы:

```
import os

import random

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, LabelEncoder, MinMaxScaler

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
confusion_matrix

import matplotlib.pyplot as plt


import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import TensorDataset, DataLoader


from sklearn.neural_network import BernoulliRBM


def set_seed(seed=42):

    random.seed(seed)
```

```
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

set_seed(42)

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

def load_wdbc():
    try:
        from sklearn.datasets import load_breast_cancer
        data = load_breast_cancer()
        X = pd.DataFrame(data.data, columns=data.feature_names)
        y = pd.Series(data.target)  # 0/1
        return X, y
    except Exception as e:
        raise RuntimeError("Не удалось загрузить WDBC через sklearn: " + str(e))

def load_credit_approval():
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/crx.data"
    df = pd.read_csv(url, header=None, na_values='?')
    cols = [f"c{i}" for i in range(df.shape[1])]
    df.columns = cols
    X = df.iloc[:, :-1].copy()
    y = df.iloc[:, -1].copy()
    y = y.map({'+':1, '-':0})
    return X, y

def preprocess_mixed(X_raw, y_raw, standardize=True):
    X = X_raw.copy()
    y = y_raw.copy()
    num_cols = X.select_dtypes(include=[np.number]).columns.tolist()
    cat_cols = [c for c in X.columns if c not in num_cols]
    for c in num_cols:
        X[c] = X[c].fillna(X[c].median())
    for c in cat_cols:
        X[c] = X[c].astype(str).fillna("NA")
        le = LabelEncoder()
```

```
        X[c] = le.fit_transform(X[c])
    if standardize:
        scaler = StandardScaler()
        if len(num_cols)>0:
            X[num_cols] = scaler.fit_transform(X[num_cols])
        if len(cat_cols)>0:
            X[cat_cols] = scaler.fit_transform(X[cat_cols])
    mask = ~y.isna()
    X = X.loc[mask].reset_index(drop=True)
    y = y.loc[mask].reset_index(drop=True).astype(int)
    return X.values.astype(np.float32), y.values.astype(np.int64)

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_sizes=[64,32,16,8], n_classes=2):
        super().__init__()
        layers = []
        layer_sizes = [input_dim] + hidden_sizes
        for i in range(len(layer_sizes)-1):
            layers.append(nn.Linear(layer_sizes[i], layer_sizes[i+1]))
            layers.append(nn.ReLU())
        self.features = nn.Sequential(*layers)
        self.classifier = nn.Linear(layer_sizes[-1], n_classes)
    def forward(self, x):
        h = self.features(x)
        out = self.classifier(h)
        return out

class AutoencoderLayer(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(in_dim, hidden_dim), nn.ReLU())
        self.decoder = nn.Sequential(nn.Linear(hidden_dim, in_dim))
    def forward(self, x):
        z = self.encoder(x)
        recon = self.decoder(z)
        return recon
    def encode(self, x):
        return self.encoder(x)
```

```python
def train_classification(model, train_loader, val_loader, epochs=50, lr=1e-3, weight_decay=1e-5,
print_every=5):
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
    best_val_f1 = 0.0
    best_state = None
    history = {"train_loss":[], "val_loss":[], "val_f1":[]}
    for epoch in range(1, epochs+1):
        model.train()
        train_losses = []
        for xb, yb in train_loader:
            xb = xb.to(device); yb = yb.to(device)
            logits = model(xb)
            loss = criterion(logits, yb)
            optimizer.zero_grad(); loss.backward(); optimizer.step()
            train_losses.append(loss.item())
        # val
        model.eval()
        val_losses = []
        ys_true = []; ys_pred = []
