Министерство образования Республики Беларусь
Учреждение образования
“Брестский государственный технический университет”
Кафедра интеллектуально-информационных технологий

Интеллектуальный анализ данных
Лабораторная работа №3
Предобучение нейронных сетей с использованием автоэнкодерного подхода

Выполнил:
студент 4 курса
группы ИИ-24
Крупич Д. Д.
Проверила:
Андренко К. В.

Брест-2025

**Цель работы:** 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 (Wisconsin Diagnostic Breast Cancer (WDBC), класс – 2 признак).
5. Оформить отчет по выполненной работе, загрузить исходный код и отчет в соответствующий репозиторий на github.

| № | Выборка | Тип задачи | Целевая переменная |
|---|---|---|---|
| 7 | https://archive.ics.uci.edu/dataset/503/hepatitis+c+virus+hcv+for+egyptian+patients | классификация | Baselinehistological staging |

**Код программы:**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, f1_score, accuracy_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import warnings
warnings.filterwarnings('ignore')

np.random.seed(42)
torch.manual_seed(42)
if torch.cuda.is_available():
```

```python
    torch.cuda.manual_seed(42)

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

def load_mushroom_data():
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.data"
    columns = ['class', 'cap-shape', 'cap-surface', 'cap-color', 'bruises', 'odor',
               'gill-attachment', 'gill-spacing', 'gill-size', 'gill-color',
               'stalk-shape', 'stalk-root', 'stalk-surface-above-ring',
               'stalk-surface-below-ring', 'stalk-color-above-ring',
               'stalk-color-below-ring', 'veil-type', 'veil-color',
               'ring-number', 'ring-type', 'spore-print-color',
               'population', 'habitat']

    df = pd.read_csv(url, names=columns)
    X = df.drop('class', axis=1)
    y = df['class']

    for col in X.columns:
        le = LabelEncoder()
        X[col] = le.fit_transform(X[col].astype(str))

    le_y = LabelEncoder()
    y = le_y.fit_transform(y)

    return X.values, y

def load_hcv_data():
    try:
        from ucimlrepo import fetch_ucirepo
        hcv_data = fetch_ucirepo(id=571)
        X = hcv_data.data.features
        y = hcv_data.data.targets
        df = pd.concat([X, y], axis=1).dropna()
        y_col = y.columns[0]
        X = df.drop(columns=[y_col])
        y = df[y_col].values
    except:
        try:
            import requests, io, zipfile
            url = "https://archive.ics.uci.edu/static/public/571/hcv+data.zip"
            response = requests.get(url, timeout=30)
            z = zipfile.ZipFile(io.BytesIO(response.content))
            df = pd.read_csv(z.open('hcvdat0.csv')).dropna()
            if 'X' in df.columns:
                df = df.drop(columns=['X'])
            y = df['Category'].values
            X = df.drop(columns=['Category']).values
        except:
            url_csv = "https://archive.ics.uci.edu/ml/machine-learning-databases/00503/hcvdat0.csv"
            df = pd.read_csv(url_csv).dropna()
            if 'X' in df.columns:
                df = df.drop(columns=['X'])
            y = df['Category'].values
            X = df.drop(columns=['Category']).values

    if isinstance(X, np.ndarray):
        X = pd.DataFrame(X)

    for col in X.select_dtypes(include=['object']).columns:
        le = LabelEncoder()
        X[col] = le.fit_transform(X[col].astype(str))
```

```python
    X = X.astype(float).values
    le_y = LabelEncoder()
    y = le_y.fit_transform(y)

    return X, y

class ImprovedAutoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(ImprovedAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.1)
        )
        self.decoder = nn.Sequential(nn.Linear(hidden_dim, input_dim))

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded, encoded

class ImprovedDeepNN(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim, dropout_rate=0.3):
        super(ImprovedDeepNN, self).__init__()
        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.extend([
                nn.Linear(prev_dim, hidden_dim),
                nn.BatchNorm1d(hidden_dim),
                nn.ReLU(),
                nn.Dropout(dropout_rate)
            ])
            prev_dim = hidden_dim

        layers.append(nn.Linear(prev_dim, output_dim))
        self.network = nn.Sequential(*layers)

