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

Копач А. В.

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
15	cardiotocography	классификация	CLASS/NSP

Код:

```
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 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')

def load_cardiotocography_data():
```

```
    try:
        df = pd.read_excel('CTG.xls', sheet_name='Data', header=1)
        print(" Данные CTG успешно загружены")

        df = df.dropna(axis=1, how='all')

        feature_columns = ['LB', 'AC', 'FM', 'UC', 'DL', 'DS', 'DP',
                           'ASTV', 'MSTV', 'ALTV', 'MLTV',
                           'Width', 'Min', 'Max', 'Nmax', 'Nzeros',
                           'Mode', 'Mean', 'Median', 'Variance', 'Tendency']

        available_features = [col for col in feature_columns if col in
df.columns]
        target_col = 'NSP' if 'NSP' in df.columns else 'CLASS'

        df_clean = df[available_features + [target_col]].dropna()

        X = df_clean[available_features]
        y = df_clean[target_col] - 1

        print(f" Данные: {X.shape[0]} samples, {X.shape[1]} features")
        print(f" Классы: {np.unique(y)}")
        print(f" Распределение классов: {np.bincount(y)}")

        return X, y, available_features

    except Exception as e:
        print(f" Ошибка загрузки: {e}")
        return None, None, None


X, y, feature_names = load_cardiotocography_data()

if X is None:
    print(" Не удалось загрузить данные CTG")
    exit()

print("\n" + "=" * 50)
print("УЛУЧШЕННАЯ РЕАЛИЗАЦИЯ RBM")
print("=" * 50)


class ImprovedRBM(nn.Module):

    def __init__(self, n_visible, n_hidden):
        super(ImprovedRBM, self).__init__()
        self.W = nn.Parameter(torch.randn(n_visible, n_hidden) * 0.01)
        self.v_bias = nn.Parameter(torch.zeros(n_visible))
        self.h_bias = nn.Parameter(torch.zeros(n_hidden))
        self.n_visible = n_visible
        self.n_hidden = n_hidden

    def sample_from_p(self, p):
        return torch.bernoulli(p)

    def v_to_h(self, v):
        activation = torch.matmul(v, self.W) + self.h_bias
        p_h = torch.sigmoid(activation)
        return p_h, self.sample_from_p(p_h)
```

```
    def h_to_v(self, h):
        activation = torch.matmul(h, self.W.t()) + self.v_bias
        p_v = torch.sigmoid(activation)
        return p_v, self.sample_from_p(p_v)

    def contrastive_divergence(self, v0, k=1):
        ph0, h0 = self.v_to_h(v0)


        vk = v0
        for _ in range(k):
            _, hk = self.v_to_h(vk)
            p_vk, vk = self.h_to_v(hk)
            vk = vk + torch.randn_like(vk) * 0.01
            vk = torch.clamp(vk, 0, 1)

        phk, _ = self.v_to_h(vk)

        return v0, vk, ph0, phk

    def free_energy(self, v):
        wx_b = torch.matmul(v, self.W) + self.h_bias
        vbias_term = torch.matmul(v, self.v_bias.unsqueeze(1)).squeeze()
        hidden_term = torch.sum(torch.log(1 + torch.exp(wx_b)), dim=1)
        return -hidden_term - vbias_term


class ImprovedRBMPretrainer:

    def __init__(self, layer_dims):
        self.layer_dims = layer_dims
        self.rbms = []

    def pretrain_layer(self, X, n_visible, n_hidden, epochs=50, lr=0.01, k=1,
momentum=0.9):
        print(f" Обучение RBM: {n_visible} → {n_hidden}")

        rbm = ImprovedRBM(n_visible, n_hidden)

        X_normalized = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0) +
1e-8)
        X_tensor = torch.FloatTensor(X_normalized)

        W_momentum = torch.zeros_like(rbm.W)
        v_bias_momentum = torch.zeros_like(rbm.v_bias)
        h_bias_momentum = torch.zeros_like(rbm.h_bias)

        losses = []
        for epoch in range(epochs):
            epoch_loss = 0
            num_batches = 0

            indices = torch.randperm(len(X_tensor))

