`

Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

Отчет по лабораторной работе 4

Специальность ИИ-23

Выполнил:

Гавришук В.Р.

Студент группы ИИ-23

Проверил:

Андренко К. В.
Преподаватель-стажёр
Кафедры ИИТ,
«___» ____________2025 г.

Брест 2025

`

Лабораторная работа № 4. Предобучение нейронных сетей с использованием RBM

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

№ варианта	Выборка	Тип задачи	Целевая переменная
4	https://archive.ics.uci.edu/dataset/925/infrared+thermography+temperature+dataset	регрессия	aveOralF/aveOralM

Ход работы:

Выборка с температурой:

```
TARGET = "aveOralF"
RANDOM_SEED = 42
BATCH_SIZE = 32
EPOCHS = 100
LR = 1e-4
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

RBM_LR = 1e-3
RBM_CD_K = 1
RBM_EPOCHS_FIRST = 50
RBM_EPOCHS_OTHER = 30

torch.manual_seed(RANDOM_SEED)
```

`

```
np.random.seed(RANDOM_SEED)

print("Fetching dataset from UCI...")
infrared = fetch_ucirepo(id=925)
X_df = infrared.data.features.copy()
y_df = infrared.data.targets.copy()

if TARGET not in y_df.columns:
    raise ValueError(f"Target {TARGET} not found. Available: {list(y_df.columns)}")

y = y_df[TARGET]
data = pd.concat([X_df, y], axis=1).replace([np.inf, -np.inf], np.nan)
data = data.dropna()

X_df = data.drop(columns=[TARGET])
y = data[TARGET].values

print(f"Dataset cleaned: X={X_df.shape}, y={y.shape}")

y_mean, y_std = y.mean(), y.std()
y_norm = (y - y_mean) / (y_std + 1e-8)

categorical_cols = []
numeric_cols = []
for col in X_df.columns:
    if X_df[col].dtype == object:
        categorical_cols.append(col)
    else:
        numeric_cols.append(col)

for c in ["Gender", "Age", "Ethnicity"]:
    if c in X_df.columns and c not in categorical_cols:
        categorical_cols.append(c)
        if c in numeric_cols:
            numeric_cols.remove(c)

print("Categorical columns:", categorical_cols)
print("Numeric columns:", numeric_cols[:10], " (total:", len(numeric_cols), ")")

cat_pipe = Pipeline([
    ("ohe", OneHotEncoder(sparse=False, handle_unknown="ignore"))
])
num_pipe = Pipeline([
    ("scaler", StandardScaler())
])
preprocessor = ColumnTransformer([
    ("num", num_pipe, numeric_cols),
    ("cat", cat_pipe, categorical_cols)
])

X_train_df, X_test_df, y_train, y_test = train_test_split(
    X_df, y_norm, test_size=0.2, random_state=RANDOM_SEED
)
X_train = preprocessor.fit_transform(X_train_df)
X_test = preprocessor.transform(X_test_df)

print("Feature matrix shapes after preprocessing:", X_train.shape, X_test.shape)

class TabularDataset(Dataset):
    def __init__(self, X, y=None):
        self.X = X.astype(np.float32)
        self.y = None if y is None else y.astype(np.float32).reshape(-1, 1)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        if self.y is None:
```