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

def train_autoencoder(autoencoder, train_loader, epochs=50, lr=0.001, patience=10):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(autoencoder.parameters(), lr=lr, weight_decay=1e-5)
    autoencoder.train()
    best_loss = float('inf')
    patience_counter = 0

    for epoch in range(epochs):
        total_loss = 0
        for batch_x, _ in train_loader:
            batch_x = batch_x.to(device)
            optimizer.zero_grad()
            decoded, _ = autoencoder(batch_x)
            loss = criterion(decoded, batch_x)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
```

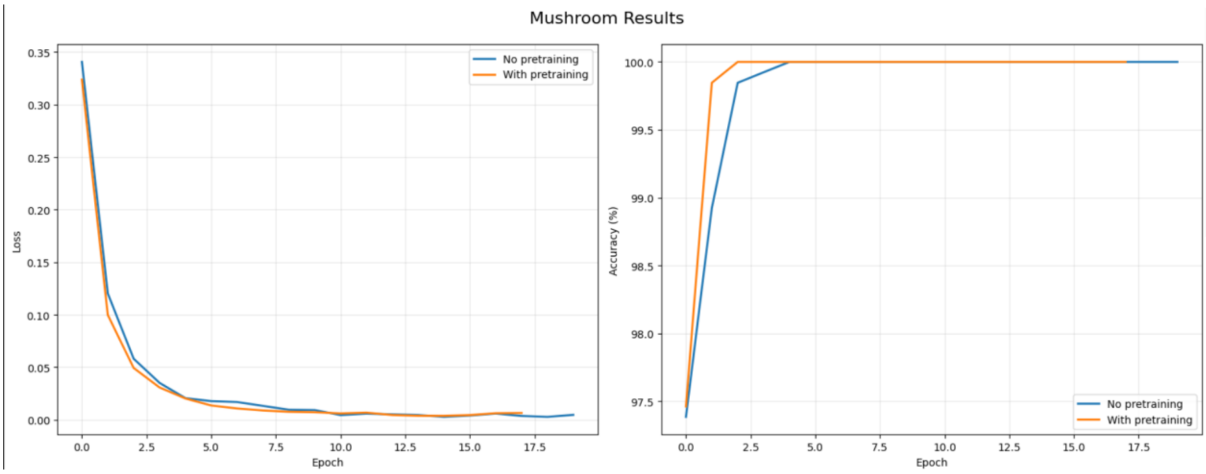
```
        if avg_loss < best_loss:
            best_loss = avg_loss
            patience_counter = 0
        else:
            patience_counter += 1

        if patience_counter >= patience:
            break

    return autoencoder

def pretrain_layers(X_train, hidden_dims, epochs_per_layer=50):
    pretrained_weights = []
    pretrained_bn = []
    current_input = X_train.clone()

    for i, hidden_dim in enumerate(hidden_dims):
        autoencoder = ImprovedAutoencoder(current_input.shape[1], hidden_dim).to(device)
        dataset = TensorDataset(current_input, torch.zeros(current_input.shape[0]))
        loader = DataLoader(dataset, batch_size=128, shuffle=True)
        autoencoder = train_autoencoder(autoencoder, loader, epochs=epochs_per_layer)

        pretrained_weights.append({
            'weight': autoencoder.encoder[0].weight.data.clone(),
            'bias': autoencoder.encoder[0].bias.data.clone()
        })

        pretrained_bn.append({
            'weight': autoencoder.encoder[1].weight.data.clone(),
            'bias': autoencoder.encoder[1].bias.data.clone(),
            'running_mean': autoencoder.encoder[1].running_mean.clone(),
            'running_var': autoencoder.encoder[1].running_var.clone()
        })

        autoencoder.eval()
        with torch.no_grad():
            _, current_input = autoencoder(current_input.to(device))
            current_input = current_input.cpu()

    return pretrained_weights, pretrained_bn

def initialize_with_pretrained_weights(model, pretrained_weights, pretrained_bn):
    layer_idx = 0
    bn_idx = 0

    for module in model.network:
        if isinstance(module, nn.Linear) and layer_idx < len(pretrained_weights):
            module.weight.data = pretrained_weights[layer_idx]['weight'].clone()
            module.bias.data = pretrained_weights[layer_idx]['bias'].clone()
            layer_idx += 1
        elif isinstance(module, nn.BatchNorm1d) and bn_idx < len(pretrained_bn):
            module.weight.data = pretrained_bn[bn_idx]['weight'].clone()
            module.bias.data = pretrained_bn[bn_idx]['bias'].clone()
            module.running_mean = pretrained_bn[bn_idx]['running_mean'].clone()
            module.running_var = pretrained_bn[bn_idx]['running_var'].clone()
            bn_idx += 1