            for i in range(0, len(X_tensor), 32):
                batch_indices = indices[i:i + 32]
                batch = X_tensor[batch_indices]
```

```
                v0, vk, ph0, phk = rbm.contrastive_divergence(batch, k=k)

                positive_grad = torch.matmul(v0.t(), ph0)
                negative_grad = torch.matmul(vk.t(), phk)

                delta_W = lr * ((positive_grad - negative_grad) / len(batch))
                delta_v_bias = lr * torch.mean(v0 - vk, dim=0)
                delta_h_bias = lr * torch.mean(ph0 - phk, dim=0)

                W_momentum = momentum * W_momentum + delta_W
                v_bias_momentum = momentum * v_bias_momentum + delta_v_bias
                h_bias_momentum = momentum * h_bias_momentum + delta_h_bias

                rbm.W.data += W_momentum
                rbm.v_bias.data += v_bias_momentum
                rbm.h_bias.data += h_bias_momentum

                loss = torch.mean(rbm.free_energy(v0)) -
torch.mean(rbm.free_energy(vk))
                epoch_loss += loss.item()
                num_batches += 1

            avg_loss = epoch_loss / num_batches if num_batches > 0 else 0
            losses.append(avg_loss)

            if (epoch + 1) % 10 == 0:
                print(f'    Эпоха [{epoch + 1}/{epochs}], Потери:
{avg_loss:.4f}')

        return rbm.W.data.clone(), rbm.h_bias.data.clone(), losses

    def pretrain_stack(self, X, epochs_per_layer=50):
        print(" Начало послойного предобучения RBM...")
        current_data = X
        all_losses = []

        for i, n_hidden in enumerate(self.layer_dims):
            n_visible = current_data.shape[1]
            print(f" Слой {i + 1}: {n_visible} → {n_hidden}")

            weights, biases, losses = self.pretrain_layer(current_data,
n_visible, n_hidden, epochs_per_layer)
            self.rbms.append((weights, biases))
            all_losses.append(losses)

            with torch.no_grad():
                rbm_temp = ImprovedRBM(n_visible, n_hidden)
                rbm_temp.W.data = weights
                rbm_temp.h_bias.data = biases

                ph, _ = rbm_temp.v_to_h(torch.FloatTensor(current_data))
                current_data = ph.numpy()

        print(" Предобучение RBM завершено!")
        return self.rbms, all_losses


class NeuralNetwork(nn.Module):
```

```
    def __init__(self, input_dim, num_classes):
        super(NeuralNetwork, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64, num_classes)
        )

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


class AutoencoderPretrainedNetwork(nn.Module):

    def __init__(self, input_dim, num_classes, pretrained_weights):
        super(AutoencoderPretrainedNetwork, self).__init__()

        self.layer1 = nn.Linear(input_dim, 256)
        self.layer2 = nn.Linear(256, 128)
        self.layer3 = nn.Linear(128, 64)
        self.output_layer = nn.Linear(64, num_classes)

        if len(pretrained_weights) >= 3:
            self.layer1.weight.data = pretrained_weights[0][0].clone()
            self.layer1.bias.data = pretrained_weights[0][1].clone()

            self.layer2.weight.data = pretrained_weights[1][0].clone()
            self.layer2.bias.data = pretrained_weights[1][1].clone()

            self.layer3.weight.data = pretrained_weights[2][0].clone()
            self.layer3.bias.data = pretrained_weights[2][1].clone()

        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(0.4)
        self.dropout2 = nn.Dropout(0.3)
        self.dropout3 = nn.Dropout(0.2)

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.dropout1(x)
        x = self.relu(self.layer2(x))
        x = self.dropout2(x)
        x = self.relu(self.layer3(x))
        x = self.dropout3(x)
        x = self.output_layer(x)
        return x


class RBMPretrainedNetwork(nn.Module):

