Greeshma_115_lab4

October 18, 2024

Data Preparation

[13]:
```
import tensorflow as tf
import tensorflow_datasets as tfds
from sklearn.model_selection import train_test_split
import numpy as np

#KMNIST dataset
kmnist_data, kmnist_info = tfds.load('kmnist', split=['train', 'test'],␣
 ↪as_supervised=True, with_info=True)

# Preprocessing the data
def preprocess_image(image, label):
    image = tf.cast(image, tf.float32) / 255.0
    image = tf.image.resize(image, [28, 28])
    return image, label

# Applying preprocessing to the training and test datasets
train_data = kmnist_data[0].map(preprocess_image)
test_data = kmnist_data[1].map(preprocess_image)

# Preparing training data
x_train, y_train = [], []
for img, label in train_data:
    x_train.append(img.numpy().flatten())
    y_train.append(label.numpy())

x_train = np.array(x_train)
y_train = np.array(y_train)

# Splitting
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.
 ↪2, random_state=42)

# Preparing test data
x_test, y_test = [], []
for img, label in test_data:
    x_test.append(img.numpy().flatten())
```

```
    y_test.append(label.numpy())

x_test = np.array(x_test)
y_test = np.array(y_test)

print(f'Training data shape: {x_train.shape}, Validation data shape: {x_val.
 ↪shape}, Test data shape: {x_test.shape}')
```

```
Training data shape: (48000, 784), Validation data shape: (12000, 784), Test
data shape: (10000, 784)
```

Radial Basis Function (RBF) Network

[14]:
```
import tensorflow as tf
from tensorflow.keras import layers, models

class RBF(tf.keras.Model):
    def __init__(self, num_hidden_units):
        super(RBF, self).__init__()
        self.num_hidden_units = num_hidden_units
        self.centers = tf.Variable(tf.random.uniform((num_hidden_units, 784),␣
 ↪0, 1))
        self.bias = tf.Variable(tf.zeros(num_hidden_units))
        self.output_layer = layers.Dense(10, activation='softmax')

    def call(self, x):
        distance = tf.norm(tf.expand_dims(x, axis=1) - self.centers, axis=2)
        rbf_output = tf.exp(-tf.square(distance))
        return self.output_layer(rbf_output + self.bias)

num_hidden_units = 64
model = RBF(num_hidden_units)
```

Training

[15]:
```
from sklearn.cluster import KMeans

#K-means to find centers
kmeans = KMeans(n_clusters=num_hidden_units, random_state=42)
kmeans.fit(x_train)
model.centers.assign(kmeans.cluster_centers_)
```

[15]:
```
<tf.Variable 'UnreadVariable' shape=(64, 784) dtype=float32, numpy=
array([[6.17648300e-04, 1.40196760e-03, 8.03930685e-04, …,
        4.15686565e-03, 1.27460342e-04, 5.82076609e-09],
       [8.38190317e-09, 3.75050120e-04, 1.08724297e-03, …,
        7.87988305e-04, 4.79559880e-04, 1.99393253e-04],
       [1.15207734e-03, 1.24242867e-03, 1.14757824e-03, …,
        3.04045249e-03, 2.19123019e-03, 2.09632097e-03],
```

```
        …,
        [4.99999907e-04, 2.53921631e-03, 6.05882332e-03, …,
         5.16666938e-03, 1.92155363e-03, 6.86321873e-05],
        [1.65940262e-03, 2.05652090e-03, 1.47504290e-03, …,
         4.72290441e-03, 2.28345767e-03, 5.31865400e-04],
        [6.75208867e-09, 7.27474689e-05, 7.56683294e-04, …,
         9.38490964e-04, 1.17866416e-03, 1.16416835e-04]], dtype=float32)>
```