`

```
            return self.X[idx]
        return self.X[idx], self.y[idx]


train_ds = TabularDataset(X_train, y_train)
test_ds = TabularDataset(X_test, y_test)
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False)
train_x_loader = DataLoader(TabularDataset(X_train), batch_size=BATCH_SIZE, shuffle=True)

in_features = X_train.shape[1]
print("Input features:", in_features)

class RBM(nn.Module):
    def __init__(self, n_visible, n_hidden, visible_type='bernoulli', device=None):
        super().__init__()
        self.n_visible = n_visible
        self.n_hidden = n_hidden
        self.visible_type = visible_type
        self.device = device if device is not None else torch.device('cpu')

        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))

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

    def sample_v(self, h):
        pre = torch.matmul(h, self.W.t()) + self.v_bias
        if self.visible_type == 'bernoulli':
            p_v = torch.sigmoid(pre)
            return p_v, torch.bernoulli(p_v)
        elif self.visible_type == 'gaussian':
            mean = pre
            sample = mean + torch.randn_like(mean)
            return mean, sample
        else:
            raise ValueError("visible_type must be 'bernoulli' or 'gaussian'")

    def free_energy(self, v):
        if self.visible_type == 'gaussian':
            vbias_term = ((v - self.v_bias) ** 2).sum(dim=1)
        else:
            vbias_term = torch.matmul(v, self.v_bias)
        wx_b = torch.matmul(v, self.W) + self.h_bias
        hidden_term = torch.sum(torch.log1p(torch.exp(wx_b)), dim=1)
        return -vbias_term - hidden_term

    def forward(self, v):
        p_h, _ = self.sample_h(v)
        return p_h

    def contrastive_divergence(self, v0, k=1, lr=1e-3):
        batch_size = v0.size(0)
        v = v0
        p_h0 = torch.sigmoid(torch.matmul(v, self.W) + self.h_bias)
        h0 = torch.bernoulli(p_h0)

        hk = h0
        vk = None
        for step in range(k):
            mean_vk, vk = self.sample_v(hk)
            p_hk, hk = self.sample_h(vk)

        pos_assoc = torch.matmul(v.t(), p_h0)
```

`

```
        neg_assoc = torch.matmul(vk.t(), p_hk)

        dW = (pos_assoc - neg_assoc) / batch_size
        dv_bias = torch.mean(v - vk, dim=0)
        dh_bias = torch.mean(p_h0 - p_hk, dim=0)

        self.W.data += lr * dW
        self.v_bias.data += lr * dv_bias
        self.h_bias.data += lr * dh_bias

        if self.visible_type == 'gaussian':
            recon_error = torch.mean((v0 - mean_vk) ** 2).item()
        else:
            recon_error = torch.mean((v0 - vk) ** 2).item()

        return recon_error


def pretrain_rbms(X_numpy, layer_sizes, device):
    rbms = []
    representations = []
    current_data = torch.tensor(X_numpy, dtype=torch.float32, device=device)

    for i, h_dim in enumerate(layer_sizes):
        v_dim = current_data.shape[1]
        if i == 0:
            vis_type = 'gaussian'
            epochs = RBM_EPOCHS_FIRST
        else:
            vis_type = 'bernoulli'
            epochs = RBM_EPOCHS_OTHER

        print(f"\nPretraining RBM layer {i + 1}/{len(layer_sizes)}: visible={v_dim}, hidden={h_dim},
visible_type={vis_type}, epochs={epochs}")
        rbm = RBM(n_visible=v_dim, n_hidden=h_dim, visible_type=vis_type, device=device).to(device)

        ds = TabularDataset(current_data.cpu().numpy())
        loader = DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True)

        for ep in range(1, epochs + 1):
            epoch_err = 0.0
            nb = 0
            for batch in loader:
                v0 = batch.to(device)
                err = rbm.contrastive_divergence(v0, k=RBM_CD_K, lr=RBM_LR)
                epoch_err += err * v0.size(0)
                nb += v0.size(0)
            epoch_err /= nb
            if ep == 1 or ep % 10 == 0 or ep == epochs:
                print(f"  RBM layer {i + 1} epoch {ep:3d} | recon MSE: {epoch_err:.6f}")

        rbms.append(rbm)

        with torch.no_grad():
            p_h = torch.sigmoid(torch.matmul(current_data, rbm.W) + rbm.h_bias)
            current_data = p_h
            representations.append(p_h.cpu().numpy())

    return rbms, representations


class FCRegressor(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 128),
            nn.ReLU(),
            nn.BatchNorm1d(128),
```

`

```
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.BatchNorm1d(64),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1)
        )

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


layer_sizes = [128, 64, 32, 16]
rbms, representations = pretrain_rbms(X_train, layer_sizes, DEVICE)
print("\nFinished pretraining RBMs.")

model = FCRegressor(in_features).to(DEVICE)

linear_layers = [m for m in model.modules() if isinstance(m, nn.Linear)]
assert len(linear_layers) >= len(rbms), "Linear layers less than RBMs - architecture mismatch"

for i, rbm in enumerate(rbms):
    lin = linear_layers[i]
    W_t = rbm.W.t().cpu().detach()
    h_b = rbm.h_bias.cpu().detach()
    if lin.weight.shape == W_t.shape:
        lin.weight.data.copy_(W_t)
    else:
        print(f"Warning: shape mismatch when assigning weights to layer {i}: lin.weight
{lin.weight.shape}, W_t {W_t.shape}")
    if lin.bias is not None and lin.bias.shape[0] == h_b.shape[0]:
        lin.bias.data.copy_(h_b)
    else:
        print(f"Warning: bias mismatch for layer {i}")

print("Initialized FC model weights from RBMs.")
```

