DNN VS CNN

April 12, 2024

[1]:
```
import zipfile
import os

# Path to the uploaded zip file
zip_path = 'archive (5).zip'
extract_folder = 'lfw_dataset'

# Extract the zip file
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_folder)

# List the contents of the extracted folder
extracted_files = os.listdir(extract_folder)
extracted_files
```

[1]:
```
['lfw-deepfunneled',
 'lfw_allnames.csv',
 'lfw_readme.csv',
 'matchpairsDevTest.csv',
 'matchpairsDevTrain.csv',
 'mismatchpairsDevTest.csv',
 'mismatchpairsDevTrain.csv',
 'pairs.csv',
 'people.csv',
 'peopleDevTest.csv',
 'peopleDevTrain.csv']
```

[2]:
```
from sklearn.datasets import fetch_lfw_people

# Load the Labeled Faces in the Wild dataset
lfw_people = fetch_lfw_people(min_faces_per_person=50, resize=0.5)

# Extract the features (images) and the target (labels)
X = lfw_people.data
y = lfw_people.target
lfw_people.images.shape, lfw_people.target_names
```

[2]:
```
((1560, 62, 47),
 array(['Ariel Sharon', 'Colin Powell', 'Donald Rumsfeld', 'George W Bush',
        'Gerhard Schroeder', 'Hugo Chavez', 'Jacques Chirac',
        'Jean Chretien', 'John Ashcroft', 'Junichiro Koizumi',
        'Serena Williams', 'Tony Blair'], dtype='<U17'))
```

[3]:
```python
from sklearn.datasets import fetch_lfw_people
from sklearn.preprocessing import MinMaxScaler

# Load the dataset with specified parameters
lfw_people = fetch_lfw_people(min_faces_per_person=50, resize=0.5)
X = lfw_people.data
y = lfw_people.target

# Apply Min-Max scaling
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Show the shape of the datasets
X_scaled.shape, y.shape
```

[3]:
```
((1560, 2914), (1560,))
```

[4]:
```python
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_lfw_people
from sklearn.preprocessing import MinMaxScaler

# Load LFW dataset
lfw_dataset = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
X = lfw_dataset.data
y = lfw_dataset.target

# Apply min-max feature scaling
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,␣
 ↪random_state=42)

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(8, activation='relu', input_shape=(X_train.
 ↪shape[1],)),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(len(set(y)), activation='softmax')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=20, batch_size=64,␣
 ↪validation_split=0.2)

# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

```
C:\Users\kusum\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/20
13/13                 3s 29ms/step -
accuracy: 0.3026 - loss: 1.8545 - val_accuracy: 0.4029 - val_loss: 1.6756
Epoch 2/20
13/13                 0s 7ms/step -
accuracy: 0.3943 - loss: 1.7111 - val_accuracy: 0.4029 - val_loss: 1.6541
Epoch 3/20
13/13                 0s 8ms/step -
accuracy: 0.4066 - loss: 1.6909 - val_accuracy: 0.4223 - val_loss: 1.6906
Epoch 4/20
13/13                 0s 7ms/step -
accuracy: 0.4172 - loss: 1.6961 - val_accuracy: 0.4175 - val_loss: 1.6379
Epoch 5/20
13/13                 0s 7ms/step -
accuracy: 0.4409 - loss: 1.6206 - val_accuracy: 0.4320 - val_loss: 1.5616
Epoch 6/20
13/13                 0s 7ms/step -
accuracy: 0.4686 - loss: 1.5373 - val_accuracy: 0.4757 - val_loss: 1.5683
Epoch 7/20
13/13                 0s 7ms/step -
accuracy: 0.4724 - loss: 1.5074 - val_accuracy: 0.4806 - val_loss: 1.5320
Epoch 8/20
13/13                 0s 6ms/step -
accuracy: 0.4756 - loss: 1.4329 - val_accuracy: 0.4951 - val_loss: 1.4681
```