        with torch.no_grad():
            for xb, yb in val_loader:
                xb = xb.to(device); yb = yb.to(device)
                logits = model(xb)
                loss = criterion(logits, yb)
                val_losses.append(loss.item())
                preds = torch.argmax(logits, dim=1).cpu().numpy()
                ys_true.append(yb.cpu().numpy()); ys_pred.append(preds)
        ys_true = np.concatenate(ys_true)
        ys_pred = np.concatenate(ys_pred)
        val_f1 = f1_score(ys_true, ys_pred, zero_division=0)
        history["train_loss"].append(np.mean(train_losses))
        history["val_loss"].append(np.mean(val_losses))
        history["val_f1"].append(val_f1)
        if val_f1 > best_val_f1:
            best_val_f1 = val_f1
            best_state = model.state_dict()
        if epoch % print_every == 0 or epoch==1 or epoch==epochs:
            print(f"Epoch {epoch}/{epochs} train_loss={history['train_loss'][-1]:.4f}
val_loss={history['val_loss'][-1]:.4f} val_f1={val_f1:.4f}")
```

```
    if best_state is not None:
        model.load_state_dict(best_state)
    return model, history

def train_autoencoder(ae_model, data_loader, epochs=50, lr=1e-3, weight_decay=1e-5, print_every=10):
    ae_model = ae_model.to(device)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(ae_model.parameters(), lr=lr, weight_decay=weight_decay)
    for epoch in range(1, epochs+1):
        ae_model.train()
        losses = []
        for xb in data_loader:
            if isinstance(xb, (list,tuple)):
                xb = xb[0]
            xb = xb.to(device)
            recon = ae_model(xb)
            loss = criterion(recon, xb)
            optimizer.zero_grad(); loss.backward(); optimizer.step()
            losses.append(loss.item())
        if epoch % print_every == 0 or epoch==1 or epoch==epochs:
            print(f"AE epoch {epoch}/{epochs} loss {np.mean(losses):.6f}")
    return ae_model

def layerwise_pretrain(X_train, hidden_sizes, ae_epochs=50, batch_size=64, lr=1e-3):
    encoders = []
    current_input = torch.tensor(X_train, dtype=torch.float32)
    dataset = TensorDataset(current_input)
    for i, hdim in enumerate(hidden_sizes):
        in_dim = current_input.shape[1]
        print(f"\nPretraining AE layer {i+1}: {in_dim} -> {hdim}")
        ae = AutoencoderLayer(in_dim, hdim)
        loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
        ae = train_autoencoder(ae, loader, epochs=ae_epochs, lr=lr)
        encoders.append(ae.encoder.state_dict())
        ae = ae.to(device)
        ae.eval()
        with torch.no_grad():
            encoded = []
            for (xb,) in loader:
                xb = xb.to(device)
```

```
            z = ae.encode(xb).cpu()
            encoded.append(z)
        encoded = torch.cat(encoded, dim=0)
    current_input = encoded
    dataset = TensorDataset(current_input)
return encoders


def rbm_layerwise_pretrain(X_train, hidden_sizes, rbm_epochs=10, batch_size=10, learning_rate=0.01,
random_state=42, verbose=True):
    """
    X_train: numpy array (N, D) with values in [0,1] (we'll scale outside)
    Returns list of fitted RBM objects and list of hidden activations (for chaining)
    """
    rbms = []
    current_input = X_train.copy()
    for i, hdim in enumerate(hidden_sizes):
        in_dim = current_input.shape[1]
        print(f"\nPretraining RBM layer {i+1}: {in_dim} -> {hdim}")
        rbm = BernoulliRBM(n_components=hdim, learning_rate=learning_rate, batch_size=batch_size,
n_iter=rbm_epochs, verbose=verbose, random_state=random_state)
        rbm.fit(current_input)
        rbms.append(rbm)
        hid = rbm.transform(current_input)
        current_input = hid
    return rbms