[16]:
```python
#loss function and optimizer
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

# Training loop
epochs = 100
batch_size = 64

for epoch in range(epochs):
    indices = np.arange(x_train.shape[0])
    np.random.shuffle(indices)

    for i in range(0, len(x_train), batch_size):
        batch_indices = indices[i:i + batch_size]
        with tf.GradientTape() as tape:
            predictions = model(x_train[batch_indices])
            loss = loss_fn(y_train[batch_indices], predictions)

        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    print(f'Epoch {epoch + 1}/{epochs}, Loss: {loss.numpy()}')
```

```
Epoch 1/100, Loss: 2.305028200149536
Epoch 2/100, Loss: 2.306403160095215
Epoch 3/100, Loss: 2.2930941581726074
Epoch 4/100, Loss: 2.2994189262390137
Epoch 5/100, Loss: 2.3085498809814453
Epoch 6/100, Loss: 2.3034515380859375
Epoch 7/100, Loss: 2.305459499359131
Epoch 8/100, Loss: 2.3018746376037598
Epoch 9/100, Loss: 2.302731513977051
Epoch 10/100, Loss: 2.301572799682617
Epoch 11/100, Loss: 2.3045997619628906
Epoch 12/100, Loss: 2.298118829727173
Epoch 13/100, Loss: 2.301370143890381
Epoch 14/100, Loss: 2.3014252185821533
Epoch 15/100, Loss: 2.301734685897827
Epoch 16/100, Loss: 2.3037033081054688
Epoch 17/100, Loss: 2.315609931945801
```

```
Epoch 18/100, Loss: 2.301865339279175
Epoch 19/100, Loss: 2.311182975769043
Epoch 20/100, Loss: 2.2949109077453613
Epoch 21/100, Loss: 2.3115458488464355
Epoch 22/100, Loss: 2.3113231658935547
Epoch 23/100, Loss: 2.303136110305786
Epoch 24/100, Loss: 2.3034801483154297
Epoch 25/100, Loss: 2.300459623336792
Epoch 26/100, Loss: 2.299020767211914
Epoch 27/100, Loss: 2.3166985511779785
Epoch 28/100, Loss: 2.3065543174743652
Epoch 29/100, Loss: 2.3073058128356934
Epoch 30/100, Loss: 2.303872585296631
Epoch 31/100, Loss: 2.3092918395996094
Epoch 32/100, Loss: 2.3076486587524414
Epoch 33/100, Loss: 2.305048942565918
Epoch 34/100, Loss: 2.299511671066284
Epoch 35/100, Loss: 2.3062925338745117
Epoch 36/100, Loss: 2.30437970161438
Epoch 37/100, Loss: 2.296316146850586
Epoch 38/100, Loss: 2.3096189498901367
Epoch 39/100, Loss: 2.3033275604248047
Epoch 40/100, Loss: 2.3022143840789795
Epoch 41/100, Loss: 2.307703733444214
Epoch 42/100, Loss: 2.307063579559326
Epoch 43/100, Loss: 2.30118465423584
Epoch 44/100, Loss: 2.3080782890319824
Epoch 45/100, Loss: 2.301595449447632
Epoch 46/100, Loss: 2.308382749557495
Epoch 47/100, Loss: 2.3054404258728027
Epoch 48/100, Loss: 2.299008369445801
Epoch 49/100, Loss: 2.3039350509643555
Epoch 50/100, Loss: 2.2991509437561035
Epoch 51/100, Loss: 2.3105688095092773
Epoch 52/100, Loss: 2.3065173625946045
Epoch 53/100, Loss: 2.3058595657348633
Epoch 54/100, Loss: 2.3011860847473145
Epoch 55/100, Loss: 2.300438404083252
Epoch 56/100, Loss: 2.312925100326538
Epoch 57/100, Loss: 2.304826021194458
Epoch 58/100, Loss: 2.295588970184326
Epoch 59/100, Loss: 2.3063066005706787
Epoch 60/100, Loss: 2.309782028198242
Epoch 61/100, Loss: 2.307143211364746
Epoch 62/100, Loss: 2.2965612411499023
Epoch 63/100, Loss: 2.307361125946045
Epoch 64/100, Loss: 2.3023550510406494
Epoch 65/100, Loss: 2.296114921569824
```