```
Epoch 9/20
13/13            0s 7ms/step -
accuracy: 0.5082 - loss: 1.3516 - val_accuracy: 0.5097 - val_loss: 1.4584
Epoch 10/20
13/13            0s 7ms/step -
accuracy: 0.5277 - loss: 1.3274 - val_accuracy: 0.5146 - val_loss: 1.4526
Epoch 11/20
13/13            0s 7ms/step -
accuracy: 0.5036 - loss: 1.3675 - val_accuracy: 0.4757 - val_loss: 1.5819
Epoch 12/20
13/13            0s 6ms/step -
accuracy: 0.5049 - loss: 1.3343 - val_accuracy: 0.5049 - val_loss: 1.5086
Epoch 13/20
13/13            0s 6ms/step -
accuracy: 0.5192 - loss: 1.2711 - val_accuracy: 0.5388 - val_loss: 1.3994
Epoch 14/20
13/13            0s 6ms/step -
accuracy: 0.5624 - loss: 1.2309 - val_accuracy: 0.5194 - val_loss: 1.4175
Epoch 15/20
13/13            0s 7ms/step -
accuracy: 0.5566 - loss: 1.2173 - val_accuracy: 0.5243 - val_loss: 1.4054
Epoch 16/20
13/13            0s 8ms/step -
accuracy: 0.5326 - loss: 1.2955 - val_accuracy: 0.4903 - val_loss: 1.5770
Epoch 17/20
13/13            0s 7ms/step -
accuracy: 0.5026 - loss: 1.3340 - val_accuracy: 0.5146 - val_loss: 1.4017
Epoch 18/20
13/13            0s 7ms/step -
accuracy: 0.5417 - loss: 1.2733 - val_accuracy: 0.5146 - val_loss: 1.4918
Epoch 19/20
13/13            0s 7ms/step -
accuracy: 0.5649 - loss: 1.2117 - val_accuracy: 0.5194 - val_loss: 1.3823
Epoch 20/20
13/13            0s 7ms/step -
accuracy: 0.5587 - loss: 1.1989 - val_accuracy: 0.5388 - val_loss: 1.3446
9/9            0s 3ms/step -
accuracy: 0.5965 - loss: 1.1717
Test accuracy: 0.565891444683075
```

[33]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam
```

[44]:
```
import numpy as np
from sklearn.datasets import fetch_lfw_people
```

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
 ↪Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping

model = Sequential([
    Input(shape=(62, 47, 1)),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    Conv2D(256, (3, 3), activation='relu', padding='same'),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(y_categorical.shape[1], activation='softmax')
])

model.compile(optimizer=Adam(learning_rate=0.001),␣
 ↪loss='categorical_crossentropy', metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_loss', patience=3,␣
 ↪restore_best_weights=True)

history = model.fit(
    X_train,
    y_train,
    epochs=30,
    batch_size=64,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping]
)

test_loss, test_accuracy = model.evaluate(X_test, y_test)
print("Test Accuracy:", test_accuracy)