    def __init__(self, input_dim, num_classes, pretrained_weights):
        super(RBMPretrainedNetwork, self).__init__()
```

```
        self.layer1 = nn.Linear(input_dim, 256)
        self.layer2 = nn.Linear(256, 128)
        self.layer3 = nn.Linear(128, 64)
        self.output_layer = nn.Linear(64, num_classes)

        if len(pretrained_weights) >= 3:
            self.layer1.weight.data = pretrained_weights[0][0].t().clone()
            self.layer1.bias.data = pretrained_weights[0][1].clone()

            self.layer2.weight.data = pretrained_weights[1][0].t().clone()
            self.layer2.bias.data = pretrained_weights[1][1].clone()

            self.layer3.weight.data = pretrained_weights[2][0].t().clone()
            self.layer3.bias.data = pretrained_weights[2][1].clone()

        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(0.4)
        self.dropout2 = nn.Dropout(0.3)
        self.dropout3 = nn.Dropout(0.2)

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.dropout1(x)
        x = self.relu(self.layer2(x))
        x = self.dropout2(x)
        x = self.relu(self.layer3(x))
        x = self.dropout3(x)
        x = self.output_layer(x)
        return x


class AutoencoderPretrainer:

    def __init__(self, layer_dims):
        self.layer_dims = layer_dims
        self.autoencoders = []

    def pretrain_layer(self, X, input_dim, encoding_dim, epochs=50):
        print(f" Предобучение слоя: {input_dim} → {encoding_dim}")

        autoencoder = nn.Sequential(
            nn.Linear(input_dim, encoding_dim),
            nn.ReLU(),
            nn.Linear(encoding_dim, input_dim)
        )

        criterion = nn.MSELoss()
        optimizer = optim.Adam(autoencoder.parameters(), lr=0.001)

        X_tensor = torch.FloatTensor(X)

        losses = []
        for epoch in range(epochs):
            autoencoder.train()
            total_loss = 0
            num_batches = 0

            for batch_idx in range(0, len(X_tensor), 32):
```

```
                batch = X_tensor[batch_idx:batch_idx + 32]
                optimizer.zero_grad()
                reconstructed = autoencoder(batch)
                loss = criterion(reconstructed, batch)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
                num_batches += 1

            avg_loss = total_loss / num_batches if num_batches > 0 else 0
            losses.append(avg_loss)

            if (epoch + 1) % 20 == 0:
                print(f'    Эпоха [{epoch + 1}/{epochs}], Потери:
{avg_loss:.4f}')

        return autoencoder[0].weight.data.clone(),
autoencoder[0].bias.data.clone(), losses

    def pretrain_stack(self, X, epochs_per_layer=50):
        print(" Начало послойного предобучения автоэнкодером...")
        current_data = X
        all_losses = []

        for i, encoding_dim in enumerate(self.layer_dims):
            input_dim = current_data.shape[1]
            print(f" Слой {i + 1}: {input_dim} → {encoding_dim}")

            weights, biases, losses = self.pretrain_layer(current_data,
input_dim, encoding_dim, epochs_per_layer)
            self.autoencoders.append((weights, biases))
            all_losses.append(losses)

            with torch.no_grad():
                linear_layer = nn.Linear(input_dim, encoding_dim)
                linear_layer.weight.data = weights
                linear_layer.bias.data = biases
                current_data =
torch.relu(linear_layer(torch.FloatTensor(current_data))).numpy()

        print(" Предобучение автоэнкодером завершено!")
        return self.autoencoders, all_losses


def train_and_evaluate_model(model, train_loader, test_loader, epochs=100,
model_name="Модель"):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    train_losses = []
    test_accuracies = []

    print(f"\n Обучение {model_name}...")