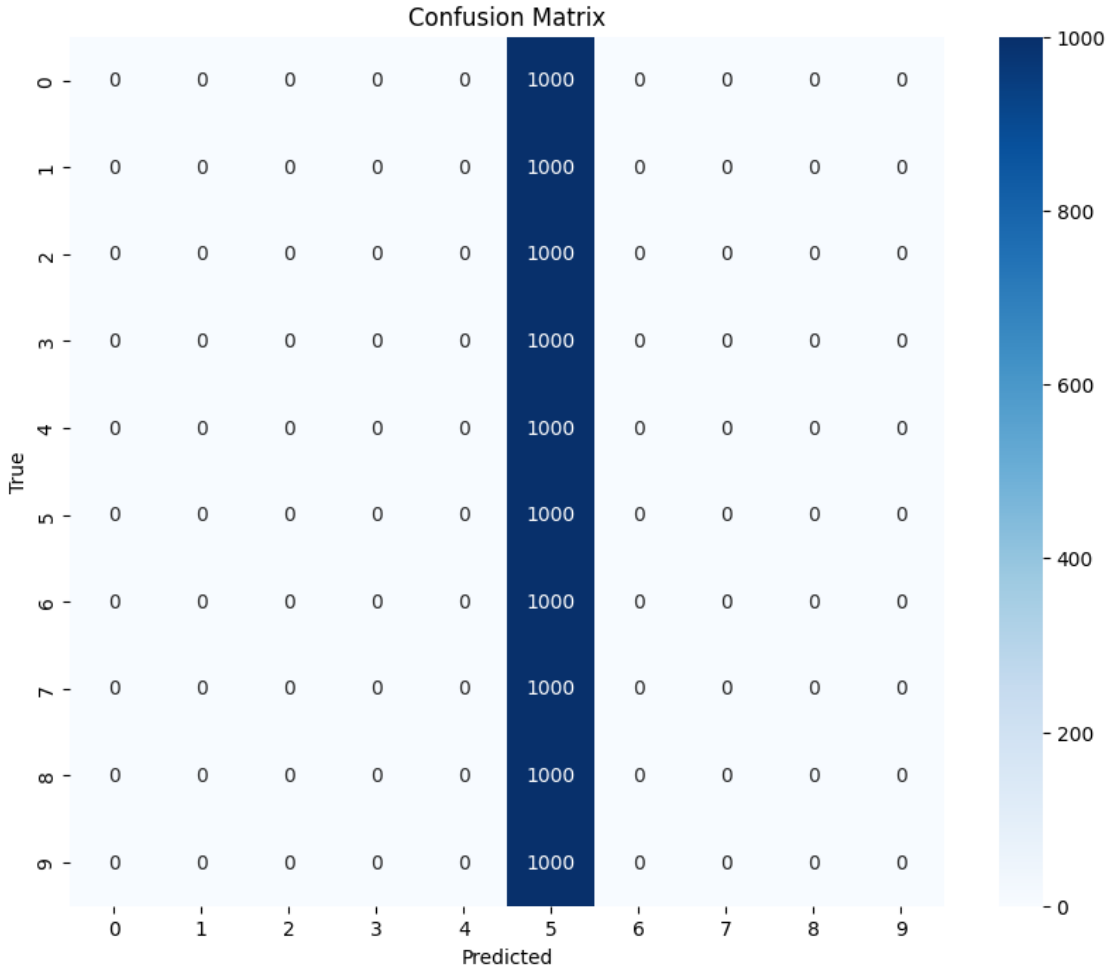
```
Epoch 66/100, Loss: 2.302973508834839
Epoch 67/100, Loss: 2.3097991943359375
Epoch 68/100, Loss: 2.3040199279785156
Epoch 69/100, Loss: 2.309020519256592
Epoch 70/100, Loss: 2.3031435012817383
Epoch 71/100, Loss: 2.3012280464172363
Epoch 72/100, Loss: 2.3028926849365234
Epoch 73/100, Loss: 2.298579216003418
Epoch 74/100, Loss: 2.3077588081359863
Epoch 75/100, Loss: 2.310840129852295
Epoch 76/100, Loss: 2.30551815032959
Epoch 77/100, Loss: 2.300353527069092
Epoch 78/100, Loss: 2.30832839012146
Epoch 79/100, Loss: 2.314821243286133
Epoch 80/100, Loss: 2.309818744659424
Epoch 81/100, Loss: 2.3131723403930664
Epoch 82/100, Loss: 2.3074374198913574
Epoch 83/100, Loss: 2.3015990257263184
Epoch 84/100, Loss: 2.2959861755371094
Epoch 85/100, Loss: 2.307415723800659
Epoch 86/100, Loss: 2.304121255874634
Epoch 87/100, Loss: 2.300257921218872
Epoch 88/100, Loss: 2.3056390285491943
Epoch 89/100, Loss: 2.3010313510894775
Epoch 90/100, Loss: 2.2989375591278076
Epoch 91/100, Loss: 2.3041763305664062
Epoch 92/100, Loss: 2.3012583255767822
Epoch 93/100, Loss: 2.2978739738464355
Epoch 94/100, Loss: 2.296121835708618
Epoch 95/100, Loss: 2.303925037384033
Epoch 96/100, Loss: 2.2980992794036865
Epoch 97/100, Loss: 2.3031370639801025
Epoch 98/100, Loss: 2.303201913833618
Epoch 99/100, Loss: 2.3019213676452637
Epoch 100/100, Loss: 2.3002774715423584
```