model.summary()
```

```
Epoch 1/30
20/20                 5s 142ms/step -
```

```
accuracy: 0.1971 - loss: 2.5220 - val_accuracy: 0.3846 - val_loss: 2.0245
Epoch 2/30
20/20            5s 126ms/step -
accuracy: 0.3444 - loss: 2.1579 - val_accuracy: 0.3846 - val_loss: 1.9650
Epoch 3/30
20/20            3s 153ms/step -
accuracy: 0.3446 - loss: 2.1375 - val_accuracy: 0.3846 - val_loss: 2.0060
Epoch 4/30
20/20            3s 147ms/step -
accuracy: 0.3470 - loss: 2.1566 - val_accuracy: 0.3846 - val_loss: 1.9843
Epoch 5/30
20/20            3s 147ms/step -
accuracy: 0.3418 - loss: 2.1675 - val_accuracy: 0.3846 - val_loss: 1.9495
Epoch 6/30
20/20            3s 157ms/step -
accuracy: 0.3346 - loss: 2.1480 - val_accuracy: 0.3846 - val_loss: 1.9380
Epoch 7/30
20/20            3s 154ms/step -
accuracy: 0.3520 - loss: 2.0825 - val_accuracy: 0.3718 - val_loss: 1.9378
Epoch 8/30
20/20            5s 128ms/step -
accuracy: 0.3565 - loss: 1.9992 - val_accuracy: 0.4167 - val_loss: 1.7605
Epoch 9/30
20/20            3s 150ms/step -
accuracy: 0.4164 - loss: 1.8325 - val_accuracy: 0.5449 - val_loss: 1.5685
Epoch 10/30
20/20            3s 152ms/step -
accuracy: 0.5015 - loss: 1.5140 - val_accuracy: 0.6282 - val_loss: 1.2682
Epoch 11/30
20/20            3s 143ms/step -
accuracy: 0.6078 - loss: 1.2044 - val_accuracy: 0.5962 - val_loss: 1.2578
Epoch 12/30
20/20            3s 151ms/step -
accuracy: 0.6373 - loss: 1.0935 - val_accuracy: 0.6795 - val_loss: 0.9722
Epoch 13/30
20/20            3s 143ms/step -
accuracy: 0.7179 - loss: 0.8680 - val_accuracy: 0.7179 - val_loss: 0.8295
Epoch 14/30
20/20            3s 151ms/step -
accuracy: 0.7568 - loss: 0.7545 - val_accuracy: 0.7308 - val_loss: 0.8416
Epoch 15/30
20/20            3s 152ms/step -
accuracy: 0.7891 - loss: 0.6295 - val_accuracy: 0.7821 - val_loss: 0.7381
Epoch 16/30
20/20            3s 152ms/step -
accuracy: 0.8212 - loss: 0.5427 - val_accuracy: 0.7885 - val_loss: 0.6402
Epoch 17/30
20/20            3s 148ms/step -
```

```
accuracy: 0.8495 - loss: 0.4380 - val_accuracy: 0.8141 - val_loss: 0.5804
Epoch 18/30
20/20                 3s 152ms/step -
accuracy: 0.8861 - loss: 0.3297 - val_accuracy: 0.8397 - val_loss: 0.5960
Epoch 19/30
20/20                 3s 152ms/step -
accuracy: 0.9021 - loss: 0.3135 - val_accuracy: 0.8462 - val_loss: 0.5143
Epoch 20/30
20/20                 3s 154ms/step -
accuracy: 0.9366 - loss: 0.1877 - val_accuracy: 0.8397 - val_loss: 0.4603
Epoch 21/30
20/20                 3s 150ms/step -
accuracy: 0.9530 - loss: 0.1503 - val_accuracy: 0.8718 - val_loss: 0.6142
Epoch 22/30
20/20                 3s 149ms/step -
accuracy: 0.9512 - loss: 0.1690 - val_accuracy: 0.8590 - val_loss: 0.5277
Epoch 23/30
20/20                 3s 150ms/step -
accuracy: 0.9614 - loss: 0.1308 - val_accuracy: 0.8782 - val_loss: 0.4879
5/5               0s 35ms/step -
accuracy: 0.8117 - loss: 0.6425
Test Accuracy: 0.8269230723381042
```

```
Model: "sequential_18"


 Layer (type)                          Output Shape                          ␣
↪Param #

 conv2d_85 (Conv2D)                    (None, 62, 47, 32)                        ␣
↪320

 max_pooling2d_72 (MaxPooling2D)       (None, 31, 23, 32)                        ␣
↪  0

 conv2d_86 (Conv2D)                    (None, 31, 23, 64)                     ␣
↪18,496

 max_pooling2d_73 (MaxPooling2D)       (None, 15, 11, 64)                        ␣
↪  0

 conv2d_87 (Conv2D)                    (None, 15, 11, 128)                    ␣
↪73,856

 max_pooling2d_74 (MaxPooling2D)       (None, 7, 5, 128)                         ␣
↪  0
```