def init_mlp_from_encoders(mlp_model, encoders_states):
    feat = mlp_model.features
    linear_layers = [m for m in feat.modules() if isinstance(m, nn.Linear)]
    for i, state in enumerate(encoders_states):
        if i < len(linear_layers):
            linear_layers[i].weight.data = state['0.weight'].data.clone()
            linear_layers[i].bias.data = state['0.bias'].data.clone()
        else:
            print("Warning: more encoders than linear layers in MLP")
    return mlp_model


def init_mlp_from_rbms(mlp_model, rbms):
    feat = mlp_model.features
    linear_layers = [m for m in feat.modules() if isinstance(m, nn.Linear)]
    for i, rbm in enumerate(rbms):
```

```python
        if i < len(linear_layers):
            comp = rbm.components_.astype(np.float32)  # shape (hidden_dim, visible_dim)
            intercept = rbm.intercept_hidden_.astype(np.float32)  # (hidden_dim,)
            linear_layers[i].weight.data = torch.tensor(comp, dtype=torch.float32)
            linear_layers[i].bias.data = torch.tensor(intercept, dtype=torch.float32)
        else:
            print("Warning: more RBMs than linear layers in MLP")
    return mlp_model


def evaluate_model(model, loader):
    model.eval()
    ys_true=[]; ys_pred=[]; ys_prob=[]
    with torch.no_grad():
        for xb, yb in loader:
            xb = xb.to(device)
            logits = model(xb)
            probs = torch.softmax(logits, dim=1)[:,1].cpu().numpy()
            preds = torch.argmax(logits, dim=1).cpu().numpy()
            ys_prob.append(probs); ys_pred.append(preds); ys_true.append(yb.numpy())
    y_true = np.concatenate(ys_true)
    y_pred = np.concatenate(ys_pred)
    y_prob = np.concatenate(ys_prob)
    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred, zero_division=0)
    rec = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)
    try:
        roc = roc_auc_score(y_true, y_prob)
    except:
        roc = np.nan
    cm = confusion_matrix(y_true, y_pred)
    return {"accuracy":acc, "precision":prec, "recall":rec, "f1":f1, "roc_auc":roc,
"confusion_matrix":cm}


def run_experiment(X, y, dataset_name="dataset", hidden_sizes=[64,32,16,8], epochs_sup=50, ae_epochs=50,
rbm_epochs=10, test_size=0.2, val_size=0.2, batch_size=64):
    print(f"\n==== Dataset: {dataset_name} | N={X.shape[0]} F={X.shape[1]} ====")
    X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=test_size, stratify=y,
random_state=42)
    X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=val_size,
stratify=y_trainval, random_state=42)
    print("Split sizes:", X_train.shape[0], X_val.shape[0], X_test.shape[0])
```

```python
    def make_loader(Xa, ya, batch_size=batch_size, shuffle=True):
        ds = TensorDataset(torch.tensor(Xa, dtype=torch.float32), torch.tensor(ya, dtype=torch.long))
        return DataLoader(ds, batch_size=batch_size, shuffle=shuffle)
    train_loader = make_loader(X_train, y_train)
    val_loader = make_loader(X_val, y_val, shuffle=False)
    test_loader = make_loader(X_test, y_test, shuffle=False)

    input_dim = X.shape[1]
    n_classes = len(np.unique(y))

    results = {}

    print("\n--- Training from scratch ---")
    model_scratch = MLP(input_dim, hidden_sizes=hidden_sizes, n_classes=n_classes)
    model_scratch, hist_scratch = train_classification(model_scratch, train_loader, val_loader,
epochs=epochs_sup, lr=1e-3)
    eval_scratch = evaluate_model(model_scratch, test_loader)
    print("Scratch eval:", eval_scratch)
    results['scratch'] = eval_scratch
    results['hist_scratch'] = hist_scratch
    results['model_scratch'] = model_scratch

    print("\n--- Layer-wise pretraining: Autoencoders ---")
    encoders_states = layerwise_pretrain(X_train, hidden_sizes, ae_epochs=ae_epochs,