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for batch_x, batch_y in train_loader:
            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
        all_preds = []
        all_labels = []

        with torch.no_grad():
            for batch_x, batch_y in test_loader:
                outputs = model(batch_x)
                _, predicted = torch.max(outputs.data, 1)
                total += batch_y.size(0)
                correct += (predicted == batch_y).sum().item()
                all_preds.extend(predicted.numpy())
                all_labels.extend(batch_y.numpy())

        accuracy = correct / total
        train_losses.append(total_loss / len(train_loader))
        test_accuracies.append(accuracy)

        if (epoch + 1) % 20 == 0:
            print(f' Эпоха [{epoch + 1}/{epochs}], Потери: {total_loss /
len(train_loader):.4f}, '
                  f'Точность: {accuracy:.4f}')

    final_accuracy = accuracy_score(all_labels, all_preds)
    final_f1 = f1_score(all_labels, all_preds, average='weighted')
    cm = confusion_matrix(all_labels, all_preds)

    print(f"\n Результаты {model_name}:")
    print(f"   Точность: {final_accuracy:.4f}")
    print(f"   F1-score: {final_f1:.4f}")

    return final_accuracy, final_f1, cm, train_losses, test_accuracies


print("\n" + "=" * 70)
print("СРАВНЕНИЕ МЕТОДОВ ПРЕДОБУЧЕНИЯ НА CTG ДАННЫХ")
print("=" * 70)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train_tensor = torch.FloatTensor(X_train_scaled)
y_train_tensor = torch.LongTensor(y_train.values)
X_test_tensor = torch.FloatTensor(X_test_scaled)
y_test_tensor = torch.LongTensor(y_test.values)

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
```

```
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

input_dim = X.shape[1]
num_classes = len(np.unique(y))
layer_dims = [256, 128, 64]
epochs = 100

print(f" Архитектура сети: {input_dim} → {layer_dims} → {num_classes}")
print(f" Эпох обучения: {epochs}")
print(f" Обучающая выборка: {X_train_scaled.shape}")
print(f" Тестовая выборка: {X_test_scaled.shape}")

print("\n" + "=" * 50)
print("1.  БАЗОВАЯ МОДЕЛЬ БЕЗ ПРЕДОБУЧЕНИЯ")
base_model = NeuralNetwork(input_dim, num_classes)
base_accuracy, base_f1, cm_base, base_train_losses, base_test_accuracies =
train_and_evaluate_model(
    base_model, train_loader, test_loader, epochs=epochs,
    model_name="Без предобучения"
)

print("\n" + "=" * 50)
print("2.  МОДЕЛЬ С ПРЕДОБУЧЕНИЕМ АВТОЭНКОДЕРОМ")
ae_pretrainer = AutoencoderPretrainer(layer_dims)
ae_weights, ae_losses = ae_pretrainer.pretrain_stack(X_train_scaled,
epochs_per_layer=50)

ae_model = AutoencoderPretrainedNetwork(input_dim, num_classes, ae_weights)