Evaluation

[17]:
```python
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

#predictions for test set
y_pred = model(x_test).numpy().argmax(axis=1)

#accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Test Accuracy: {accuracy * 100:.2f}%')
```

```
#confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Visualizing
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Test Accuracy: 10.00%
```



Analysis

Discuss the strengths and limitations of using an RBF network for this dataset.

Strengths of Using an RBF Network

Good for Non-linear Problems:

- RBF networks are particularly effective for problems where the decision boundary is non-linear. The Gaussian basis functions allow the model to adapt to complex patterns in the data, making them suitable for character recognition tasks like classifying ancient Japanese characters.

Locality:

- The RBF function has a localized response to inputs, meaning that it can effectively model complex local structures in the input space. This is beneficial in image classification, where specific features (like strokes or shapes) are important.

Robustness to Overfitting:

- With proper regularization and tuning of the number of RBF units, these networks can avoid overfitting, especially in cases where the training dataset is small relative to the input dimension (as is often the case with image data).

Fast Training:

- Once the centers of the RBF units are determined (using techniques like K-means), the training process is relatively fast since it primarily involves updating the weights, making it computationally efficient.

Interpretability:

- The structure of RBF networks allows for some interpretability. The influence of each RBF unit can be traced back to specific areas in the input space, helping to understand which features contribute most to the classifications.

Limitations of Using an RBF Network

Choice of Centers:

The performance of an RBF network heavily relies on the selection of RBF centers. If the centers are not well chosen, the model may fail to capture the underlying data distribution effectively.

Sensitivity to Parameters:

- The model's performance can be sensitive to the width parameter of the Gaussian functions. If the width is too small, the RBF units may respond too narrowly, leading to overfitting.

Scalability:

- As the dataset size increases, RBF networks can become computationally expensive. The K-means clustering step and the calculations for Gaussian activations can be slow with large datasets, especially when the input dimensionality is high.

Limited Generalization:

- RBF networks can struggle to generalize well beyond the training set if not carefully tuned. They may not perform as well on unseen data compared to other neural network architectures, like convolutional neural networks, which are often preferred for image data.

Not Widely Used in Practice:

- While RBF networks are theoretically interesting, they are less commonly used in practical applications compared to deep learning approaches. As a result, there may be fewer resources and community support for troubleshooting and optimization.

How does the number of RBF units affect model performance?

Underfitting vs. Overfitting:

- Few RBF Units: If the number of RBF units is too low, the network may not have enough capacity to capture the complexity of the data. This can lead to underfitting, where the model fails to learn the underlying patterns, resulting in poor performance on both training and test datasets.
- Too Many RBF Units: Conversely, if the number of RBF units is excessively high, the model may become overly complex, capturing noise in the training data. This can lead to overfitting, where the model performs well on training data but poorly on unseen data, resulting in a decrease in generalization ability.

Computational Complexity:

- The computational complexity of the model increases with the number of RBF units. Training and inference times may rise, which can be a limitation for real-time applications or when working with large datasets.

Performance Tuning:

- Finding the optimal number of RBF units is crucial. Typically, this is done through cross-validation, where different configurations are tested to see which provides the best trade-off between bias and variance. The optimal number may vary based on the dataset characteristics, such as complexity and size.

Influence on Decision Boundaries:

- The decision boundaries of the RBF network are influenced by the positions of the RBF centers and the number of units. More units can lead to more complex decision boundaries, which can be beneficial for datasets with intricate class distributions.