batch_size=batch_size, lr=1e-3)
    print("\n--- Initializing MLP from pretrained autoencoders and finetune ---")
    model_pre_ae = MLP(input_dim, hidden_sizes=hidden_sizes, n_classes=n_classes)
    model_pre_ae = init_mlp_from_encoders(model_pre_ae, encoders_states)
    model_pre_ae, hist_pre_ae = train_classification(model_pre_ae, train_loader, val_loader,
epochs=epochs_sup, lr=1e-4)
    eval_pre_ae = evaluate_model(model_pre_ae, test_loader)
    print("AE Pretrained eval:", eval_pre_ae)
    results['pretrained_ae'] = eval_pre_ae
    results['hist_pre_ae'] = hist_pre_ae
    results['model_pre_ae'] = model_pre_ae

    print("\n--- Layer-wise pretraining: RBMs ---")
    mm = MinMaxScaler(feature_range=(0,1))
    X_train_rbm = mm.fit_transform(X_train)
    X_val_rbm = mm.transform(X_val)
```

```
    X_test_rbm = mm.transform(X_test)

    rbms = rbm_layerwise_pretrain(X_train_rbm, hidden_sizes, rbm_epochs=rbm_epochs, batch_size=min(10,
X_train_rbm.shape[0]), learning_rate=0.01)
    print("\n--- Initializing MLP from pretrained RBMs and finetune ---")
    model_pre_rbm = MLP(input_dim, hidden_sizes=hidden_sizes, n_classes=n_classes)
    model_pre_rbm = init_mlp_from_rbms(model_pre_rbm, rbms)
    model_pre_rbm, hist_pre_rbm = train_classification(model_pre_rbm, train_loader, val_loader,
epochs=epochs_sup, lr=1e-4)
    eval_pre_rbm = evaluate_model(model_pre_rbm, test_loader)
    print("RBM Pretrained eval:", eval_pre_rbm)
    results['pretrained_rbm'] = eval_pre_rbm
    results['hist_pre_rbm'] = hist_pre_rbm
    results['model_pre_rbm'] = model_pre_rbm

    return results

if __name__ == "__main__":
    hidden_sizes = [64, 32, 16, 8]
    epochs_sup = 40
    ae_epochs = 30
    rbm_epochs = 10
    batch_size = 64

    X_wdbc, y_wdbc = load_wdbc()
    Xw, yw = preprocess_mixed(X_wdbc, y_wdbc, standardize=True)
    res_wdbc = run_experiment(Xw, yw, dataset_name="WDBC", hidden_sizes=hidden_sizes,
epochs_sup=epochs_sup, ae_epochs=ae_epochs, rbm_epochs=rbm_epochs, batch_size=batch_size)

    X_cr, y_cr = load_credit_approval()
    Xc, yc = preprocess_mixed(X_cr, y_cr, standardize=True)
    res_credit = run_experiment(Xc, yc, dataset_name="CreditApproval", hidden_sizes=hidden_sizes,
epochs_sup=epochs_sup, ae_epochs=ae_epochs, rbm_epochs=rbm_epochs, batch_size=batch_size)

    def print_summary(name, res):
        print(f"\n=== Summary for {name} ===")
        for mode in ['scratch', 'pretrained_ae', 'pretrained_rbm']:
            print(f"\n--- {mode} ---")
            d = res[mode]
            for key in ["accuracy","precision","recall","f1","roc_auc","confusion_matrix"]:
                print(key, ":", d.get(key))
    print_summary("WDBC", res_wdbc)
```

```
    print_summary("CreditApproval", res_credit)


    summary = []

    for name, r in [("WDBC",res_wdbc), ("CreditApproval",res_credit)]:

        for mode in ["scratch","pretrained_ae","pretrained_rbm"]:

            d = r[mode]

            summary.append({

                "dataset": name,

                "mode": mode,

                "accuracy": d["accuracy"],

                "precision": d["precision"],

                "recall": d["recall"],

                "f1": d["f1"],

                "roc_auc": d["roc_auc"],

                "cm": d["confusion_matrix"].tolist()

            })

    df_summary = pd.DataFrame(summary)

    df_summary.to_csv("pretrain_compare_summary_rbm.csv", index=False)

    print("\nSaved summary to pretrain_compare_summary_rbm.csv")
```

Результат работы программы:

```
==== Dataset: WDBC | N=569 F=30 ====
Split sizes: 364 91 114

--- Training from scratch ---
Epoch 1/40 train_loss=0.7143 val_loss=0.7042 val_f1=0.0000