ae_accuracy, ae_f1, cm_ae, ae_train_losses, ae_test_accuracies =
train_and_evaluate_model(
    ae_model, train_loader, test_loader, epochs=epochs,
    model_name="С предобучением (Autoencoder)"
)

print("\n" + "=" * 50)
print("3.  МОДЕЛЬ С ПРЕДОБУЧЕНИЕМ RBM")
rbm_pretrainer = ImprovedRBMPretrainer(layer_dims)
rbm_weights, rbm_losses = rbm_pretrainer.pretrain_stack(X_train_scaled,
epochs_per_layer=50)

rbm_model = RBMPretrainedNetwork(input_dim, num_classes, rbm_weights)
rbm_accuracy, rbm_f1, cm_rbm, rbm_train_losses, rbm_test_accuracies =
train_and_evaluate_model(
    rbm_model, train_loader, test_loader, epochs=epochs,
    model_name="С предобучением (RBM)"
)

print("\n" + "=" * 70)
print("ВИЗУАЛИЗАЦИЯ РЕЗУЛЬТАТОВ СРАВНЕНИЯ")
print("=" * 70)

fig = plt.figure(figsize=(20, 15))

ax1 = plt.subplot2grid((3, 3), (0, 0), colspan=2)
methods = ['Без предобучения', 'Autoencoder', 'RBM']
accuracies = [base_accuracy, ae_accuracy, rbm_accuracy]
```

```
f1_scores = [base_f1, ae_f1, rbm_f1]

x = np.arange(len(methods))
width = 0.35

bars1 = ax1.bar(x - width / 2, accuracies, width, label='Точность', alpha=0.8,
color='skyblue')
bars2 = ax1.bar(x + width / 2, f1_scores, width, label='F1-score', alpha=0.8,
color='lightcoral')

ax1.set_title('Сравнение метрик моделей\n(Данные CTG)', fontweight='bold',
fontsize=14)
ax1.set_ylabel('Score')
ax1.set_xticks(x)
ax1.set_xticklabels(methods, rotation=45)
ax1.legend()
ax1.grid(True, alpha=0.3)

for bars in [bars1, bars2]:
    for bar in bars:
        height = bar.get_height()
        ax1.text(bar.get_x() + bar.get_width() / 2., height + 0.01,
                 f'{height:.3f}', ha='center', va='bottom', fontweight='bold')

class_names = ['Нормальный', 'Подозрительный', 'Патологический']

ax2 = plt.subplot2grid((3, 3), (0, 2))
sns.heatmap(cm_base, annot=True, fmt='d', cmap='Blues', ax=ax2,
            xticklabels=class_names, yticklabels=class_names)
ax2.set_title('Матрица ошибок\n(Без предобучения)', fontweight='bold')
ax2.set_xlabel('Предсказанный класс')
ax2.set_ylabel('Истинный класс')

ax3 = plt.subplot2grid((3, 3), (1, 0))
sns.heatmap(cm_ae, annot=True, fmt='d', cmap='Blues', ax=ax3,
            xticklabels=class_names, yticklabels=class_names)
ax3.set_title('Матрица ошибок\n(Autoencoder)', fontweight='bold')
ax3.set_xlabel('Предсказанный класс')
ax3.set_ylabel('Истинный класс')

ax4 = plt.subplot2grid((3, 3), (1, 1))
sns.heatmap(cm_rbm, annot=True, fmt='d', cmap='Blues', ax=ax4,
            xticklabels=class_names, yticklabels=class_names)
ax4.set_title('Матрица ошибок\n(RBM)', fontweight='bold')
ax4.set_xlabel('Предсказанный класс')
ax4.set_ylabel('Истинный класс')

ax5 = plt.subplot2grid((3, 3), (1, 2))