Выборка с качеством вина:

```
TARGET = "quality"
RANDOM_SEED = 42
BATCH_SIZE = 32
EPOCHS = 100
LR = 1e-4
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

RBM_LR = 1e-3
RBM_CD_K = 1
RBM_EPOCHS_FIRST = 50
RBM_EPOCHS_OTHER = 30

torch.manual_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

print("Fetching Wine Quality dataset...")
wine = fetch_ucirepo(id=186)
X_df = wine.data.features.copy()
y_df = wine.data.targets.copy()

if TARGET not in y_df.columns:
    raise ValueError(f"Target {TARGET} not found. Available: {list(y_df.columns)}")

y = y_df[TARGET].values
X = X_df.values.astype(np.float32)
```

`

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
y_mean, y_std = y.mean(), y.std()
y_norm = (y - y_mean) / (y_std + 1e-8)

X_train, X_test, y_train, y_test = train_test_split(
    X, y_norm, test_size=0.2, random_state=RANDOM_SEED
)

class TabularDataset(Dataset):
    def __init__(self, X, y=None):
        self.X = X.astype(np.float32)
        self.y = None if y is None else y.astype(np.float32).reshape(-1, 1)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        if self.y is None:
            return self.X[idx]
        return self.X[idx], self.y[idx]


train_ds = TabularDataset(X_train, y_train)
test_ds = TabularDataset(X_test, y_test)
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False)
train_x_loader = DataLoader(TabularDataset(X_train), batch_size=BATCH_SIZE, shuffle=True)

in_features = X_train.shape[1]
print("Input features:", in_features)


class RBM(nn.Module):
    def __init__(self, n_visible, n_hidden, visible_type='gaussian', device=None):
        super().__init__()
        self.n_visible = n_visible
        self.n_hidden = n_hidden
        self.visible_type = visible_type
        self.device = device if device is not None else torch.device("cpu")

        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))

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

    def sample_v(self, h):
        pre = torch.matmul(h, self.W.t()) + self.v_bias
        if self.visible_type == 'bernoulli':
            p_v = torch.sigmoid(pre)
            return p_v, torch.bernoulli(p_v)
        elif self.visible_type == 'gaussian':
            mean = pre
            sample = mean + torch.randn_like(mean)
            return mean, sample
        else:
            raise ValueError("visible_type must be 'bernoulli' or 'gaussian'")

    def contrastive_divergence(self, v0, k=1, lr=1e-3):
        batch_size = v0.size(0)
        v = v0
        p_h0 = torch.sigmoid(torch.matmul(v, self.W) + self.h_bias)
        h0 = torch.bernoulli(p_h0)
```

`

```
        hk = h0
        vk = None
        for step in range(k):
            mean_vk, vk = self.sample_v(hk)
            p_hk, hk = self.sample_h(vk)

        pos_assoc = torch.matmul(v.t(), p_h0)
        neg_assoc = torch.matmul(vk.t(), p_hk)

        dW = (pos_assoc - neg_assoc) / batch_size
        dv_bias = torch.mean(v - vk, dim=0)
        dh_bias = torch.mean(p_h0 - p_hk, dim=0)

        self.W.data += lr * dW
        self.v_bias.data += lr * dv_bias
        self.h_bias.data += lr * dh_bias

        if self.visible_type == 'gaussian':
            recon_error = torch.mean((v0 - mean_vk) ** 2).item()
        else:
            recon_error = torch.mean((v0 - vk) ** 2).item()

        return recon_error


def pretrain_rbms(X_numpy, layer_sizes, device):
    rbms = []
    current_data = torch.tensor(X_numpy, dtype=torch.float32, device=device)