Epoch 5/40 train_loss=0.5943 val_loss=0.5768 val_f1=0.9483
Epoch 10/40 train_loss=0.1919 val_loss=0.1661 val_f1=0.9739
Epoch 15/40 train_loss=0.0674 val_loss=0.0813 val_f1=0.9825
Epoch 20/40 train_loss=0.0444 val_loss=0.0929 val_f1=0.9825
Epoch 25/40 train_loss=0.0312 val_loss=0.0932 val_f1=0.9735
Epoch 30/40 train_loss=0.0239 val_loss=0.0948 val_f1=0.9735
Epoch 35/40 train_loss=0.0187 val_loss=0.0880 val_f1=0.9825
Epoch 40/40 train_loss=0.0147 val_loss=0.0935 val_f1=0.9735
Scratch eval: {'accuracy': 0.956140350877193, 'precision': 0.9855072463768116, 'recall': 0.9444444444444444, 'f1': 0.9645390070921985, 'roc_auc': np.float64(0.9943783068783069), 'confusion_matrix': array([[41,  1],
       [ 4, 68]])}

--- Layer-wise pretraining: Autoencoders ---

Pretraining AE layer 1: 30 -> 64
AE epoch 1/30 loss 1.099238
AE epoch 10/30 loss 0.323581
AE epoch 20/30 loss 0.144776
AE epoch 30/30 loss 0.088569

Pretraining AE layer 2: 64 -> 32
AE epoch 1/30 loss 0.640508
AE epoch 10/30 loss 0.195796
AE epoch 20/30 loss 0.094007
AE epoch 30/30 loss 0.061484

Pretraining AE layer 3: 32 -> 16
AE epoch 1/30 loss 1.224143
AE epoch 10/30 loss 0.388969
```

```
AE epoch 20/30 loss 0.222152
AE epoch 30/30 loss 0.141597

Pretraining AE layer 4: 16 -> 8
AE epoch 1/30 loss 2.004041
AE epoch 10/30 loss 1.760936
AE epoch 20/30 loss 0.989153
AE epoch 30/30 loss 0.447652

--- Initializing MLP from pretrained autoencoders and finetune ---
Epoch 1/40 train_loss=0.7672 val_loss=0.7600 val_f1=0.0000
Epoch 5/40 train_loss=0.7373 val_loss=0.7299 val_f1=0.0000
Epoch 10/40 train_loss=0.6997 val_loss=0.6986 val_f1=0.0000
Epoch 15/40 train_loss=0.6711 val_loss=0.6691 val_f1=0.0000
Epoch 20/40 train_loss=0.6383 val_loss=0.6383 val_f1=0.0345
Epoch 25/40 train_loss=0.6058 val_loss=0.6054 val_f1=0.4384
Epoch 30/40 train_loss=0.5705 val_loss=0.5690 val_f1=0.8000
Epoch 35/40 train_loss=0.5259 val_loss=0.5272 val_f1=0.9259
Epoch 40/40 train_loss=0.4829 val_loss=0.4799 val_f1=0.9464
AE Pretrained eval: {'accuracy': 0.8596491228070176, 'precision': 1.0, 'recall': 0.7777777777777778,
'f1': 0.875, 'roc_auc': np.float64(0.9867724867724867), 'confusion_matrix': array([[42,  0],
       [16, 56]])}

--- Layer-wise pretraining: RBMs ---

Pretraining RBM layer 1: 30 -> 64
[BernoulliRBM] Iteration 1, pseudo-likelihood = -15.04, time = 0.01s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -13.90, time = 0.01s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -13.40, time = 0.01s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -13.14, time = 0.01s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -13.09, time = 0.01s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -13.01, time = 0.01s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -13.02, time = 0.01s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -12.93, time = 0.01s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -12.89, time = 0.01s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -12.87, time = 0.01s

Pretraining RBM layer 2: 64 -> 32
[BernoulliRBM] Iteration 1, pseudo-likelihood = -43.37, time = 0.00s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -42.98, time = 0.01s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -42.79, time = 0.01s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -42.64, time = 0.01s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -42.56, time = 0.01s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -42.46, time = 0.01s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -42.42, time = 0.01s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -42.39, time = 0.01s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -42.38, time = 0.01s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -42.37, time = 0.01s

