`

Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

Отчет по лабораторной работе 3

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 автоэнкодерного подхода.

Общее задание:

1. Взять за основу любую сверточную или полносвязную архитектуру с количеством слоев более 3. Осуществить ее обучение (без предобучения) в соответствии с вариантом задания. Получить оценку эффективности модели, используя метрики, специфичные для решаемой задачи (например, MAPE – для регрессионной задачи или F1/Confusion matrix для классификационной).

2. Выполнить обучение с предобучением, используя автоэнкодерный подход, алгоритм которого изложен в лекции. Условие останова (например, по количеству эпох) при обучении отдельных слоев с использованием автоэнкодера выбрать самостоятельно.

3. Сравнить результаты, полученные при обучении с/без предобучения, сделать выводы.

4. Выполните пункты 1-3 для датасетов из ЛР 2.

5. Оформить отчет по выполненной работе, загрузить исходный код и отчет в соответствующий репозиторий на github.

Вариант: 11
Выборка : Wisconsin Diagnostic Breast Cancer (WDBC)
Класс: 2-й признак

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

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
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
```

```python
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):

    X = X_raw.copy()
```

```python
    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])

    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
```

```python
class AutoencoderLayer(nn.Module):
    """
    Autoencoder for one layer: enc: in_dim -> hidden, dec: hidden -> in_dim
    We'll use MSE loss to reconstruct.
    """
    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)


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())
        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):
    """
```

```
    X_train: numpy array, shape (N, D)
    hidden_sizes: list of ints, e.g. [64,32,16,8]
    Returns: list of encoder weight/state dicts to initialize MLP
    """
    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 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 init_mlp_from_encoders(mlp_model, encoders_states):
    """
    mlp_model.features includes layers like Linear, ReLU, Linear, ReLU...
    encoders_states: list of state_dicts for each encoder (Linear + ReLU) saved from
AutoencoderLayer.encoder
    We'll assign weight & bias to corresponding Linear layers.
    """
    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 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,
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])


    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))


    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)


    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 encoders and finetune ---")

    model_pre = MLP(input_dim, hidden_sizes=hidden_sizes, n_classes=n_classes)

    model_pre = init_mlp_from_encoders(model_pre, encoders_states)


    model_pre, hist_pre = train_classification(model_pre, train_loader, val_loader, epochs=epochs_sup,
lr=1e-4)

    eval_pre = evaluate_model(model_pre, test_loader)

    print("Pretrained eval:", eval_pre)



    results = {

        "scratch": eval_scratch,

        "pretrained": eval_pre,

        "hist_scratch": hist_scratch,

        "hist_pre": hist_pre,

        "model_scratch": model_scratch,
```

```python
        "model_pre": model_pre
    }
    return results


if __name__ == "__main__":

    hidden_sizes = [64, 32, 16, 8]
    epochs_sup = 40
    ae_epochs = 30
    batch_size = 64



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



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

    def print_summary(name, res):
        print(f"\n=== Summary for {name} ===")
        for key in ["accuracy","precision","recall","f1","roc_auc","confusion_matrix"]:
            print("Scratch", key, ":", res["scratch"].get(key))
        for key in ["accuracy","precision","recall","f1","roc_auc","confusion_matrix"]:
            print("Pretrained", key, ":", res["pretrained"].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"]:
            d = r[mode]
            summary.append({
                "dataset": name,
                "mode": mode,
                "accuracy": d["accuracy"],
```

```python
                "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.csv", index=False)
    print("\nSaved summary to pretrain_compare_summary.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 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 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 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 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 encoders 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

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]])}
```

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

```
Split sizes: 441 111 138


--- Training from scratch ---

Epoch 1/40 train_loss=0.6813 val_loss=0.6739 val_f1=0.0000

Epoch 5/40 train_loss=0.6049 val_loss=0.5791 val_f1=0.7273

Epoch 10/40 train_loss=0.3928 val_loss=0.3602 val_f1=0.8750

Epoch 15/40 train_loss=0.3149 val_loss=0.2937 val_f1=0.8842

Epoch 20/40 train_loss=0.2872 val_loss=0.2796 val_f1=0.8817

Epoch 25/40 train_loss=0.2621 val_loss=0.2826 val_f1=0.8913

Epoch 30/40 train_loss=0.2349 val_loss=0.2845 val_f1=0.9032
```

```
Epoch 35/40 train_loss=0.2111 val_loss=0.2816 val_f1=0.8936

Epoch 40/40 train_loss=0.1869 val_loss=0.2874 val_f1=0.8936

Scratch eval: {'accuracy': 0.8623188405797102, 'precision': 0.85, 'recall': 0.8360655737704918, 'f1': 0.8429752066115702, 'roc_auc': np.float64(0.9416648924845646), 'confusion_matrix': array([[68,  9],