```
conv2d_88 (Conv2D)                   (None, 7, 5, 128)                 ␣
↪147,584

conv2d_89 (Conv2D)                   (None, 7, 5, 256)                 ␣
↪295,168

flatten_18 (Flatten)                 (None, 8960)                          ␣
↪  0

dense_67 (Dense)                     (None, 512)                     ␣
↪4,588,032

dropout_2 (Dropout)                  (None, 512)                           ␣
↪  0

dense_68 (Dense)                     (None, 12)                          ␣
↪6,156


Total params: 15,388,838 (58.70 MB)

Trainable params: 5,129,612 (19.57 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 10,259,226 (39.14 MB)
```

[45]:
```
import numpy as np
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
 ↪Dropout, BatchNormalization, Activation
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2

# Load the dataset
lfw_people = fetch_lfw_people(min_faces_per_person=50, resize=0.5)
X = lfw_people.data
y = lfw_people.target
```

```
# Apply Min-Max feature scaling
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Reshape X for CNN and convert y to categorical
X_reshaped = X_scaled.reshape((-1, 62, 47, 1))  # Adjusting the dimensions for␣
 ↪CNN input
y_categorical = to_categorical(y)

# Split the data into training, validation, and test sets
X_train, X_temp, y_train, y_temp = train_test_split(X_reshaped, y_categorical,␣
 ↪test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)

# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
datagen.fit(X_train)

# Build the CNN model with Batch Normalization and L2 Regularization
model = Sequential([
    Conv2D(32, (3, 3), padding='same', kernel_regularizer=l2(0.001),␣
 ↪input_shape=(62, 47, 1)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), padding='same', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), padding='same', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), padding='same', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Activation('relu'),

    Conv2D(256, (3, 3), padding='same', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Activation('relu'),

    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(y_categorical.shape[1], activation='softmax')
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),␣
 ↪loss='categorical_crossentropy', metrics=['accuracy'])

# Set up early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)

# Train the model with data augmentation
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=64),
    epochs=50,  # Increased epochs
    validation_data=(X_val, y_val),
    callbacks=[early_stopping]
)

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print("Test Accuracy:", test_accuracy)

# Print the model architecture
model.summary()
```

```
Epoch 1/50
```

```
C:\Users\kusum\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(
C:\Users\kusum\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:120:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
20/20            8s 243ms/step -
accuracy: 0.2231 - loss: 3.2241 - val_accuracy: 0.1859 - val_loss: 2.8748
Epoch 2/50
20/20            5s 227ms/step -
accuracy: 0.3219 - loss: 2.5316 - val_accuracy: 0.3846 - val_loss: 2.7810
Epoch 3/50
20/20            5s 228ms/step -
accuracy: 0.3427 - loss: 2.4728 - val_accuracy: 0.3846 - val_loss: 2.7765
Epoch 4/50
20/20            5s 232ms/step -
accuracy: 0.3781 - loss: 2.3632 - val_accuracy: 0.3846 - val_loss: 2.7484
Epoch 5/50
20/20            5s 231ms/step -
accuracy: 0.3843 - loss: 2.3630 - val_accuracy: 0.3846 - val_loss: 2.7061
Epoch 6/50
20/20            5s 218ms/step -
accuracy: 0.4214 - loss: 2.2317 - val_accuracy: 0.3846 - val_loss: 2.6199
Epoch 7/50
20/20            5s 227ms/step -
accuracy: 0.4442 - loss: 2.2473 - val_accuracy: 0.4231 - val_loss: 2.6590
Epoch 8/50
20/20            5s 225ms/step -
accuracy: 0.4464 - loss: 2.1765 - val_accuracy: 0.1410 - val_loss: 2.6388
Epoch 9/50
20/20            5s 232ms/step -
accuracy: 0.4390 - loss: 2.1808 - val_accuracy: 0.3846 - val_loss: 2.5645
Epoch 10/50
20/20            5s 223ms/step -
accuracy: 0.4740 - loss: 2.0563 - val_accuracy: 0.3846 - val_loss: 2.5460
Epoch 11/50
20/20            5s 238ms/step -
accuracy: 0.4643 - loss: 2.0942 - val_accuracy: 0.3846 - val_loss: 2.4825
Epoch 12/50
20/20            5s 233ms/step -
accuracy: 0.4968 - loss: 1.9854 - val_accuracy: 0.3846 - val_loss: 2.4511
Epoch 13/50
20/20            5s 226ms/step -
accuracy: 0.4993 - loss: 1.9266 - val_accuracy: 0.3846 - val_loss: 2.5005
Epoch 14/50
20/20            5s 232ms/step -
accuracy: 0.4965 - loss: 1.9621 - val_accuracy: 0.3846 - val_loss: 2.4461
Epoch 15/50
20/20            5s 240ms/step -
accuracy: 0.5190 - loss: 1.8800 - val_accuracy: 0.3846 - val_loss: 2.4018
```