Pretraining RBM layer 3: 32 -> 16
[BernoulliRBM] Iteration 1, pseudo-likelihood = -20.39, time = 0.00s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -19.49, time = 0.00s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -18.91, time = 0.00s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -18.50, time = 0.01s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -18.17, time = 0.01s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -17.92, time = 0.00s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -17.69, time = 0.00s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -17.54, time = 0.00s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -17.41, time = 0.00s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -17.30, time = 0.00s

Pretraining RBM layer 4: 16 -> 8
[BernoulliRBM] Iteration 1, pseudo-likelihood = -10.62, time = 0.00s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -10.33, time = 0.00s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -10.13, time = 0.00s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -9.96, time = 0.00s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -9.83, time = 0.00s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -9.72, time = 0.00s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -9.63, time = 0.00s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -9.57, time = 0.00s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -9.53, time = 0.00s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -9.47, time = 0.00s

--- Initializing MLP from pretrained RBMs and finetune ---
Epoch 1/40 train_loss=0.6756 val_loss=0.6761 val_f1=0.7703
Epoch 5/40 train_loss=0.6754 val_loss=0.6757 val_f1=0.7703
Epoch 10/40 train_loss=0.6756 val_loss=0.6753 val_f1=0.7703
Epoch 15/40 train_loss=0.6748 val_loss=0.6748 val_f1=0.7703
```

```
Epoch 20/40 train_loss=0.6745 val_loss=0.6744 val_f1=0.7703
Epoch 25/40 train_loss=0.6755 val_loss=0.6740 val_f1=0.7703
Epoch 30/40 train_loss=0.6740 val_loss=0.6737 val_f1=0.7703
Epoch 35/40 train_loss=0.6734 val_loss=0.6733 val_f1=0.7703
Epoch 40/40 train_loss=0.6730 val_loss=0.6729 val_f1=0.7703
RBM Pretrained eval: {'accuracy': 0.631578947368421, 'precision': 0.631578947368421, 'recall': 1.0, 'f1':
0.7741935483870968, 'roc_auc': np.float64(0.5), 'confusion_matrix': array([[ 0, 42],
       [ 0, 72]])}
```

==== Dataset: CreditApproval | N=690 F=15 ====

```
Split sizes: 441 111 138

--- Training from scratch ---
Epoch 1/40 train_loss=0.7086 val_loss=0.7059 val_f1=0.6125
Epoch 5/40 train_loss=0.6811 val_loss=0.6720 val_f1=0.6316
Epoch 10/40 train_loss=0.4935 val_loss=0.4494 val_f1=0.8667
Epoch 15/40 train_loss=0.3462 val_loss=0.3193 val_f1=0.8750
Epoch 20/40 train_loss=0.3126 val_loss=0.2922 val_f1=0.8842
Epoch 25/40 train_loss=0.2936 val_loss=0.2846 val_f1=0.8723
Epoch 30/40 train_loss=0.2748 val_loss=0.2815 val_f1=0.8723
Epoch 35/40 train_loss=0.2571 val_loss=0.2783 val_f1=0.8817
Epoch 40/40 train_loss=0.2353 val_loss=0.2770 val_f1=0.8817
Scratch eval: {'accuracy': 0.8913043478260869, 'precision': 0.859375, 'recall': 0.9016393442622951, 'f1':
0.88, 'roc_auc': np.float64(0.9520970832446242), 'confusion_matrix': array([[68,  9],
       [ 6, 55]])}

--- Layer-wise pretraining: Autoencoders ---

Pretraining AE layer 1: 15 -> 64
AE epoch 1/30 loss 1.017450
AE epoch 10/30 loss 0.451695
AE epoch 20/30 loss 0.163276
AE epoch 30/30 loss 0.067016

Pretraining AE layer 2: 64 -> 32
AE epoch 1/30 loss 0.439885
AE epoch 10/30 loss 0.200020
AE epoch 20/30 loss 0.101035
AE epoch 30/30 loss 0.056365

Pretraining AE layer 3: 32 -> 16
AE epoch 1/30 loss 1.207882
AE epoch 10/30 loss 0.407012
AE epoch 20/30 loss 0.272588
