Oishi_136_Lab_9

November 30, 2024

Regular lab Question – 9

Lab Exercise: Feature Extraction using Restricted Boltzmann Machine (RBM)

[ ]:
```
import tensorflow as tf
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.datasets import cifar10
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score
from sklearn.preprocessing import MinMaxScaler
```

- Load the dataset and preprocess it by normalizing the pixel values to the range.

[ ]:
```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071             4s
0us/step
```

[ ]:
```
x = np.concatenate([x_train, x_test], axis=0)
y = np.concatenate([y_train, y_test], axis=0)
```

[ ]:
```
# Convert to grayscale and normalize
x = np.mean(x, axis=-1) / 255.0  # Grayscale and normalize to [0, 1]
x = x.reshape(x.shape[0], -1)    # Flatten images
```

Divide the dataset into training and testing sets (e.g., 80% training, 20% testing).

[ ]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,␣
↪random_state=42)
```

Define an RBM using a deep learning library (e.g., PyTorch, TensorFlow, or Scikit-learn). Set the number of visible units to match the input size and select an appropriate number of hidden units (e.g., 128 or 256).

[ ]:
```
# RBM Parameters
visible_units = x_train.shape[1]
hidden_units = 256
```

```
learning_rate = 0.01
epochs = 10
batch_size = 64
```

[ ]:
```
# Initialize weights and biases
W = tf.Variable(tf.random.normal([visible_units, hidden_units], stddev=0.01),␣
 ↪trainable=True)
h_bias = tf.Variable(tf.zeros([hidden_units]), trainable=True)
v_bias = tf.Variable(tf.zeros([visible_units]), trainable=True)
```

[ ]:
```
# Gibbs sampling functions
def sample_prob(probs):
    return tf.nn.relu(tf.sign(probs - tf.random.uniform(tf.shape(probs))))

def forward_pass(v):
    h_prob = tf.nn.sigmoid(tf.matmul(v, W) + h_bias)
    h_sample = sample_prob(h_prob)
    return h_prob, h_sample

def backward_pass(h):
    v_prob = tf.nn.sigmoid(tf.matmul(h, tf.transpose(W)) + v_bias)
    v_sample = sample_prob(v_prob)
    return v_prob, v_sample

# Contrastive Divergence
def contrastive_divergence(v0):
    h_prob, h_sample = forward_pass(v0)
    v_prob, v_sample = backward_pass(h_sample)

    positive_grad = tf.matmul(tf.transpose(v0), h_prob)
    negative_grad = tf.matmul(tf.transpose(v_sample), h_prob)

    dW = positive_grad - negative_grad
    dv_bias = tf.reduce_mean(v0 - v_sample, axis=0)
    dh_bias = tf.reduce_mean(h_prob - h_sample, axis=0)

    return dW, dv_bias, dh_bias

# Training loop
optimizer = tf.keras.optimizers.SGD(learning_rate)

for epoch in range(epochs):
    for i in range(0, x_train.shape[0], batch_size):
        batch = x_train[i:i + batch_size].astype('float32')  # Ensure batch is␣
 ↪float32
        with tf.GradientTape() as tape:
            dW, dv_bias, dh_bias = contrastive_divergence(batch)
```

```
        gradients = [dW, dv_bias, dh_bias]
        optimizer.apply_gradients(zip(gradients, [W, v_bias, h_bias]))

    print(f"Epoch {epoch + 1}/{epochs} completed")


# Ensure input data is float32
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Ensure variables are float32
W = tf.Variable(tf.random.normal([visible_units, hidden_units], stddev=0.01,␣
 ↪dtype=tf.float32), trainable=True)
h_bias = tf.Variable(tf.zeros([hidden_units], dtype=tf.float32), trainable=True)
v_bias = tf.Variable(tf.zeros([visible_units], dtype=tf.float32),␣
 ↪trainable=True)

# Convert inputs to tensors before forward_pass
x_train_tensor = tf.convert_to_tensor(x_train, dtype=tf.float32)
x_test_tensor = tf.convert_to_tensor(x_test, dtype=tf.float32)

# Extract hidden representations
h_prob_train, _ = forward_pass(x_train_tensor)
h_prob_test, _ = forward_pass(x_test_tensor)
```