```
Epoch 16/50
20/20            5s 232ms/step -
accuracy: 0.5371 - loss: 1.8608 - val_accuracy: 0.3846 - val_loss: 2.3937
Epoch 17/50
20/20            5s 238ms/step -
accuracy: 0.5158 - loss: 1.8335 - val_accuracy: 0.3846 - val_loss: 2.2987
Epoch 18/50
20/20            5s 242ms/step -
accuracy: 0.5522 - loss: 1.7610 - val_accuracy: 0.3846 - val_loss: 2.3264
Epoch 19/50
20/20            5s 231ms/step -
accuracy: 0.5449 - loss: 1.8070 - val_accuracy: 0.3846 - val_loss: 2.3706
Epoch 20/50
20/20            5s 240ms/step -
accuracy: 0.5757 - loss: 1.7366 - val_accuracy: 0.3846 - val_loss: 2.3294
Epoch 21/50
20/20            5s 226ms/step -
accuracy: 0.5944 - loss: 1.6802 - val_accuracy: 0.3910 - val_loss: 2.1556
Epoch 22/50
20/20            5s 234ms/step -
accuracy: 0.5554 - loss: 1.7215 - val_accuracy: 0.3846 - val_loss: 2.2242
Epoch 23/50
20/20            5s 239ms/step -
accuracy: 0.5879 - loss: 1.6664 - val_accuracy: 0.4295 - val_loss: 2.0908
Epoch 24/50
20/20            5s 231ms/step -
accuracy: 0.5898 - loss: 1.6348 - val_accuracy: 0.5192 - val_loss: 1.8884
Epoch 25/50
20/20            5s 233ms/step -
accuracy: 0.6150 - loss: 1.5855 - val_accuracy: 0.4936 - val_loss: 1.8796
Epoch 26/50
20/20            5s 234ms/step -
accuracy: 0.6192 - loss: 1.5626 - val_accuracy: 0.6282 - val_loss: 1.5755
Epoch 27/50
20/20            5s 231ms/step -
accuracy: 0.6266 - loss: 1.5609 - val_accuracy: 0.6090 - val_loss: 1.5515
Epoch 28/50
20/20            5s 230ms/step -
accuracy: 0.6124 - loss: 1.5844 - val_accuracy: 0.6859 - val_loss: 1.3925
Epoch 29/50
20/20            5s 236ms/step -
accuracy: 0.6445 - loss: 1.5174 - val_accuracy: 0.7115 - val_loss: 1.3510
Epoch 30/50
20/20            5s 230ms/step -
accuracy: 0.6376 - loss: 1.5436 - val_accuracy: 0.6474 - val_loss: 1.3725
Epoch 31/50
20/20            5s 232ms/step -
accuracy: 0.6436 - loss: 1.5154 - val_accuracy: 0.7051 - val_loss: 1.3042
```