AE epoch 30/30 loss 0.237393

Pretraining AE layer 4: 16 -> 8
AE epoch 1/30 loss 1.395242
AE epoch 10/30 loss 0.518655
AE epoch 20/30 loss 0.150982
AE epoch 30/30 loss 0.083678

--- Initializing MLP from pretrained autoencoders and finetune ---
Epoch 1/40 train_loss=0.9903 val_loss=0.9953 val_f1=0.6125
Epoch 5/40 train_loss=0.9125 val_loss=0.9153 val_f1=0.6125
Epoch 10/40 train_loss=0.8413 val_loss=0.8436 val_f1=0.6125
Epoch 15/40 train_loss=0.7899 val_loss=0.7902 val_f1=0.6125
Epoch 20/40 train_loss=0.7522 val_loss=0.7492 val_f1=0.6125
Epoch 25/40 train_loss=0.7200 val_loss=0.7156 val_f1=0.6125
Epoch 30/40 train_loss=0.6933 val_loss=0.6858 val_f1=0.6164
Epoch 35/40 train_loss=0.6672 val_loss=0.6570 val_f1=0.6400
Epoch 40/40 train_loss=0.6390 val_loss=0.6273 val_f1=0.6870
AE Pretrained eval: {'accuracy': 0.6304347826086957, 'precision': 0.55, 'recall': 0.9016393442622951,
'f1': 0.6832298136645962, 'roc_auc': np.float64(0.8237172663402172), 'confusion_matrix': array([[32, 45],
       [ 6, 55]])}

--- Layer-wise pretraining: RBMs ---

Pretraining RBM layer 1: 15 -> 64
[BernoulliRBM] Iteration 1, pseudo-likelihood = -7.75, time = 0.00s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -7.44, time = 0.01s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -7.60, time = 0.01s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -7.37, time = 0.01s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -7.17, time = 0.01s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -7.28, time = 0.01s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -7.35, time = 0.01s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -7.18, time = 0.01s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -7.27, time = 0.01s
```

```
[BernoulliRBM] Iteration 10, pseudo-likelihood = -7.09, time = 0.01s

Pretraining RBM layer 2: 64 -> 32
[BernoulliRBM] Iteration 1, pseudo-likelihood = -41.05, time = 0.01s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -40.34, time = 0.01s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -40.02, time = 0.01s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -39.89, time = 0.01s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -39.72, time = 0.01s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -39.67, time = 0.01s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -39.59, time = 0.01s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -39.47, time = 0.01s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -39.44, time = 0.01s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -39.42, time = 0.01s

Pretraining RBM layer 3: 32 -> 16
[BernoulliRBM] Iteration 1, pseudo-likelihood = -14.79, time = 0.00s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -11.66, time = 0.01s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -9.77, time = 0.01s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -8.46, time = 0.01s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -7.54, time = 0.01s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -6.83, time = 0.01s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -6.43, time = 0.01s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -6.03, time = 0.01s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -5.83, time = 0.01s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -5.65, time = 0.01s

Pretraining RBM layer 4: 16 -> 8