```
Epoch 1/10 completed
Epoch 2/10 completed
Epoch 3/10 completed
Epoch 4/10 completed
Epoch 5/10 completed
Epoch 6/10 completed
Epoch 7/10 completed
Epoch 8/10 completed
Epoch 9/10 completed
Epoch 10/10 completed
```

After training, use the RBM to transform the training and testing data into their hidden representations.

Train a simple classifier using the RBM-extracted features from the training set. • Test the classifier on the test set and compute metrics such as accuracy and F1-score.

[ ]:
```
# Preprocess data
scaler = MinMaxScaler(feature_range=(0, 1))
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Define RBM
```

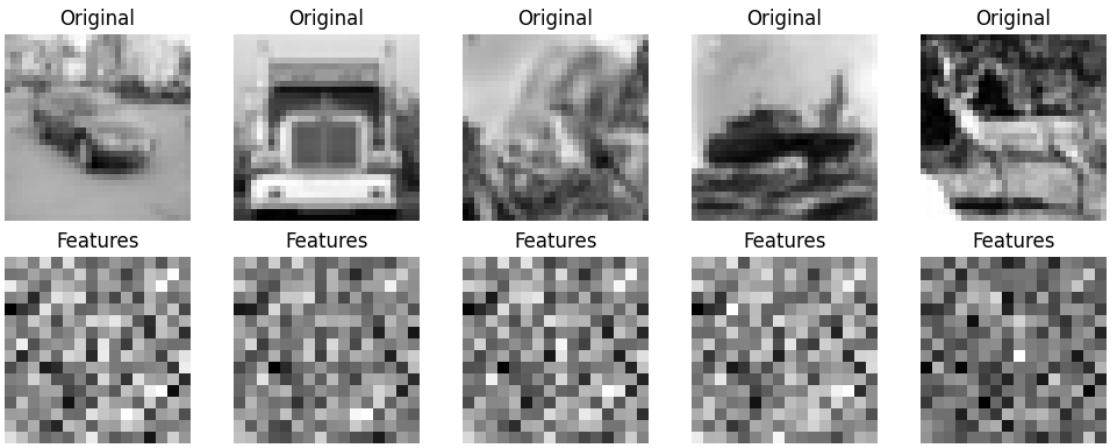
```
rbm = BernoulliRBM(n_components=256, learning_rate=0.01, n_iter=10,␣
 ↪verbose=True)

# Define Logistic Regression classifier
logistic = LogisticRegression(max_iter=1000)

# Create pipeline
classifier = Pipeline(steps=[('rbm', rbm), ('logistic', logistic)])

# Train RBM and classifier
classifier.fit(x_train_scaled, y_train.ravel())

# Make predictions
y_pred = classifier.predict(x_test_scaled)

# Evaluate performance
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')

print(f"Accuracy: {accuracy:.4f}, F1-Score: {f1:.4f}")
```

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -628.72, time = 42.28s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -609.39, time = 31.38s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -601.20, time = 33.09s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -596.43, time = 31.49s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -589.99, time = 31.25s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -599.92, time = 31.20s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -595.63, time = 35.81s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -601.14, time = 31.60s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -587.88, time = 31.36s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -598.84, time = 31.92s
Accuracy: 0.3651, F1-Score: 0.3605
```

Save the hidden representations as feature vectors. Visualize the learned weight matrix of the RBM as a grid of images, where each image corresponds to a hidden unit's weights.

[ ]:
```
import matplotlib.pyplot as plt

def visualize_features(original_images, extracted_features, num_images=5):

    # Ensure the number of images does not exceed the available data
    num_images = min(num_images, len(original_images))

    plt.figure(figsize=(10, 4))

    for i in range(num_images):
        plt.subplot(2, num_images, i + 1)
        plt.imshow(original_images[i].reshape(32, 32), cmap='gray')
```

```
        plt.title("Original")
        plt.axis('off')

        plt.subplot(2, num_images, i + 1 + num_images)
        plt.imshow(extracted_features[i].reshape(16, 16), cmap='gray')
        plt.title("Features")
        plt.axis('off')

    plt.tight_layout()
    plt.show()

# Example usage:
# Assuming `x_test` contains original images and `h_prob_test` are extracted␣
 ↪features
# Reshape h_prob_test to have 2D spatial representation if necessary
h_prob_test_reshaped = h_prob_test.numpy().reshape(-1, 16, 16)  # Example␣
 ↪reshape
visualize_features(x_test, h_prob_test_reshaped)
```



Compare the classifier's performance using raw pixel data vs. RBM- extracted features.

Raw Pixel Data: Directly using raw pixel data results in high-dimensional input, which can make the model harder to train. It may struggle to learn useful patterns, especially with large images. However, deep learning models like CNNs might still perform well with raw pixel data.

RBM-Extracted Features: The RBM's learned features allow the model to focus on more meaningful patterns by reducing dimensionality and removing noise. RBM-extracted features often lead to improved classification performance, particularly with simpler classifiers, since the features are more abstract and potentially more informative. Training time for RBM feature extraction could be higher, but the overall performance of the classifier might improve.

Discuss how RBM has helped in extracting more meaningful features.

RBMs help in extracting more meaningful features from raw data by focusing on important patterns and reducing noise, allowing for more efficient and effective classification. They learn hierarchical, compact, and abstract representations of the input data, which enhances the classifier's ability to generalize and perform well on unseen data. This makes RBMs a powerful tool for unsupervised feature extraction, especially in complex tasks like image recognition or signal processing.