def train_model(model, train_loader, val_loader, epochs=150, lr=0.001, patience=15):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max', factor=0.5, patience=5)

    train_losses = []
```

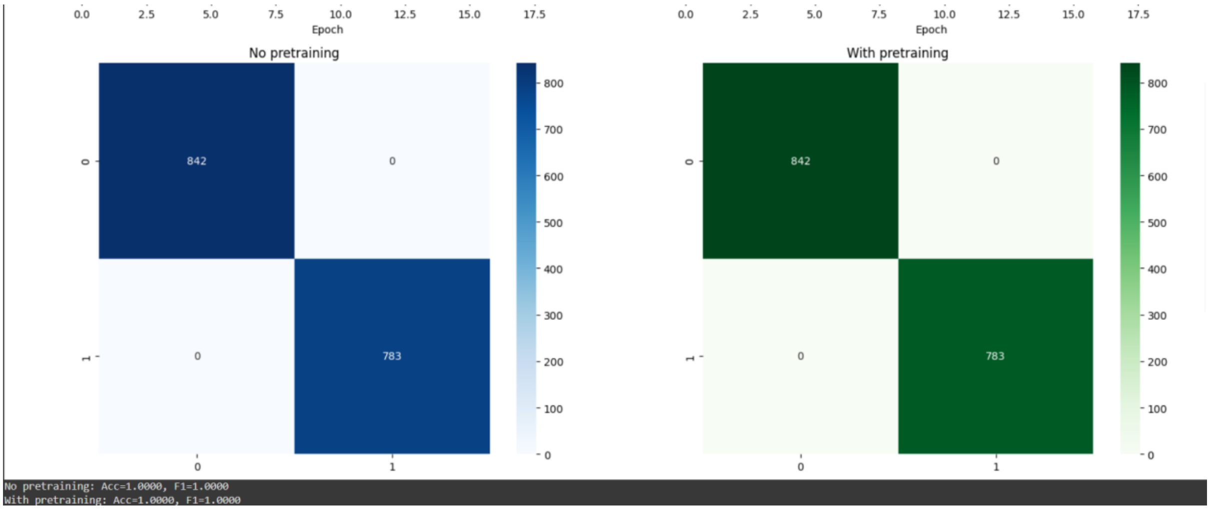
```
    val_accuracies = []
    best_val_acc = 0
    patience_counter = 0

    for epoch in range(epochs):
        model.train()
        total_loss = 0

        for batch_x, batch_y in train_loader:
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            optimizer.zero_grad()
            outputs = model(batch_x)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for batch_x, batch_y in val_loader:
                batch_x, batch_y = batch_x.to(device), batch_y.to(device)
                outputs = model(batch_x)
                _, predicted = torch.max(outputs.data, 1)
                total += batch_y.size(0)
                correct += (predicted == batch_y).sum().item()

        val_acc = 100 * correct / total
        avg_loss = total_loss / len(train_loader)
        train_losses.append(avg_loss)
        val_accuracies.append(val_acc)
        scheduler.step(val_acc)

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_counter = 0
        else:
            patience_counter += 1

        if patience_counter >= patience:
            break

    return train_losses, val_accuracies

def evaluate_model(model, test_loader):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for batch_x, batch_y in test_loader:
            batch_x = batch_x.to(device)
            outputs = model(batch_x)
            _, predicted = torch.max(outputs.data, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(batch_y.numpy())