ax5.plot(base_test_accuracies, label='Без предобучения', linewidth=2)
ax5.plot(ae_test_accuracies, label='Autoencoder', linewidth=2)
ax5.plot(rbm_test_accuracies, label='RBM', linewidth=2)
ax5.set_title('Точность во время обучения', fontweight='bold')
ax5.set_xlabel('Эпоха')
ax5.set_ylabel('Точность')
ax5.legend()
ax5.grid(True, alpha=0.3)

ax6 = plt.subplot2grid((3, 3), (2, 0))
ax6.plot(base_train_losses, label='Без предобучения', linewidth=2)
```

```
ax6.plot(ae_train_losses, label='Autoencoder', linewidth=2)
ax6.plot(rbm_train_losses, label='RBM', linewidth=2)
ax6.set_title('Потери во время обучения', fontweight='bold')
ax6.set_xlabel('Эпоха')
ax6.set_ylabel('Потери')
ax6.legend()
ax6.grid(True, alpha=0.3)

ax7 = plt.subplot2grid((3, 3), (2, 1))
for i, losses in enumerate(ae_losses):
    ax7.plot(losses, label=f'AE Слой {i + 1}', linestyle='--')
for i, losses in enumerate(rbm_losses):
    ax7.plot(losses, label=f'RBM Слой {i + 1}', linestyle='-')
ax7.set_title('Потери при предобучении', fontweight='bold')
ax7.set_xlabel('Эпоха')
ax7.set_ylabel('Потери')
ax7.legend()
ax7.grid(True, alpha=0.3)

ax8 = plt.subplot2grid((3, 3), (2, 2))
improvements_acc = [ae_accuracy - base_accuracy, rbm_accuracy - base_accuracy]
improvements_f1 = [ae_f1 - base_f1, rbm_f1 - base_f1]

x_imp = np.arange(2)
width_imp = 0.35

bars_imp1 = ax8.bar(x_imp - width_imp / 2, improvements_acc, width_imp,
                    label='Улучшение точности', alpha=0.8,
                    color=['green' if x > 0 else 'red' for x in
improvements_acc])
bars_imp2 = ax8.bar(x_imp + width_imp / 2, improvements_f1, width_imp,
                    label='Улучшение F1', alpha=0.8,
                    color=['green' if x > 0 else 'red' for x in
improvements_f1])

ax8.set_title('Улучшение относительно базовой модели', fontweight='bold')
ax8.set_ylabel('Улучшение')
ax8.set_xticks(x_imp)
ax8.set_xticklabels(['Autoencoder', 'RBM'])
ax8.axhline(y=0, color='black', linestyle='-', alpha=0.3)
ax8.legend()
ax8.grid(True, alpha=0.3)

for bars in [bars_imp1, bars_imp2]:
    for bar in bars:
        height = bar.get_height()
        ax8.text(bar.get_x() + bar.get_width() / 2., height + (0.001 if height
>= 0 else -0.01),
                 f'{height:+.3f}', ha='center', va='bottom' if height >= 0 else
'top',
                 fontweight='bold', color='black')

plt.tight_layout()
plt.show()

print(f"\n СРАВНИТЕЛЬНАЯ ТАБЛИЦА РЕЗУЛЬТАТОВ:")
print(f"{'Метод':<25} {'Точность':<12} {'F1-Score':<12} {'Улучшение
точности':<18}")
print(f"{'-' * 70}")
```

```
print(f"{'Без предобучения':<25} {base_accuracy:.4f}     {base_f1:.4f}     {'-
':<18}")
print(f"{'Autoencoder':<25} {ae_accuracy:.4f}     {ae_f1:.4f}     {ae_accuracy