[BernoulliRBM] Iteration 1, pseudo-likelihood = -10.76, time = 0.00s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -10.60, time = 0.00s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -10.49, time = 0.00s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -10.40, time = 0.00s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -10.34, time = 0.00s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -10.29, time = 0.00s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -10.27, time = 0.00s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -10.24, time = 0.00s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -10.21, time = 0.00s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -10.20, time = 0.00s

--- Initializing MLP from pretrained RBMs and finetune ---
Epoch 1/40 train_loss=0.7023 val_loss=0.7024 val_f1=0.6125
Epoch 5/40 train_loss=0.7018 val_loss=0.7020 val_f1=0.6125
Epoch 10/40 train_loss=0.7014 val_loss=0.7015 val_f1=0.6125
Epoch 15/40 train_loss=0.7009 val_loss=0.7011 val_f1=0.6125
Epoch 20/40 train_loss=0.7005 val_loss=0.7006 val_f1=0.6125
Epoch 25/40 train_loss=0.7000 val_loss=0.7001 val_f1=0.6125
Epoch 30/40 train_loss=0.6996 val_loss=0.6997 val_f1=0.6125
Epoch 35/40 train_loss=0.6992 val_loss=0.6993 val_f1=0.6125
Epoch 40/40 train_loss=0.6987 val_loss=0.6988 val_f1=0.6125
RBM Pretrained eval: {'accuracy': 0.4420289855072464, 'precision': 0.4420289855072464, 'recall': 1.0,
'f1': 0.6130653266331658, 'roc_auc': np.float64(0.5), 'confusion_matrix': array([[ 0, 77],
       [ 0, 61]])}
```

=== Summary for WDBC ===

```
--- scratch ---
accuracy : 0.956140350877193
precision : 0.9855072463768116
recall : 0.9444444444444444
f1 : 0.9645390070921985
roc_auc : 0.9943783068783069
confusion_matrix : [[41  1]
 [ 4 68]]

--- pretrained_ae ---
accuracy : 0.8596491228070176
precision : 1.0
recall : 0.7777777777777778
f1 : 0.875
roc_auc : 0.9867724867724867
confusion_matrix : [[42  0]
 [16 56]]

--- pretrained_rbm ---
accuracy : 0.631578947368421
precision : 0.631578947368421
recall : 1.0
f1 : 0.7741935483870968
roc_auc : 0.5
```

```
confusion_matrix : [[ 0 42]
 [ 0 72]]
```

=== Summary for CreditApproval ===

```
--- scratch ---
accuracy : 0.8913043478260869
precision : 0.859375
recall : 0.9016393442622951
f1 : 0.88
roc_auc : 0.9520970832446242
confusion_matrix : [[68  9]
 [ 6 55]]

--- pretrained_ae ---
accuracy : 0.6304347826086957
precision : 0.55
recall : 0.9016393442622951
f1 : 0.6832298136645962
roc_auc : 0.8237172663402172
confusion_matrix : [[32 45]
 [ 6 55]]

--- pretrained_rbm ---
accuracy : 0.4420289855072464
precision : 0.4420289855072464
recall : 1.0
f1 : 0.6130653266331658
roc_auc : 0.5
confusion_matrix : [[ 0 77]
 [ 0 61]]
```

Анализ для WDBC :

- при обучении с нуля модель показывает очень высокие результаты по всем метрикам, что говорит о хорошей способности обобщать. Ошибок почти нет — только 4 пропущенные положительные случаи и 1 ложное срабатывание;

- autoencoder не дал улучшения - recall значительно снизился (−17%), то есть модель чаще пропускает положительные случаи (16 FN). Precision = 1.0 говорит, что если модель уже решает, что объект положительный — она почти не ошибается, но делает это слишком редко.

- модель предсказывает все примеры как положительные (все TP и FP). Это объясняет то, что recall = 1.0 и при этом accuracy низкая. ROC-AUC = 0.5 - модель не различает классы.

Анализ для CreditApproval :

- аналогично предыдущему случаю при обучении с нуля получаем отличные результаты — модель хорошо сбалансирована: высокая точность и полнота.

- recall остался высоким то есть модель по-прежнему ловит почти все положительные, но precision упал сильно - стало много ложных тревог (FP=45) и снизилась accuracy.

- опять модель классифицирует все как положительное.

Можем сделать вывод, что RBM-предобучение в данном эксперименте некорректно работает или не адаптируется при fine-tuning, возможно, из-за неверной инициализации или несогласования размерностей.

Вывод: научилась осуществлять предобучение нейронных сетей с помощью автоэнкодерного подхода.