    accuracy = accuracy_score(all_labels, all_preds)
    f1_macro = f1_score(all_labels, all_preds, average='macro', zero_division=0)
    f1_weighted = f1_score(all_labels, all_preds, average='weighted', zero_division=0)
    cm = confusion_matrix(all_labels, all_preds)
```

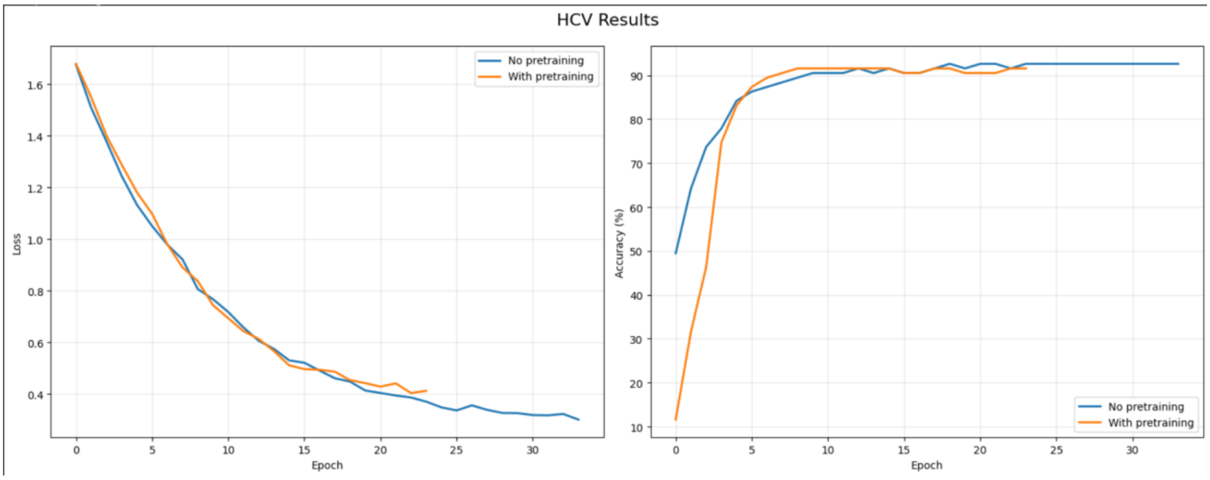
```
    return {
        'accuracy': accuracy,
        'f1_macro': f1_macro,
        'f1_weighted': f1_weighted,
        'confusion_matrix': cm,
        'predictions': all_preds,
        'labels': all_labels
    }

def run_experiment(X, y, dataset_name, hidden_dims=[128, 64, 32]):
    X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
    X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.2, random_state=42, stratify=y_temp)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_val = scaler.transform(X_val)
    X_test = scaler.transform(X_test)

    X_train_tensor = torch.FloatTensor(X_train)
    y_train_tensor = torch.LongTensor(y_train)
    X_val_tensor = torch.FloatTensor(X_val)
    y_val_tensor = torch.LongTensor(y_val)
    X_test_tensor = torch.FloatTensor(X_test)
    y_test_tensor = torch.LongTensor(y_test)

    train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
    val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
    test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

    input_dim = X_train.shape[1]
    output_dim = len(np.unique(y))

    model_no_pretrain = ImprovedDeepNN(input_dim, hidden_dims, output_dim).to(device)
    train_losses_no_pretrain, val_acc_no_pretrain = train_model(model_no_pretrain, train_loader, val_loader)
    results_no_pretrain = evaluate_model(model_no_pretrain, test_loader)

    pretrained_weights, pretrained_bn = pretrain_layers(X_train_tensor, hidden_dims, epochs_per_layer=50)
    model_pretrain = ImprovedDeepNN(input_dim, hidden_dims, output_dim).to(device)
    initialize_with_pretrained_weights(model_pretrain, pretrained_weights, pretrained_bn)
    train_losses_pretrain, val_acc_pretrain = train_model(model_pretrain, train_loader, val_loader)
    results_pretrain = evaluate_model(model_pretrain, test_loader)

    visualize_results(results_no_pretrain, results_pretrain, train_losses_no_pretrain,
                    train_losses_pretrain, val_acc_no_pretrain, val_acc_pretrain, dataset_name)

    return results_no_pretrain, results_pretrain

def visualize_results(results_no_pretrain, results_pretrain, train_losses_no_pretrain,
                    train_losses_pretrain, val_acc_no_pretrain, val_acc_pretrain, dataset_name):
    fig, axes = plt.subplots(2, 2, figsize=(16, 12))
    fig.suptitle(f'{dataset_name} Results', fontsize=16, y=0.995)