```
Epoch 32/50
20/20            5s 231ms/step -
accuracy: 0.6474 - loss: 1.4627 - val_accuracy: 0.7372 - val_loss: 1.1416
Epoch 33/50
20/20            5s 237ms/step -
accuracy: 0.6531 - loss: 1.4566 - val_accuracy: 0.7500 - val_loss: 1.1510
Epoch 34/50
20/20            5s 237ms/step -
accuracy: 0.6629 - loss: 1.4182 - val_accuracy: 0.7756 - val_loss: 1.0660
Epoch 35/50
20/20            5s 246ms/step -
accuracy: 0.6935 - loss: 1.3830 - val_accuracy: 0.7564 - val_loss: 1.1145
Epoch 36/50
20/20            5s 237ms/step -
accuracy: 0.6720 - loss: 1.3822 - val_accuracy: 0.7821 - val_loss: 1.0187
Epoch 37/50
20/20            5s 245ms/step -
accuracy: 0.6623 - loss: 1.3736 - val_accuracy: 0.7244 - val_loss: 1.1866
Epoch 38/50
20/20            5s 238ms/step -
accuracy: 0.6717 - loss: 1.3363 - val_accuracy: 0.7756 - val_loss: 1.0657
Epoch 39/50
20/20            5s 229ms/step -
accuracy: 0.7062 - loss: 1.3203 - val_accuracy: 0.7244 - val_loss: 1.1949
Epoch 40/50
20/20            5s 220ms/step -
accuracy: 0.6901 - loss: 1.3164 - val_accuracy: 0.7885 - val_loss: 1.1204
Epoch 41/50
20/20            5s 223ms/step -
accuracy: 0.6867 - loss: 1.3079 - val_accuracy: 0.7692 - val_loss: 1.1023
5/5            0s 25ms/step -
accuracy: 0.7059 - loss: 1.2397
Test Accuracy: 0.7243589758872986
```

```
Model: "sequential_19"


 Layer (type)                       Output Shape                        ␣
↪Param #

 conv2d_90 (Conv2D)                 (None, 62, 47, 32)                    ␣
↪320

 batch_normalization                (None, 62, 47, 32)                    ␣
↪128
 (BatchNormalization)                                                     ␣
↪
```

```
activation_5 (Activation)           (None, 62, 47, 32)                  ␣
↪  0

max_pooling2d_75 (MaxPooling2D)     (None, 31, 23, 32)                  ␣
↪  0

conv2d_91 (Conv2D)                  (None, 31, 23, 64)               ␣
↪18,496

batch_normalization_1               (None, 31, 23, 64)                  ␣
↪256
(BatchNormalization)                                                    ␣
↪

activation_6 (Activation)           (None, 31, 23, 64)                  ␣
↪  0

max_pooling2d_76 (MaxPooling2D)     (None, 15, 11, 64)                  ␣
↪  0

conv2d_92 (Conv2D)                  (None, 15, 11, 128)              ␣
↪73,856

batch_normalization_2               (None, 15, 11, 128)                 ␣
↪512
(BatchNormalization)                                                    ␣
↪

activation_7 (Activation)           (None, 15, 11, 128)                 ␣
↪  0

max_pooling2d_77 (MaxPooling2D)     (None, 7, 5, 128)                   ␣
↪  0

conv2d_93 (Conv2D)                  (None, 7, 5, 128)               ␣
↪147,584

batch_normalization_3               (None, 7, 5, 128)                   ␣
↪512
(BatchNormalization)                                                    ␣
↪

activation_8 (Activation)           (None, 7, 5, 128)                   ␣
↪  0
```

```
 conv2d_94 (Conv2D)                   (None, 7, 5, 256)                    ␣
↪295,168

 batch_normalization_4                (None, 7, 5, 256)                      ␣
↪1,024
 (BatchNormalization)                                                          ␣
↪

 activation_9 (Activation)            (None, 7, 5, 256)                        ␣
↪  0

 flatten_19 (Flatten)                 (None, 8960)                             ␣
↪  0

 dense_69 (Dense)                     (None, 512)                        ␣
↪4,588,032

 dropout_3 (Dropout)                  (None, 512)                              ␣
↪  0

 dense_70 (Dense)                     (None, 12)                             ␣
↪6,156
```