    for i, h_dim in enumerate(layer_sizes):
        v_dim = current_data.shape[1]
        if i == 0:
            vis_type = 'gaussian'
            epochs = RBM_EPOCHS_FIRST
        else:
            vis_type = 'bernoulli'
            epochs = RBM_EPOCHS_OTHER

        print(f"\nPretraining RBM {i + 1}/{len(layer_sizes)}: visible={v_dim}, hidden={h_dim},
type={vis_type}")
        rbm = RBM(n_visible=v_dim, n_hidden=h_dim, visible_type=vis_type, device=device).to(device)

        ds = TabularDataset(current_data.cpu().numpy())
        loader = DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True)

        for ep in range(1, epochs + 1):
            epoch_err = 0.0
            nb = 0
            for batch in loader:
                v0 = batch.to(device)
                err = rbm.contrastive_divergence(v0, k=RBM_CD_K, lr=RBM_LR)
                epoch_err += err * v0.size(0)
                nb += v0.size(0)
            epoch_err /= nb
            if ep == 1 or ep % 10 == 0 or ep == epochs:
                print(f"  Epoch {ep:3d} | Recon MSE: {epoch_err:.6f}")

        with torch.no_grad():
            p_h, _ = rbm.sample_h(current_data)
            current_data = p_h

        rbms.append(rbm)

    return rbms

class FCRegressor(nn.Module):
    def __init__(self, in_dim):
```

`

```
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 128),
            nn.ReLU(),
            nn.BatchNorm1d(128),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.BatchNorm1d(64),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1)
        )

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


layer_sizes = [128, 64, 32, 16]
rbms = pretrain_rbms(X_train, layer_sizes, DEVICE)
print("\nFinished pretraining RBMs.")


model = FCRegressor(in_features).to(DEVICE)

linear_layers = [m for m in model.modules() if isinstance(m, nn.Linear)]
assert len(linear_layers) >= len(rbms), "Linear layers < RBMs"

for i, rbm in enumerate(rbms):
    lin = linear_layers[i]
    W_t = rbm.W.t().cpu().detach()
    h_b = rbm.h_bias.cpu().detach()
    if lin.weight.shape == W_t.shape:
        lin.weight.data.copy_(W_t)
    else:
        print(f" Shape mismatch at layer {i}: {lin.weight.shape} vs {W_t.shape}")
    if lin.bias is not None and lin.bias.shape[0] == h_b.shape[0]:
        lin.bias.data.copy_(h_b)

print("Initialized model weights from pretrained RBMs.")


criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode="min", patience=8, factor=0.5)
```

Результат работы программы:

Выборка с температурой:

```
Pretraining RBM layer 1/4: visible=46, hidden=128, visible_type=gaussian, epochs=50

  RBM layer 1 epoch   1 | recon MSE: 0.703566

  RBM layer 1 epoch  10 | recon MSE: 0.663555

  RBM layer 1 epoch  20 | recon MSE: 0.476659

  RBM layer 1 epoch  30 | recon MSE: 0.300904

  RBM layer 1 epoch  40 | recon MSE: 0.287099

  RBM layer 1 epoch  50 | recon MSE: 0.286975

Pretraining RBM layer 2/4: visible=128, hidden=64, visible_type=bernoulli, epochs=30

  RBM layer 2 epoch   1 | recon MSE: 0.276963
```

`

```
  RBM layer 2 epoch  10 | recon MSE: 0.277161
  RBM layer 2 epoch  20 | recon MSE: 0.276183
  RBM layer 2 epoch  30 | recon MSE: 0.276385
Pretraining RBM layer 3/4: visible=64, hidden=32, visible_type=bernoulli, epochs=30
  RBM layer 3 epoch   1 | recon MSE: 0.250314
  RBM layer 3 epoch  10 | recon MSE: 0.250076
  RBM layer 3 epoch  20 | recon MSE: 0.249922
  RBM layer 3 epoch  30 | recon MSE: 0.249898
Pretraining RBM layer 4/4: visible=32, hidden=16, visible_type=bernoulli, epochs=30
  RBM layer 4 epoch   1 | recon MSE: 0.250070
  RBM layer 4 epoch  10 | recon MSE: 0.249973
  RBM layer 4 epoch  20 | recon MSE: 0.249876
  RBM layer 4 epoch  30 | recon MSE: 0.249953