       [10, 51]])}


--- Layer-wise pretraining autoencoders ---


Pretraining layer 1: 15 -> 64

AE epoch 1/30 loss 1.001679

AE epoch 10/30 loss 0.422423

AE epoch 20/30 loss 0.139974

AE epoch 30/30 loss 0.058336


Pretraining layer 2: 64 -> 32

AE epoch 1/30 loss 0.489889

AE epoch 10/30 loss 0.197786

AE epoch 20/30 loss 0.103182

AE epoch 30/30 loss 0.060854


Pretraining layer 3: 32 -> 16

AE epoch 1/30 loss 1.100480

AE epoch 10/30 loss 0.376653

AE epoch 20/30 loss 0.248061

AE epoch 30/30 loss 0.202386


Pretraining layer 4: 16 -> 8

AE epoch 1/30 loss 1.749125

AE epoch 10/30 loss 0.942819

AE epoch 20/30 loss 0.321574

AE epoch 30/30 loss 0.141986


--- Initializing MLP from pretrained encoders and finetune ---

Epoch 1/40 train_loss=0.7616 val_loss=0.7616 val_f1=0.6125

Epoch 5/40 train_loss=0.7234 val_loss=0.7246 val_f1=0.6125

Epoch 10/40 train_loss=0.6852 val_loss=0.6871 val_f1=0.6125

Epoch 15/40 train_loss=0.6562 val_loss=0.6586 val_f1=0.6906

Epoch 20/40 train_loss=0.6311 val_loss=0.6337 val_f1=0.7778

Epoch 25/40 train_loss=0.6073 val_loss=0.6083 val_f1=0.7835

Epoch 30/40 train_loss=0.5822 val_loss=0.5816 val_f1=0.7912

Epoch 35/40 train_loss=0.5554 val_loss=0.5540 val_f1=0.8043
```

```
Epoch 40/40 train_loss=0.5302 val_loss=0.5262 val_f1=0.8000

Pretrained eval: {'accuracy': 0.7898550724637681, 'precision': 0.8809523809523809, 'recall': 0.6065573770491803, 'f1': 0.7184466019417476, 'roc_auc': np.float64(0.9046199701937407), 'confusion_matrix': array([[72,  5],
       [24, 37]])}


=== Summary for WDBC ===
Scratch accuracy : 0.956140350877193
Scratch precision : 0.9855072463768116
Scratch recall : 0.9444444444444444
Scratch f1 : 0.9645390070921985
Scratch roc_auc : 0.9943783068783069
Scratch confusion_matrix : [[41  1]
 [ 4 68]]
Pretrained accuracy : 0.8596491228070176
Pretrained precision : 1.0
Pretrained recall : 0.7777777777777778
Pretrained f1 : 0.875
Pretrained roc_auc : 0.9867724867724867
Pretrained confusion_matrix : [[42  0]
 [16 56]]


=== Summary for CreditApproval ===
Scratch accuracy : 0.8623188405797102
Scratch precision : 0.85
Scratch recall : 0.8360655737704918
Scratch f1 : 0.8429752066115702
Scratch roc_auc : 0.9416648924845646
Scratch confusion_matrix : [[68  9]
 [10 51]]
Pretrained accuracy : 0.7898550724637681
Pretrained precision : 0.8809523809523809
Pretrained recall : 0.6065573770491803
Pretrained f1 : 0.7184466019417476
Pretrained roc_auc : 0.9046199701937407
Pretrained confusion_matrix : [[72  5]
 [24 37]]
```

Для обоих наборов данных обучение «с нуля» оказалось более эффективным, чем предобучение с помощью автоэнкодера.

Автоэнкодер привёл к снижению чувствительности и F1-метрики, хотя сохранил высокие значения ROC-AUC, что говорит о частичном сохранении разделяющей способности, но ухудшении калибровки классификатора.

В данных условиях автоэнкодер не дал преимуществ.

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