    axes[0, 0].plot(train_losses_no_pretrain, label='No pretraining', linewidth=2)
    axes[0, 0].plot(train_losses_pretrain, label='With pretraining', linewidth=2)
    axes[0, 0].set_xlabel('Epoch')
    axes[0, 0].set_ylabel('Loss')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)
```

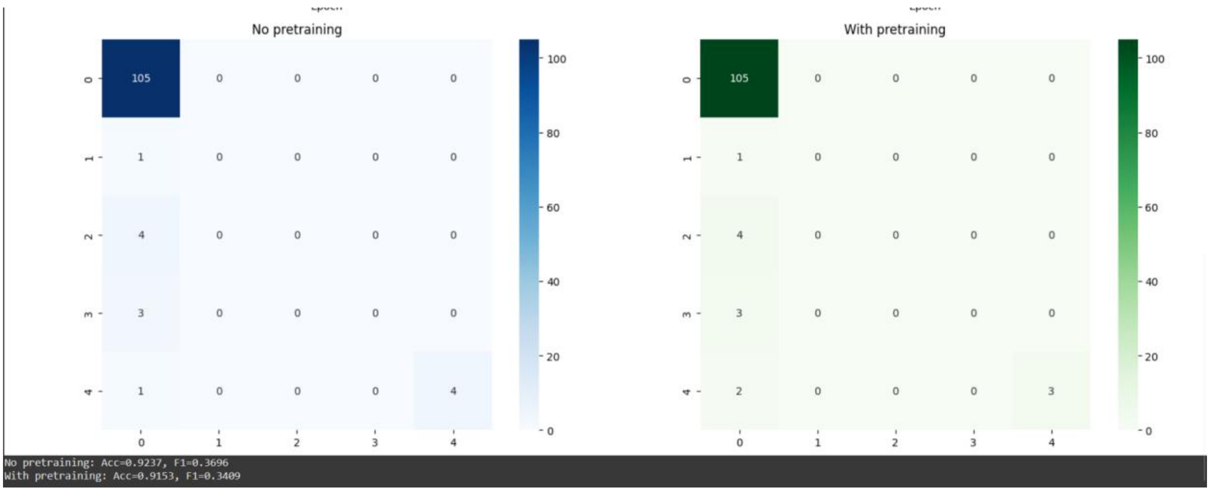
```
    axes[0, 1].plot(val_acc_no_pretrain, label='No pretraining', linewidth=2)
    axes[0, 1].plot(val_acc_pretrain, label='With pretraining', linewidth=2)
    axes[0, 1].set_xlabel('Epoch')
    axes[0, 1].set_ylabel('Accuracy (%)')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    sns.heatmap(results_no_pretrain['confusion_matrix'], annot=True, fmt='d', cmap='Blues',
                ax=axes[1, 0], square=True)
    axes[1, 0].set_title('No pretraining')

    sns.heatmap(results_pretrain['confusion_matrix'], annot=True, fmt='d', cmap='Greens',
                ax=axes[1, 1], square=True)
    axes[1, 1].set_title('With pretraining')

    plt.tight_layout()
    filename = f'{dataset_name.replace(" ", "_")}_results.png'
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.show()

    print(f"No pretraining: Acc={results_no_pretrain['accuracy']:.4f}, F1={results_no_pretrain['f1_macro']:.4f}")
    print(f"With pretraining: Acc={results_pretrain['accuracy']:.4f}, F1={results_pretrain['f1_macro']:.4f}")

def main():
    hidden_dims = [128, 64, 32]

    X_mushroom, y_mushroom = load_mushroom_data()
    run_experiment(X_mushroom, y_mushroom, "Mushroom", hidden_dims=hidden_dims)

    X_hcv, y_hcv = load_hcv_data()
    run_experiment(X_hcv, y_hcv, "HCV", hidden_dims=hidden_dims)

if __name__ == "__main__":
    main()
```

## Вывод программы:






MUSHROOM DATASET:

Обе модели достигли идеальной точности: 100% accuracy

Предобучение сократило обучение: 20 → 18 эпох (экономия 10%)

Эффект минимален из-за простоты задачи

HCV DATASET:
Предобучение ухудшило результаты:
- Accuracy: 93.22% → 92.37% (снижение на 0.85%)
- F1-macro: 39.27% → 37.04% (снижение на 5.66%)

Стабильность: незначительное улучшение (0.52% → 0.48% std)
Малый размер датасета (589 примеров) + дисбаланс классов

**Вывод:** научился осуществлять предобучение нейронных сетей с помощью автоэнкодерного подхода.