Finished pretraining RBMs.
Initialized FC model weights from RBMs.
Epoch   1 | Train MSE: 1.065370 | Val MSE: 0.723301
Epoch  10 | Train MSE: 0.547689 | Val MSE: 0.344510
Epoch  20 | Train MSE: 0.358021 | Val MSE: 0.322501
Epoch  30 | Train MSE: 0.308996 | Val MSE: 0.316003
Epoch  40 | Train MSE: 0.271722 | Val MSE: 0.334508
Epoch  50 | Train MSE: 0.267370 | Val MSE: 0.319342
Epoch  60 | Train MSE: 0.271190 | Val MSE: 0.338369
Epoch  70 | Train MSE: 0.261766 | Val MSE: 0.337625
Epoch  80 | Train MSE: 0.247283 | Val MSE: 0.338336
Epoch  90 | Train MSE: 0.265363 | Val MSE: 0.327035
Epoch 100 | Train MSE: 0.273899 | Val MSE: 0.334618
Sample predictions (first 10):
   y_true    y_pred
36.850002 36.982014
36.750000 36.803013
36.850002 36.876415
36.650002 36.934189
37.550003 38.510044
37.250000 37.515713
36.750000 36.901562
36.850002 36.740887
37.150002 36.928856
37.100002 36.823154
```

Выборка с качеством вина:

`

```
Pretraining RBM 1/4: visible=11, hidden=128, type=gaussian
  Epoch   1 | Recon MSE: 0.988136
  Epoch  10 | Recon MSE: 0.978129
  Epoch  20 | Recon MSE: 0.943695
  Epoch  30 | Recon MSE: 0.838537
  Epoch  40 | Recon MSE: 0.738106
  Epoch  50 | Recon MSE: 0.690689
Pretraining RBM 2/4: visible=128, hidden=64, type=bernoulli
  Epoch   1 | Recon MSE: 0.255985
  Epoch  10 | Recon MSE: 0.255898
  Epoch  20 | Recon MSE: 0.255712
  Epoch  30 | Recon MSE: 0.255896
Pretraining RBM 3/4: visible=64, hidden=32, type=bernoulli
  Epoch   1 | Recon MSE: 0.250068
  Epoch  10 | Recon MSE: 0.249904
  Epoch  20 | Recon MSE: 0.249897
  Epoch  30 | Recon MSE: 0.249945
Pretraining RBM 4/4: visible=32, hidden=16, type=bernoulli
  Epoch   1 | Recon MSE: 0.250024
  Epoch  10 | Recon MSE: 0.249951
  Epoch  20 | Recon MSE: 0.249966
  Epoch  30 | Recon MSE: 0.249973

Finished pretraining RBMs.
Initialized model weights from pretrained RBMs.
Epoch   1 | Train MSE: 0.955982 | Val MSE: 0.789334
Epoch  10 | Train MSE: 0.641580 | Val MSE: 0.616697
Epoch  20 | Train MSE: 0.603817 | Val MSE: 0.606996
Epoch  30 | Train MSE: 0.592281 | Val MSE: 0.587153
Epoch  40 | Train MSE: 0.575033 | Val MSE: 0.592050
Epoch  50 | Train MSE: 0.550654 | Val MSE: 0.587816
Epoch  60 | Train MSE: 0.513500 | Val MSE: 0.588377
Epoch  70 | Train MSE: 0.492217 | Val MSE: 0.578146
Epoch  80 | Train MSE: 0.483889 | Val MSE: 0.586919
Epoch  90 | Train MSE: 0.474817 | Val MSE: 0.586109
Epoch 100 | Train MSE: 0.469056 | Val MSE: 0.588792
Sample predictions (first 10):
 y_true   y_pred
 8.000 6.097408
 5.000 5.077805
```

```
`
 7.000 6.869720
 6.000 5.602548
 6.000 5.230566
 6.000 6.399209
 5.000 5.326153
 6.000 6.463978
 5.000 4.996253
 7.000 6.394671
```

Можно сделать вывод: особых отличий не наблюдается по следующим причинам:
1. Модель без предобучения уже обучается хорошо;

2. И автоэнкодер, и RBM фактически пытаются найти скрытые представления X**,** которые восстанавливают сами X.

Вывод: научился осуществлять предобучение нейронных сетей с помощью RBM.