```
Total params: 15,393,702 (58.72 MB)

Trainable params: 5,130,828 (19.57 MB)

Non-trainable params: 1,216 (4.75 KB)

Optimizer params: 10,261,658 (39.15 MB)
```

[46]:
```
# Define the accuracies obtained from the models
dnn_test_accuracy = 0.5659  # Update this with your DNN's test accuracy if it␣
 ↪changes
cnn_test_accuracy = 0.8269  # Update this with your CNN's test accuracy if it␣
 ↪changes

print("Classification Results:")
print("Deep Neural Network (DNN) Test Accuracy: {:.2f}%".
 ↪format(dnn_test_accuracy * 100))
print("Convolutional Neural Network (CNN) Test Accuracy: {:.2f}%".
 ↪format(cnn_test_accuracy * 100))
```

```
Classification Results:
Deep Neural Network (DNN) Test Accuracy: 56.59%
Convolutional Neural Network (CNN) Test Accuracy: 82.69%
```

Report: Classification Results and Analysis Classification Results: Deep Neural Network (DNN) Test Accuracy: 56.59% Convolutional Neural Network (CNN) Test Accuracy: 82.69% These results indicate that the CNN significantly outperformed the DNN in this classification task.

Analysis Based on Classification Results a. How do the learning rate and filter size value affect the CNN classification accuracy?

The learning rate and filter size are fundamental hyperparameters in training convolutional neural networks:

Learning Rate: This controls the step size during the gradient descent optimization process. An optimally set learning rate helps the model converge efficiently to a good solution. If the learning rate is too high, the model may converge too quickly to a suboptimal solution or overshoot the minima. If it's too low, the model may take too long to converge, leading to lengthy training times or getting stuck in local minima. The CNN's higher accuracy suggests that the learning rate was well-tuned for balancing the speed of convergence with the stability of the training process.

Filter Size: This determines the size of the kernel that moves over the input image to extract features. Smaller filters might capture finer details, while larger filters tend to capture broader features of the image. The choice of filter size affects the model's ability to generalize from the training data to unseen data. In this case, the effective use of filter sizes likely contributed to the CNN's ability to recognize and generalize critical features from the input data, resulting in higher accuracy.

b. Does the increased hidden layer in DNN size improve the classification accuracy?

Increasing the size or number of hidden layers in a neural network can improve its capacity to learn more complex features from the data, which is often beneficial for tasks involving high-dimensional input such as images. However, there are several caveats:

Capacity vs. Overfitting: While more layers can increase the model's capacity to learn, they can also make the model prone to overfitting, especially if there isn't enough data to support the complexity of the model. Overfitting results in high performance on the training data but poor generalization to new, unseen data.

Training Challenges: Larger networks are more challenging to train. They are more susceptible to issues like vanishing gradients, where gradients become too small for effective training as they propagate back through the layers during training. This can be mitigated with techniques like proper initialization, using ReLU activations, or employing batch normalization.

In this scenario, despite increasing the DNN's hidden layer size, the CNN still outperformed it. This outcome suggests that merely increasing the size of the network was not sufficient to achieve better results compared to the CNN, which efficiently leverages spatial hierarchies and parameter sharing through convolutional layers. It's possible that the DNN could improve with more data, better regularization strategies, or an optimized architecture specifically suited for the dataset in use.

Conclusion: The comparison clearly demonstrates the advantages of using CNNs for image-based tasks over DNNs due to their architectural benefits in handling spatial data. Adjusting learning rates and filter sizes critically influences CNN performance, while simply increasing the DNN size without strategic considerations can lead to inefficiencies and suboptimal learning.