- base_accuracy:+.4f}")
print(f"{'RBM':<25} {rbm_accuracy:.4f}     {rbm_f1:.4f}     {rbm_accuracy -
base_accuracy:+.4f}")

print(f"\n ПАРАМЕТРЫ ЭКСПЕРИМЕНТА:")
print(f"   • Данные: CTG ({X.shape[0]} samples, {X.shape[1]} features)")
print(f"   • Классы: {num_classes} ({class_names})")
print(f"   • Архитектура: {input_dim} → {layer_dims} → {num_classes}")
print(f"   • Эпох предобучения: 50 на слой")
print(f"   • Эпох обучения: {epochs}")

print(f"\n АНАЛИЗ РЕЗУЛЬТАТОВ:")
best_method = np.argmax([base_accuracy, ae_accuracy, rbm_accuracy])
best_method_name = methods[best_method]
best_accuracy = [base_accuracy, ae_accuracy, rbm_accuracy][best_method]

print(f"   • Лучший метод: {best_method_name} (точность: {best_accuracy:.4f})")

if ae_accuracy > base_accuracy and rbm_accuracy > base_accuracy:
    print(f"   • Оба метода предобучения улучшили производительность")
elif ae_accuracy > base_accuracy:
    print(f"   • Только Autoencoder улучшил производительность")
elif rbm_accuracy > base_accuracy:
    print(f"   • Только RBM улучшил производительность")
else:
    print(f"   • Предобучение не дало улучшения")

print(f"\n ВЫВОДЫ:")
print(f"   • RBM показал {'лучшие' if rbm_accuracy > ae_accuracy else 'худшие'}
результаты чем Autoencoder")
print(f"   • Разница между методами: {abs(rbm_accuracy - ae_accuracy):.4f}")
```

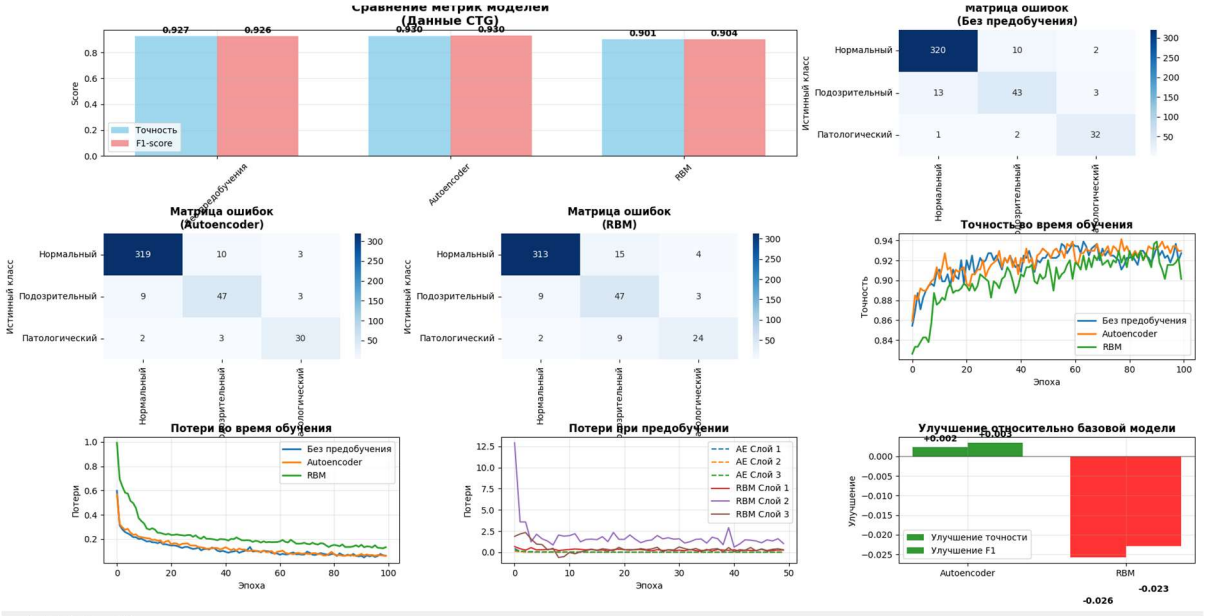
Вывод программы:

```
C:\Users\sasha\PyCharmMiscProject\.venv\Scripts\python.exe
C:\Users\sasha\PyCharmMiscProject\IAD4.py

 Данные CTG успешно загружены

 Данные: 2126 samples, 21 features

 Классы: [0. 1. 2.]

 Распределение классов: [1655  295  176]


==================================================

УЛУЧШЕННАЯ РЕАЛИЗАЦИЯ RBM

==================================================
```

```
================================================================================
СРАВНЕНИЕ МЕТОДОВ ПРЕДОБУЧЕНИЯ НА CTG ДАННЫХ
================================================================================
 Архитектура сети: 21 → [256, 128, 64] → 3
 Эпох обучения: 100
 Обучающая выборка: (1700, 21)
 Тестовая выборка: (426, 21)

==================================================
1.  БАЗОВАЯ МОДЕЛЬ БЕЗ ПРЕДОБУЧЕНИЯ

 Обучение Без предобучения...
 Эпоха [20/100], Потери: 0.1523, Точность: 0.8967
 Эпоха [40/100], Потери: 0.1038, Точность: 0.9155
 Эпоха [60/100], Потери: 0.0820, Точность: 0.9366
 Эпоха [80/100], Потери: 0.0780, Точность: 0.9225
 Эпоха [100/100], Потери: 0.0629, Точность: 0.9272

 Результаты Без предобучения:
   Точность: 0.9272
   F1-score: 0.9264

==================================================
2.  МОДЕЛЬ С ПРЕДОБУЧЕНИЕМ АВТОЭНКОДЕРОМ
 Начало послойного предобучения автоэнкодером...
 Слой 1: 21 → 256
 Предобучение слоя: 21 → 256
    Эпоха [20/50], Потери: 0.0022
    Эпоха [40/50], Потери: 0.0013
 Слой 2: 256 → 128
 Предобучение слоя: 256 → 128
    Эпоха [20/50], Потери: 0.0060
    Эпоха [40/50], Потери: 0.0040
```

```
 Слой 3: 128 → 64
 Предобучение слоя: 128 → 64
    Эпоха [20/50], Потери: 0.0166
    Эпоха [40/50], Потери: 0.0094
 Предобучение автоэнкодером завершено!

 Обучение С предобучением (Autoencoder)...
 Эпоха [20/100], Потери: 0.1867, Точность: 0.9131
 Эпоха [40/100], Потери: 0.1230, Точность: 0.9061
 Эпоха [60/100], Потери: 0.0955, Точность: 0.9390
 Эпоха [80/100], Потери: 0.0956, Точность: 0.9343
 Эпоха [100/100], Потери: 0.0616, Точность: 0.9296

 Результаты С предобучением (Autoencoder):
   Точность: 0.9296
   F1-score: 0.9299
==================================================
3.  МОДЕЛЬ С ПРЕДОБУЧЕНИЕМ RBM
 Начало послойного предобучения RBM...
 Слой 1: 21 → 256
 Обучение RBM: 21 → 256
    Эпоха [10/50], Потери: 0.2658
    Эпоха [20/50], Потери: 0.3639
    Эпоха [30/50], Потери: 0.1810
    Эпоха [40/50], Потери: 0.2289
    Эпоха [50/50], Потери: 0.2556
 Слой 2: 256 → 128
 Обучение RBM: 256 → 128
    Эпоха [10/50], Потери: 1.9013
    Эпоха [20/50], Потери: 1.5209
    Эпоха [30/50], Потери: 1.5396
    Эпоха [40/50], Потери: 2.9061
    Эпоха [50/50], Потери: 1.0293
```

```
Слой 3: 128 → 64
Обучение RBM: 128 → 64
    Эпоха [10/50], Потери: -0.5177
    Эпоха [20/50], Потери: 0.5765
    Эпоха [30/50], Потери: 0.2681
    Эпоха [40/50], Потери: 0.5514
    Эпоха [50/50], Потери: 0.2812
Предобучение RBM завершено!

Обучение С предобучением (RBM)...
Эпоха [20/100], Потери: 0.2440, Точность: 0.8967
Эпоха [40/100], Потери: 0.1992, Точность: 0.8944
Эпоха [60/100], Потери: 0.1764, Точность: 0.9014
Эпоха [80/100], Потери: 0.1469, Точность: 0.9131
Эпоха [100/100], Потери: 0.1339, Точность: 0.9014

Результаты С предобучением (RBM):
  Точность: 0.9014
  F1-score: 0.9036
================================================================================
ВИЗУАЛИЗАЦИЯ РЕЗУЛЬТАТОВ СРАВНЕНИЯ
================================================================================
```



СРАВНИТЕЛЬНАЯ ТАБЛИЦА РЕЗУЛЬТАТОВ:

Метод	Точность	F1-Score	Улучшение точности
Без предобучения	0.9272	0.9264	-
Autoencoder	0.9296	0.9299	+0.0023
RBM	0.9014	0.9036	-0.0258

ПАРАМЕТРЫ ЭКСПЕРИМЕНТА:

- Данные: CTG (2126 samples, 21 features)
- Классы: 3 (['Нормальный', 'Подозрительный', 'Патологический'])
- Архитектура: 21 → [256, 128, 64] → 3
- Эпох предобучения: 50 на слой
- Эпох обучения: 100

АНАЛИЗ РЕЗУЛЬТАТОВ:

- Лучший метод: Autoencoder (точность: 0.9296)
- Только Autoencoder улучшил производительность

ВЫВОДЫ:

- RBM показал худшие результаты чем Autoencoder
- Разница между методами: 0.0282

Вывод: научился осуществлять предобучение нейронных сетей с помощью RBM